Dash Platform

Release latest

thephez

Sep 21, 2023

CONTENTS

Dash aims to be the most user-friendly and scalable payments-focused cryptocurrency in the world. The Dash network features instant transaction confirmation, double spend protection, optional privacy equal to that of physical cash, a self-governing, and a self-funding model driven by incentivized full nodes. While Dash is based on Bitcoin and compatible with many key components of the Bitcoin ecosystem, its two-tier network structure offers significant improvements in transaction speed, privacy and governance. This section of the documentation describes these and many more key features that set Dash apart in the blockchain economy.

Check out the official Dash website to learn how individuals and businesses can use Dash.

User Docs Learn what Dash is and how it works. Topics include how to obtain and store Dash, the governance system, and masternode setup.

Click to begin

Core Docs Find technical details about the Dash Core blockchain, along with protocol and API reference material.

Click to begin

Platform Docs Start working with Dash Platform and discover how you can use its powerful capabilities to power your Web3 project.

Click to begin

CHAPTER

ONE

PLATFORM DOCS

Welcome to the Dash Platform developer documentation. You'll find guides and documentation to help you start working with Dash Platform and building decentralized applications based on the Dash cryptocurrency. Let's jump right in!

Introduction Background information about Dash

Click to begin

Tutorials Basics of building with Dash Platform

Click to begin

Explanations Descriptions of Dash Platform features

Click to begin

Reference API endpoint details and technical information

Click to begin

Platform Protocol Reference Dash Platform protocol reference

Click to begin

Resources Links to helpful sites and tools

Click to begin

Dash SDK JavaScript SDK documentation

Click to begin

DAPI Client JavaScript DAPI-Client documentation

Click to begin

1.1 What is Dash

Dash is the world's first and longest-running DAO, a cryptocurrency that has stood the test of time, a truly decentralized and open source project built without a premine, ICO, or venture capital investment. Dash is the only solution on the market today developing a decentralized API as an integral part of its Web3 stack, making it the first choice for developers creating unstoppable apps.

Dash is built on battle-tested technology including Bitcoin and Cosmos Tendermint, and implements cutting-edge threshold signing features on masternodes to guarantee transaction finality in little more than a second. Dash is fast,

private, secure, decentralized, and open-source, your first choice for truly decentralized Web3 services and for earning yield in return for providing infrastructure services.

1.1.1 Key Advantages

Industry Leading Security

The Dash network is the most secure blockchain-based payments network, thanks to technological innovations such as *ChainLocks*. This mitigates the risk of 51% attacks, forcing any would-be malicious actor to successfully attack both the mining layer and the *masternode* layer. To attack both layers, a malicious actor would have to spend a large amount of Dash in order to dictate false entries to the blockchain, thereby raising the price of Dash in the process. Therefore, a successful attack would be cost prohibitive due to the large percentage of Dash's total market required to attempt it.

Stable and Long Lasting Governance

The Dash *decentralized autonomous organization (DAO)* is the oldest and most successful example of decentralized governance. In that regard, one of Dash's most notable innovations is the creation of a treasury, which funds project proposals that advance the Dash network and ecosystem. This treasury is funded by 10% of the block reward, which is a combination of transaction fees collected on the network and newly minted Dash awarded to miners for securing the blockchain. Nodes that maintain a minimum of 1000 Dash (*masternodes*) receive voting rights on how to distribute treasury funds. Voting on project proposals encourages engagement with the overall network and ecosystem, resulting in numerous projects being funded that advance Dash in terms of technology development, marketing, and business development.

Established History of Technological Innovation

Most of Dash's technical innovations are described in greater detail elsewhere in this developer hub. However, its record speaks for itself with innovations in governance (masternodes, treasury system), security (ChainLocks), usability (automatic InstantSend), and scalability (*long-living masternode quorums*).

Instantly Confirmed Transactions

All transactions are automatically sent and received instantly at no extra cost. Transaction security and decentralization are not compromised, due to the ChainLocks innovation. As a result, using Dash to transact means getting the speed and fungibility of fiat currency, while simultaneously having the lower costs, privacy, and security of funds of a blockchain-based network.

1.1.2 Key Features

Masternodes

The most important differentiating feature of the Dash payments network is the concept of a **masternode**. On a traditional p2p network, nodes participate equally in the sharing of data and network resources. These nodes are all compensated equally for their contributions toward preserving the network.

However, the Dash network has a second layer of network participants that provide enhanced functionality in exchange for greater compensation. This second layer of masternodes is the reason why Dash is the most secure payments network, and can provide industry-leading features such as instant transaction settlement and usernames.

Long-Living Masternode Quorums

Dash's long-living masternode quorums (LLMQs) are used to facilitate the operation of masternode-provided features in a decentralized, deterministic way. These LLMQs are deterministic subsets of the overall masternode list that are formed via a *distributed key generation* protocol and remain active for long periods of time (e.g. hours to days). The main task of LLMQs is to perform threshold signing of consensus-related messages for features like InstantSend and ChainLocks.

InstantSend

InstantSend provides a way to lock transaction inputs and enable secure, instantaneous transactions. Long-living masternode quorums check whether or not a submitted transaction is valid. If it is valid, the masternodes "lock" the inputs to that specific transaction and broadcast this information to the network, effectively promising that the transaction will be included in subsequently mined blocks and not allowing any other transaction to spend any of the locked inputs.

ChainLocks

ChainLocks are a feature provided by the Dash Network which provides certainty when accepting payments. This technology, particularly when used in parallel with InstantSend, creates an environment in which payments can be accepted immediately and without the risk of "Blockchain Reorganization Events".

The risk of blockchain reorganization is typically addressed by requiring multiple "confirmations" before a transaction can be safely accepted as payment. This type of indirect security is effective, but at a cost of time and user experience. ChainLocks are a solution for this problem.

Proof-of-Service

The Proof of Service (PoSe) scoring system helps incentivize masternodes to provide network services. Masternodes that fail to participate in quorums that provide core services are penalized, which eventually results in them being excluded from masternode payment eligibility.

1.2 What is Dash Platform

Dash Platform is a Web3 technology stack for building decentralized applications on the Dash network. The two main architectural components, *Drive* and *DAPI*, turn the Dash P2P network into a cloud that developers can integrate with their applications.

1.2.1 Key Advantages

Decentralized Cloud Storage

Store your application data in the safest place on the Internet. All data stored on the Dash network is protected by Dash's consensus algorithm, ensuring data integrity and availability.

Reduced Data Silos

Because your application data is stored across many nodes on the Dash network, it is safe and always available for customers, business partners, and investors.

Client Libraries

Write code and integrate with Dash Platform using the languages that matter to your business. Don't worry about understanding blockchain infrastructure: a growing number of client libraries abstract away the complexity typically associated with working on blockchain-based networks.

Instant Data Confirmation

Unlike many blockchain-based networks, data stored on the platform is instantly confirmed by the Dash consensus algorithm to ensure the best user experience for users. With Dash Platform, you can gain the advantages of a blockchain-based storage network without the usual UX compromises.

1.2.2 Key Components

DAPI - A decentralized API

DAPI is a *decentralized* HTTP API exposing JSON-RPC and gRPC endpoints. Through these endpoints, developers can send and retrieve application data and query the Dash blockchain.

DAPI provides developers the same access and security as running their own Dash node without the cost and maintenance overhead. Unlike traditional APIs which have a single point of failure, DAPI allows clients to connect to different instances depending on resource availability in the Dash network.

Developers can connect to DAPI directly or use a client library. This initial client library, dapi-client, is a relatively simple API wrapper developed by Dash Core Group to provide function calls to the DAPI endpoints.

The source for both DAPI and dapi-client are available on GitHub:

- DAPI: https://github.com/dashpay/platform/tree/master/packages/dapi
- DAPI-Client: https://github.com/dashpay/platform/tree/master/packages/js-dapi-client

Drive - Decentralized Storage

Drive is Dash Platform's storage component, allowing for consensus-based verification and validation of user-created data. In order for this to occur, developers create a *data contract*. This data contract describes the data structures that comprise an application, similar to creating a schema for a document-oriented database like MongoDB.

Data created by users of the application is validated and verified against this contract. Upon successful validation/verification, application data is submitted to Drive (via DAPI), where it is stored on the masternode network. Drive uses Dash's purpose-built database, GroveDB, to provide efficient proofs with query responses, so you don't have to trust the API provider to be certain your data is authentic.

The source is available on GitHub:

- Drive: https://github.com/dashpay/platform/tree/master/packages/js-drive

1.3 Intro to Testnet

Testnet is the Dash testing network used for experimentation and evaluation of Dash Core and Dash Platform features. As a testing network, Testnet may be subject to occasional updates and changes that break backwards compatibility.

1.3.1 Network Details

Infrastructure

Dash Core Group provides the core Testnet infrastructure consisting of 150 masternodes running Dash Core along with the platform services that provide the *decentralized API (DAPI)* and *storage (Drive)* functionality.

Testnet also includes a block explorer for the core blockchain and a test Dash faucet that dispenses funds to users/developers experimenting on the network.

Features

The Dash Platform features available on testnet include:

- *Dash Platform Name Service* (DPNS): a data contract and supporting logic for name registration
- *Identities*: creation of identities
- *Names*: creation of DPNS names that link to an identity
- *Data Contracts*: creation of data contracts
- *Documents*: used to store/update/delete data associated with data contracts
- *DashPay*: a data contract enableing a decentralized application that creates bidirectional direct settlement payment channels between identities and supports contact (name) based payments

1.3.2 Getting involved

This network is open for all who are interested in testing and interacting with Dash Platform. To learn how to connect, please jump to the *Connecting to a Network tutorial*.

1.4 Introduction

The tutorials in this section walk through the steps necessary to begin building on Dash Platform using the Dash JavaScript SDK. As all communication happens via the masternode hosted decentralized API (DAPI), you can begin using Dash Platform immediately without running a local blockchain node.

Building on Dash Platform requires first registering an Identity and then registering a Data Contract describing the schema of data to be stored. Once that is done, data can be stored and updated by submitting Documents that comply with the Data Contract.

> Tutorial code
>
> You can clone a repository containing the code for all tutorials from GitHub or download it as a zip file.

1.4.1 Prerequisites

The tutorials in this section are written in JavaScript and use Node.js. The following prerequisites are necessary to complete the tutorials:

- Node.js (v12+)
- Familiarity with JavaScript asynchronous functions using async/await
- The Dash JavaScript SDK (see *Connecting to a Network*)

1.4.2 Quickstart

While going through each tutorial is advantageous, the subset of tutorials listed below get you from a start to storing data on Dash Platform most quickly:

- *Obtaining test funds*
- *Registering an Identity*
- *Registering a Data Contract*
- *Submitting data*

1.5 Connect to a network

The purpose of this tutorial is to walk through the steps necessary to access the network.

1.5.1 Overview

Platform services are provided via a combination of HTTP and gRPC connections to DAPI, and some connections to an Insight API. Although one could interact with DAPI by connecting to these directly, or by using DAPI-client, the easiest approach is to use the JavaScript Dash SDK. The Dash SDK connects to the testnet by default.

1.5.2 Prerequisites

- An installation of NodeJS v12 or higher

1.5.3 Connect via Dash SDK

1. Install the Dash SDK

The JavaScript SDK package is available from npmjs.com and can be installed by running `npm install dash` from the command line:

```
npm install dash
```

2. Connect to Dash Platform

Create a file named `dashConnect.js` with the following contents. Then run it by typing `node dashConnect.js` from the command line:

```
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

async function connect() {
  return await client.getDAPIClient().core.getBestBlockHash();
}

connect()
  .then((d) => console.log('Connected. Best block hash:\n', d))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Once this returns successfully, you're ready to begin developing! See the *Quickstart* for recommended next steps. For details on all SDK options and methods, please refer to the *SDK documentation*.

1.5.4 Connect to a Devnet

The SDK also supports connecting to development networks (devnets). Since devnets can be created by anyone, the client library will be unaware of them unless connection information is provided using one of the options described below.

Connect via Seed

Using a seed node is the preferred method in most cases. The client uses the provided seed node to a retrieve a list of available masternodes on the network so requests can be spread across the entire network.

