Datona-Lib

Release 0.0.1a

Aug 01, 2020

User Documenation

Version 0.0.2.

This is an alpha version designed for experimental use only.

CHAPTER 1

What is Datona-Lib?

Datona-Lib is an open-source Node.js library that encapsulates the core cryptographic, blockchain, vault and communications features needed to develop on the datona.io platform. It is available on github here. It is intended for developers of vault servers, identity apps and requester software, and for those wanting to experiment with *Smart Data Access*.

Datona-Lib consists of four components:

- *datona-crypto* implements the core cryptographic functions such as hashing and digital signatures.
- *datona-vault* allows developers of owner and requester apps to interface with and manage a remote vault without needing to implement the *Datona Vault Application-Layer Request Protocol*. For developers of vault servers it fully encapsulates the *Vault Keeper function*.
- *datona-comms* implements the Datona *Application Layer Protocol*.
- *datona-blockchain* provides the interface to *Smart Data Access Contracts* on the blockchain.

CHAPTER 2

Contents

2.1 What Is Smart Data Access?

Smart Data Access is a technology that gives individuals control of their online and offline data to combat the rising unethical use and abuse of our data. When someone shares their data with someone else, Smart Data Access acts like a piece of elastic ensuring the owner of the data always knows who has their data and can update or withdraw it any time, wherever the data is held.

From an organisation's perspective, Smart Data Access helps to automate compliance with data protection regulations and enables new use cases for decentralised applications. It aims to work closely with existing front-end and back-end infrastructure, rather than reinvent it.

From a regulator's perspective, Smart Data Access is a technological solution to the GDPR 8 Rights For Individuals. In addition it enables new capabilities such as remote auditing and automated auditing of organisations to ensure they are only using data within owners' permissions.

More details can be found in the Technical White Paper.

2.1.1 How Does It Work?

Data is protected by Smart Data Access software and controlled by a Smart Data Access Contract (S-DAC). The S-DAC is the piece of elastic that connects the owner with their data. It specifies the terms and conditions for the management, use, access, update, expiry and deletion of the data. A Smart Data Access software component, called the Vault Keeper, installed on each data server creates a firewall around the data and ensures it is only accessible if the contract permits it. If the S-DAC is terminated, the Vault Keeper will automatically delete the data. Data protected by a Vault Keeper is said to be in a *Vault*.

The S-DAC is deployed on the platform's blockchain - currently Ethereum - so is always accessible from anywhere in the world, whether the data server is available or not. This allows the data owner to indirectly control the data in the vault at any time. The Smart Data Access software in the data server includes a full Ethereum node so has an up-to-date view of all S-DACs on the blockchain and can query contracts locally for high performance.

The data itself can be stored anywhere. Organisations can integrate Smart Data Access into their existing customer databases or can use a third-party 'data vault' cloud service. Alternatively, owners can host the data themselves. The data can be accessible from the web or be isolated behind a firewall. It can even be held completely offline, with some limitations.

Smart Data Access can also be used to control single access data shares where the data is used immediately and then deleted. Single access shares do not need the blockchain or a data vault service.

Datona labs hosts an experimental cloud server on the Ethereum testnet at datonavault.com for use by all developers. See *How To Use*.

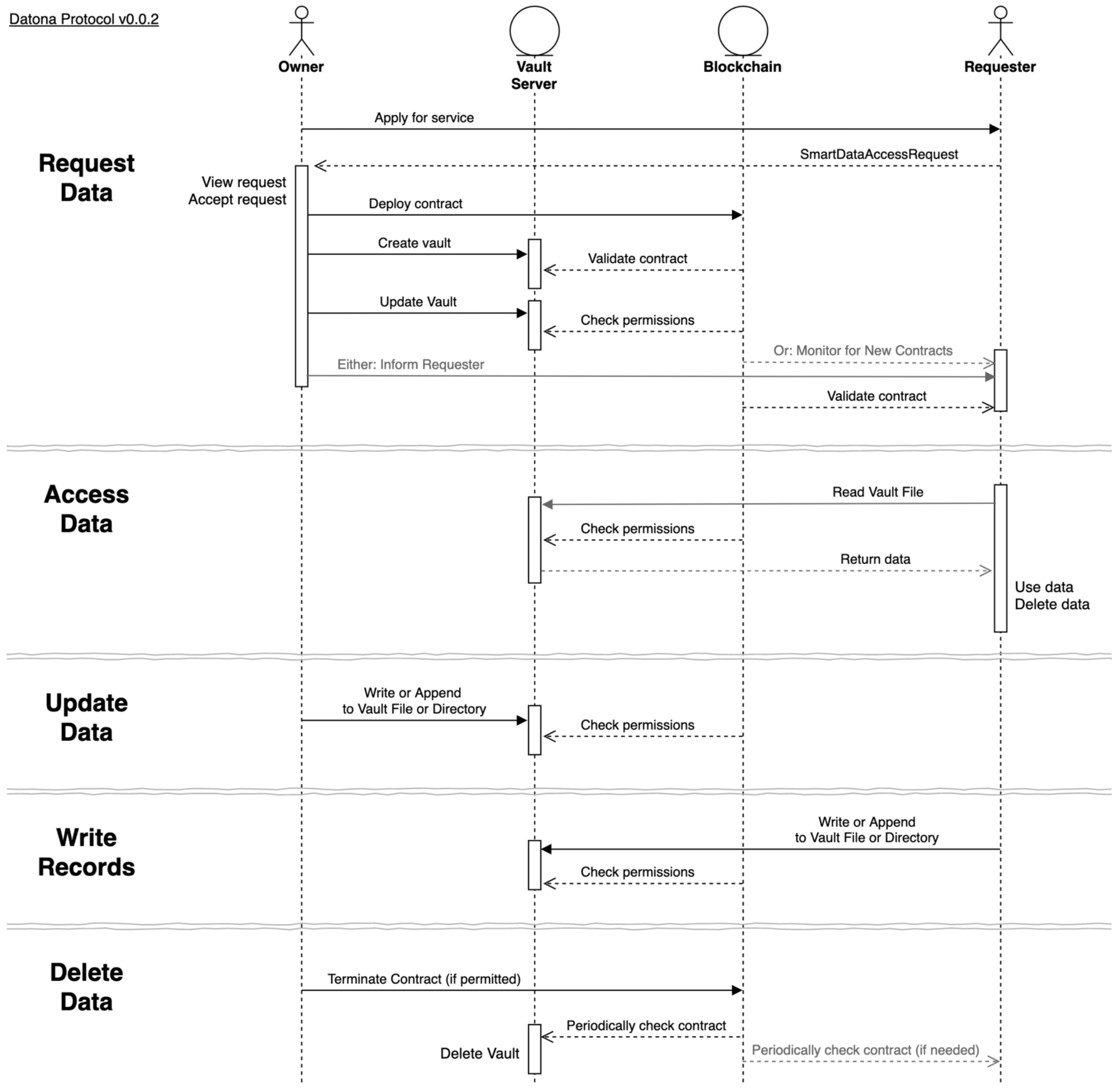
2.1.2 Smart Data Access Contracts

An S-DAC must conform to the *Datona S-DAC Interface Specification* but its implementation is entirely user defined. An S-DAC is a state machine that encodes the life-cycle of the data it controls, specifying who can access the data at what phase and when the owner can update or delete the data. An S-DAC can be as simple or as complex as needed for the particular use case. See *Build a Smart Data Access Contract*.

At this time, the *Datona S-DAC Interface Specification* is a minimal specification and will evolve over time. In the future a formal standard will be published.

S-DACs will eventually be designed to contain (or reference) a summary of terms for presentation to and acceptance by the user. To ensure that the terms match the life-cycle encoded in the contract, all S-DACs will need to be independently reviewed and validated by an expert community. It's possible there will eventually be a community library containing off-the-shelf, pre-validated contract templates for general use and an active community of developers creating and validating bespoke contracts for organisations.

2.1.3 Life-Cycle of Smart Data



The Smart Data Access process begins with a **Request** for data between the *Requester* and the data *Owner*, known as a *Smart Data Access Request*. This request identifies the S-DAC to be used to control the contract and how to inform the Requester if the request is accepted. (In future the request may also contain the summary of terms and a list of trusted vault services). The request can be presented to the owner in any way. For example, it could be a QR code on a leaflet; a url behind a button on a web page; or a contactless hardware device on a desk. In all cases the data owner initiates the process by applying for the service.

Once the Owner accepts a request their app must deploy the contract to the blockchain, store the data in a vault and finally inform the requester of the contract address and vault service used. The Requester must validate the deployed contract to ensure the correct one has been deployed.

Whenever the requester needs to use the data it can **Access** the vault. The vault server will check the S-DAC to ensure the Requester is permitted before returning the data. The requester can then use the data and immediately delete its copy. The Requester can access the data any time and as many times as the S-DAC allows.

If the Owner's information changes, they can **Update** the data through the vault server's public api. The vault server will check the S-DAC to ensure the owner is permitted to update the data before updating the vault.

If the use case requires it, the Requester can **Write Records** back to the vault, for example to pass a policy certificate back to the Owner or to append to a log file. The S-DAC can be configured to give different actors permission to write to different files and directories in the vault.

If the S-DAC permits, the Owner can **Delete** the data at any time by terminating the contract. Once terminated, any attempt by the Requester to access the vault will be denied. The vault server will periodically check for contract expiry and when found will permanently delete the vault.

2.2 How To Use

This section describes the use of datona-lib by the three primary types of developers: Requesters, Owner App Developers and Data Vault Service Providers.

See the *Smart Data Access Life-Cycle* for the overall process.

2.2.1 Requesters

Building a Smart Data Access Contract

All S-DACs must comply with the *Smart Data Access Contract Interface* but the implementation will depend on the use case. S-DACs can be simple, for example to give indefinite access until terminated; or they can be highly complex giving access to different Requesters depending on a complex workflow supported by external blockchain oracles.

Here is an example of a simple contract that automatically terminates after a given number of days. It permits access for a single Requester and permits the Owner or Requester to terminate at any time.

```
pragma solidity ^0.6.3;

import "SDAC.sol";

contract Duration_SDAC is SDAC {

    address public permittedRequester;
    uint public contractDuration;
    uint public contractStart;
    bool terminated = false;


    modifier onlyOwnerOrRequester {
        require( msg.sender == owner || msg.sender == permittedRequester );
        _;
    }

    constructor( address _permittedRequester, uint _contractDuration ) public {
        permittedRequester = _permittedRequester;
        contractDuration = _contractDuration;
        contractStart = block.timestamp;
    }

    function getPermissions( address requester, address file ) public view override␣
↪returns (byte) {
        if ( file == address(0) && !hasExpired() ) {
            if (requester == owner) return NO_PERMISSIONS | READ_BIT | WRITE_BIT |␣
↪APPEND_BIT;
            if (requester == permittedRequester) return NO_PERMISSIONS | READ_BIT;
        }
        return NO_PERMISSIONS;
    }

    function isPermitted( address requester ) public view returns (bool) {
        return ( getPermissions(requester, address(0)) & READ_BIT ) > 0;
    }

    function hasExpired() public view override returns (bool) {
        return terminated ||
               (block.timestamp - contractStart) >= (contractDuration * 1 days);
    }

    function terminate() public override onlyOwnerOrRequester {
        terminated = true;
    }

    function getOwner() public view returns (address) {
        return owner;
    }

}
```

File Permissions

Protocol v0.0.2 introduced file-based read, write and append permissions to S-DACs. This allows a vault to be split into compartments (files and directories) each having different access permissions for different actors. This could be used, for example, to allow the Owner's name to be accessible to the Requester while their name and address is accessible to a third-party delivery company.

The S-DAC interface does not support standard file names. Each file and directory is instead uniquely identified by a hash. What hash name is given to each file is at the discretion of the user and should form part of the Smart Data Access Request.

The getPermissions function in the S-DAC is responsible for returning the correct permissions for the requester and file passed as its input parameters. Permissions are returned as a single byte of the binary form `d----rwa`, where d is the most significant bit and if set (1) indicates the file is a directory. The read-bit, write-bit and append-bit will be set (1) if that permission is granted.

Read and *write* file permissions behave in the standard way. The *append* permission allows the user to append data to a file but not to overwrite what has been written before. This can be useful for log files and audit trails. The append permission for a directory allows new files to be written to that directory but does not allow existing files to be overwritten. There is no execute permission since files cannot be executed on a vault server.

The distinction between files and directories is in how the vault server responds to an access request. For files the response will contain the data within the file, if the requester is permitted to access it. For directories it will contain a list of filenames. The files within a directory inherit their permissions from the parent directory and must be accessed with separate requests.

Here is an example abstract S-DAC that implements UNIX-like user/group/others permissions for individual files.

```
pragma solidity ^0.6.3;

import "SDAC.sol";


/*
 * Abstract file based SDAC that allows a vault server to manage multiple files and␣
↪directories within a vault.
 * Each file or directory has its own unix-like user/group/others permissions of the␣
↪form rwa (read, write, append).
 *
 * Groups and files are set on construction and remain static throughout the life of␣
↪the contract. File owner, group and
 * permissions are also set on construction but can be modified later. As with unix␣
↪file systems only the file's owner
 * can modify its group and permissions. Unlike unix systems there is no admin, root␣
↪or sudo group.
 */

struct FilePermissions {
    address user;
    address group;
    bytes2 permissions;
}


abstract contract FileBasedSdac is SDAC {

    mapping (address => FilePermissions) internal files;
    mapping (address => mapping(address => bool)) internal groups;

    // Internal permissions bitmap
    uint8 internal constant INTERNAL_PERMISSIONS_USER_BIT = 6;
    uint8 internal constant INTERNAL_PERMISSIONS_GROUP_BIT = 3;
    uint8 internal constant INTERNAL_PERMISSIONS_OTHERS_BIT = 0;
```

(continues on next page)

(continued from previous page)

```
    bytes2 internal constant INTERNAL_PERMISSIONS_DIRECTORY_MASK = 0x0200;
    bytes2 internal constant INTERNAL_PERMISSIONS_USER_WRITE_MASK = 0x0080;


    // create a new user group
    function addGroup(address id, address[] memory users) internal {
        for (uint i=0; i<users.length; i++) {
            groups[id][users[i]] = true;
        }
    }


    // add a new file with the given permissions.  Permissions are a 2-byte field␣
↪with the bit form ------dr warw arwa,
    // reflecting unix-like permissions for user, group, other.
    //   e.g. 0x01E0 describes a file (not a directory) with permissions rwar-----
    //   i.e. user (owner) has read, write, append permissions, group has read␣
↪permissions and others have no permissions.
    function addFile(address id, FilePermissions memory permissions) internal {
        files[id] = permissions;
    }


    // File based permissions returned as a byte with the form d----rwa.
    // Mimics unix file permissions:
    //   - returns the owner permissions if the requester is the owner of the file
    //   - returns the group permissions if the requester is not the owner but␣
↪belongs to the file's group
    //   - returns the other permissions if the requester is neither the owner nor a␣
↪group member
    // Deliberately does not throw if a file does not exist, returns 0 instead.
    function getPermissions( address requester, address file ) public view override␣
↪returns (byte) {
        address fileOwner = files[file].user;
        address fileGroup = files[file].group;
        byte directoryFlag = files[file].permissions & INTERNAL_PERMISSIONS_DIRECTORY_
↪MASK > 0 ? DIRECTORY_BIT : byte(0);
        if ( fileOwner == address(0) || this.hasExpired() ) {
            return NO_PERMISSIONS;
        }
        else if (requester == fileOwner) {
            return (byte)(files[file].permissions >> INTERNAL_PERMISSIONS_USER_BIT) &␣
↪ALL_PERMISSIONS | directoryFlag;
        }
        else if (groups[fileGroup][requester]) {
            return (byte)(files[file].permissions >> INTERNAL_PERMISSIONS_GROUP_BIT) &
↪ ALL_PERMISSIONS | directoryFlag;
        }
        else {
            return (byte)(files[file].permissions >> INTERNAL_PERMISSIONS_OTHERS_BIT)␣
↪& ALL_PERMISSIONS | directoryFlag;
        }
    }


    // change a file's permissions
    function chmod(address file, bytes2 permissions) public {
```

(continues on next page)

(continued from previous page)

```
        require( files[file].user == msg.sender, 'Operation not permitted' );
        require( (files[file].permissions & INTERNAL_PERMISSIONS_USER_WRITE_MASK) > 0,
↪ 'Operation not permitted' );
        files[file].permissions = permissions;
    }


    // change a file's owner
     function chown(address file, address user) public {
        require( files[file].user == msg.sender, 'Operation not permitted' );
        require( (files[file].permissions & INTERNAL_PERMISSIONS_USER_WRITE_MASK) > 0,
↪ 'Operation not permitted' );
        files[file].user = user;
    }


    // change a file's owner and group
    function chown(address file, address user, address group) public {
        chown(file, user);
        files[file].group = group;
    }


    // change a file's group
    function chgrp(address file, address group) public {
        require( files[file].user == msg.sender, 'Operation not permitted' );
        require( (files[file].permissions & INTERNAL_PERMISSIONS_USER_WRITE_MASK) > 0,
↪ 'Operation not permitted' );
        files[file].group = group;
    }

}
```

Building a Smart Data Access Request

Here is an example *Smart Data Access Request Packet* for passing to a data owner. The *hash* in this request is a hash of the runtime bytecode of the Duration_SDAC above. The *url* in this request is the URL of the Requester's server that will handle a *Smart Data Access Response Packet* from the Owner.

In this case the Requester has added a *customerId* field to the accept and reject transaction templates. This number will be added to the response that the Owner returns to the Requester.

```
{
  "txnType": "SmartDataAccessRequest",
  "version": "0.0.1",
  "contract": {
    "hash": "5573012304cc4d87a7a07253c728e08250db6821a3dfdbbbcac9a24f8cd89ad4",
  },
  "api": {
    "url": {
      "scheme": "file",
      "host": "my.server.io",
      "port": "8601"
    },
    "acceptTransaction": {
```

(continues on next page)

(continued from previous page)

```
      "customerId": "10001"
    },
    "rejectTransaction": {
      "customerId": "10001"
    }
  }
}
```

Creating a Server to Handle a Smart Data Access Response

If the Owner accepts the Smart Data Access Request then they will inform the Requester of the S-DAC's blockchain address and where the data is being held. To do this the Requester must run a server to handle the *Smart Data Access Response Packet*.

Example of a basic server. When handling a response the server must perform some validation on the deployed contract. As a minimum it must check that the deployed contract is of the expected type by checking its runtime bytecode. In this example it also checks that the signatory of the response is the owner of the contract.

```
const datona = require('datona-lib');
const assert = datona.assertions;

//
// Constants
//

const myKey = new datona.crypto.key(
↪"e68e40257cfee330038c49637fcffff82fae04b9c563f4ea071c20f2eb55063c");
const sdacHash = "5573012304cc4d87a7a07253c728e08250db6821a3dfdbbbcac9a24f8cd89ad4";
const sdacSourceCode = require("./contracts/" + sdacHash + ".json");


//
// Server
//

var customers = [];

const myServer = net.createServer(connection);
myServer.listen(8601);

connection(c){

  c.on('data', (buffer) => {
    try {
      // Decode the transaction and validate the structure of the response packet. ␣
↪These will throw if not valid
      const txn = datona.comms.decodeTransaction(data);
      const sdaResponse = txn.txn;
      assert.equals(sdaResponse.txnType, "SmartDataAccessResponse", "SDA Response is␣
↪invalid: txnType")

      // Handle depending on the response type
      switch (sdaResponse.responseType) {
        case "accept":
          assert.isAddress(sdaResponse.contract, "SDA Response is invalid: contract")
```

(continues on next page)

(continued from previous page)

```
          assert.isAddress(sdaResponse.vaultAddress, "SDA Response is invalid:␣
↪vaultAddress")
          assert.isUrl(sdaResponse.vaultUrl, "SDA Response is invalid: vaultUrl")

          // Connect to the Owner's S-DAC on the blockchain
          const contract = new datona.blockchain.Contract(sdacSourceCode.abi,␣
↪sdaResponse.contract);

          // Verify the signatory is the owner of the contract and that the correct␣
↪contract has been deployed,
          contract.assertOwner(txn.signatory)
            .then( () => { contract.assertBytecode(sdacSourceCode.runtimeBytecode) })
            .then( () => {
              // Contract is valid so record the new customer and return a success␣
↪response
              customers.push(txn.data);
              sendResponse(datona.comms.createSuccessResponse());
            })
            .catch( (error) => {
              sendResponse(datona.comms.createErrorResponse(error));
            });
          break;
        case "reject":
          logger.log("Customer reject: "+sdaResponse.reason);
          sendResponse(datona.comms.createSuccessResponse());
          break;
        default:
          throw new datona.errors.TransactionError("Invalid responseType:
↪"+sdaResponse.responseType);
      }
    }
    catch (error) {
      sendResponse(datona.comms.createErrorResponse(error));
    }
  });

}

function sendResponse(c, response) {
  c.write(encodeTransaction(response, myKey));
  c.end();
}
```

Monitoring For New Contracts

An alternative to using a server to receive Smart Data Access Responses is to monitor the blockchain directly for new vaults that you are permitted to access. This method will only work if you know the address and url of the vault server used by all customers, or if you require customers to identify the vault service in the contract itself. The datona-blockchain *subscribe* function supports the registering of a callback to be called whenever a new contract of a given type (with a given runtime bytecode) is deployed on the blockchain and you are permitted to access the data it controls.

Example:

```
const myContract = require("../contracts/myContract.json");
const subscription = subscribe(datona.crypto.hash(myContract.runtimeBytecode),␣
↪registerNewCustomer, myKey.address);

function registerNewCustomer(contractAddress) {
  const newCustomer = { contract: contractAddress };
  customers.push(newCustomer);
}
```

Accessing a Customer's Data

To access a data from a customer's vault you will need the contract address, vault URL and vault server's public address from the SmartDataAccessResponse received from the data owner. The datona-vault *RemoteVault* class is used to access the vault.

```
const customer = customers[0];
const remoteVault = new RemoteVault(customer.vaultUrl, customer.contract, myKey,␣
↪customer.vaultAddress);

remoteVault.read()
  .then( (data) => { console.log("vault contains: "+data) )
  .catch( console.error );
```

If the vault contains specific files then they should be read individually:

```
const customersFolder = "0xF000000000000000000000000000000000000001"

remoteVault.read(customersFolder)
  .then( (data) => { console.log("folder contains files:\n"+data) )
  .catch( console.error );

remoteVault.read(customersFolder+"/name")
  .then( (data) => { console.log("Customer name: "+data) )
  .catch( console.error );

remoteVault.read(customersFolder+"/email")
  .then( (data) => { console.log("Customer email: "+data) )
  .catch( console.error );
```

2.2.2 Owner App Developers

Receiving a Smart Data Access Request

A Smart Data Access Request is passed from Requester to Owner as a *Signed Transaction*. Once received, the *SmartDataAccessRequest* class is used to decode and validate it. The app can then display the request to the Owner for acceptance or rejection.

```
const datona = require('datona-lib');

const myKey = new datona.crypto.key(
↪"b94452c533536500e30f2253c96d123133ca1cbdb987556c2dc229573a2cd53c");

const request = new datona.comms.SmartDataAccessRequest(signedTxnStr, myKey);
```

Accepting a Smart Data Access Request

The following example demonstrates the use of the *Contract* class to deploy a new S-DAC on the blockchain, and the *RemoteVault* class to create the vault. It uses the *accept* method of the *SmartDataAccessRequest* class to inform the Requester.

```
const vaultServerAddress = "0x288b32F2653C1d72043d240A7F938a114Ab69584",

const vaultUrl = {
  scheme: "file",
  host: "datonavault.com",
  port: 8964
}

var myDataShares = [];

//
// Accept Request
//

// Read contract bytecode and ABI from file system and create a Contract object
const contractSourceCode = require("./contracts/" + request.data.contract.hash);
const sdac = new datona.blockchain.Contract(contractSourceCode.abi);

// Function to create a new vault and store the data.  Returns a Promise.
function createAndDeployVault(){
  const vault = new datona.vault.RemoteVault( vaultUrl, sdac.address, myKey,␣
↪vaultServerAddress );
  return vault.create()
    .then( vault.write("Hello World!") );
}

// Function to send the contract address and vault URL to the requester.  Returns a␣
↪Promise.
function recordContractAndInformRequester(){
  myDataShares.push( {
    contract: sdac.address,
    vault: {
      address: vaultServerAddress,
      url: vaultUrl
    }
  });
  return request.accept(sdac.address, vaultServerAddress, vaultUrl);
}

// Deploy the contract, create the vault and inform the requester
sdac.deploy(myKey, contractSourceCode.bytecode, [request.signatory])
  .then( createAndDeployVault )
  .then( recordContractAndInformRequester )
  .catch( console.error );
```

Accessing a Vault

To access all data in the vault use the datona-vault *RemoteVault* class, in the same way as a Requester *accesses a customer's data* above.

```
const dataShare = myDataShares[0];

const remoteVault = new RemoteVault(dataShare.vault.url, dataShare.contract, myKey,␣
↪dataShare.vault.address);

remoteVault.read()
  .then( (data) => { console.log("vault contains: "+data) )
  .catch( console.error );
```

Reading from a Specific Vault File

To read the data from a specific file in the vault include the filename as part of the read request.

```
remoteVault.read("0xF000000000000000000000000000000000000001/name.txt")
  .catch( console.error );
```

In this case the file is `name.txt` and it inherits its permissions from the parent directory `0xF000000000000000000000000000000000000001`. The permissions for this directory must be encoded in the contract.

Listing a Directory

If the contract supports directories then its possible to list the files held a directory by simply reading it.

```
remoteVault.read("0xF000000000000000000000000000000000000001")
  .then( console.log );
  .catch( console.error );
```

If `0xF000000000000000000000000000000000000001` is a directory (has the directory bit set in its contract permissions) then the vault server will return a list of names of all the files in the vault directory, separated by newlines. If the file is not a directory then the contents of the file will be returned.

Writing to a Vault

To write (or overwrite) the data in the vault use the *RemoteVault* class.

```
const dataShare = myDataShares[0];

const remoteVault = new RemoteVault(dataShare.vault.url, dataShare.contract, myKey,␣
↪dataShare.vault.address);

remoteVault.write("Hi World!")
  .catch( console.error );
```

Writing to a Specific Vault File

To write (or overwrite) the data in a file include the filename as part of the write request.

```
remoteVault.write("Barney Rubble", "0xF000000000000000000000000000000000000001/name.
↪txt")
  .catch( console.error );
```

In this case the file is `name.txt` and it inherits its permissions from the parent directory `0xF000000000000000000000000000000000000001`. The permissions for this directory must be encoded in the contract.