```
const Dash = require('dash');

const client = new Dash.Client({
  seeds: [{
    host: 'seed-1.testnet.networks.dash.org:1443',
  }],
});

async function connect() {
  return await client.getDAPIClient().core.getBestBlockHash();
}

connect()
  .then((d) => console.log('Connected. Best block hash:\n', d))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Connect via Address

Custom addresses may be directly specified via `dapiAddresses` in cases where it is beneficial to know exactly what node(s) are being accessed (e.g. debugging, local development, etc.).

```
const Dash = require('dash');

const client = new Dash.Client({
  dapiAddresses: [
    '127.0.0.1:3000:3010',
    '127.0.0.2:3000:3010',
  ],
});

async function connect() {
  return await client.getDAPIClient().core.getBestBlockHash();
}

connect()
  .then((d) => console.log('Connected. Best block hash:\n', d))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

1.5.5 Connect Directly to DAPI (Optional)

> Advanced Topic
>
> Normally, the Dash SDK, dapi-client, or another library should be used to interact with DAPI. This may be helpful for debugging in some cases, but generally is not required.

The example below demonstrates retrieving the hash of the best block hash directly from a DAPI node via command line and several languages:

SHELL

```
curl --request POST \
  --url https://seed-1.testnet.networks.dash.org:1443/ \
  --header 'content-type: application/json' \
  --data '{"method":"getBlockHash","id":1,"jsonrpc":"2.0","params":{"height": 100 }}'
```

PYTHON

```
import requests

url = "https://seed-1.testnet.networks.dash.org:1443/"

payload = "{\"method\":\"getBlockHash\",\"id\":1,\"jsonrpc\":\"2.0\",\"params\":{\
↪"height\":100}}"
headers = {'content-type': 'application/json'}
```

(continues on next page)

(continued from previous page)

```
response = requests.request("POST", url, data=payload, headers=headers)

print(response.text)
```

RUBY

```
require 'uri'
require 'net/http'

url = URI("https://seed-1.testnet.networks.dash.org:1443/")

http = Net::HTTP.new(url.host, url.port)

request = Net::HTTP::Post.new(url)
request["content-type"] = 'application/json'
request.body = "{\"method\":\"getBlockHash\",\"id\":1,\"jsonrpc\":\"2.0\",\"params\":{\
↪"height\":100}}"

response = http.request(request)
puts response.read_body
```

1.6 Create and fund a wallet

In order to make changes on Dash Platform, you need a wallet with a balance. This tutorial explains how to generate a new wallet, retrieve an address from it, and transfer test funds to the address from a faucet.

1.6.1 Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)

1.7 Code

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: null, // this indicates that we want a new wallet to be generated
    // if you want to get a new address for an existing wallet
    // replace 'null' with an existing wallet mnemonic
    offlineMode: true,  // this indicates we don't want to sync the chain
    // it can only be used when the mnemonic is set to 'null'
  },
};
```

(continues on next page)

(continued from previous page)

```
const client = new Dash.Client(clientOpts);

const createWallet = async () => {
  const account = await client.getWalletAccount();

  const mnemonic = client.wallet.exportWallet();
  const address = account.getUnusedAddress();
  console.log('Mnemonic:', mnemonic);
  console.log('Unused address:', address.address);
};

createWallet()
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());

// Handle wallet async errors
client.on('error', (error, context) => {
  console.error(`Client error: ${error.name}`);
  console.error(context);
});
```

```
Mnemonic: thrive wolf habit timber birth service crystal patient tiny depart tower focus
Unused address: yXF7LsyajRvJGX96vPHBmo9Dwy9zEvzkbh
```

Please save your mnemonic for the next step and for re-use in subsequent tutorials throughout the documentation.

1.8 What's Happening

Once we connect, we output the newly generated mnemonic from `client.wallet.exportWallet()` and an unused address from the wallet from `account.getUnusedAddress()`.

1.9 Next Step

Using the faucet at https://testnet-faucet.dash.org/, send test funds to the "unused address" from the console output. You will need to wait until the funds are confirmed to use them. There is a block explorer running at https://testnet-insight.dashevo.org/insight/ which can be used to check confirmations.

1.10 Identities and names

The following tutorials cover creating and managing identities as well as creating and retrieving names.

- *Register an Identity*
- *Retrieve an Account's Identities*
- *Topup an Identity's Balance*
- *Register a Name for an Identity*
- *Retrieve a Name*

Tutorial code

You can clone a repository containing the code for all tutorials from GitHub or download it as a zip file.

1.10.1 Register an Identity

The purpose of this tutorial is to walk through the steps necessary to register an identity.

Overview

Identities serve as the basis for interactions with Dash Platform. They consist primarily of a public key used to register a unique entity on the network. Additional details regarding identities can be found in the *Identity description*.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *How to Create and Fund a Wallet*

Code

Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
          mnemonic: 'a Dash wallet mnemonic with testnet funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const createIdentity = async () => {
  return client.platform.identities.register();
};

createIdentity()
  .then((d) => console.log('Identity:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

The Identity will be output to the console. The Identity will need to have one confirmation before it is accessible via `client.platform.identity.get`.

> **Make a note of the returned identity `id` as it will be used used in subsequent tutorials throughout the documentation.**

What's Happening

After connecting to the Client, we call `platform.identities.register`. This will generate a keypair and submit an *Identity Create State Transaction*. After the Identity is registered, we output it to the console.

1.10.2 Retrieve an identity

In this tutorial we will retrieve the identity created in the *Register an Identity tutorial*.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A Dash Platform Identity: *Tutorial: Register an Identity*

1.10.3 Code

```
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

const retrieveIdentity = async () => {
  return client.platform.identities.get('an identity ID goes here');
};

retrieveIdentity()
  .then((d) => console.log('Identity retrieved:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

1.10.4 Example Identity

The following example response shows a retrieved identity:


```
{
   "protocolVersion":0,
   "id":"6Jz8pFZFhssKSTacgQmZP14zGZNnFYZFKSbx4WVAJFy3",
   "publicKeys":[
      {
         "id":0,
         "type":0,
         "data":"A4zZl0EaRBB6IlDbyR80YUM2l02qqNUCoIizkQxubtxi"
      }
   ],
   "balance":10997588,
   "revision":0
}
```


1.10.5 What's Happening

After we initialize the Client, we request an identity. The `platform.identities.get` method takes a single argument: an identity ID. After the identity is retrieved, it is displayed on the console.

1.10.6 Topup an identity's balance

The purpose of this tutorial is to walk through the steps necessary to add credits to an identity's balance.

1.10.7 Overview

As users interact with Dash Platform applications, the credit balance associated with their identity will decrease. Eventually it will be necessary to topup the balance by converting some Dash to credits. Additional details regarding credits can be found in the *Credits description*.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- A Dash Platform Identity: *Tutorial: Register an Identity*

1.10.8 Code

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with testnet funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const topupIdentity = async () => {
  const identityId = 'an identity ID goes here';
  const topUpAmount = 1000; // Number of duffs

  await client.platform.identities.topUp(identityId, topUpAmount);
  return client.platform.identities.get(identityId);
};

topupIdentity()
  .then((d) => console.log('Identity credit balance: ', d.balance))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

1.10.9 What's Happening

After connecting to the Client, we call `platform.identities.topUp` with an identity ID and a topup amount in duffs (1 duff = 1000 credits). This creates a lock transaction and increases the identity's credit balance by the relevant amount (minus fee). The updated balance is output to the console.

> Wallet Operations
>
> The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.10.10 Update an identity

Since Dash Platform v0.23, it is possible to update identities to add new keys or disable existing ones. Platform retains disabled keys so that any existing data they signed can still be verified while preventing them from signing new data.

1.10.11 Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- A Dash Platform Identity: *Tutorial: Register an Identity*

1.10.12 Code

The two examples below demonstrate updating an existing identity to add a new key and disabling an existing key:

The current SDK version signs all state transitions with public key id 1. If it is disabled, the SDK will be unable to use the identity. Future SDK versions will provide a way to also sign using keys added in an identity update.

JAVASCRIPT

```
// Disable identity key
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const updateIdentityDisableKey = async () => {
  const identityId = 'an identity ID goes here';
  const keyId = 'a public key ID goes here'; // One of the identity's public key IDs

  // Retrieve the identity to be updated and the public key to disable
  const existingIdentity = await client.platform.identities.get(identityId);
  const publicKeyToDisable = existingIdentity.getPublicKeyById(keyId);

  const updateDisable = {
```

(continues on next page)

(continued from previous page)

```
    disable: [publicKeyToDisable],
  };

  await client.platform.identities.update(existingIdentity, updateDisable);
  return client.platform.identities.get(identityId);
}

updateIdentityDisableKey()
  .then((d) => console.log('Identity updated:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// Add identity key
const Dash = require('dash');
const { IdentityPublicKey, IdentityPublicKeyWithWitness } = require('@dashevo/wasm-dpp');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const updateIdentityAddKey = async () => {
  const identityId = 'an identity ID goes here';
  const existingIdentity = await client.platform.identities.get(identityId);
  const newKeyId = existingIdentity.toJSON().publicKeys.length;

  // Get an unused identity index
  const account = await client.platform.client.getWalletAccount();
  const identityIndex = await account.getUnusedIdentityIndex();

  // Get unused private key and construct new identity public key
  const { privateKey: identityPrivateKey } =
    account.identities.getIdentityHDKeyByIndex(identityIndex, 0);

  const identityPublicKey = identityPrivateKey.toPublicKey().toBuffer();

  const newPublicKey = new IdentityPublicKeyWithWitness({
    id: newKeyId,
    type: IdentityPublicKey.TYPES.ECDSA_SECP256K1,
    data: identityPublicKey,
    purpose: IdentityPublicKey.PURPOSES.AUTHENTICATION,
    securityLevel: IdentityPublicKey.SECURITY_LEVELS.CRITICAL,
```

(continues on next page)

(continued from previous page)

```
    readOnly: false,
    signature: Buffer.alloc(0),
  });

  const updateAdd = {
    add: [newPublicKey],
  };

  // Submit the update signed with the new key
  await client.platform.identities.update(existingIdentity, updateAdd, {
    [newPublicKey.getId()]: identityPrivateKey,
  });

  return client.platform.identities.get(identityId);};
};

updateIdentityAddKey()
  .then((d) => console.log('Identity updated:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

1.10.13 What's Happening

Disabling keys

After we initialize the Client, we retrieve our existing identity and provide the `id` of one (or more) of the identity keys to disable. The update is submitted to DAPI using the `platform.identities.update` method with two arguments:

1. An identity
2. An object containing the key(s) to be disabled

Internally, the method creates a State Transition containing the updated identity, signs the state transition, and submits the signed state transition to DAPI. After the identity is updated, we output it to the console.

Adding keys

After we initialize the Client, we retrieve our existing identity and set an `id` for the key to be added. Next, we get an unused private key from our wallet and use it to derive a public key to add to our identity. The update is submitted to DAPI using the `platform.identities.update` method with three arguments:

1. An identity
2. An object containing the key(s) to be added
3. An object containing the id and private key for each public key being added

> When adding new public keys, they must be signed using the associated private key to prove ownership of the keys.

Internally, the method creates a State Transition containing the updated identity, signs the state transition, and submits the signed state transition to DAPI. After the identity is updated, we output it to the console.

1.10.14 Retrieve an account's identities

In this tutorial we will retrieve the list of identities associated with a specified mnemonic-based account. Since multiple identities may be created using the same mnemonic, it is helpful to have a way to quickly retrieve all these identities (e.g. if importing the mnemonic into a new device).

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic
- A Dash Platform Identity: *Tutorial: Register an Identity*

1.10.15 Code

```
const Dash = require('dash');

const client = new Dash.Client({
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with testnet funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
});

const retrieveIdentityIds = async () => {
  const account = await client.getWalletAccount();
  return account.identities.getIdentityIds();
};

retrieveIdentityIds()
  .then((d) => console.log('Mnemonic identities:\n', d))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Example Response

```
[
  "6Jz8pFZFhssKSTacgQmZP14zGZNnFYZFKSbx4WVAJFy3",
  "8XoJHG96Vfm3eGh1A7HiDpMb1Jw2B9opRJe8Z38urapt",
  "CEPMcuBgAWeaCXiP2gJJaStANRHW6b158UPvL1C8zw2W",
  "GTGZrkPC72tWeBaqopSCKgiBkVVQR3s3yBsVeMyUrmiY"
]
```

1.10.16 What's Happening

After we initialize the Client and getting the account, we call `account.identities.getIdentityIds()` to retrieve a list of all identities created with the wallet mnemonic. The list of identities is output to the console.

> Wallet Operations
>
> The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.
>
> A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.10.17 Register a name for an identity

The purpose of this tutorial is to walk through the steps necessary to register a *Dash Platform Name Service (DPNS)* name.

Overview

Dash Platform names make cryptographic identities easy to remember and communicate. An identity may have multiple alias names (`dashAliasIdentityId`) in addition to its default name (`dashUniqueIdentityId`). Additional details regarding identities can be found in the *Identity description*.

Note: An identity must have a default name before any aliases can be created for the identity.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- A Dash Platform identity: *Tutorial: Register an Identity*
- A name you want to register: *Name restrictions*

Code

The examples below demonstrate creating both the default name and alias names.

Note: the name must be the full domain name including the parent domain (i.e. `myname.dash` instead of just `myname`). Currently `dash` is the only top-level domain that may be used.

JAVASCRIPT

```
// Register Name for Identity
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with testnet funds goes here',
```

(continues on next page)

(continued from previous page)

```
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerName = async () => {
  const { platform } = client;

  const identity = await platform.identities.get('an identity ID goes here');
  const nameRegistration = await platform.names.register(
    '<identity name goes here>.dash',
    { dashUniqueIdentityId: identity.getId() },
    identity,
  );

  return nameRegistration;
};

registerName()
  .then((d) => console.log('Name registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// Register Alias for Identity
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with testnet funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerAlias = async () => {
  const platform = client.platform;
  const identity = await platform.identities.get('an identity ID goes here');
  const aliasRegistration = await platform.names.register(
    '<identity alias goes here>.dash',
    { dashAliasIdentityId: identity.getId() },
    identity,
  );
```

(continues on next page)

(continued from previous page)

```
  return aliasRegistration;
};

registerAlias()
  .then((d) => console.log('Alias registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

What's Happening

After initializing the Client, we fetch the Identity we'll be associating with a name. This is an asynchronous method so we use *await* to pause until the request is complete. Next, we call `platform.names.register` and pass in the name we want to register, the type of identity record to create, and the identity we just fetched. We wait for the result, and output it to the console.

> Wallet Operations
>
> The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.
>
> A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.10.18 Retrieve a name

In this tutorial we will retrieve the name created in the *Register a Name for an Identity tutorial*. Additional details regarding identities can be found in the *Identity description*.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)

Code

JAVASCRIPT

```
//  Resolve by Name
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

const retrieveName = async () => {
  // Retrieve by full name (e.g., myname.dash)
  return client.platform.names.resolve('<identity name>.dash');
};

retrieveName()
  .then((d) => console.log('Name retrieved:\n', d.toJSON()))
```

(continues on next page)

(continued from previous page)

```
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// Revolve by Record
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

const retrieveNameByRecord = async () => {
  // Retrieve by a name's identity ID
  return client.platform.names.resolveByRecord(
    'dashUniqueIdentityId',
    '<identity id>',
  );
};

retrieveNameByRecord()
  .then((d) => console.log('Name retrieved:\n', d[0].toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// Search for Name
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

const retrieveNameBySearch = async () => {
  // Search for names (e.g. `user*`)
  return client.platform.names.search('user', 'dash');
};

retrieveNameBySearch()
  .then((d) => {
    for (const name of d) {
      console.log('Name retrieved:\n', name.toJSON());
    }
  })
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Example Name

The following example response shows a retrieved name (`user-9999.dash`):


```
{
  "$protocolVersion": 0,
  "$id": "4veLBZPHDkaCPF9LfZ8fX3JZiS5q5iUVGhdBbaa9ga5E",
  "$type": "domain",
  "$dataContractId": "566vcJkmebVCAb2Dkj2yVMSgGFcsshupnQqtsz1RFbcy",
  "$ownerId": "HBNMY5QWuBVKNFLhgBTC1VmpEnscrmqKPMXpnYSHwhfn",
  "$revision": 1,
  "label": "user-9999",
  "records": {
    "dashUniqueIdentityId": "HBNMY5QWuBVKNFLhgBTC1VmpEnscrmqKPMXpnYSHwhfn"
  },
  "preorderSalt": "BzQi567XVqc8wYiVHS887sJtL6MDbxLHNnp+UpTFSB0",
  "subdomainRules": { "allowSubdomains": false },
  "normalizedLabel": "user-9999",
  "normalizedParentDomainName": "dash"
}
```


What's Happening

After we initialize the Client, we request a name. The *code examples* demonstrate the three ways to request a name:

1. Resolve by name. The `platform.names.resolve` method takes a single argument: a fully-qualified name (e.g., `user-9999.dash`).
2. Resolve by record. The `platform.names.resolveByRecord` method takes two arguments: the record type (e.g., `dashUniqueIdentityId`) and the record value to resolve.
3. Search. The `platform.names.search` method takes two arguments: the leading characters of the name to search for and the domain to search (e.g., `dash` for names in the `*.dash` domain). The search will return names that begin the with string provided in the first parameter.

After the name is retrieved, it is displayed on the console.

1.11 Contracts and documents

The following tutorials cover working with data contracts as well as storing and updating related data using the documents they define.

- *Register a Data Contract*
- *Retrieve a Data Contract*
- *Update a Data Contract*
- *Submit Documents*
- *Retrieve Documents*
- *Update Documents*
- *Delete Documents*

Tutorial code

You can clone a repository containing the code for all tutorials from GitHub or download it as a zip file.

1.11.1 Register a data contract

In this tutorial we will register a data contract.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- A Dash Platform Identity: *Tutorial: Register an Identity*

1.11.2 Code

Defining contract documents

As described in the *data contract explanation*, data contracts must include one or more developer-defined *documents*.

The most basic example below (tab 1) demonstrates a data contract containing a single document type (`note`) which has a single string property (`message`).

The second tab shows the same data contract with an index defined on the `$ownerId` field. This would allow querying for documents owned by a specific identity using a *where clause*.

The third tab shows a data contract using the JSON-Schema $ref feature that enables reuse of defined objects. Note that the $ref keyword has been temporarily disabled since Platform v0.22.

The fourth tab shows a data contract requiring the optional `$createdAt` and `$updatedAt` *base fields*. Using these fields enables retrieving timestamps that indicate when a document was created or modified.

Since Platform v0.23, an index can only use the ascending order (`asc`). Future updates will remove this restriction.

JSON

```
// 1. Minimal contract
{
  "note": {
    "type": "object",
    "properties": {
      "message": {
        "type": "string"
      }
    },
    "additionalProperties": false
  }
}
```

JSON

```
//  2. Indexed
{
  "note": {
    "type": "object",
    "indices": [
      {
        "name": "ownerId",
        "properties": [{ "$ownerId": "asc" }], "unique": false }
    ],
    "properties": {
      "message": {
        "type": "string"
      }
    },
    "additionalProperties": false
  }
}

/*
An identity's documents are accessible via a query including a where clause like:
{
  where: [['$ownerId', '==', 'an identity id']],
}
*/
```

JSON

```
//  3. References ($ref)
// NOTE: The `$ref` keyword is temporarily disabled for Platform v0.22.
{
  "customer": {
    "type": "object",
    "properties": {
      "name": { "type": "string" },
      "billing_address": { "$ref": "#/$defs/address" },
      "shipping_address": { "$ref": "#/$defs/address" }
    },
    "additionalProperties": false
  }
}

/*
The contract document defined above is dependent on the following object
being added to the contract via the contracts `.setDefinitions` method:

{
  address: {
    type: "object",
    properties: {
```

(continues on next page)

(continued from previous page)

```
      street_address: { type: "string" },
      city:           { type: "string" },
      state:          { type: "string" }
    },
    required: ["street_address", "city", "state"],
    additionalProperties: false
  }
}
*/
```

JSON

```
//  4. Timestamps
{
  "note": {
    "type": "object",
    "properties": {
      "message": {
        "type": "string"
      }
    },
    "required": ["$createdAt", "$updatedAt"],
    "additionalProperties": false
  }
}

/*
If $createdAt and/or $updatedAt are added to the list of required properties
for a document, all documents of that type will store a timestamp indicating
when the document was created or modified.

This information will be returned when the document is retrieved.
*/
```

JSON

```
// 5. Binary data
{
 "block": {
   "type": "object",
    "properties": {
      "hash": {
        "type": "array",
        "byteArray": true,
        "maxItems": 64,
        "description": "Store block hashes"
      }
    },
    "additionalProperties": false
```

(continues on next page)

(continued from previous page)

```
  }
}

/*
Setting `"byteArray": true` indicates that the provided data will be an
array of bytes (e.g. a NodeJS Buffer).
*/
```

Please refer to the *data contract reference page* for more comprehensive details related to contracts and documents.

Registering the data contract

The following examples demonstrate the details of creating contracts using the features *described above*:

JAVASCRIPT

```
// 1. Minimal contract
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerContract = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  const contractDocuments = {
    note: {
      type: 'object',
      properties: {
        message: {
          type: 'string',
        },
      },
      additionalProperties: false,
    },
  };

  const contract = await platform.contracts.create(contractDocuments, identity);
  console.dir({ contract: contract.toJSON() });
```

(continues on next page)

(continued from previous page)

```
  // Make sure contract passes validation checks
  const validationResult = await platform.dpp.dataContract.validate(contract);

  if (validationResult.isValid()) {
    console.log('Validation passed, broadcasting contract..');
    // Sign and submit the data contract
    return platform.contracts.publish(contract, identity);
  }
  console.error(validationResult); // An array of detailed validation errors
  throw validationResult.errors[0];
};

registerContract()
  .then((d) => console.log('Contract registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// 2. Indexed
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerContract = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  const contractDocuments = {
    note: {
      type: 'object',
      indices: [{
        name: 'ownerId',
        properties: [{ $ownerId: 'asc' }],
        unique: false,
      }],
      properties: {
        message: {
          type: 'string',
        },
```

(continues on next page)

(continued from previous page)

```
      },
      additionalProperties: false,
    },
  };

  const contract = await platform.contracts.create(contractDocuments, identity);
  console.dir({ contract: contract.toJSON() });

  // Make sure contract passes validation checks
  const validationResult = await platform.dpp.dataContract.validate(contract);

  if (validationResult.isValid()) {
    console.log('Validation passed, broadcasting contract..');
    // Sign and submit the data contract
    return platform.contracts.publish(contract, identity);
  }
  console.error(validationResult); // An array of detailed validation errors
  throw validationResult.errors[0];
};

registerContract()
  .then((d) => console.log('Contract registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// 3. References ($ref)
// NOTE: The `$ref` keyword is temporarily disabled for Platform v0.22.
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerContract = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  // Define a reusable object
  const definitions = {
    address: {
      type: 'object',
```

(continues on next page)

(continued from previous page)

```
      properties: {
        street_address: { type: 'string' },
        city: { type: 'string' },
        state: { type: 'string' },
      },
      required: ['street_address', 'city', 'state'],
      additionalProperties: false,
    },
  };

  // Create a document with properties using a definition via $ref
  const contractDocuments = {
    customer: {
      type: 'object',
      properties: {
        name: { type: 'string' },
        billing_address: { $ref: '#/$defs/address' },
        shipping_address: { $ref: '#/$defs/address' },
      },
      additionalProperties: false,
    },
  };

  const contract = await platform.contracts.create(contractDocuments, identity);

  // Add reusable definitions referred to by "$ref" to contract
  contract.setDefinitions(definitions);
  console.dir({ contract: contract.toJSON() });

  // Make sure contract passes validation checks
  const validationResult = await platform.dpp.dataContract.validate(contract);

  if (validationResult.isValid()) {
    console.log('Validation passed, broadcasting contract..');
    // Sign and submit the data contract
    return platform.contracts.publish(contract, identity);
  }
  console.error(validationResult); // An array of detailed validation errors
  throw validationResult.errors[0];
};

registerContract()
  .then((d) => console.log('Contract registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// 4. Timestamps
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerContract = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  const contractDocuments = {
    note: {
      type: 'object',
      properties: {
        message: {
          type: 'string',
        },
      },
      required: ['$createdAt', '$updatedAt'],
      additionalProperties: false,
    },
  };

  const contract = await platform.contracts.create(contractDocuments, identity);
  console.dir({ contract: contract.toJSON() });

  // Make sure contract passes validation checks
  const validationResult = await platform.dpp.dataContract.validate(contract);

  if (validationResult.isValid()) {
    console.log('Validation passed, broadcasting contract..');
    // Sign and submit the data contract
    return platform.contracts.publish(contract, identity);
  }
  console.error(validationResult); // An array of detailed validation errors
  throw validationResult.errors[0];
};

registerContract()
  .then((d) => console.log('Contract registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

JAVASCRIPT

```
// 5. Binary data
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
const client = new Dash.Client(clientOpts);

const registerContract = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  const contractDocuments = {
    block: {
      type: 'object',
      properties: {
        hash: {
          type: 'array',
          byteArray: true,
          maxItems: 64,
          description: 'Store block hashes',
        },
      },
      additionalProperties: false,
    },
  };

  const contract = await platform.contracts.create(contractDocuments, identity);
  console.dir({ contract: contract.toJSON() }, { depth: 5 });

  // Make sure contract passes validation checks
  const validationResult = await platform.dpp.dataContract.validate(contract);

  if (validationResult.isValid()) {
    console.log('Validation passed, broadcasting contract..');
    // Sign and submit the data contract
    return platform.contracts.publish(contract, identity);
  }
  console.error(validationResult); // An array of detailed validation errors
  throw validationResult.errors[0];
};

registerContract()
  .then((d) => console.log('Contract registered:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
```

(continues on next page)

(continued from previous page)

```
  .finally(() => client.disconnect());
```

Make a note of the returned data contract `$id` as it will be used used in subsequent tutorials throughout the documentation.