Appending to a Specific Vault File

Appending data to a file is done in the same way as writing but uses the `append` method.

```
const logfile = "0xF000000000000000000000000000000000000002";

remoteVault.append("\nThu 16 Apr 2020 14:34:47 BST - Name updated", logfile)
  .catch( console.error );
```

Deleting a Vault

To delete the data in the vault simply terminate the contract. No-one can access the vault once the contract has been terminated, and the data vault server will delete the data when it next checks the contract. If required the *delete* method of the *RemoteVault* class can be used to force the Data Vault Server to delete the data right away (not shown).

```
const dataShare = myDataShares[0];

// Read contract bytecode and ABI from file system and create a Contract object
const contractSourceCode = require("./contracts/" + dataShare.contract.hash);
const sdac = new datona.blockchain.Contract(contractSourceCode.abi, dataShare.
↪contract);

// Terminate contract
sdac.terminate(myKey)
  .catch( console.error );
```

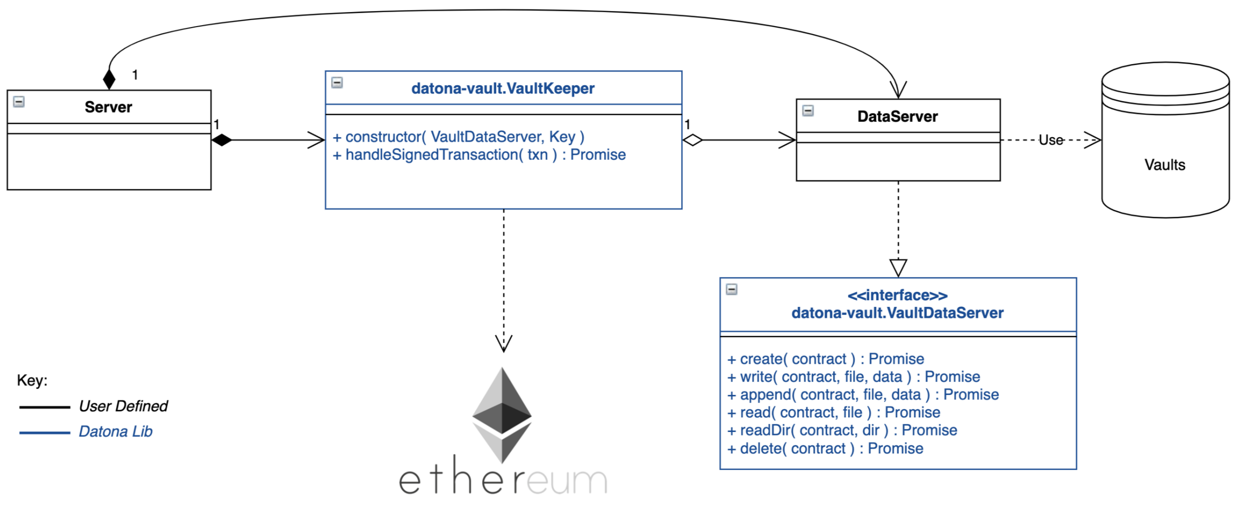
2.2.3 Vault Service Providers

Creating a Data Vault Server

A Data Vault Server can be a public cloud-based service, a locally hosted server within an organisation or a home-based server. Whatever the type of server, it must implement the Datona *Application Layer Protocol* and undertake the appropriate permission checks before accepting a create, update, access or delete request.

The Datona Lib *VaultKeeper* class provides these capabilities leaving the developer to implement the server's data layer. The VaultKeeper provides the following capabilities:

- decoding and validating incoming *SignedTransaction* packets and the *VaultRequest* packet within;
- verifying the appropriate permissions for accepting requests against the S-DAC on the blockchain;
- if permitted, calls a user-defined *VaultDataServer* instance to handle the request;
- constructing the appropriate success or error *VaultResponse* packet and encoding it as a *SignedTransaction*.


1
Server
1
datona-vault.VaultKeeper
+ constructor(VaultDataServer, Key)
+ handleSignedTransaction(txn) : Promise
1
DataServer
Use
Vaults
<<interface>>
datona-vault.VaultDataServer
+ create(contract) : Promise
+ write(contract, file, data) : Promise
+ append(contract, file, data) : Promise
+ read(contract, file) : Promise
+ readDir(contract, dir) : Promise
+ delete(contract) : Promise
Key:
User Defined
Datona Lib
ethereum

The diagram above shows the class relationships between the user-defined classes in black and the datona-lib classes in blue. The user-defined `DataServer` class must implement the VaultDataServer interface and promise to handle the 4 types of data request. All permission checks will have already been performed by the `VaultKeeper` so the `DataServer` need only perform the requests unconditionally.

Example bare-minimal server and VaultDataServer implementation. This example is a plain TCP server. It could instead be written as an HTTP or WebSocket server.

```
const datona = require("datona-lib");
const net = require('net');

const myKey = new datona.crypto.Key(
↪"ae139af24306ecac804cfe974398d6d76361287d7b96d9e165d9bcb99a64b6ce");


//
// Example Server.  Has no logging or sigterm detection.
//

const vaultManager = new RamBasedVaultDataServer();
const vaultKeeper = new datona.vault.VaultKeeper(vaultManager, myKey);
const server = net.createServer(connection);

function connection(c){

  c.on('data', (buffer) => {
    vaultKeeper.handleSignedRequest(buffer.toString())
      .then( (response) => {
        c.write(response);
        c.end();
      })
      .catch( console.error ); // should never happen
  });

}


//
// Example VaultDataServer.  All vaults are held in RAM!
//
```

(continues on next page)

(continued from previous page)

```
class RamBasedVaultDataServer extends datona.vault.VaultDataServer {

  constructor() {
    super();
    this.vaults = {};
  }

  create(contract) {
    if (this.vaults[contract] != undefined) {
      throw new datona.errors.VaultError("attempt to create a vault that already␣
↪exists: " + contract);
    }
  }

  write(contract, file, data) {
    if (this.vaults[contract] == undefined) {
      throw new datona.errors.VaultError("attempt to write to a vault that does not␣
↪exist: " + contract);
    }
    this.vaults[contract][file] = data;
  };

  append(contract, file, data) {
    if (this.vaults[contract] == undefined) {
      throw new datona.errors.VaultError("attempt to append to a vault that does not␣
↪exist: " + contract);
    }
    if (this.vaults[contract][file] === undefined) { this.vaults[contract][file] =␣
↪data; }
    else this.vaults[contract][file] += data;
  };

  read(contract, file) {
    if (this.vaults[contract] == undefined) {
      throw new datona.errors.VaultError("attempt to access a vault that does not␣
↪exist: " + contract);
    }
    if (this.vaults[contract][file] === undefined) {
      throw new datona.errors.VaultError("attempt to access a file that does not␣
↪exist: " + contract+"/"+file);
    }
    return this.vaults[contract];
  };

  readDir(contract, dir) {
    if (this.vaults[contract] === undefined) {
      throw new datona.errors.VaultError("attempt to access a vault that does not␣
↪exist: " + contract);
    }
    var contents = "";
    for (var file in this.vaults[contract]) {
      if (file.substring(0,43) === dir+"/") contents += (contents.length===0) ? file.
↪substring(43) : "\n"+file.substring(43);
    }
    return contents;
  };
```

(continues on next page)

(continued from previous page)

```
  delete(contract) {
    if (this.vaults[contract] == undefined) {
      throw new datona.errors.VaultError("attempt to delete a vault that does not␣
↪exist: " + contract);
    }
    this.vaults[contract] = undefined;
  };

}
```

2.3 datona-blockchain

Gives access to the Datona blockchain (Ethereum right now), providing functions to deploy, manage and access S-DACs. Is designed to be used by both owner-end software (identity apps) and vault software. Uses web3.

2.3.1 Constants

- `ZERO_ADDRESS` *(Address)* = `"0x0000000000000000000000000000000000000000"`

2.3.2 Class Permissions

Encapsulates file/directory permissions returned by the `getPermissions` method of an SDAC. Provides accessor functions to read properties of the permissions.

Properties

- `permissions` *(byte)* - the raw permissions provided to the constructor. Is expected to be of the form specified in the *SDAC Interface*.

Constructor

Decodes a raw permissions byte.

```
new Permissions(permissionsByte);
```

Parameters

1. `permissionsByte` *(byte or String)* - the raw permissions byte either as an integer or a string of the form `0xNN`.

Throws

- `TypeError` if the permissionsByte is a string and does not have the form `0xNN`

Example

```
// using string format
const permissions = new Permissions("0x87");

// using byte returned from contract
myContract.getPermissions(myAddress)
  .then( function(rawPermissions) {
      const permissions = new Permissions(rawPermissions);
      if (permissions.canRead()) { ...
      }
  });
```

---

canRead

Accesses the READ bit in the raw permissions.

```
if (permissions.canRead()) { ... }
```

Returns

`boolean` - true if the bit is set in the raw permissions.

---

canWrite

Accesses the WRITE bit in the raw permissions.

```
if (permissions.canWrite()) { ... }
```

Returns

`boolean` - true if the bit is set in the raw permissions.

---

canAppend

Accesses the APPEND bit in the raw permissions.

```
if (permissions.canAppend()) { ... }
```

Returns

`boolean` - true if the bit is set in the raw permissions.

---

isDirectory

Accesses the DIRECTORY bit in the raw permissions.

```
if (permissions.isDirectory()) { ... }
```

Returns

`boolean` - true if the bit is set in the raw permissions.

2.3.3 Class Contract

Represents a Smart Data Access Contract on the blockchain. Provides functions to interact with the contract.

Constants

Bit masks for the permissions byte returned by a Smart Data Access Contract:

- static `NO_PERMISSIONS` *(byte)* = `0x00`
- static `ALL_PERMISSIONS` *(byte)* = `0x07`;
- static `READ_BIT` *(byte)* = `0x04`;
- static `WRITE_BIT` *(byte)* = `0x02`;
- static `APPEND_BIT` *(byte)* = `0x01`;
- static `DIRECTORY_BIT` *(byte)* = `0x80`;

Reserved Addresses used by a Smart Data Access Contract:

- static `ROOT_DIRECTORY` *(Address)* = `"0x0000000000000000000000000000000000000000"`;

Properties

- `address` *(Address)* - the public blockchain address of the contract. Will be `undefined` unless given in the constructor, set using *setAddress*, or deployed using the *deploy* function.

Constructor

Creates a new Contract instance. Connects with the blockchain (if not connected already).

```
new Contract(abi, [address]);
```

Parameters

1. `abi` *(Object)* - The smart contract's abi
2. `address` *(Address)* - (Optional) The address of the contract on the blockchain, if already deployed. Exclude if constructing a new contract. Note, the address can be set later via *setAddress* if preferred.

Throws

- `BlockchainError` - if it can't connect with the blockchain or the abi is invalid.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi);
```

---

setAddress

Sets the address of this contract on the blockchain. Can be used as an alternative to passing it in the constructor.

```
setAddress(address);
```

Parameters

1. `address` *(Address)* - address of the contract on the blockchain

Throws

- `BlockchainError` - if the address is already set

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi);
contract.setAddress("0xfb3e6dd29d01c1b5b99e46db3fe26df1138b73d1");
```

---

deploy

Deploys this contract on the blockchain.

```
deploy(key, bytecode, [constructorArgs]);
```

Parameters

1. `key` *(Key)* - the Datona Key object used to sign the transaction
2. `bytecode` *(string)* - the contract creation bytecode (in hex with no leading 0x)
3. `constructorArgs` *(Array)* - (Optional) arguments to pass to the contract's constructor

Returns

`Promise` - A promise to deploy the contract on the blockchain, returning the contract address.

Resolves With

`Address` - The blockchain address of the deployed contract. Resolves after the transaction has been mined.

Rejects With

- `BlockchainError` - if deployment failed. If the blockchain VM reverted the transaction then examine the blockchain receipt in the error details.

Throws

- `BlockchainError` - if the bytecode is invalid

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi);

var contractAddress;

contract.deploy(myKey, myContract.bytecode, [1, requesterAddress])
  .then( function(address){
    contractAddress = address;
    const vault = new datona.vault.RemoteVault( vaultUrl, contractAddress, myKey );
    return vault.write("Hello World");
  })
  .catch( function(error){
    console.error(error);
  });
```

---

getOwner

Gets the owner of the contract

```
getOwner();
```

Returns

`Promise` - A promise to return owner's address

Resolves With

`Address` - The owner's address

Rejects With

- `BlockchainError` - if the contract owner could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.getOwner()
  .then(console.log)
  .catch(console.error);
```

---

hasExpired

Resolves true if the smart data access contract has expired.

```
hasExpired();
```

Returns

`Promise` - A promise to return the expiry status

Resolves With

`boolean` - True if the contract has expired. False otherwise.

Rejects With

- `BlockchainError` - if the expiry status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.hasExpired()
  .then( function(expired){
    if (expired) {
      console.log("contract has expired");
    }
  })
  .catch(console.error);
```

---

getPermissions

Promises to call the contract's getPermissions method and return the permissions byte as a *Permissions* object.