1.11.3 What's Happening

After we initialize the Client, we create an object defining the documents this data contract requires (e.g. a `note` document in the example). The `platform.contracts.create` method takes two arguments: a contract definitions JSON-schema object and an identity. The contract definitions object consists of the document types being created (e.g. `note`). It defines the properties and any indices.

Once the data contract has been created, we still need to submit it to DAPI. The `platform.contracts.publish` method takes a data contract and an identity parameter. Internally, it creates a State Transition containing the previously created contract, signs the state transition, and submits the signed state transition to DAPI. A response will only be returned if an error is encountered.

Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.11.4 Retrieve a data contract

In this tutorial we will retrieve the data contract created in the *Register a Data Contract tutorial*.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A Dash Platform Contract ID: *Tutorial: Register a Data Contract*

1.11.5 Code

Retrieving a data contract

```
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

const retrieveContract = async () => {
  const contractId = '3iaEhdyAVbmSjd59CT6SCrqPjfAfMdPTc8ksydgqSaWE';
  return client.platform.contracts.get(contractId);
};
```

(continues on next page)

(continued from previous page)

```
retrieveContract()
  .then((d) => console.dir(d.toJSON(), { depth: 5 }))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Updating the client app list

In many cases it may be desirable to work with a newly retrieved data contract using the <contract name>.<contract document> syntax (e.g. dpns.domain). Data contracts that were created after the client was initialized or not included in the initial client options can be added via client.getApps().set(...).

```
const Dash = require('dash');
const { PlatformProtocol: { Identifier } } = Dash;

const myContractId = 'a contract ID';
const client = new Dash.Client();

client.platform.contracts.get(myContractId)
  .then((myContract) => {
    client.getApps().set('myNewContract', {
      contractId: Identifier.from(myContractId),
      contract: myContract,
    });
  });
```

1.11.6 Example Data Contract

The following example response shows a retrieved contract:

```
{
  "protocolVersion":1,
  "$id":"G1FVmxxrnbT6CiQU7w2xgY9oMMqkkZb7vS6fkeRrSTXG",
  "$schema":"https://schema.dash.org/dpp-0-4-0/meta/data-contract",
  "version":2,
  "ownerId":"8uFQj2ptknrcwykhQbTzQatoQUyxn4VJQn1J25fxeDvk",
  "documents":{
    "note":{
      "type":"object",
      "properties":{
        "author":{
          "type":"string"
        },
        "message":{
          "type":"string"
        }
      },
      "additionalProperties":false
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

Please refer to the *data contract reference page* for more comprehensive details related to contracts and documents.

1.11.7 What's Happening

After we initialize the Client, we request a contract. The `platform.contracts.get` method takes a single argument: a contract ID. After the contract is retrieved, it is displayed on the console.

The second code example shows how the contract could be assigned a name to make it easily accessible without initializing an additional client.

1.11.8 Update a data contract

Since Dash Platform v0.22, it is possible to update existing data contracts in certain backwards-compatible ways. This includes:

- Adding new documents
- Adding new optional properties to existing documents
- Adding *non-unique* indices for properties added in the update.

In this tutorial we will update an existing data contract.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- A Dash Platform Identity: *Tutorial: Register an Identity*
- A Dash Platform Contract ID: *Tutorial: Register a Data Contract*

1.11.9 Code

The following example demonstrates updating an existing contract to add a new property to an existing document:

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
```

(continues on next page)

(continued from previous page)

```
  },
};
const client = new Dash.Client(clientOpts);

const updateContract = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  const existingDataContract = await platform.contracts.get('a contract ID goes here');
  const documents = existingDataContract.getDocuments();

  documents.note.properties.author = {
    type: 'string',
  };

  existingDataContract.setDocuments(documents);

  // Make sure contract passes validation checks
  const validationResult = await platform.dpp.dataContract.validate(
    existingDataContract,
  );

  if (validationResult.isValid()) {
    console.log('Validation passed, broadcasting contract..');
    // Sign and submit the data contract
    return platform.contracts.update(existingDataContract, identity);
  }
  console.error(validationResult); // An array of detailed validation errors
  throw validationResult.errors[0];
};

updateContract()
  .then((d) => console.log('Contract updated:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

> Please refer to the *data contract reference page* for more comprehensive details related to contracts and documents.

1.11.10 What's Happening

After we initialize the Client, we retrieve an existing contract owned by our identity. We then get the contract's documents and modify a document (adding an `author` property to the `note` document in the example).The `setDocuments` method takes one argument: the object containing the updated document types.

Once the data contract has been updated, we still need to submit it to DAPI. The `platform.contracts.update` method takes a data contract and an identity parameter. Internally, it creates a State Transition containing the updated contract, signs the state transition, and submits the signed state transition to DAPI. A response will only be returned if an error is encountered.

> Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.11.11 Submit documents

In this tutorial we will submit some data to an application on Dash Platform. Data is stored in the form of *documents* which are encapsulated in a *state transition* before being submitted to DAPI.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- A Dash Platform Identity: *Tutorial: Register an Identity*
- A Dash Platform Contract ID: *Tutorial: Register a Data Contract*

1.11.12 Code

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
  apps: {
    tutorialContract: {
      contractId: '3iaEhdyAVbmSjd59CT6SCrqPjfAfMdPTc8ksydgqSaWE',
    },
  },
};
const client = new Dash.Client(clientOpts);

const submitNoteDocument = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');

  const docProperties = {
    message: `Tutorial Test @ ${new Date().toUTCString()}`,
  };

  // Create the note document
  const noteDocument = await platform.documents.create(
```

(continues on next page)

(continued from previous page)

```
    'tutorialContract.note',
    identity,
    docProperties,
  );

  const documentBatch = {
    create: [noteDocument], // Document(s) to create
    replace: [], // Document(s) to update
    delete: [], // Document(s) to delete
  };
  // Sign and submit the document(s)
  return platform.documents.broadcast(documentBatch, identity);
};

submitNoteDocument()
  .then((d) => console.log(d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Initializing the Client with a contract identity

The example above shows how access to contract documents via `<contract name>.<contract document>` syntax (e.g. `tutorialContract.note`) can be enabled by passing a contract identity to the constructor. Please refer to the Dash SDK documentation for details.

1.11.13 What's happening

After we initialize the Client, we create a document that matches the structure defined by the data contract of the application being referenced (e.g. a `note` document for the contract registered in the *data contract tutorial*). The `platform.documents.create` method takes three arguments: a document locator, an identity, and the document data. The document locator consists of an application name (e.g. `tutorialContract`) and the document type being created (e.g. `note`). The document data should contain values for each of the properties defined for it in the data contract (e.g. `message` for the tutorial contract's note).

Once the document has been created, we still need to submit it to *DAPI*. Documents are submitted in batches that may contain multiple documents to be created, replaced, or deleted. In this example, a single document is being created. The `documentBatch` object defines the action to be completed for the document (the empty action arrays - `replace` and `delete` in this example - may be excluded and are shown for reference only here).

The `platform.documents.broadcast` method then takes the document batch and an identity parameter. Internally, it creates a *State Transition* containing the previously created document, signs the state transition, and submits the signed state transition to DAPI.

Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.11.14 Retrieve documents

In this tutorial we will retrieve some of the current data from a data contract. Data is stored in the form of documents as described in the Dash Platform Protocol *Document explanation*.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A Dash Platform Contract ID: *Tutorial: Register a Data Contract*

1.11.15 Code

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  apps: {
    tutorialContract: {
      contractId: '3iaEhdyAVbmSjd59CT6SCrqPjfAfMdPTc8ksydgqSaWE',
    },
  },
};
const client = new Dash.Client(clientOpts);

const getDocuments = async () => {
  return client.platform.documents.get('tutorialContract.note', {
    limit: 2, // Only retrieve 2 document
  });
};

getDocuments()
  .then((d) => {
    for (const n of d) {
      console.log('Document:\n', n.toJSON());
    }
  })
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Initializing the Client with a contract identity

The example above shows how access to contract documents via `<contract name>.<contract document>` syntax (e.g. `tutorialContract.note`) can be enabled by passing a contract identity to the constructor. Please refer to the Dash SDK documentation for details.

Queries

The example code uses a very basic query to return only one result. More extensive querying capabilities are covered in the *query syntax reference*.

1.11.16 Example Document

The following examples show the structure of a `note` document (from the data contract registered in the tutorial) returned from the SDK when retrieved with various methods.

The values returned by `.toJSON()` include the base document properties (prefixed with `$`) present in all documents along with the data contract defined properties.

> Note: When using `.toJSON()`, binary data is displayed as a base64 string (since JSON is a text-based format).

The values returned by `.getData()` (and also shown in the console.dir() `data` property) represent *only* the properties defined in the `note` document described by the *tutorial data contract*.

JSON

```
//  .toJSON()
{
  "$protocolVersion": 0,
  "$id": "6LpCQhkXYV2vqkv1UWByew4xQ6BaxxnGkhfMZsN3SV9u",
  "$type": "note",
  "$dataContractId": "3iaEhdyAVbmSjd59CT6SCrqPjfAfMdPTc8ksydgqSaWE",
  "$ownerId": "CEPMcuBgAWeaCXiP2gJJaStANRHW6b158UPvL1C8zw2W",
  "$revision": 1,
  "message": "Tutorial CI Test @ Fri, 23 Jul 2021 13:12:13 GMT"
}
```

JSON

```
// .getData()
{
  "Tutorial CI Test @ Fri, 23 Jul 2021 13:12:13 GMT"
}
```

TEXT

```
# .data.message
Tutorial CI Test @ Fri, 23 Jul 2021 13:12:13 GMT
```

JSON

```
// console.dir(document)
Document {
  dataContract: DataContract {
    protocolVersion: 0,
    id: Identifier(32) [Uint8Array] [
       40,  93, 196, 112,  38, 188,  51, 122,
      149,  59,  21,  39, 147, 119,  87,  53,
      236,  60,  97,  42,  31,  82, 135, 120,
       68, 188,  55, 153, 226, 198, 181, 139
    ],
    ownerId: Identifier(32) [Uint8Array] [
      166, 222,  98,  87, 193,  19,  82,  37,
       50, 118, 210,  64, 103, 122,  28, 155,
      168,  21, 198, 134, 142, 151, 153, 136,
       46,  64, 223,  74, 215, 153, 158, 167
    ],
    schema: 'https://schema.dash.org/dpp-0-4-0/meta/data-contract',
    documents: { note: [Object] },
    '$defs': undefined,
    binaryProperties: { note: {} },
    metadata: Metadata { blockHeight: 526, coreChainLockedHeight: 542795 }
  },
  entropy: undefined,
  protocolVersion: 0,
  id: Identifier(32) [Uint8Array] [
     79,  93, 213, 226,  76,  79, 205, 191,
    165, 190,  68,  28,   8,  83,  61, 226,
    222, 248,  48, 235, 147, 110, 181, 229,
      7,  66,  65, 230, 100, 194, 192, 156
  ],
  type: 'note',
  dataContractId: Identifier(32) [Uint8Array] [
     40,  93, 196, 112,  38, 188,  51, 122,
    149,  59,  21,  39, 147, 119,  87,  53,
    236,  60,  97,  42,  31,  82, 135, 120,
     68, 188,  55, 153, 226, 198, 181, 139
  ],
  ownerId: Identifier(32) [Uint8Array] [
    166, 222,  98,  87, 193,  19,  82,  37,
     50, 118, 210,  64, 103, 122,  28, 155,
    168,  21, 198, 134, 142, 151, 153, 136,
     46,  64, 223,  74, 215, 153, 158, 167
  ],
  revision: 1,
  data: { message: 'Tutorial CI Test @ Fri, 23 Jul 2021 13:12:13 GMT' },
  metadata: Metadata { blockHeight: 526, coreChainLockedHeight: 542795 }
}
```

1.11.17 What's happening

After we initialize the Client, we request some documents. The `client.platform.documents.get` method takes two arguments: a record locator and a query object. The records locator consists of an app name (e.g. `tutorialContract`) and the top-level document type requested, (e.g. `note`).