```
getPermissions(requester, [file]);
```

Parameters

1. `requester` *(Address)* - the address of the requester that wants to read the data
2. `file` *(Address)* - (Optional) the specific file to check. Defaults to the `ROOT_DIRECTORY` if not given.

Returns

`Promise` - A promise to return the permissions byte encapsulated in a Permissions object

Resolves With

`Permissions` - the *Permissions* object representing the permissions byte returned by the SDAC.

Rejects With

- `BlockchainError` - if the permission status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);
const fileId = "0x0000000000000000000000000000000000000001";
const vaultUrl = "file://datonavault.com:8124";
const vaultOwner = "0x288b32F2653C1d72043d240A7F938a114Ab69584";

contract.getPermissions(myKey.address, fileId)
  .then( function(permissions){
    if (permissions.canRead() && !permissions.isDirectory()) {
      const vault = new datona.vault.RemoteVault( vaultUrl, contract.address, myKey,␣
↪vaultOwner );
      return vault.read(fileId);
    }
  })
  .catch(console.error);
```

canRead

Resolves true if the owner of the given address is permitted to read the data from a given file in the vault controlled by this contract.

```
canRead(requester, [file]);
```

Parameters

1. `requester` *(Address)* - the address of the requester that wants to read the data
2. `file` *(Address)* - (Optional) the specific file to check. Defaults to the `ROOT_DIRECTORY` if not given.

Returns

`Promise` - A promise to return the permission status

Resolves With

`boolean` - True if the address is permitted to read the file. False otherwise.

Rejects With

- `BlockchainError` - if the permission status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);
const fileId = "0x0000000000000000000000000000000000000001";
const vaultUrl = "file://datonavault.com:8124";
const vaultOwner = "0x288b32F2653C1d72043d240A7F938a114Ab69584";

contract.canRead(myKey.address, fileId)
  .then( function(permitted){
    if (permitted) {
      const vault = new datona.vault.RemoteVault( vaultUrl, contract.address, myKey,␣
↪vaultOwner );
      return vault.read(fileId);
    }
  })
  .catch(console.error);
```

---

canWrite

Resolves true if the owner of the given address is permitted to write to (or overwrite) a given file or directory in the vault controlled by this contract.

If permitted to write to a directory, the user can add a new file to the directory or can overwrite any file within that directory.

```
canWrite(requester, [file]);
```

Parameters

1. `requester` *(Address)* - the address of the requester that wants to write to the file
2. `file` *(Address)* - (Optional) the specific file or directory to check. Defaults to the `ROOT_DIRECTORY` if not given.

Returns

`Promise` - A promise to return the permission status

Resolves With

`boolean` - True if the address is permitted to write to the file. False otherwise.

Rejects With

- `BlockchainError` - if the permission status could not be retrieved from the blockchain.

Throws

- BlockchainError - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);
const fileId = "0x0000000000000000000000000000000000000001";
const vaultUrl = "file://datonavault.com:8124";
const vaultOwner = "0x288b32F2653C1d72043d240A7F938a114Ab69584";

contract.canWrite(myKey.address, fileId)
  .then( function(permitted){
    if (permitted) {
      const vault = new datona.vault.RemoteVault( vaultUrl, contract.address, myKey,␣
↪vaultOwner );
      return vault.write("hello world", fileId);
    }
  })
  .catch(console.error);
```

canAppend

Resolves true if the owner of the given address is permitted to append to (or overwrite) a given file or directory in the vault controlled by this contract.

If permitted to append to a directory, the user can add a new file to the directory or can append to any file within that directory. It does not mean that existing files in that directory are (over)writable - use canWrite to determine this.

```
canAppend(requester, [file]);
```

Parameters

1. requester *(Address)* - the address of the requester that wants to append to the file
2. file *(Address)* - (Optional) the specific file to check. Defaults to the ROOT_DIRECTORY if not given.

Returns

Promise - A promise to return the permission status

Resolves With

boolean - True if the address is permitted to append to the file. False otherwise.

Rejects With

- `BlockchainError` - if the permission status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);
const fileId = "0x0000000000000000000000000000000000000001";
const vaultUrl = "file://datonavault.com:8124";
const vaultOwner = "0x288b32F2653C1d72043d240A7F938a114Ab69584";

contract.canAppend(myKey.address, fileId)
  .then( function(permitted){
    if (permitted) {
      const vault = new datona.vault.RemoteVault( vaultUrl, contract.address, myKey,␣
↪vaultOwner );
      return vault.append("some more info", fileId);
    }
  })
  .catch(console.error);
```

---

getBytecode

Gets the runtime bytecode of this contract from the blockchain

```
getBytecode();
```

Returns

`Promise` - A promise to return the bytecode

Resolves With

`String` - The runtime bytecode (in hex)

Rejects With

- `BlockchainError` - if the bytecode could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.getBytecode()
  .then(console.log)
  .catch(console.error);
> 60806040526004361061009e576000357c0100000000000...
```

---

call

Calls the given view or pure contract method with the given arguments. Use *transact* to call a state-modifying method instead.

```
call(method, [args);
```

Parameters

1. `method` *(String)* - the name of the contract method to call
2. `args` *(Array)* - (Optional) arguments to pass to the method

Returns

`Promise` - A promise to return the output from the method.

Resolves With

The datatype that the contract method returns, e.g. `string`, `boolean`, `integer`.

Rejects With

- `BlockchainError` - if the call failed. Examine the error details for more information.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address, the method does not exist or the method arguments are invalid.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi);

contract.call("isPermitted", [myKey.address])
  .then( function(permitted){
      console.log("isPermitted returned "+permitted);
  })
  .catch(console.error);
> isPermitted returned true
```

---

transact

Calls the given state-modifying contract method with the given arguments. Use *call* to call a view or pure method instead.

```
call(key, method, [args], [options]);
```

Parameters

1. `key` *(Key)* - the key used to sign the transaction
2. `method` *(String)* - the name of the contract method to call
3. `args` *(Array)* - (Optional) arguments to pass to the method
4. `options` *(Object)* - (Optional) any fields in this object will be included in the blockchain transaction

Returns

`Promise` - A promise to return the output from the method.

Resolves With

The datatype that the contract method returns, e.g. `string`, `boolean`, `integer`. Resolves after the transaction has been mined.

Rejects With

- `BlockchainError` - if the call failed. Examine the error details for more information.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address, the method does not exist or the method arguments are invalid.

Example

```
// In this example the user has created a smart contract with an additional
↪'pay(address payee)' method

const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi);

contract.transact(myKey, "pay", [theirAddress], {value: 100})
  .then( function(){
      console.log("payment successful");
  })
  .catch(console.error);
```

terminate

Terminates this contract by calling it's `terminate` method.

```
terminate(key);
```

Parameters

1. `key` *(Key)* - the key used to sign the transaction

Returns

`Promise` - A promise to attempt to terminate the contract

Resolves With

Resolves with no data if successful. Resolves after the transaction has been mined.

Rejects With

- `BlockchainError` - if the contract could not be terminated.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.terminate()
  .then( function(){
    console.log("contract terminated");
  })
  .catch(console.error);
```

---

assertBytecode

Asserts that the contract's runtime bytecode equals the expected bytecode given.

```
assertBytecode(expectedBytecode);
```

Parameters

1. `expectedBytecode` *(String)* - the bytecode to test

Returns

`Promise` - A promise to resolve if the bytecodes match, and to reject if not.

Resolves With

Resolves with no data if the contract's bytecode matches the bytecode given.

Rejects With

- `ContractTypeError` - if the bytecodes do not match
- `BlockchainError` - if the bytecode could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.assertBytecode(myContract.runtimeBytecode)
  .then( function(){
    console.log("contract bytecode is as expected");
```

(continues on next page)

(continued from previous page)

```
  })
  .catch(console.error);
```

---

assertOwner

Asserts that the contract's runtime bytecode equals the expected bytecode given.

```
assertOwner(expectedOwner);
```

Parameters

1. `expectedOwner` *(Address)* - the owner address to test

Returns

`Promise` - A promise to resolve if the addresses match, and to reject if not.

Resolves With

Resolves with no data if the contract's owner matches the address given.

Rejects With

- `ContractOwnerError` - if the owner does not match
- `BlockchainError` - if the owner could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.assertOwner(myKey.address)
  .then( function(){
    console.log("I am the owner of contract "+contract.address);
  })
  .catch(console.error);
```

---

assertNotExpired

Resolves provided the contract has not expired.

```
assertNotExpired();
```

Returns

`Promise` - A promise to resolve if the contract has not expired, and to reject if not.

Resolves With

Resolves with no data if the contract has not expired.

Rejects With

- `ContractExpiryError` - if the contract has expired
- `BlockchainError` - if the expiry status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.assertOwner(myKey.address)
  .then( contract.assertNotExpired )
  .then( updateMyData )
  .catch(console.error);
```

---

assertHasExpired

Resolves provided the contract has expired.

```
assertNotExpired();
```

Returns

`Promise` - A promise to resolve if the contract has expired, and to reject if not.

Resolves With

Resolves with no data if the contract has expired.

Rejects With

- `ContractExpiryError` - if the contract has not expired
- `BlockchainError` - if the expiry status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const myContract = require("../contracts/myContract.json");
const contract = new Contract(myContract.abi, myContractAddress);

contract.terminate(myKey)
  .then( contract.assertHasExpired )
  .then( function(){
    console.log("Double checked. Contract has been terminated.");
  })
  .catch(console.error);
```

---

assertCanRead

Resolves provided the given address is permitted to read the given file or directory in the vault controlled by this contract.

```
assertCanRead(requester, [file]);
```

Parameters

1. `requester` *(Address)* - the address of the requester that wants to read the data
2. `file` *(Address)* - (Optional) the specific file to check. Defaults to the `ROOT_DIRECTORY` if not given.

Returns

`Promise` - A promise to resolve if the given address is permitted to read the given file, and to reject if not.

Resolves With

`Permissions` - the *Permissions* object representing the permissions byte returned by the SDAC. Only resolves if the requester is permitted.

Rejects With

- `PermissionError` - if permission is not granted
- `BlockchainError` - if the expiry status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const expectedContract = require("../contracts/myContract.json");
const contract = new Contract(expectedContract.abi, customer.contractAddress);
const fileId = "0x0000000000000000000000000000000000000001";

contract.assertBytecode(expectedContract.runtimeBytecode)
  .then( () => { return contract.assertOwner(customer.address) })
  .then( () => { return contract.assertCanRead(myKey.address, fileId) })
  .then( function(){
    console.log("Confirmed customer's contract is valid");
  })
  .catch(console.error);
```

---

assertCanWrite

Resolves provided the given address is permitted to write to the given file or directory in the vault controlled by this contract.

```
assertCanWrite(requester, [file]);
```

Parameters

1. `requester` *(Address)* - the address of the requester that wants to write the data
2. `file` *(Address)* - (Optional) the specific file to check. Defaults to the `ROOT_DIRECTORY` if not given.

Returns

`Promise` - A promise to resolve if the given address is permitted to write to the given file, and to reject if not.

Resolves With

`Permissions` - the *Permissions* object representing the permissions byte returned by the SDAC. Only resolves if the requester is permitted.

Rejects With

- `PermissionError` - if permission is not granted
- `BlockchainError` - if the expiry status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const expectedContract = require("../contracts/myContract.json");
const contract = new Contract(expectedContract.abi, customer.contractAddress);
const ownersFile = "0x0000000000000000000000000000000000000001";
const resultsFile = "0x0000000000000000000000000000000000000002";

contract.assertBytecode(expectedContract.runtimeBytecode)
  .then( () => { return contract.assertOwner(customer.address) })
  .then( () => { return contract.assertCanRead(myKey.address, ownersFile) })
  .then( () => { return contract.assertCanWrite(myKey.address, resultsFile) })
  .then( function(){
    console.log("Confirmed customer's contract is valid");
  })
  .catch(console.error);
```

---

assertCanAppend

Resolves provided the given address is permitted to append to the given file or directory in the vault controlled by this contract.

```
assertCanAppend(requester, [file]);
```

Parameters

1. `requester` *(Address)* - the address of the requester that wants to append the data
2. `file` *(Address)* - (Optional) the specific file to check. Defaults to the `ROOT_DIRECTORY` if not given.

Returns

`Promise` - A promise to resolve if the given address is permitted to append to the given file, and to reject if not.

Resolves With

`Permissions` - the *Permissions* object representing the permissions byte returned by the SDAC. Only resolves if the requester is permitted.

Rejects With

- `PermissionError` - if permission is not granted
- `BlockchainError` - if the expiry status could not be retrieved from the blockchain.

Throws

- `BlockchainError` - if the contract hasn't been deployed or mapped to a blockchain address.

Example

```
const expectedContract = require("../contracts/myContract.json");
const contract = new Contract(expectedContract.abi, customer.contractAddress);
const ownersFile = "0x0000000000000000000000000000000000000001";
const logFile = "0x0000000000000000000000000000000000000002";

contract.assertBytecode(expectedContract.runtimeBytecode)
  .then( () => { return contract.assertOwner(customer.address) })
  .then( () => { return contract.assertCanRead(myKey.address, ownersFile) })
  .then( () => { return contract.assertCanAppend(myKey.address, logFile) })
  .then( function(){
    console.log("Confirmed customer's contract is valid");
  })
  .catch(console.error);
```

2.3.4 Class GenericSmartDataAccessContract

Instance of *Contract* providing an interface to any Smart Data Access Contract. Maps to a contract at the given address using the standard *SDAC Interface* ABI.

Constructor

Creates a new Contract instance. Connects with the blockchain (if not connected already).

```
new GenericSmartDataAccessContract(address);
```

Parameters

1. `address` *(Address)* - The address of the contract on the blockchain.

Throws

- `BlockchainError` - if it can't connect with the blockchain.

Example

```
const contract = new GenericSmartDataAccessContract(customer.contractAddress);
```

2.3.5 Functions

setProvider

Overrides the default gateway service to the blockchain (that configured in config.json). Supported URL schemes (protocols) are ws, wss, http and https.

```
setProvider(url);
```

Parameters

1. `url` *(URL)* - url of the blockchain provider

Throws

- `BlockchainError` - if the url is invalid or the scheme (protocol) is not supported.

Example

```
const myGateway = { scheme: "https", host: "kovan.infura.com:, port: "" };

datona.blockchain.setProvider(myGateway);
```

---

sendTransaction

Promises to publish a transaction on the blockchain. There are three types of transactions: balance transfers between accounts; contract deployment and contract function calls. The `Contract` class above should be used for contract deployment and calls.

```
sendTransaction(key, transaction);
```

Parameters

1. `key` *(Key)* - the Datona Key object used to sign the transaction
2. `transaction` *(Object)* - Object containing the transaction

The Transaction parameter has the following structure:

- `from` *(String)* - (optional) source address. Will be auto populated from the `key` parameter if not given.

- `to` *(String)* - destination address (account or contract address). Leave undefined if this is a deployment transaction.
- `value` *(Number|String|BN|BigNumber)* - eth to transfer to the destination in wei, if any. Can be omitted if this is a contract deployment or call.
- `data` *(String)* - hex string data (with `0x` prefix) to include in the transaction. If a contract deployment, this is the contract bytecode. If a contract call this is the call information.
- `gasPrice` *(Number)* - (optional) the price you are offering to pay per unit of gas in wei. If not given, the gas price will be auto populated using the median gas price of the last few blocks.
- `gas` *(Number|String|BN|BigNumber)* - (optional) the maximum amount of gas allowed for the transaction (gas limit). Warning - A high gas limit is used if the `gas` parameter is not given.
- `nonce` *(Number)* - (optional) must be the correct value for the sending address. The EVM expects this nonce to increment each time a transaction is successfully published from the sending address. The nonce will be automatically calculated so it should be left undefined unless you know what you are doing.

Returns

`Promise` - A promise to publish the transaction.

Resolves With

`receipt` - An object containing the EVM receipt. Will only resolve if the receipt's status is `0x01` - will reject if not. See here for information on the receipt structure.

Rejects With

- `BlockchainError` - if the transaction is invalid, could not be published or was rejected by the EVM.

Example

```
const myTransaction = {
  to: "0xc16a409a39EDe3F38E212900f8d3afe6aa6A8929",
  value: 1000,
  data: "0x01020304
}

datona.blockchain.sendTransaction(myKey, myTransaction)
  .then(console.log)
  .catch(console.error);
```

---

subscribe

Subscribes the client to receive notification of a new contract deployed to the blockchain with the given code hash. Optionally, the client can receive notification only if the given address is permitted to access the data controlled by the contract.

```
subscribe(bytecodeHash, callback, [permittedAddress]);
```

Parameters

1. `bytecodeHash` *(Hash)* - hash of the runtime bytecode of the new contract to monitor for
2. `callback` *(function)* - function to call if new contract is found: `function callback(contractAddress, bytecodeHash)`
3. `permittedAddress` *(String)* - (Optional) address to check if permitted

callback parameters

1. `contractAddress` *(Address)* - blockchain address of the new contract
2. `bytecodeHash` *(Hash)* - (Optional) hash of the runtime bytecode of the new contract. Allows the same callback to be used for multiple subscriptions.

Returns

`Hash` - unique subscription id (can be used to *unsubscribe* later).

Throws

- `BlockchainError` - if web3 cannot be subscribed to

Example

```
const myContract = require("../contracts/myContract.json");
const subscription = subscribe(datona.crypto.hash(myContract.runtimeBytecode),␣
↪registerNewCustomer, myKey.address);

function registerNewCustomer(contractAddress) {
  const contract = new Contract(myContract.abi, contractAddress);
  contract.getOwner()
    .then( function(ownerAddress){
      const newCustomer = { owner: ownerAddress, contract: contractAddress };
      customers.push(newCustomer);
    })
    .catch( function(error){
      console.error("Couldn't get owner of new customer contract.  Try again later.
↪"+contractAddress+" - "+error.message);
      const newCustomer = { owner: undefined, contract: contractAddress };
      customers.push(newCustomer);
    });
}
```

unsubscribe

Unsubscribes a previous subscription. The subscription is identified by the subscription id returned from the original call to *subscribe*.

```
unsubscribe(subscriptionId);
```

Parameters

1. `subscriptionId` *(Hash)* - the subscription to unsubscribe

Returns

`uint` - the number of subscriptions unsubscribed.

Example

```
const myContract = require("../contracts/myContract.json");
const subscription = subscribe(datona.crypto.hash(myContract.runtimeBytecode),␣
↪registerNewCustomer, myKey.address);

...

if( unsubscribe(subscription) == 0 ) console.error("failed to unsubscribe");
```

---

close

Closes the connection to the blockchain. Should be called on program exit if any blockchain functions have been used.

```
close();
```

Throws

- `BlockchainError` - if the connection cannot be closed.

---

getGasPrice

Returns the median gas price of the last few blocks.

```
getGasPrice();
```

Returns

`Promise` - A promise to return the gas price

Resolves With

`String` - String representation of the gas price in wei

Example

```
getGasPrice().then(console.log);

> "20000000000"
```

2.4 datona-vault

2.4.1 Class VaultFilename

A parsed vault filename with extracted file and directory parts and other properties. See the *VaultFile* type.

Properties

- `fullFilename` *(String)* - the original filename passed to the constructor
- `file` *(String)* - the file part of the filename.
- `directory` *(String)* - the directory part of the filename (an Address).
- `hasDirectory` *(boolean)* - true if the filename has a directory part.
- `isValid` *(boolean)* - true if the filename is valid in accordance with the *protocol*.

Constructor

Parses and validates a filename string, setting the properties above.

```
new VaultFilename(filename);
```

Parameters

1. `filename` *(String)* - name of the vault file to parse

Example

```
const vaultFile = new VaultFilename("0x0000000000000000000000000000000000000002/my_
↪file.txt");
if (!vaultFile.isValid) throw new datona.errors.TypeError("invalid filename");
```

2.4.2 Class RemoteVault

Represents a single vault within a vault server controlled by a single S-DAC. Is designed to be used by data owners to create, update and delete a vault, and by requesters to access a vault. Extends *Class DatonaConnector*.

Properties

- `url` *(URL)* - the URL given to the constructor
- `remoteAddress` *(Address)* - the public blockchain address of the remote data vault server given to the constructor.

Constructor

Creates a new RemoteVault instance with a network client suitable for the given url scheme.

```
new RemoteVault(url, contractAddress, localPrivateKey, remoteAddress);
```

Parameters

1. `url` *(URL)* - the URL object identifying the server, port and URI scheme of the remote data vault server
2. `contractAddress` *(Address)* - The address of the contract that controls this vault
3. `localPrivateKey` *(Key)* - The Key object used to sign any transactions
4. `remoteAddress` - the public blockchain address of the remote data vault server. Used for verifying received responses.

Throws

- `VaultError` - if the url scheme is unsupported

Example

```
const url = { scheme: "file", host: "datonavault.com", port: "8643" };
const myContractAddress = "0x008Cd346b65F5aFa306Ef9160a84455D308e6851";
const remoteAddress = "0x41A60F71063CD7c9e5247d3E7d551f91f94b5C3b";
const remoteVault = new RemoteVault(url, myContractAddress, myKey, remoteAddress);
```

---

create

Promises to create a new vault on the remote data vault server containing the given data. This method creates the data request, signs it, initiates the vault request and handles the vault response.

```
create([options]);
```

Parameters

1. `options` *(Object)* - (Optional) this object will be passed unchanged to the remote data vault server.

Returns

`Promise` - A promise to create this vault and resolve if successful. Promises to reject if the vault was not created for any reason.

Resolves With

`{ txn:  VaultResponse, signatory:  Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the `remoteAddress` given in the constructor. See *VaultResponse*. If the response is an error type then the promise will reject instead.

Rejects With

- `ContractOwnerError` - if you are not the vault owner (the contract owner)
- `ContractExpiryError` - if the contract has expired
- `VaultError` - if the vault server failed to create the vault for any reason.
- `CommunicationError` - if communication with the vault server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the vault server's remote.
- `MalformedRequestError` - if the request form is invalid or fields are missing or invalid
- `InvalidSignatureError` - if the signatory cannot be recovered from the signature

Example

```
const remoteVault = new RemoteVault(url, myContractAddress, myKey, remoteAddress);

remoteVault.create()
  .then( () => { console.log("vault created successfully") })
  .catch( console.error );
```

---

write

Promises to write data to the vault, to a specific file if specified. This method creates the data request, signs it, initiates the vault request and handles the vault response.

```
write(data, [file], [options]);
```

Parameters

1. `data` *(Object)* - the data to be stored
2. `file` *(Address)* - (Optional) the specific file to write to. Defaults to the *ROOT_DIRECTORY* if not given.
3. `options` *(Object)* - (Optional) this object will be passed unchanged to the remote data vault server.

Returns

`Promise` - A promise to write the data to the given file in this vault and resolve if successful. Promises to reject if the vault was not updated for any reason.

Resolves With

`{ txn:  VaultResponse, signatory:  Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the `remoteAddress` given in the constructor. See *VaultResponse*. If the response is an error type then the promise will reject instead.

Rejects With

- `ContractOwnerError` - if you are not the vault owner (the contract owner)
- `ContractExpiryError` - if the contract has expired
- `VaultError` - if the vault server failed to update the vault for any reason.
- `CommunicationError` - if communication with the vault server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the vault server's remote.
- `MalformedRequestError` - if the request form is invalid or fields are missing or invalid
- `InvalidSignatureError` - if the signatory cannot be recovered from the signature

Example

```
const remoteVault = new RemoteVault(url, myContractAddress, myKey, remoteAddress);

remoteVault.write("Hello World", "0xF000000000000000000000000000000000000002")
  .then( () => { console.log("vault updated successfully") })
  .catch( console.error );
```

---

append

Promises to append data to the vault, to a specific file or directory if specified. This method creates the data request, signs it, initiates the vault request and handles the vault response.

When appending data to a directory, the data is written to a new file in that directory. The `file` parameter must contain a unique file name, e.g. "0x0000000000000000000000000000000000000001/myfile1.txt"

```
append(data, [file], [options]);
```

Parameters

1. `data` *(Object)* - the data to be appended
2. `file` *(Address)* - (Optional) the specific file to write to. Defaults to the *ROOT_DIRECTORY* if not given.
3. `options` *(Object)* - (Optional) this object will be passed unchanged to the remote data vault server.

Returns

`Promise` - A promise to write the data to the given file in this vault and resolve if successful. Promises to reject if the vault was not updated for any reason.

Resolves With

`{ txn: VaultResponse, signatory: Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the `remoteAddress` given in the constructor. See *VaultResponse*. If the response is an error type then the promise will reject instead.

Rejects With

- `ContractOwnerError` - if you are not the vault owner (the contract owner)
- `ContractExpiryError` - if the contract has expired
- `VaultError` - if the vault server failed to update the vault for any reason.
- `CommunicationError` - if communication with the vault server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the vault server's remote.
- `MalformedRequestError` - if the request form is invalid or fields are missing or invalid
- `InvalidSignatureError` - if the signatory cannot be recovered from the signature

Example

```
const remoteVault = new RemoteVault(url, myContractAddress, myKey, remoteAddress);

remoteVault.append("some additional info", "0xF0000000000000000000000000000000000002
↪")
  .then( () => { console.log("vault appended successfully") })
  .catch( console.error );
```

read

Promises to retrieve the data from this vault if permitted. This method creates the data request, signs it, initiates the vault request and handles the vault response.

```
read([file], [options]);
```

Parameters

1. `file` *(Address)* - (Optional) the specific file or directory to read from. Defaults to the *ROOT_DIRECTORY* if not given.
2. `options` *(Object)* - (Optional) this object will be passed unchanged to the remote data vault server.

Returns

`Promise` - A promise to retrieve the data and resolve if successful. Promises to reject if the vault could not be accessed for any reason.

Resolves With

`Object` - the data returned from the vault in whatever format it was written.

Rejects With

- `PermissionError` - if you are not permitted to access the vault
- `ContractExpiryError` - if the contract has expired
- `VaultError` - if the vault server could not handle the request for any reason.
- `CommunicationError` - if communication with the vault server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the vault server's remote.
- `MalformedRequestError` - if the request form is invalid or fields are missing or invalid
- `InvalidSignatureError` - if the signatory cannot be recovered from the signature

Example

```
const remoteVault = new RemoteVault(url, myContractAddress, myKey, remoteAddress);

remoteVault.read("0xF00000000000000000000000000000000000000002")
  .then( (data) => { console.log("vault contains: "+data) )
  .catch( console.error );
```

delete

Promises to delete this vault and its data provided the contract has expired or has been terminated. This method creates the data request, signs it, initiates the vault request and handles the vault response.

```
delete([options]);
```

Parameters

1. `options` *(Object)* - (Optional) this object will be passed unchanged to the remote data vault server.