> DPNS Contract
>
> Note: Access to the DPNS contract is built into the Dash SDK. DPNS documents may be accessed via the `dpns` app name (e.g. `dpns.domain`).

If you need more than the first 100 documents, you'll have to make additional requests with `startAt` incremented by 100 each time. In the future, the Dash SDK may return documents with paging information to make this easier and reveal how many documents are returned in total.

1.11.18 Update documents

In this tutorial we will update existing data on Dash Platform. Data is stored in the form of *documents* which are encapsulated in a *state transition* before being submitted to DAPI.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- Access to a previously created document (e.g., one created using the *Submit Documents tutorial*)

1.11.19 Code

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
  apps: {
    tutorialContract: {
      contractId: '3iaEhdyAVbmSjd59CT6SCrqPjfAfMdPTc8ksydgqSaWE',
    },
  },
};
const client = new Dash.Client(clientOpts);

const updateNoteDocument = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');
  const documentId = 'an existing document ID goes here';
```

(continues on next page)

(continued from previous page)

```
  // Retrieve the existing document
  const [document] = await client.platform.documents.get(
    'tutorialContract.note',
    { where: [['$id', '==', documentId]] },
  );

  // Update document
  document.set('message', `Updated document @ ${new Date().toUTCString()}`);

  // Sign and submit the document replace transition
  return platform.documents.broadcast({ replace: [document] }, identity);
};

updateNoteDocument()
  .then((d) => console.log('Document updated:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Initializing the Client with a contract identity

The example above shows how access to contract documents via `<contract name>.<contract document>` syntax (e.g. `tutorialContract.note`) can be enabled by passing a contract identity to the constructor. Please refer to the Dash SDK documentation for details.

1.11.20 What's happening

After we initialize the Client, we retrieve the document to be updated via `platform.documents.get` using its `id`. Once the document has been retrieved, we must submit it to *DAPI* with the desired data updates. Documents are submitted in batches that may contain multiple documents to be created, replaced, or deleted. In this example, a single document is being updated.

The `platform.documents.broadcast` method then takes the document batch (e.g. `{replace: [noteDocument]}`) and an identity parameter. Internally, it creates a *State Transition* containing the previously created document, signs the state transition, and submits the signed state transition to DAPI.

Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.11.21 Delete documents

In this tutorial we will update delete data from Dash Platform. Data is stored in the form of *documents* which are encapsulated in a *state transition* before being submitted to DAPI.

Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)
- A wallet mnemonic with some funds in it: *Tutorial: Create and Fund a Wallet*
- Access to a previously created document (e.g., one created using the *Submit Documents tutorial*)

1.11.22 Code

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'a Dash wallet mnemonic with funds goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
  apps: {
    tutorialContract: {
      contractId: '3iaEhdyAVbmSjd59CT6SCrqPjfAfMdPTc8ksydgqSaWE',
    },
  },
};
const client = new Dash.Client(clientOpts);

const deleteNoteDocument = async () => {
  const { platform } = client;
  const identity = await platform.identities.get('an identity ID goes here');
  const documentId = 'an existing document ID goes here';

  // Retrieve the existing document
  const [document] = await client.platform.documents.get(
    'tutorialContract.note',
    { where: [['$id', '==', documentId]] },
  );

  // Sign and submit the document delete transition
  return platform.documents.broadcast({ delete: [document] }, identity);
};

deleteNoteDocument()
  .then((d) => console.log('Document deleted:\n', d.toJSON()))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

Initializing the Client with a contract identity

The example above shows how access to contract documents via `<contract name>.<contract document>` syntax (e.g. `tutorialContract.note`) can be enabled by passing a contract identity to the constructor. Please refer to the Dash SDK documentation for details.

1.11.23 What's happening

After we initialize the Client, we retrieve the document to be deleted via `platform.documents.get` using its `id`.

Once the document has been retrieved, we must submit it to *DAPI*. Documents are submitted in batches that may contain multiple documents to be created, replaced, or deleted. In this example, a single document is being deleted.

The `platform.documents.broadcast` method takes the document batch (e.g. `{delete: [documents[0]]}`) and an identity parameter. Internally, it creates a *State Transition* containing the previously created document, signs the state transition, and submits the signed state transition to DAPI.

Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

1.12 Send funds

Once you have a wallet and some funds (*tutorial*), another common task is sending Dash to an address. (Sending Dash to a contact or a DPNS identity requires the Dashpay app, which has not been registered yet.)

1.13 Code

Wallet Operations

The JavaScript SDK does not cache wallet information. It re-syncs the entire Core chain for some wallet operations (e.g. `client.getWalletAccount()`) which can result in wait times of 5+ minutes.

A future release will add caching so that access is much faster after the initial sync. For now, the `skipSynchronizationBeforeHeight` option can be used to sync the wallet starting at a certain block height.

```
const Dash = require('dash');

const clientOpts = {
  network: 'testnet',
  wallet: {
    mnemonic: 'your wallet mnemonic goes here',
    unsafeOptions: {
      skipSynchronizationBeforeHeight: 650000, // only sync from early-2022
    },
  },
};
```

(continues on next page)

(continued from previous page)

```
const client = new Dash.Client(clientOpts);

const sendFunds = async () => {
  const account = await client.getWalletAccount();

  const transaction = account.createTransaction({
    recipient: 'yP8A3cbdxRtLRduy5mXDsBnJtMzHWs6ZXr', // Testnet2 faucet
    satoshis: 100000000, // 1 Dash
  });
  return account.broadcastTransaction(transaction);
};

sendFunds()
  .then((d) => console.log('Transaction broadcast!\nTransaction ID:', d))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());

// Handle wallet async errors
client.on('error', (error, context) => {
  console.error(`Client error: ${error.name}`);
  console.error(context);
});
```

1.14 What's Happening

After initializing the Client, we build a new transaction with `account.createTransaction`. It requires a recipient and an amount in satoshis (often called "duffs" in Dash). 100 million satoshis equals one Dash. We pass the transaction to `account.broadcastTransaction` and wait for it to return. Then we output the result, which is a transaction ID. After that we disconnect from the Client so node can exit.

1.15 Use DAPI client methods

In addition to the SDK methods for interacting with identities, names, contracts, and documents, the SDK also provides direct access to DAPI client methods.

1.15.1 Prerequisites

- *General prerequisites* (Node.js / Dash SDK installed)

1.16 Code

The following example demonstrates several of the Core DAPI client methods. DAPI client also has several Platform methods accessible via `getDAPIClient().platform.*`. The methods can be found here in the js-dapi-client repository.

```
const Dash = require('dash');

const client = new Dash.Client({ network: 'testnet' });

async function dapiClientMethods() {
  console.log(await client.getDAPIClient().core.getBlockHash(1));
  console.log(await client.getDAPIClient().core.getBestBlockHash());
  console.log(await client.getDAPIClient().core.getBlockByHeight(1));

  return client.getDAPIClient().core.getStatus();
}

dapiClientMethods()
  .then((d) => console.log('Core status:\n', d))
  .catch((e) => console.error('Something went wrong:\n', e))
  .finally(() => client.disconnect());
```

> Examples using DAPI client to access many of the DAPI endpoints can be found in the *DAPI Endpoint Reference section*.

1.17 Set up a node

Since Dash Platform is accessible through DAPI, running a node is typically unnecessary. The information provided in this section is for advanced users interested in running their own network for development or participating in testing the project by running a testnet node.

1.17.1 Dash masternode

The purpose of this tutorial is to walk through the steps necessary to set up a masternode with Dash Platform services.

Prerequisites

- Docker (v20.10.0+) and docker-compose (v1.25.0+) installed
- An installation of NodeJS (v16, NPM v8.0+)

The following is not necessary for setting up a local network for development, but is helpful if setting up a testnet masternode:

- Access to a Linux system configured with a non-root user (guide)

More comprehensive details of using the dashmate tool can be found in the dashmate README.

Use NPM to install dashmate globally in your system:

```
npm install -g dashmate
```

Local Network

Dashmate can be used to create a local network on a single computer. This network contains multiple nodes to mimic conditions and features found in testnet/mainnet settings.

Dashmate local networks use the *regtest network type* so layer 1 blocks can be easily mined as needed.

Setup

Run the following command to start the setup wizard, then accept the default values at each step to create a local network:

```
dashmate setup local
```

Example (partial) output of the setup wizard showing important information:

```
  ✓ Initialize SDK
    › HD private key:␣
↪tprv8ZgxMBicQKsPfLTCjh8vdHkDHYM369tUeQ4aqpV9GzUfQyBKutfstB1sDfQyLERACTEYy5Qjph42gBiqqnqYmXJZZqRc4PQss
  ✓ Register DPNS identity
    › DPNS identity: 6whgUd1LzwzU4ob7K8FGCLV765K7dp2JbEmVgdTQEFxD
  ✓ Register DPNS contract
    › DPNS contract ID: EpCvWuoh3JcFetFY83HdwuzRUvwxF2hc3mU19MtBg2kK
  ✓ Obtain DPNS contract commit block height
    › DPNS contract block height: 5
  ✓ Register top level domain "dash"
  ✓ Register identity for Dashpay
    › Dashpay's owner identity: 2T7kLcbJzQrLhBV6BferW42Jimb3BJ5zAAore42mfNyE
  ✓ Register Dashpay Contract
    › Dashpay contract ID: EAv8ePXREdJ719ntcRiKuEYxv9XooMwL1mJmPHMGuW9r
  ✓ Obtain Dashpay contract commit block height
    › Dashpay contract block height: 15
  ✓ Register Feature Flags identity
    › Feature Flags identity: 8BsvV4RCbW7srWj81kgjJCykRBF2rzyigys8XkBchY96
```

(continues on next page)

(continued from previous page)

```
  ✓ Register Feature Flags contract
    › Feature Flags contract ID: JDrDAGVqTWsM9k7KGBsSjcyC11Vd2UdPxPoPf4NzyyrP
  ✓ Obtain Feature Flags contract commit block height
    › Feature Flags contract block height: 20
```

Make a note of the key and identity information displayed during setup as they may be required in the future.

Operation

Once the setup completes, start/stop/restart the network via the following commands:

```
dashmate group start
dashmate group stop
dashmate group restart
```

The status of the network's nodes can be check via the group status command:

```
dashmate group status
```

Mining Dash

During development it may be necessary to obtain Dash to create and topup *identities*. This can be done using the dashmate `wallet:mint` command. First obtain an address to fund via the *Create and Fund a Wallet* tutorial and then mine Dash to it as shown below:

SHELL

```
# Mine to provided address

# Stop the devnet first
dashmate group stop

# Mine 10 Dash to a provided address
dashmate wallet mint 10 --address=<your address> --config=local_seed

# Restart the devnet
dashmate group start
```

SHELL

```
# Mine to new address

# Stop the devnet first
dashmate group:stop

# Mine 10 Dash to a random address/key
# The address and private key will be displayed
dashmate wallet:mint 10 --config=local_seed

# Restart the devnet
dashmate group:start
```

Example output of `dashmate wallet mint 10 --address=yYqfdpePzn2kWtMxr9nz22HBFM7WBRmAqG --config=local_seed`:

```
✓ Generate 10 dash to address
  ✓ Start Core
  ↓ Use specified address yYqfdpePzn2kWtMxr9nz22HBFM7WBRmAqG [SKIPPED]
  ✓ Generate 10 dash to address yYqfdpePzn2kWtMxr9nz22HBFM7WBRmAqG
    › Generated 172.59038279 dash
  ✓ Wait for balance to confirm
  ✓ Stop Core
```

Using the network

Once the address is funded, you can begin creating identities, data contracts, etc. and experimenting with Dash Platform. The *other tutorials* in this section will help you get started.

To make the Dash SDK connect to your local network, set the `network` option to `'local'`:

```
const clientOpts = {
  network: 'local',
  ...
};

const client = new Dash.Client(clientOpts);
```

Testnet Masternode Setup

> Advanced Topic
>
> Running a masternode requires familiarity with Dash Platform services. Improper configuration may impact testing so please exercise caution if running a masternode.

To setup a testnet masternode, please refer to the comprehensive documentation of the process as described here. The following video also details how to complete the process.

> Full Platform Node
>
> A full node that with all Platform services can be started by simply running the setup command with the node type setup parameter set to `fullnode` and then starting the node.

```
dashmate setup testnet fullnode
dashmate start
```

Remote Development Network

Connecting to a remote development network

In order to connect to a remote *devnet* (e.g. one run by Dash Core Group), please use one of the methods described in the *Connect to a Devnet* section.

For development we recommend using either a local network created via dashmate as *described above* or using Testnet. While configuring a remote development network is possible using the Dash network deployment tool, it is beyond the scope of this documentation. For details regarding this tool, please refer to the GitHub repository.

1.17.2 Dash Core full node

Since Dash Platform is fully accessible via DAPI, running a full node is unnecessary and generally provides no particular benefit. Regardless, the steps below provide the necessary information for advanced users to connect.

Config File

The config file shown below may be used to connect a Dash Core node to Testnet. Testnet currently operates using Dash Core v19.3.0.

```
# dash-testnet.conf
testnet=1

# Hard-coded first node
addnode=seed-1.testnet.networks.dash.org:19999
```

Starting Dash Core

To start Dash Core and connect to Testnet, simply run dashd or dash-qt with the `conf` parameter set to the configuration file created above: `<path to binary> -conf=<path to configuration file>`

```
dashd -conf=/home/dash/.dashcore/dash-testnet.conf
```

1.18 Decentralized API (DAPI)

1.18.1 Overview

Historically, nodes in most cryptocurrency networks communicated with each other, and the outside world, according to a peer-to-peer (P2P) protocol. The use of P2P protocols presented some downsides for developers, namely, network resources were difficult to access without specialized knowledge or trusted third-party services.

To overcome these obstacles, the Dash decentralized API (DAPI) uses Dash's robust masternode infrastructure to provide an API for accessing the network. DAPI supports both layer 1 (Core blockchain) and layer 2 (Dash Platform) functionality so all developers can interact with Dash via a single interface.

Fig. 1: DAPI Overview

1.18.2 Security

DAPI protects connections by using TLS to encrypt communication between clients and the masternodes. This encryption safeguards transmitted data from unauthorized access, interception, or tampering. *Platform gRPC endpoints* provide an additional level of security by optionally returning cryptographic proofs. Successful proof verification guarantees that the server responded without modifying the requested data.

1.18.3 Endpoint Overview

DAPI currently provides 2 types of endpoints: JSON-RPC and gRPC. The JSON-RPC endpoints expose some layer 1 information while the gRPC endpoints support layer 2 as well as streaming of events related to blocks and transactions/transitions. For a list of all endpoints and usage details, please see the *DAPI endpoint reference section*.

1.19 Platform Protocol (DPP)

1.19.1 Overview

To ensure the consistency and integrity of data stored on Layer 2, all data is governed by the Dash Platform Protocol (DPP). Dash Platform Protocol describes serialization and validation rules for the platform's 3 core data structures: data contracts, documents, and state transitions. Each of these structures are briefly described below.

1.19.2 Structure Descriptions

Data Contract

A data contract is a database schema that a developer needs to register with the platform in order to start using any decentralized storage functionality. Data contracts are described using the JSON Schema language and must follow some basic rules as described in the platform protocol repository. Contracts are serialized to binary form using CBOR.

> Contract updates
>
> Dash's data contracts support backwards-compatible modifications after their initial deployment unlike many smart contract based systems. This provides developers with additional flexibility when designing applications.

For additional detail, see the *Data Contract* explanation.

Document

A document is an atomic entity used by the platform to store user-submitted data. It resembles the documents stored in a document-oriented DB (e.g. MongoDB). All documents must follow some specific rules that are defined by a generic document schema. Additionally, documents are always related to a particular application, so they must comply with the rules defined by the application's data contract. Documents are submitted to the platform API (*DAPI*) by clients during their use of the application.

For additional detail, see the *Document* explanation.

State Transition

A state transition represents a change made by a user to the application and platform states. It consists of:

- Either:
  - An array of documents, or
  - One data contract
- The contract ID of the application to which the change is made
- The user's signature.

The user signature is made for the binary representation of the state transition using a private key associated with an *identity*. A state transition is constructed by a client-side library when the user creates documents and submits them to the platform API.

For additional detail, see the *State Transition* explanation.

1.19.3 Versions

Version	Information
0.24	See details in the GitHub release.
0.23	See details in the GitHub release.
0.22	See details in the GitHub release.
0.21	See details in the GitHub release.
0.20	This release updated to a newer version of JSON Schema (2020-12 spec) and also switched to a new regex module (Re2) for improved security. See more details in the GitHub release.

Data Contract

Overview

As described briefly in the *Dash Platform Protocol explanation*, Dash Platform uses data contracts to define the schema (structure) of data it stores. Therefore, an application must first register a data contract before using the platform to store its data. Then, when the application attempts to store or change data, the request will only succeed if the new data matches the data contract's schema.

The first two data contracts are the DashPay wallet and *Dash Platform Name Service (DPNS)*. The concept of the social, username-based DashPay wallet served as the catalyst for development of the platform, with DPNS providing the mechanism to support usernames.

Details

Ownership

Data contracts are owned by the *identity* that registers them. Each identity may be used to create multiple data contracts and data contract updates can only be made using the identity that owns it.

Structure

Each data contract must define several fields. When using the JavaScript implementation of the Dash Platform Protocol, some of these fields are automatically set to a default value and do not have to be explicitly provided. These include:

- The platform protocol schema it uses (default: defined by js-dpp)
- A contract ID (generated from a hash of the data contract's owner identity plus some entropy)
- One or more documents

In the *example contract* shown below, a `contact` document and a `profile` document are defined. Each of these documents then defines the properties and indices it requires.

Registration

Once a *Dash Platform Protocol* compliant data contract has been defined, it may be registered on the platform. Registration is completed by submitting a state transition containing the data contract to *DAPI*.

The drawing below illustrates the steps an application developer follows to complete registration.

Fig. 2: Data Contract Registration

Updates

Since Dash Platform v0.22, it is possible to update existing data contracts in certain backwards-compatible ways. This includes adding new documents, adding new optional properties to existing documents, and adding non-unique indices for properties added in the update.

> For more detailed information, see the *Platform Protocol Reference - Data Contract* page.

Example Contract

An example contract for DashPay is included below:

```
{
  "profile": {
    "type": "object",
    "indices": [
      {
        "properties": [
          {
            "$ownerId": "asc"
          }
        ],
        "unique": true
      },
      {
        "properties": [
          {
```

(continues on next page)

(continued from previous page)

```
            "$ownerId": "asc"
          },
          {
            "$updatedAt": "asc"
          }
        ]
      }
    ],
    "properties": {
      "avatarUrl": {
        "type": "string",
        "format": "url",
        "maxLength": 2048
      },
      "avatarHash": {
        "type": "array",
        "byteArray": true,
        "minItems": 32,
        "maxItems": 32,
        "description": "SHA256 hash of the bytes of the image specified by avatarUrl"
      },
      "avatarFingerprint": {
        "type": "array",
        "byteArray": true,
        "minItems": 8,
        "maxItems": 8,
        "description": "dHash the image specified by avatarUrl"
      },
      "publicMessage": {
        "type": "string",
        "maxLength": 140
      },
      "displayName": {
        "type": "string",
        "maxLength": 25
      }
    },
    "required": [
      "$createdAt",
      "$updatedAt"
    ],
    "additionalProperties": false
  },
  "contactInfo": {
    "type": "object",
    "indices": [
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
```

(continues on next page)

(continued from previous page)

```
            "rootEncryptionKeyIndex": "asc"
          },
          {
            "derivationEncryptionKeyIndex": "asc"
          }
        ],
        "unique": true
      },
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "$updatedAt": "asc"
          }
        ]
      }
    ],
    "properties": {
      "encToUserId": {
        "type": "array",
        "byteArray": true,
        "minItems": 32,
        "maxItems": 32
      },
      "rootEncryptionKeyIndex": {
        "type": "integer",
        "minimum": 0
      },
      "derivationEncryptionKeyIndex": {
        "type": "integer",
        "minimum": 0
      },
      "privateData": {
        "type": "array",
        "byteArray": true,
        "minItems": 48,
        "maxItems": 2048,
        "description": "This is the encrypted values of aliasName + note + displayHidden␣
↪encoded as an array in cbor"
      }
    },
    "required": [
      "$createdAt",
      "$updatedAt",
      "encToUserId",
      "privateData",
      "rootEncryptionKeyIndex",
      "derivationEncryptionKeyIndex"
    ],
    "additionalProperties": false
```

(continues on next page)

(continued from previous page)

```
  },
  "contactRequest": {
    "type": "object",
    "indices": [
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "toUserId": "asc"
          },
          {
            "accountReference": "asc"
          }
        ],
        "unique": true
      },
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "toUserId": "asc"
          }
        ]
      },
      {
        "properties": [
          {
            "toUserId": "asc"
          },
          {
            "$createdAt": "asc"
          }
        ]
      },
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "$createdAt": "asc"
          }
        ]
      }
    ],
    "properties": {
      "toUserId": {
        "type": "array",
```

(continues on next page)

(continued from previous page)

```
        "byteArray": true,
        "minItems": 32,
        "maxItems": 32,
        "contentMediaType": "application/x.dash.dpp.identifier"
      },
      "encryptedPublicKey": {
        "type": "array",
        "byteArray": true,
        "minItems": 96,
        "maxItems": 96
      },
      "senderKeyIndex": {
        "type": "integer",
        "minimum": 0
      },
      "recipientKeyIndex": {
        "type": "integer",
        "minimum": 0
      },
      "accountReference": {
        "type": "integer",
        "minimum": 0
      },
      "encryptedAccountLabel": {
        "type": "array",
        "byteArray": true,
        "minItems": 48,
        "maxItems": 80
      },
      "autoAcceptProof": {
        "type": "array",
        "byteArray": true,
        "minItems": 38,
        "maxItems": 102
      },
      "coreHeightCreatedAt": {
        "type": "integer",
        "minimum": 1
      }
    },
    "required": [
      "$createdAt",
      "toUserId",
      "encryptedPublicKey",
      "senderKeyIndex",
      "recipientKeyIndex",
      "accountReference"
    ],
    "additionalProperties": false
  }
}
```

This is a visualization of the JSON data contract as UML class diagram for better understanding of the structure:

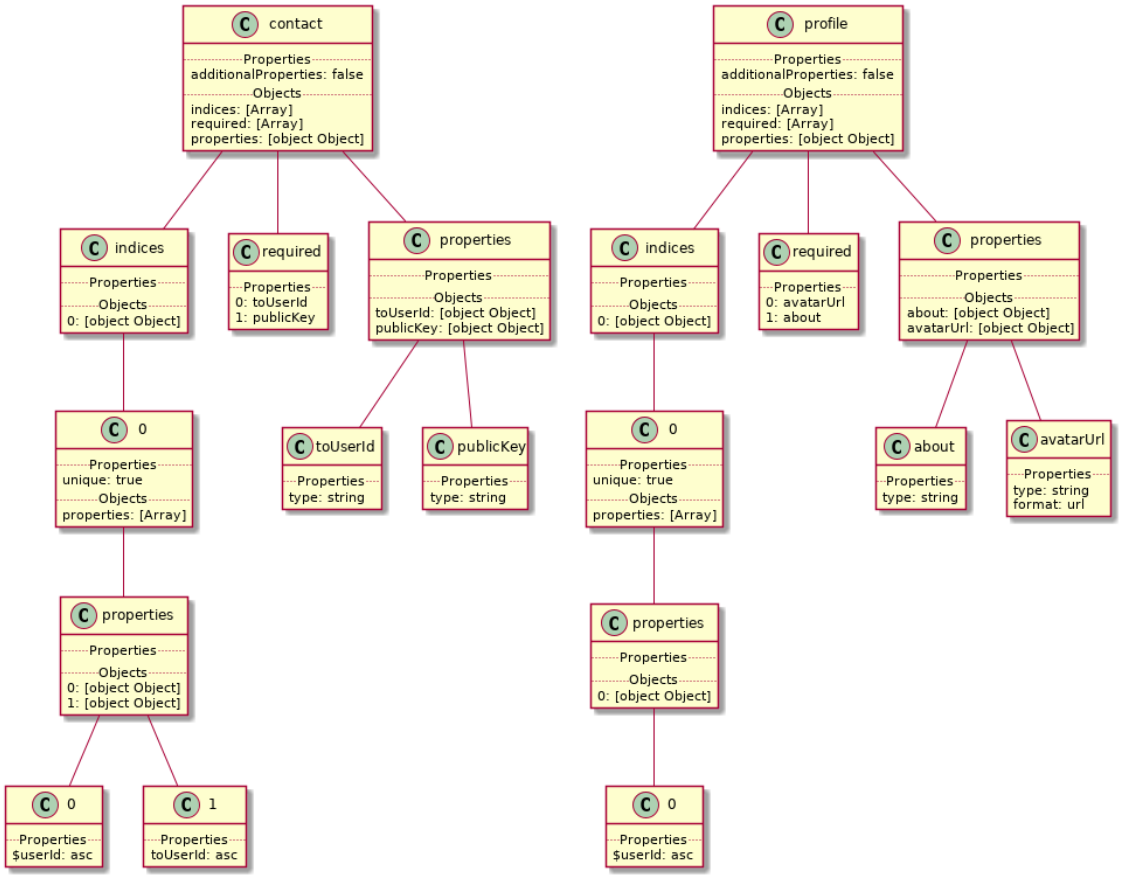

Fig. 3: Dashpay Contract Diagram

View a full-size copy of this diagram.