Returns

`Promise` - A promise to delete the vault and resolve if successful. Promises to reject if the vault could not be deleted for any reason.

Resolves With

`{ txn: VaultResponse, signatory: Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the `remoteAddress` given in the constructor. See *VaultResponse*. If the response is an error type then the promise will reject instead.

Rejects With

- `ContractOwnerError` - if you are not the vault owner (the contract owner)
- `ContractExpiryError` - if the contract has not expired
- `VaultError` - if the vault server could not handle the request for any reason.
- `CommunicationError` - if communication with the vault server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the vault server's remote.
- `MalformedRequestError` - if the request form is invalid or fields are missing or invalid
- `InvalidSignatureError` - if the signatory cannot be recovered from the signature

Example

```
const remoteVault = new RemoteServer(url, myContractAddress, myKey, remoteAddress);

remoteVault.delete()
  .then( () => { console.log("vault deleted") })
  .catch( console.error );
```

2.4.3 Class VaultKeeper

Guardian of a Vault Data Server. Designed to be used by developers of data vault servers, whether cloud based or locally hosted.

All create, update, access and delete requests go through the Vault Keeper, where they are approved or rejected against the Datona Smart Data Access Protocol. If approved and permission granted by the vault's Smart Data Access Contract, the VaultKeeper passes the raw request to the *VaultDataServer* object given to the constructor.

Properties

- `vaultDataServer` *(VaultDataServer)* - the *VaultDataServer* instance given to the constructor

Constructor

Creates a new VaultKeeper instance

```
new VaultKeeper(vaultDataServer, key);
```

Parameters

1. `vaultDataServer` *(VaultDataServer)* - the *VaultDataServer* instance that provides the data server service.
2. `key` *(Key)* - The vault server's private key as a Key object. Used to sign any transactions. The signature is used by the remote client to authenticate the vault server and so this key must correspond to the vault server's public identity.

Example

```
DataServer = require('MyDataServer.js');
const vaultManager = new DataServer();
const vaultKeeper = new VaultKeeper(vaultManager, myKey);
```

---

handleSignedRequest

Primary method to process a signed VaultRequest from a client. Decodes and processes the request, checks the validity of the signature, validates the request and passes the raw data request to the *VaultDataServer* instance given to the constructor.

```
handleSignedRequest(signedRequestStr);
```

Parameters

1. `signedRequestStr` *(SignedTransaction)* - the data to be stored

Returns

`Promise` - A promise to resolve with a signed success or error *VaultResponse*.

Resolves With

`SignedTransaction` - containing the VaultResponse and transaction signature, ready to send back to the client.

Rejects With

Does not reject. Any error is converted to signed error VaultResponse and resolved.

Example

```
const myDataVaultServer = net.createServer(connection);

connection(c){

  c.on('data', (buffer) => {
    const data = buffer.toString();
    vaultKeeper.handleSignedRequest(data)
      .then( function(response){
        c.write(response);
        c.end();
      })
      .catch( console.error ); // should never happen
  });

}
```

---

createVault

Can be used if *handleSignedRequest* is not appropriate. Handles a valid create request. This method checks the validity of the signature and validates the request before creating a new vault via the VaultDataServer.

```
createVault(request, signatory);
```

Parameters

1. `request` *(VaultRequest)* - VaultRequest of type 'create' containing the contract address and data to put in the vault
2. `signatory` *(Address)* - signatory the address that signed the request. Must be the owner of the contract.

Returns

`Promise` - A promise to create the vault and resolve a success or error response.

Resolves With

`SignedTransaction` - containing the VaultResponse and transaction signature, ready to send back to the client.

Rejects With

Does not reject. Any error is converted to signed error VaultResponse and resolved.

An error response will be resolved if:

(a) the request is not a valid "create" request

(b) the signature is invalid;

(c) the signatory is not the owner of the contract

(d) the contract has expired

(e) the VaultDataServer returns an error

Example

```
const {txn, signatory} = comms.decodeTransaction(signedRequestStr);
if (txn.requestType == "create") {
  vaultKeeper.createVault(txn, signatory)
    .then( myServer.sendResponse )
    .catch( console.error );  // should never happen
}
```

---

writeVault

Can be used if *handleSignedRequest* is not appropriate. Handles a valid write request. This method checks the validity of the signature and validates the request before updating the vault via the VaultDataServer.

```
writeVault(request, signatory);
```

Parameters

1. `request` *(VaultRequest)* - VaultRequest of type 'write' containing the contract address, file to write and data to put in the vault
2. `signatory` *(Address)* - signatory the address that signed the request. Must be the owner of the contract.

Returns

`Promise` - A promise to write to the vault and resolve a success or error response.

Resolves With

`SignedTransaction` - containing the VaultResponse and transaction signature, ready to send back to the client.

Rejects With

Does not reject. Any error is converted to signed error VaultResponse and resolved.

An error response will be resolved if:

(a) the request is not a valid "create" request

(b) the signature is invalid;

(c) the signatory is not the owner of the contract

(d) the contract has expired

(e) the VaultDataServer returns an error

Example

```
const {txn, signatory} = comms.decodeTransaction(signedRequestStr);
if (txn.requestType == "write") {
  vaultKeeper.writeVault(txn, signatory)
    .then( myServer.sendResponse )
    .catch( console.error );  // should never happen
}
```

---

appendVault

Can be used if *handleSignedRequest* is not appropriate. Handles a valid append request. This method checks the validity of the signature and validates the request before updating the vault via the VaultDataServer.

```
appendVault(request, signatory);
```

Parameters

1. `request` *(VaultRequest)* - VaultRequest of type 'append' containing the contract address, file to append and data to put in the vault
2. `signatory` *(Address)* - signatory the address that signed the request. Must be the owner of the contract.

Returns

`Promise` - A promise to append to the vault and resolve a success or error response.

Resolves With

`SignedTransaction` - containing the VaultResponse and transaction signature, ready to send back to the client.

Rejects With

Does not reject. Any error is converted to signed error VaultResponse and resolved.

An error response will be resolved if:

(a) the request is not a valid "append" request

(b) the signature is invalid;

(c) the signatory is not the owner of the contract

(d) the contract has expired

(e) the VaultDataServer returns an error

Example

```
const {txn, signatory} = comms.decodeTransaction(signedRequestStr);
if (txn.requestType == "append") {
  vaultKeeper.appendVault(txn, signatory)
    .then( myServer.sendResponse )
    .catch( console.error );  // should never happen
}
```

---

readVault

Can be used if *handleSignedRequest* is not appropriate. Handles a valid read request. This method checks the validity of the signature and validates the request before accessing the vault via the VaultDataServer.

```
readVault(request, signatory);
```

Parameters

1. `request` *(VaultRequest)* - VaultRequest of type 'read' containing the contract address and file to read
2. `signatory` *(Address)* - signatory the address that signed the request. Must be permitted to access the vault.

Returns

`Promise` - A promise to access the vault and resolve a success or error response.

Resolves With

`SignedTransaction` - containing the VaultResponse and transaction signature, ready to send back to the client. A successful VaultResponse will contain the data from the vault.

Rejects With

Does not reject. Any error is converted to a signed error VaultResponse and resolved.

An error response will be resolved if:

(a) the request is not a valid "access" request

(b) the signature is invalid;

(c) the signatory is not permitted to access the vault (contract's isPermitted function returns false)

(d) the contract has expired

(e) the VaultDataServer returns an error

Example

```
const {txn, signatory} = comms.decodeTransaction(signedRequestStr);
if (txn.requestType == "read") {
  vaultKeeper.readVault(txn, signatory)
    .then( myServer.sendResponse )
    .catch( console.error );  // should never happen
}
```

---

deleteVault

Can be used if *handleSignedRequest* is not appropriate. Handles a valid delete request. This method checks the validity of the signature and validates the request before deleting the vault via the VaultDataServer. The contract must have expired (contract's hasExpired function returns true) before a vault can be deleted.

```
deleteVault(request, signatory);
```

Parameters

1. `request` *(VaultRequest)* - VaultRequest of type 'delete' containing the contract address and data to put in the vault

2. `signatory` *(Address)* - signatory the address that signed the request. Must be the owner of the contract.

Returns

`Promise` - A promise to delete the vault and resolve a success or error response.

Resolves With

`SignedTransaction` - containing the VaultResponse and transaction signature, ready to send back to the client.

Rejects With

Does not reject. Any error is converted to signed error VaultResponse and resolved.

An error response will be resolved if:

(a) the request is not a valid "delete" request

(b) the signature is invalid;

(c) the signatory is not the owner of the contract

(d) the contract has not expired

(e) the VaultDataServer returns an error

Example

```
const {txn, signatory} = comms.decodeTransaction(signedRequestStr);
if (txn.requestType == "create") {
  vaultKeeper.deleteVault(txn, signatory)
    .then( myServer.sendResponse )
    .catch( console.error );  // should never happen
}
```

2.4.4 Interface VaultDataServer

To use the Datona *VaultKeeper*, data vault developers must develop a class of this type that provides the data vault's data server capability. For example, a class could be developed to interface with an existing database, a remote file server or a local file system. If extending this interface, override the functions supported by your data server.

---

create

Must create a new vault identified by the given contract address. Must fail if the vault already exists.

```
create(contract, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault. Future write, append, read and delete requests will identify the vault using this contract address.
2. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Returns

`Promise` - A promise to create the vault. Must reject with a VaultError object if unsuccessful.

---

write

Must unconditionally write the given data to the given file in the vault identified by the given contract address, overwriting its contents if it already exists. Will fail if the vault does not exist.

```
write(contract, file, data, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault.
2. `file` *(Address)* - the specific file to write to.
3. `data` *(Object)* - the data to store in the vault
4. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Returns

`Promise` - A promise to write the data to the file. Must reject with a VaultError object if unsuccessful.

---

createFile

The same as `write` but only if the file does not already exist. Will fail if the vault does not exist or the file already exists.

```
createFile(contract, file, data, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault.
2. `file` *(Address)* - the specific file to write to.
3. `data` *(Object)* - the data to write to the file
4. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Returns

`Promise` - A promise to write the data to the file. Must reject with a VaultError object if the vault does not exist or the file already exists.

---

append

Must unconditionally append the given data to the given file in the vault identified by the given contract address, creating the file if it does not exist. Will fail if the vault does not exist.

```
append(contract, file, data, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault.
2. `file` *(Address)* - the specific file to write to.
3. `data` *(Object)* - the data to append to the file
4. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Returns

`Promise` - A promise to append the data to the file. Must reject with a VaultError object if unsuccessful.

---

read

Must unconditionally return the data from the given file in the vault identified by the given contract address. Will fail if the vault or file does not exist.

```
read(contract, file, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault.
2. `file` *(Address)* - the specific file to write to.
3. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Returns

`Promise` - A promise to resolve the vault contents in the same form given when the file was written. Must reject with a VaultError object if unsuccessful.

---

readDir

Must promise to unconditionally return a list of the names of files in the given directory within the vault identified by the given contract address. Will fail if the vault does not exist.

```
read(contract, file, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault.
2. `file` *(Address)* - the specific file to write to.
3. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Throws

`VaultError` - if the vault does not exist.

Returns

`Promise` - A promise to resolve the directory listing in the format `[<filename1>][\n<filename2>]...` Equivalent to `ls -c1` in linux. If the directory does not exist then then the empty string is resolved. Must reject with a VaultError object if unsuccessful.

---

delete

Must promise to unconditionally delete the vault identified by the given contract address, including all files within. Will fail if the vault does not exist.

```
deleteVault(contract, [options]);
```

Parameters

1. `contract` *(Address)* - the address of the contract to identify the vault.
2. `options` *(Object)* - (Optional) options from the end user. Allows the server developer to provide server-specific features to end user applications.

Returns

`Promise` - A promise to delete the vault and all data within it. Must reject with a VaultError object if unsuccessful.

2.5 datona-crypto

The datona-crypto component provides all the datona-lib cryptographic classes and functions.

2.5.1 Class Key

Encapsulates a private key and provides cryptographic functions that use it. The Key is a core class of datona-lib.

Properties

- `privateKey` *(PrivateKey)* - the private key given to the constructor
- `publicKey` *(PublicKey)* - the public key derived from the private key
- `address` *(Address)* - the public blockchain address derived from the private key

Constructor

Constructs the instance with the given private key.

```
new Key(privateKey);
```

Parameters

1. `privateKey` *(PrivateKey)* - 32-byte private key in hex (64 hex characters)

Example

```
const myKey = new Key(
↪"e68e40257cfee330038c49637fcffff82fae04b9c563f4ea071c20f2eb55063c");
console.log(myKey.address);
> 0x41A60F71063CD7c9e5247d3E7d551f91f94b5C3b
```

---

sign

Signs the given hash with this key.

```
sign(hash);
```

Parameters

1. `hash` *(Hash)* - 32-byte *hash* to sign in hex (64 hex characters)

Returns

`DatonaSignature` - the signature of the given hash derived from this key

Example

```
const signature = myKey.sign(hash("Hello World!"));
```

---

encrypt

Encrypts the given data using the Elliptic Curve Integrated Encryption Scheme. The symmetric encryption key is generated from this private key and the given public key. The resulting encrypted data can be decrypted with this public key and the private part of the given public key.