State Transition

Overview

At any given point in time, the data stored by each application (and more broadly, the entire platform) is in a specific state. State transitions are the means for submitting data that creates, updates, or deletes platform data and results in a change to a new state.

For example, Alice may have already added Bob and Carol as friends in *DashPay* while also having a pending friend request to Dan. If Dan declines the friend request, the state will transition to a new one where Alice and Bob remain in Alice's friend list while Dan moves to the declined list.

Fig. 4: State Transition Example

Implementation Overview

To ensure the consistency and integrity of data stored on Layer 2, all data is governed by the *Dash Platform Protocol* (DPP) which describes serialization and validation rules. Since state transitions are the vehicle for delivering data to the platform, the implementation of state transitions resides in DPP alongside the validation logic.

Structure

To support the various data types used on the platform and enable future updates, state transitions were designed to be flexible. Each state transition consists of a:

1. Header - version and payload type
2. Payload - contents vary depending on payload type
3. Signature - signature of the header/payload by the identity submitting to state transition

The following table contains a list of currently defined payload types:

Payload Type	Payload Description
Data Contract Create (0)	*Database schema* for a single application
Documents Batch (1)	An array of 1 or more *document* transition objects containing application data
Identity Create (2)	Information including the public keys required to create a new *Identity*
Identity Topup (3)	Information including proof of a transaction containing an amount to add to the provided identity's balance
Data Contract Update (4)	An updated *database schema* to modify an existing application

Application Usage

State transitions are constructed by client-side libraries and then submitted to the platform via *DAPI*. Based on the validation rules described in *DPP* (and an application *data contract* where relevant), Dash Platform first validates the state transition.

Some state transitions (e.g. data contracts, identity) are validated solely by rules explicitly defined in DPP, while others (e.g. documents) are also subject to the rules defined by the relevant application's data contract. Once the state transition has been validated, the platform stores the data and updates the platform state.

> For more detailed information, see the *Platform Protocol Reference - State Transition* page

Document

Overview

Dash Platform is based on document-oriented database concepts and uses related terminology. In short, JSON documents are stored into document collections which can then be fetched back using a *query language* similar to common document-oriented databases like MongoDB, CouchDB, or Firebase.

Documents are defined in an application's *Data Contract* and represent the structure of application-specific data. Each document consists of one or more fields and the indices necessary to support querying.

Details

Base Fields

Dash Platform Protocol (DPP) defines a set of base fields that must be present in all documents. For the `js-dpp` reference implementation, the base fields shown below are defined in the document base schema.

Field Name	Description
protocolVersion	The platform protocol version (currently 1)
$id	The document ID (32 bytes)
$type	Document type defined in the referenced contract
$revision	Document revision (=>1)
$dataContractId	Data contract ID generated from the data contract's `ownerId` and `entropy` (32 bytes)
$ownerId	*Identity* of the user submitting the document (32 bytes)
$createdAt	Time (in milliseconds) the document was created
$updatedAt	Time (in milliseconds) the document was last updated

> Timestamp fields
>
> Note: The `$createdAt` and `$updatedAt` fields will only be present in documents that add them to the list of *required properties*.

Data Contract Fields

Each application defines its own fields via document definitions in its data contract. Details of the DPNS data contract documents are described below as an example. This contract defines two document types (`preorder` and `domain`) and provides the functionality described in the *Name Service explanation*.

Document Type	Field Name	Data Type
preorder	saltedDomainHash	string
—	—	—
domain	label	string
domain	normalizedLabel	string
domain	normalizedParentvDomainName	string
domain	preorderSalt	array (bytes)
domain	records	object
domain	records.dashUniqueIdentityId	array (bytes)
domain	records.dashAliasIdentityId	array (bytes)
domain	subdomainRules	object
domain	subdomainRules.allowSubdomains	boolean

Example Document

The following example shows the structure of a DPNS `domain` document as output from `JSON.stringify()`. Note the $ prefix indicating the base fields.


```
{
  "$protocolVersion": 1,
  "$id": "5D8U1k6t6ax8TnyL6QGFFbtMhn39zsixrSMQaxZrYKf1",
  "$type": "domain",
  "$dataContractId": "GWRSAVFMjXx8HpQFaNJMqBV7MBgMK4br5UESsB4S31Ec",
  "$ownerId": "9gU2ZnDhkakHgB4eLbqvEAwQPDBwhW12KD5xPZxybNjE",
  "$revision": 1,
  "label": "RT-Sylvan-71605",
  "normalizedLabel": "rt-sylvan-71605",
  "normalizedParentDomainName": "dash",
  "preorderSalt": "zKaLWLe+kKHiRoBXdfSd7TSU9HdIseeoOly1eTYZ670=",
  "records": {
    "dashUniqueIdentityId": "9gU2ZnDhkakHgB4eLbqvEAwQPDBwhW12KD5xPZxybNjE"
  },
  "subdomainRules": {
    "allowSubdomains": false
  }
}
```

Document Submission

Once a document has been created, it must be encapsulated in a State Transition to be sent to the platform. The structure of a document state transition is shown below. For additional details, see the *State Transition* explanation.

Field Name	Description
protocolVersion	Dash Platform Protocol version (currently `0`)
type	State transition type (1 for documents)
ownerId	Identity submitting the document(s)
transitions	Document `create`, `replace`, or `delete` transitions (up to 10 objects)
signaturePublicKeyId	The `id` of the identity public key that signed the state transition
signature	Signature of state transition data

Document Create

The document create transition is used to create a new document on Dash Platform. The document create transition extends the *base schema* to include the following additional fields:

Field	Type	Description
$entropy	array (32 bytes)	Entropy used in creating the document ID
$createdAt	integer	(Optional) Time (in milliseconds) the document was created
$updatedAt	integer	(Optional) Time (in milliseconds) the document was last updated

Document Replace

The document replace transition is used to update the data in an existing Dash Platform document. The document replace transition extends the *base schema* to include the following additional fields:

Field	Type	Description
$revision	integer	Document revision (=> 1)
$updatedAt	integer	(Optional) Time (in milliseconds) the document was last updated

Document Delete

The document delete transition is used to delete an existing Dash Platform document. It only requires the fields found in the base document transition.

For more detailed information, see the *Platform Protocol Reference - Document* page.

Data Trigger

This page is intended to provide a brief description of how data triggers work in the initial version of Dash Platform. The design will likely undergo changes in the future.

Overview

Although *data contracts* provide much needed constraints on the structure of the data being stored on Dash Platform, there are limits to what they can do. Certain system data contracts may require server-side validation logic to operate effectively. For example, *DPNS* must enforce some rules to ensure names remain DNS compatible. *Dash Platform Protocol* (DPP) supports this application-specific custom logic using Data Triggers.

Constraints

Given a number of technical considerations (security, masternode processing capacity, etc.), data triggers are not considered a platform feature at this time. They are currently hard-coded in Dash Platform Protocol and only used in system data contracts.

Details

Since all application data is submitted in the form of documents, data triggers are defined in the context of documents. To provide even more granularity, they also incorporate the document `action` so separate triggers can be created for the `CREATE`, `REPLACE`, or `DELETE` actions.

As an example, DPP contains several data triggers for DPNS. The `domain` document has added constraints for creation. All DPNS document types have constraints on replacing or deleting:

Data Contract	Document	Action(s)	Trigger Description
DPNS	`domain`	`CREATE`	Enforces DNS compatibility, validate provided hashes, and restrict top-level domain (TLD) registration
—-	—-	—-	—-
DPNS	All Document Types	`REPLACE`	Prevents updates to any DPNS document type
DPNS	All Document Types	`DELETE`	Prevents deletion of any DPNS document type

When document state transitions are received, DPP checks if there is a trigger associated with the document type and action. If a trigger is found, DPP executes the trigger logic. Successful execution of the trigger logic is necessary for the document to be accepted and applied to the *platform state*.

1.20 Identity

1.20.1 Overview

Identities are foundational to Dash Platform. They provide a familiar, easy-to-use way for users to interact and identify one another using names rather than complicated cryptocurrency identifiers such as public key hashes.

Identities are separate from names and can be thought of as a lower-level primitive that provides the foundation for various user-facing functionality. An identity consists primarily of one or more public keys recorded on the platform chain that can be used to control a user's profile and sign their documents. Each identity also has a balance of *credits* that is established by locking funds on layer 1. These credits are used to pay fees associated with the *state transitions* used to perform actions on the platform.

> Identities DIP
>
> The Identities Dash Improvement Proposal (DIP) provides more extensive background information and details.

1.20.2 Identity Management

In order to *create an identity*, a user pays the network to store their public key(s) on the platform chain. Since new users may not have existing Dash funds, an invitation process will allow users to create an identity despite lacking their own funds. The invitation process will effectively separate the funding and registration steps that are required for any new identity to be created.

Once an identity is created, its credit balance is used to pay for activity (e.g. use of applications). The *topup process* provides a way to add additional funds to the balance when necessary.

Identity Create Process

> Testnet Faucet
>
> On Testnet, a test Dash faucet is available. It dispenses small amounts to enable all users to directly acquire the funds necessary to create an identity and username.

First, a sponsor (which could be a business, another person, or even the same user who is creating the identity) spends Dash in a transaction to create an invitation. The transaction contains one or more outputs which lock some Dash funds to establish credits within Dash platform.

After the transaction is broadcast and confirmed, the sponsor sends information about the invitation to the new user. This may be done as a hyperlink that the core wallet understands, or as a QR code that a mobile wallet can scan. Once the user has the transaction data from the sponsor, they can use it to fund an identity create state transition within Dash platform.

Users who already have Dash funds can act as their own sponsor if they wish, using the same steps listed here.

Identity Balance Topup Process

The identity balance topup process works in a similar way to the initial identity creation funding. As with identity creation, a lock transaction is created on the layer 1 core blockchain. This lock transaction is then referenced in the identity topup state transition which increases the identity's balance by the designated amount.

> Since anyone can topup either their own account or any other account, application developers can easily subsidize the cost of using their application by topping up their user's identities.

Identity Update Process

> Added in Dash Platform Protocol v0.23

Identity owners may find it necessary to update their identity keys periodically for security purposes. The identity update state transition enables users to add new keys and disable existing ones.

Identity updates only require the creation of a state transition that includes a list of keys being added and/or disabled. Platform retains disabled keys so that any existing data they signed can still be verified while preventing them from signing new data.

Masternode Identities

Dash Platform v0.22 introduced identities for masternode owners and operators, and a future release will introduce identities for masternode voters. The system automatically creates and updates these identities based on information in the layer 1 masternode registration transactions. For example, owner/operator withdraw keys on Platform are aligned with the keys assigned on the Core blockchain.

In a future release, the credits paid as fees for state transitions will be distributed to masternode-related identities similar to how rewards are currently distributed to masternodes on the core blockchain. Credits will be split between owner and operator in the same ratio as on layer 1, and masternode owners will also have the flexibility to further split their portion between multiple identities to support reward-sharing use cases.

1.20.3 Credits

> Added in Dash Platform Protocol v0.13
>
> DPP v0.13 introduced the initial implementation of credits. Future releases will expand the functionality available.

As mentioned above, credits provide the mechanism for paying fees that cover the cost of platform usage. Once a user locks Dash on the core blockchain and proves ownership of the locked value in an identity create or topup transaction, their credit balance increases by that amount. As they perform platform actions, these credits are deducted to pay the associated fees.

> As of Dash Platform Protocol v0.13, credits deducted to pay state transition fees cannot be converted by masternodes back into Dash. This aspect of the credit system will come in a future release.

1.21 Name Service (DPNS)

1.21.1 Overview

Dash Platform Name Service (DPNS) is a service used to register names on Dash Platform. It is a general service that is used to provide usernames and application names for *identities* but can also be extended in the future to resolve other cryptocurrency addresses, websites, and more. DPNS is implemented as an application on top of the platform and leverages platform capabilities.

DPNS DIP

The DPNS Dash Improvement Proposal (DIP) provides more extensive background information and details.