The key derivation function used is the standard datona crypto hash function. The encryption scheme used is AES-GCM. ECIES has been selected instead of an asymmetric scheme like RSA for performance reasons.

```
encrypt(publicKeyTo, data);
```

Parameters

1. `publicKeyTo` *(address)* public part of the remote key that will be used to decrypt this data
2. `data` *(bytes)* data to encrypt (e.g. as a string)

Returns

`bytes` - the encrypted data

Example

```
const encryptedData = myKey.encrypt(theirPublicKey, "Hello World"))
```

---

decrypt

Decrypts the given data that has been encrypted with the `encrypt` function. The given public key must be the public part of the private key used to encrypt the data and this key must be the private part of the public key used to encrypt the data.

```
decrypt(publicKeyFrom, data);
```

Parameters

1. `publicKeyFrom` *(address)* public part of the remote key that was used to encrypt this data
2. `data` *(bytes)* the encrypted data

Returns

`bytes` - the decrypted data

Example

```
const key1 = new Key("e68e40257cfee330038c49637fcffff82fae04b9c563f4ea071c20f2eb55063c
↪");
const key2 = new Key("b692ef5519cd87854b9bd97dd47a8929cbe473fe7a0da53e4ec79efec540cd2b
↪");
const encryptedData = key1.encrypt(key2.publicKey, "Hello World"));
const decryptedData = key2.decrypt(key1.publicKey, encryptedData));
assert(decryptedData == "Hello World");
```

2.5.2 Functions

generateKey

Generates a new Key object with a random private key. NB: This function does not use a true random source. Use only for experimental and test purposes.

```
generateKey();
```

Returns

`Key` - a new Key object with a random private key.

Example

```
const myPrivateKey = datona.crypto.generateKey();
```

---

sign

Signs the given hash using the given private key.

```
sign(hash, privateKey);
```

Parameters

1. `hash` *(Hash)* - 32-byte *hash* to sign in hex (64 hex characters)
2. `privateKey` *(PrivateKey)* - 32-byte *private key* in hex (64 hex characters)

Returns

`DatonaSignature` - the signature of the given hash derived from the given key

Example

```
const myPrivateKey = "e68e40257cfee330038c49637fcffff82fae04b9c563f4ea071c20f2eb55063c
↪";
const signature = sign(hash("Hello World!"), myPrivateKey);
```

---

verify

Verifies that the signatory of the given hash and signature matches the given address

```
verify(hash, signature, address);
```

Parameters

1. `hash` *(Hash)* - 32-byte *hash* to sign in hex (64 hex characters)
2. `signature` *(DatonaSignature)* - 65-byte *DatonaSignature* in hex (130 hex characters)
3. `address` *(Address)* - expected signatory *address* to verify against

Returns

`bool` - true if signatory matches the given address

Throws

- `InvalidHashError` if the hash is invalid
- `InvalidSignatureError` if the signatory could not be recovered

Example

```
const myKey = new Key(
↪"e68e40257cfee330038c49637fcffff82fae04b9c563f4ea071c20f2eb55063c");
const myHash = hash("Hello World!");
const signature = myKey.sign(myHash);
const matches = verify(myHash, signature, myKey.address);

console.log(matches);
> true
```

---

recover

Recovers the address of the signatory of the given hash and signature

```
recover(hash, signature);
```

Parameters

1. `hash` *(Hash)* - 32-byte *hash* to sign in hex (64 hex characters)
2. `signature` *(DatonaSignature)* - 65-byte *DatonaSignature* in hex (130 hex characters)

Returns

`Address` - address of the signatory (with leading 0x)

Throws

- `InvalidHashError` if the hash is invalid
- `InvalidSignatureError` if the signatory could not be recovered

Example

```
const myKey = new Key(
↪"e68e40257cfee330038c49637fcffff82fae04b9c563f4ea071c20f2eb55063c");
const myHash = hash("Hello World!");
const signature = myKey.sign(myHash);
const address = recover(myHash, signature);

console.log(address);
> 0x41A60F71063CD7c9e5247d3E7d551f91f94b5C3b

console.log(myKey.address == address);
> true
```

---

hash

Generates a keccak256 hash of the given data string

```
hash(data);
```

Parameters

1. `data` *(Buffer)* - the data to be hashed

Returns

`Hash` - *hash* of the given data as a 32-byte hex string (64 hex characters)

Example

```
const myHash = hash("Hello World!");

console.log(myHash);
> 3ea2f1d0abf3fc66cf29eebb70cbd4e7fe762ef8a09bcc06c8edf641230afec0
```

---

fileToHash

Generates a keccak256 hash of the given file's contents. Can handle files of any length.

```
fileToHash(path, nonce);
```

Parameters

1. `path` *(String)* - the file path
2. `nonce` *(String)* - (Optional) if present the nonce is appended to the file contents to form part of the hash

Returns

`Promise` - A promise to resolve with the hash of the file contents

Resolves With

`Hash` - *hash* of the given data as a 32-byte hex string (64 hex characters)

Rejects With

- `FileSystemError` - if the file cannot be read

Example

```
fileToHash("../myFiles/myFile.txt")
  .then( (hash) => { console.log("hash="+hash) })
  .catch(console.error);

> hash=3ea2f1d0abf3fc66cf29eebb70cbd4e7fe762ef8a09bcc06c8edf641230afec0
```

---

calculateContractAddress

Generates a contract address

```
calculateContractAddress(ownerAddress, nonce);
```

Parameters

1. `ownerAddress` *(Address)* - the blockchain address of the deployer
2. `nonce` *(uint)* - the owner's next transaction nonce

Returns

`Address` - blockchain address of the contract

Example

```
const contractAddress = calculateContractAddress(myKey.address, 1);
```

---

publicKeyToAddress

Calculates the address from a public key

```
publicKeyToAddress(publicKey);
```

Parameters

1. `publicKey` *(Uint8Array)* - public key as a byte buffer

Returns

`Address` - blockchain address of the public key

Example

```
const address = publicKeyToAddress(myKey.publicKey);
```

---

hexToUint8Array

Basic conversion function to convert a hex string to a Uint8Array

```
hexToUint8Array(hex);
```

Parameters

1. `hex` *(String)* - string of hex characters (without `0x` prefix)

Returns

`Uint8Array` - Uint8Array representation of the hex string

Example

```
const array = hexToUint8Array("010203fdfeff");
```

---

uint8ArrayToHex

Basic conversion function to convert a Uint8Array to a hex string

```
uint8ArrayToHex(array);
```

Parameters

1. `array` *(Uint8Array)* - array to convert

Returns

`String` - hex representation of the array (without `0x` prefix)

Example

```
const myArray = new Uint8Array([1, 2, 3, 253, 254, 255]);
const hex = uint8ArrayToHex(myArray);
console.log(hex)

> 010203fdfeff
```

2.6 datona-comms

The `datona-comms` component implements the Datona application layer protocol, providing classes and functions to facilitate communication between two datona enabled software applications.

2.6.1 Class SmartDataAccessRequest

Encapsulates a Smart Data Access request from a Requester to an Owner. This class validates the request and allows the user to accept or reject the request. If accept or reject is called, this class connects to the Requester's remote server to send the response.

Properties

- `data` *(Object)* - the transaction data object decoded from the signed transaction given to the constructor
- `remoteAddress` *(Address)* - the public blockchain address of the remote server, as decoded from the signed transaction given to the constructor

Constructor

Decodes and validates a raw request transaction from a requester, creating a new SmartDataAccessRequest instance.

```
new SmartDataAccessRequest(signedTxnStr, localPrivateKey);
```

Parameters

1. `signedTxnStr` *(String)* - The raw, signed request from the Requester
2. `localPrivateKey` *(Key)* - The Key object used to sign any transaction responses

Throws

- `TransactionError` - if the general transaction structure or signature is invalid
- `RequestError` - if the transaction is not a *SmartDataAccessRequestPacket* type or does not have the necessary request data

Example

```
const request = new SmartDataAccessRequest(rawTxn, myKey);
```

---

accept

Sends a *SmartDataAccessResponse* to the requester, giving the blockchain address of the SDAC and the URL of the vault that contains the data. Before responding to the requester it is expected that the contract is already deployed on the blockchain and a vault has already been created with the shared data.

```
accept(contractAddress, vaultUrl);
```

Parameters

1. `contractAddress` *(Address)* - The blockchain address of the deployed SDAC
2. `vaultUrl` *(URL)* - The URL of the vault holding the data

Returns

`Promise` - a promise to return the remote server response.

Resolves With

`{ txn:  Object, signatory:  Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the requester (i.e. the same signatory as the original request).

Rejects With

- `CommunicationError` - if communication with the requester's server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the requester.
- `InvalidTransactionError` - if the server response is not a valid *GeneralServerResponse*.

Example

```
const request = new SmartDataAccessRequest(rawTxn, myKey);
const vaultUrl = { scheme: "file", host: "datonavault.com", port: "8643" };

// Read contract bytecode and ABI from file system and create a Contract object
const contractSourceCode = require("./contracts/" + request.contract.hash);
const contract = new datona.blockchain.Contract(contractSourceCode.abi);

// Function to create a new vault and store the data.  Returns a Promise.
function createAndDeployVault(){
  const vault = new datona.vault.RemoteVault( vaultUrl, contract.address,␣
↪myPrivateKey );
  return vault.create("Hello World!");
}

// Function to send the contract address and vault URL to the requester.  Returns a␣
↪Promise.
function informRequester(){
  return request.accept(contract.address, vaultUrl);
}

return contract.deploy(contractSourceCode.bytecode, request.signatory)
  .then( createAndDeployVault })
  .then( informRequester )
  .then( console.log )
  .catch( console.error );
```

reject

Sends a *SmartDataAccessResponse* to the requester rejecting the request. This is not strictly necessary since the requester cannot rely on receiving a response. However, it is polite!

```
reject([reason]);
```

Parameters

1. `reason` *(String)* - *(Optional)* the reason for rejecting the request

Returns

`Promise` - a promise to return the remote server response.

Resolves With

`{ txn: Object, signatory: Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the requester (i.e. the same signatory as the original request).

Rejects With

- `CommunicationError` - if communication with the requester's server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the requester.
- `InvalidTransactionError` - if the server response is not a valid *GeneralServerResponse*.

Example

```
const request = new SmartDataAccessRequest(rawTxn, myKey);
request.reject("you are sharing my data with mail spammers")
  .then(console.log)
  .catch(console.error);
```

2.6.2 Class DatonaConnector

Enables communications with a remote server, abstracting away the underlying network protocols. Supported protocols are: *file* (plain tcp connection), *ws* (websocket) and *http*. Encrypted protocols, such as https and wss, will be supported in the future.

Designed to be used as a superclass, this class is extended by the *SmartDataAccessRequest* and *Class VaultFilename* classes.

Properties

- `remoteAddress` *(Address)* - the public blockchain address of the remote server, as given to the constructor

Constructor

Creates a new DatonaConnector instance with a network client suitable for the given url scheme.

```
new DatonaConnector(url, localPrivateKey, remoteAddress);
```

Parameters

1. `url` *(URL)* - the URL object identifying the server, port and URI scheme
2. `localPrivateKey` *(Key)* - The Key object used to sign any transactions
3. `remoteAddress` - the public blockchain address of the remote application. Used for verifying received responses.

Throws

- `RequestError` - if the url scheme is unsupported

Example

```
const url = { scheme: "file", host: "datonavault.com", port: "8643" };
const remoteAddress = "0x41A60F71063CD7c9e5247d3E7d551f91f94b5C3b";
const remoteServer = new DatonaConnector( url, myKey, remoteAddress);
```

---

send

Serialises the given object, signs it and returns a promise to send it to the requester.

```
send(txn);
```

Parameters

1. `txn` *(Object)* - the transaction to sign and send

Returns

`Promise` - a promise to return the remote server response.

Resolves With

`{ txn: Object, signatory: Address }` - the server response transaction and signatory's address, validated to confirm it was sent by the `remoteAddress` given in the constructor.

Rejects With

- `CommunicationError` - if communication with the requester's server failed
- `TransactionError` - if the structure of the server response was invalid or was not signed by the requester.

Example

```
const txn = { txnType: "VaultRequest", requestType: "access", contract:␣
↪myContractAddress };

remoteServer.send(txn)
  .then( console.log )
  .catch( console.error );
```

2.6.3 Functions

encodeTransaction

Signs the given transaction object and encodes it ready for transmission.

```
encodeTransaction(txn, key);
```

Parameters

1. `txn` *(Object)* - The transaction to encode and sign
2. `key` *(Key)* - The key used to sign the transaction

Returns

`String` - a *SignedTransaction* object as a JSON formatted string

Example

```
const myKey = new datona.crypto.Key(
↪"b94452c533536500e30f2253c96d123133ca1cbdb987556c2dc229573a2cd53c");

const txn = { txnType: "GeneralResponse", responseType: "success" };

const signedTxnStr = encodeTransaction(txn, myKey);
```

---

decodeTransaction

Decodes the given transaction object and returns the data payload.

```
decodeTransaction(signedTxnStr);
```

Parameters

1. `signedTxnStr` *(String)* - The JSON formatted *SignedTransaction*

Returns

`Object` - object containing the transaction and the signatory's address

```
{
  txn: Object,
  signatory: Address
}
```

Throws

`TransactionError` - if the transaction data or signature is invalid

Example

```
try {
  const txn = decodeTransaction(signedTxnStr);
  console.log("transaction type: "+txn.txn.txnType);
  console.log("signatory: "+txn.signatory);
}
catch (error) {
  console.error(error.toString());
}
```

---

validateResponse

Validates the a pre-decoded response transaction against the *GeneralServerResponse* format. If the response is valid and the response type

```
validateResponse(txn, [expectedTxnType]);
```

Parameters

1. `txn` *(Object)* - the transaction to validate
2. `expectedTxnType` *(String)* - (optional) the expected txnType of the response to override the default of *GeneralResponse*

Throws

`InvalidTransactionError` - if the transaction structure is invalid or the txnType does not match the expectedTxnType.

Example

```
try {
  validateResponse(myTransaction, "VaultResponse");
  // no error was thrown so must be a valid VaultResponse
}
catch (error) {
  console.error(error.toString());
}
```

---

createSuccessResponse

Constructs a *GeneralServerResponse* Success transaction, optionally of the given type.

```
createSuccessResponse([txnType]);
```

Parameters

1. `txnType` *(String)* - (optional) txnType to override default of *GeneralResponse*

Returns

`Object` - (optional) txnType the *GeneralServerResponse* transaction

Example

```
const response = createSuccessResponse();
const sdarResponse = createSuccessResponse("SmartDataAccessResponse");
```

---

createErrorResponse

Constructs a *GeneralServerResponse* Error transaction, optionally of the given type.

```
createErrorResponse(error, [txnType]);
```

Parameters

1. `error` *(Error)* - the error to insert in the transaction
2. `txnType` *(String)* - (optional) txnType to override default of *GeneralResponse*

Returns

`Object` - (optional) txnType the *GeneralServerResponse* transaction

Example

```
...
catch (error) {
  const response = createErrorResponse(error);
  const signedTxnStr = encodeTransaction(response, myKey);
  ...
}
```

2.7 Core Types

Type	Definition
Address	Blockchain address in the format `/^0x[0-9a-fA-F]{40}$/`
DatonaSignature	ECDSA secp256k1 signature - 130 hex-character string representing 64 byte signature (s) concatenated with 1 byte recovery (r) in that order
Hash	keccak256 hash in the format `/^[0-9a-fA-F]{64}$/`
PrivateKey	Private key in the format `/^[0-9a-fA-F]{64}$/`
PublicKey	Public key in the format `/^[0-9a-fA-F]{130}$/`
URL	Server URL of the form: `{ scheme: String, host: String, port: Number }`
VaultFile	Name of a file or directory in a vault. See *VaultFile* below.

2.7.1 VaultFile

A VaultFile name has the form `[directory/]<file>`

If the directory part is present it must be a single blockchain address and the file part can be any POSIX file name except `.` and `..` If not present then the file part must be a single blockchain address. Nested directories are not permitted.

Example valid files:

- `0x0000000000000000000000000000000000000001`
- `0x0000000000000000000000000000000000000002/my_file.txt`
- `0x0000000000000000000000000000000000000002/0x0000000000000000000000000000000000000001`

Example invalid files:

- `my_file.txt`

- `0x0000000000000000000000000000000000000002/0x0000000000000000000000000000000000000001/my_file.txt`

2.8 Application Layer Protocol

Version: 0.0.2

WARNING - This protocol is experimental and subject to change without notice. The version will be updated if any change is made.

This protocol uses Semantic Versioning.

2.8.1 Smart Data Access Contract Interface

All S-DACs must comply with the following interface. In future the protocolVersion may be used to support backward compatibility.

```
pragma solidity ^0.6.3;

abstract contract SDAC {

    string public constant DatonaProtocolVersion = "0.0.2";

    // constants describing the permissions-byte structure of the form d----rwa.
    byte public constant NO_PERMISSIONS = 0x00;
    byte public constant ALL_PERMISSIONS = 0x07;
    byte public constant READ_BIT = 0x04;
    byte public constant WRITE_BIT = 0x02;
    byte public constant APPEND_BIT = 0x01;
    byte public constant DIRECTORY_BIT = 0x80;

    address public owner = msg.sender;

    // File based d----rwa permissions.  Assumes the data vault has validated the␣
↪requester's ID.
    // Address(0) is a special file representing the vault's root
    function getPermissions( address requester, address file ) public virtual view␣
↪returns (byte);

    // returns true if the contract has expired either automatically or has been␣
↪manually terminated
    function hasExpired() public virtual view returns (bool);

    // terminates the contract if the sender is permitted and any termination␣
↪conditions are met
    function terminate() public virtual;

}
```

2.8.2 General Protocol

SignedTransaction

All Datona transactions are sent as a SignedTransaction, which contains the raw *Transaction* and a digital signature.

```
{
  "txn": Transaction,
  "signature": DatonaSignature
}
```

where:

- `signature` is the DatonaSignature of the keccak256 hash of the `txn` element.

Transaction

All transactions in the Datona Protocol have the following JSON structure:

```
{
  "txnType": String,
  ...
}
```

Field	Description
txnType	*(String)*. The name of the transaction type used to identify the type of transaction.

GeneralServerResponse

A basic acknowledgement or error response from a server to a client.

Acknowledgement

```
{
  "txnType": "GeneralResponse",
  "responseType":"success"
}
```

Field	Description
responseType	*(String)* The type of the response: either “success” or “error”

Error

Error responses contain the fields of a DatonaError.

```
{
  "txnType": "GeneralResponse",
  "responseType":"error",
  "error": {
    "name": String,
    "message": String,
    "details": String
  }
}
```

Field	Description
name	*(String)* Name of error
message	*(String)* Natural language error message
details	*(String)* Detailed error message, usually not suitable for displaying to the average user. Possibly empty.

2.8.3 Smart Data Access Request Protocol

A SmartDataAccessRequest is sent from a requester to a data owner, to request data to be shared in a vault controlled by a Smart Data Access Contract. The data owner can respond with a SmartDataAccessResponse accepting or rejecting the request.

The format of the response is specific to the requester's use case. Therefore, the SmartDataAccessRequest contains user defined acceptTransaction and rejectTransaction elements that the requester is free to tailor as needed.

If accepting the request, the owner's application software is required to construct a SmartDataAccessResponse using the template given in the acceptTransaction element and extend it with (a) the url of the data vault server holding the data, and (b) the blockchain address of the deployed S-DAC.

If rejecting the request, the owner's application software is required to construct a SmartDataAccessResponse using the template given in the rejectTransaction and extend it with the reason for the rejection.

SmartDataAccessRequestPacket

The following JSON gives the minimal template spec for a Smart Data Access request from Requester to Owner.

```
{
  "txnType": "SmartDataAccessRequest",
  "version": "0.0.1",
  "contract": {
    "hash": Hash
  },
  "api": {
    "url": {
      "scheme": String,
      "host": String,
      "port": uint
    },
    "acceptTransaction": {},
    "rejectTransaction": {}
  }
}
```

Field	Description
version	*(String)* The version of the Smart Data Access Request protocol with which this request is compliant.
contract	*(Object)* The requested S-DAC and associated details
contract.hash	*(Hash)* keccak256 hash of the requested S-DAC's runtime bytecode
api	*(Object)*. Details of how the owner-end software should respond to the request.
api.url	*(URL)* URL of the Requester's server that will handle the response. See Type Definitions.
api.acceptTransaction	*(Object)* Template for the transaction that will be returned to the requester if the request is accepted. Requester specific - for example can be configured to include an internal reference number. Shall be extended with the following fields: • *contract*: *Address* of the S-DAC deployed on the blockchain, staring with `0x` • *vaultUrl*: *URL* of the vault service that is hosting the data, in the same format as *api.url* defined above.
api.rejectTransaction	*(Object)* Template for the transaction that will be returned to the requester if the request is rejected. Will be extended with the following fields: • *reason*: `String` message containing the reason for the rejection

SmartDataAccessResponse

An accept response consists of copying the acceptTransaction object from the SmartDataAccessRequestPacket and adding the following elements:

```
{
  "txnType": "SmartDataAccessResponse",
  "responseType": "accept",
  "contract": Address,
  "vaultAddress": Address,
  "vaultUrl": {
    "scheme": String,
    "host": String,
    "port": uint
  }
  ... elements copied from the acceptTransaction object (if any)
}
```

Field	Description
contract	*(Address)* Blockchain address of the deployed S-DAC
vaultAddress	*(Address)* Public address of the vault server (used to authenticate all comms with the server)
vaultUrl	*(URL)* URL of the Requester's server that will handle the response.

A reject response consists of copying the rejectTransaction object from the SmartDataAccessRequestPacket and adding the following elements:

```
{
  "txnType": "SmartDataAccessResponse",
  "responseType": "reject",
  "reason": String
```

(continues on next page)

(continued from previous page)

```
}
... elements copied from the rejectTransaction object (if any)
```

2.8.4 Vault Request Protocol

VaultRequest packets are sent to a Data Vault Server to create, write, append, read or delete a vault. The server promises to respond to any request with a VaultResponse packet indicating success or error. The protocol consists of a single request and response.

VaultRequest

One of the following JSON requests:

create

```
{
  "txnType": "VaultRequest",
  "requestType": "create",
  "contract": Address,
}
```

write

```
{
  "txnType": "VaultRequest",
  "requestType": "write",
  "contract": Address,
  "data": Object
}
```

append

```
{
  "txnType": "VaultRequest",
  "requestType": "append",
  "contract": Address,
  "data": Object
}
```

read

```
{
  "txnType": "VaultRequest",
  "requestType": "read",
  "contract": Address
}
```

delete

```
{
  "txnType": "VaultRequest",
  "requestType": "delete",
  "contract": Address
}
```

Field	Description
type	*(String)* The type of request: either "create", "write", "append", "read" or "delete"
contract	*(Address)* The blockchain address of the Smart Data Access Contract that controls the vault. The S-DAC must already be deployed on the blockchain.
data	Any type. The data to store in the vault or retrieved from the vault

---

VaultResponse

Every Vault Request from the client is responded to with a Vault Response. There are two types of response - success and error.

success

A success response conforms with the *GeneralServerResponse* Acknowledgement format. If responding to a read request, the response will additionally contain a `data` field with returned vault contents.

```
{
  "txnType": "VaultResponse",
  "responseType":"success",
  "data": Object
}
```

error

An error response conforms with the *GeneralServerResponse* Error format.

```
{
  "txnType": "VaultResponse",
  "responseType":"error",
  "error": {
    "name": String,
    "message": String,
    "details": String
```

(continues on next page)

(continued from previous page)

```
  }
}
```

2.9 Errors

2.9.1 Class DatonaError

Root class for all errors thrown by Datona software. Extends `Error`.

Properties

- `name` *(String)* - the name of the error. Same as the class name.
- `message` *(String)* - single line error message suitable for display to the end user
- `details` *(String)* - (Optional) detailed error message unsuitable for display to the end user

Constructor

Creates a new RemoteVault instance with a network client suitable for the given url scheme.

```
new DatonaError(message, details);
```

Parameters

1. `message` *(String)* - single line error message suitable for display to the end user
2. `details` *(String)* - (Optional) detailed error message unsuitable for display to the end user

Example

```
const url = { scheme: "file", host: "datonavault.com", port: "8643" };
const myContractAddress = "0x008Cd346b65F5aFa306Ef9160a84455D308e6851";
const remoteAddress = "0x41A60F71063CD7c9e5247d3E7d551f91f94b5C3b";
const remoteVault = new RemoteVault(url, myContractAddress, myKey, remoteAddress);
```

---

toJSON

Converts this error into a JSON formatted string, excluding the stacktrace.

```
toJSON();
```

Returns

`String` - A JSON formatted string representation of this error with name, message and details.

---

toObject

Converts this error into a simple struct with just name, message and details, excluding the stacktrace.

```
toObject();
```

Returns

`{ name:  String, message:  String, details:  String }`

---

toString

Converts this error into a single line string suitable for logging, excluding the stacktrace. If the error details property is longer than 96 chars then it will be truncated.

```
toString();
```

Returns

`String` - String version of this object

2.9.2 Classes of DatonaError

All error classes listed below are derived from *DatonaError* and have the same constructor parameters. Each class may have its own subclass allowing catch blocks to switch based on error class.

Internal Errors

`InternalError` Class of exception for defensive programming checks. These errors are not expected to be raised and indicate a low-level software problem that needs raising with the software developer.

Developer Errors

`DeveloperError`. Class of exception for software usage errors. These errors indicate a problem with how the developer is interfacing with or using this software.

Subclasses

`ArgumentError` The caller of this method passed an invalid or missing argument

`TypeError` The caller of this method passed an argument with an invalid type

`InvalidHashError` The caller of this method passed an invalid hash

Cryptographic Errors

`CryptographicError` Class of cryptographic errors

Subclasses

`InvalidSignatureError` The caller of this method passed an invalid hash

`HashingError` The data could not be hashed

Blockchain Errors

`BlockchainError` Class of errors related to blockchain access and contract management

Subclasses

`ContractOwnerError` This request must be made by the contract owner

`ContractTypeError` Indicates the contract class is invalid

`ContractExpiryError` This request must be made by the contract owner

`PermissionError` Indicates the signatory does not have permission to perform this action

Transaction Errors

`TransactionError` Class of errors related to a communications transaction

Subclasses

`InvalidTransactionError` Indicates the transaction type is invalid

`MalformedTransactionError` Indicates the transaction has an invalid form

`RequestError` Indicates the transaction contains an invalid request

`CommunicationError` Class of errors related to a communications transaction

Vault Errors

`VaultError` Class of errors related to vault management and guardianship

Subclasses

`FileSystemError` Error resulting from filesystem access