Relationship to identities

DPNS names and *Identities* are tightly integrated. Identities provide a foundation that DPNS builds on to enable name-based interactions – a user experience similar to what is found in non-cryptocurrency applications. With DPNS, users or application developers register a name and associate it with an identity. Once linked, the identity's private keys allow them to prove ownership of the name to establish trust when they interact with other users and applications.

1.21.2 Details

Name Registration Process

Given the DNS-compatible nature of DPNS, all DPNS names are technically domain names and are registered under a top-level DPNS domain (`.dash`). Some applications may abstract the top-level domain away, but it is important to be aware that it exists.

To prevent front-running, name registration is split into a two phase process consisting of:

1. Pre-ordering the domain name
2. Registering the domain name

In the pre-order phase, the domain name is salted to obscure the actual domain name being registered (e.g. `hash('alice.dash' + salt)`) and submitted to platform. This is done to prevent masternodes from seeing the names being registered and "stealing" them for later resale. Once the pre-order receives a sufficient number of confirmations, the registration can proceed.

In the registration phase, the domain name (e.g. `alice.dash`) is once again submitted along with the salt used in the pre-order. The salt serves as proof that the registration is from the user that submitted the pre-order. This registration also references the identity being associated with the domain name to complete the identity-domain link.

Implementation

DPNS names currently have several constraints as defined in the DPNS data contract. The constraints exist to maintain compatibility with DNS:

- Maximum length - 63 characters
- Character set - `0-9`, `-` (hyphen), and `A-Z` (case insensitive)

> Note: Use of `-` as a prefix/suffix to a name is *not* allowed (e.g. `-name` or `name-`). This constraint is defined by this JSON-Schema pattern in the DPNS data contract:

```
"^[a-zA-Z0-9][a-zA-Z0-9-]{0,61}[a-zA-Z0-9]$"
```

Additionally, the DPNS *data triggers* defined in js-dpp enforce additional validation rules related to the `domain` document.

For more implementation details, please reference the open-source JavaScript DPNS client reference implementation found in the js-dpns-client repository. Additionally, the DPNS data contract is available in the dpns-contract repository.

Contract Diagram

This is a visualization of the JSON data contract as UML class diagram for better understanding of the structure. The left side shows the `domain` document and the right side shows the `preorder` document:

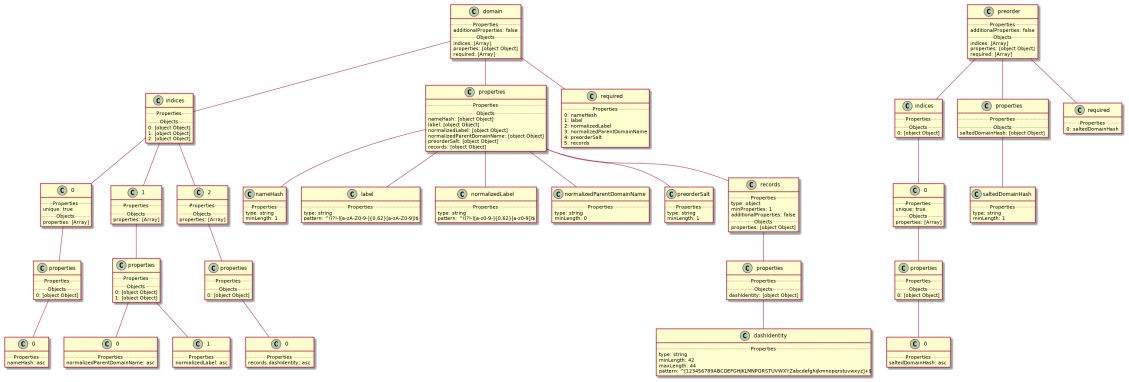

Fig. 5: DPNS Contract Diagram

View a full-size copy of this diagram.

1.22 Drive

1.22.1 Overview

Using the traditional, layer 1 blockchain for data storage is widely known to be expensive and inefficient. Consequently, data for Dash Platform applications is stored in Drive, a layer 2 component that provides decentralized storage hosted by masternodes. As data changes over time, Drive maintains a record of the current state of each item to support easy retrieval using *DAPI*.

1.22.2 Details

Drive Components

There are a number of components working together to facilitate Drive's overall functionality. These components are listed below along with a brief description of service they provide:

- *Platform chain* (orders state transitions; creates and propagates blocks of state transitions)
- Platform state machine (validates data against the *Dash platform protocol*; applies data to state and storage)
- *Platform state* (represents current data)
- Storage (record of state transitions)

Data Update Process

The process of adding or updating data in Drive consists of several steps to ensure data is validated, propagated, and stored properly. This description provides a simplified overview of the process:

1. *State transitions* are submitted to the platform via *DAPI*
2. DAPI sends the state transitions to the platform chain where they are validated, ordered, and committed to a block
3. Valid state transitions are applied to the platform state
4. The platform chain propagates a block containing the state transitions
5. Receiving nodes update Drive data based on the valid state transitions in the block

Fig. 6: Storing data in Drive

Platform Chain

Overview

The platform chain is the *Drive* component responsible for replicating the platform state across all masternodes participating in the network. Masternodes operate this Proof of Service (PoSe) chain to provide layer 2 consensus and support Dash Platform-specific requirements without impacting layer 1 functionality. Although the platform chain can read from the Dash layer 1 core blockchain, the core blockchain is not dependent on it or aware of it.

Details

Evolution of design

Early designs of Drive were based on using on the layer 1 core blockchain and IPFS to replicate layer 2 data. As the design matured, a number of challenges led to a re-evaluation of how to efficiently secure, propagate, and finalize this data. Ultimately, meeting the requirements for a trustless, decentralized system led to choosing a blockchain-based solution over some seemingly obvious choices that work fine in a centralized setting.

Characteristics

In order to support Dash Platform's performance requirements, the platform chain has the following design characteristics:

- Relies on masternode Proof of Service, not miner Proof of Work (PoW)
- Hosted exclusively on masternodes
- Uses a *practical Byzantine Fault Tolerance (pBFT)* consensus algorithm
- Has a deterministic fee structure
- Provides fast (< 10 seconds) and absolute block finality (no reorgs)

Blocks and Transitions

Similar to transactions on the Dash core chain, state transitions are aggregated and put into blocks periodically on the platform chain. Each block has a header that points back to the previous block, thus forming a chain of blocks that is shared among all masternodes. The platform's pBFT consensus algorithm is responsible for ordering the state transitions into a block and then committing the block. As soon as a block is accepted by a + 1 majority of validators, it becomes final and cannot be changed. Thus, the platform chain is not susceptible to blockchain reorganizations.

Platform State

Platform state represents the current state of all the data stored on the platform. You can think about this as one large database, where each application has its own database (Application State) defined by the Data Contract associated with the application. Therefore, the platform state can be thought of as a global view of all Dash Platform data, whereas the application state is a local view of one application's data.

The Platform Chain is processed by a state machine to reach consensus on how to build the state and what it should look like. The last block of the Platform Chain contains the root of the tree structure built from all documents in the platform state. By checking the root of the state tree stored in the block, the node can confirm that it has the correct state.

Fig. 7: Platform State Propagation

1.23 Platform Consensus

Dash Platform is a decentralized network that requires its own consensus algorithm for decision-making and verifying state transitions. This consensus algorithm must fulfill the following three requirements:

** * Fast write operations:** The Drive block time needs to be small since state transitions must be confirmed and applied to the state as quickly as possible.
** * Fast reads:** Each block should update the state so that the data and cryptographic proofs can be read directly from the database. However, this needs to be done fast, so a consensus algorithm with faster reads is needed.
** * Data consistency:** Nodes should always respond with the same data for a given block height to negate instances of blockchain reorgs.

Tendermint was selected as the consensus solution that most closely aligned with the requirements and goals of Dash Platform.

1.23.1 Tendermint

Tendermint is a mostly asynchronous, pBFT-based consensus protocol. Here is a quick overview of how it works:

- Validators participate by taking turns to propose. They validate state transitions by voting on them.
- If a validator successfully validates a block, it gets added to the chain. Do note that voting on state transitions is indirect. Plus, validators don't work on individual transitions, but vote on a block of transitions. This method is a lot more resource-friendly.
- If a validator fails to add a block, the protocol automatically moves to the next round, and a new validator is chosen to propose the block.
- Following the proposal, Tendermint goes through two stages to voting – Pre-vote and Pre-Commit.
- A block gets committed when it gets >2/3rd of the total validators pre-committing for it in one round. The sequence of Propose -> Pre-vote -> Pre-commit is one round.
- In the event of a network dispute, Tendermint prefers consistency over availability.No additional blocks are confirmed or finalized until the dispute is resolved. This takes network reorg out of the equation.

Tendermint has been mainly designed to enable efficient verification and authentication of the latest state of the blockchain. It does so by embedding cryptographic commitments for certain information in the block "header." This information includes:

- Contents of the block.
- The Validator set committing the block.
- Various results returned by the application.

  Notes about Tendermint

  - Block execution only occurs after a block is committed. So, cryptographic proofs for the latest state are only available in the subsequent block.
  - Information like the transaction results and the validator set is never directly included in the block - only their Merkle roots are.
  - Verification of a block requires a separate data structure to store this information. We call this the "State."
  - Block verification also requires access to the previous block.

  Additional information about Tendermint is available in the Tendermint Core spec.

Tendermint Limitations

While Tendermint provided a great starting point, implementing the classic version of the algorithm would have required us to start from scratch. For example, Tendermint validators use EdDSA cryptographic keys to sign votes during the consensus process.

However, Dash already has a well-established network of Masternodes that use BLS keys and a BLS threshold signing mechanism to produce a single signature that mobile wallets and other light clients can easily verify. In addition, subsets of masternodes, called Long-living Masternode Quorums (LLMQ), can perform BLS threshold signing on arbitrary messages.

Rather than reinventing the wheel, Dash chose to fork the Tendermint code and integrate masternode quorums into the process to create a new consensus algorithm called "Tenderdash."

1.23.2 Tenderdash

As with Tendermint, Tenderdash provides Byzantine Fault Tolerant (BFT) State Machine Replication via blocks containing transactions. Additionally, it has been updated to integrate some improvements that leverage Dash's LLMQs. Key mechanisms of the Tenderdash algorithm include:

- If enough members have signed the same message, a valid recovered threshold signature can be created and propagated to the rest of the network.
- Quorums are formed and rotated from time to time through distributed key generation (DKG) sessions.
- DKG chooses pseudorandom nodes from the deterministic masternode list.
- The resulting quorum is then committed to the core blockchain as a transaction.
- The members of a quorum operate somewhat like validators but do so more efficiently due to the pre-existing BLS threshold signature.
- BLS threshold signing results in more compact block headers since only a single BLS threshold signature is required instead of individual signatures from each validator. Notably, this means that any client can easily verify the block signatures using the deterministic masternode list.
- The validators' signature is produced by an LLMQ, which is secured by the core blockchain's Proof-of-Work (PoW).

This allows Dash Platform to leverage the best of both worlds – the speed and finality of Tendermint and the security of PoW.

Dynamic Validator Set Rotation

Rather than having a static validator set, Tenderdash periodically changes to a new set of validator nodes. These validator sets are a subset of masternodes that belong to the LLMQs.

The validator set is assigned to a new masternode quorum every 15 blocks (~2 mins). To determine the next quorum, the BLS threshold signature of the previous block is used as a verifiable random function to choose one of the available quorums.

There are many advantages to adopting this dynamic rotation approach:

- The validator set is less predictable, which reduces the window for attacks like DoS.
- The process balances the performance and security of platform chains like InstantSend and ChainLock quorum changes on the core chain.

1.23.3 How Does Tenderdash Differ From Tendermint?

Here are the differences between Tenderdash and Tendermint:

- **Threshold Signatures**: Tenderdash employs threshold signatures for signing, adding an extra layer of security.
- **Quorum-Based Voting**: Tenderdash implements quorums, meaning not all validators participate in every voting round; only active quorum members are involved, enhancing efficiency.
- **Execution Timing**: Tenderdash facilitates same-block execution, optimizing transaction processing, whereas Tendermint traditionally relies on next-block execution.
- **Consensus Module Refactoring**: Tenderdash has undergone a complete overhaul of its vote-extensions and consensus module, working diligently to eliminate deadlocks and increase stability.

- **Dynamic Validator Management**: Tenderdash incorporates logic to actively connect with new validators in a set and disconnect those that are no longer in the validator set, thereby ensuring an adaptable and efficient network.
- **Project Activity**: Whereas Tenderdash continues to evolve and improve, Tendermint appears somewhat inactive lately, though this observation might be subjective.

1.24 DashPay

1.24.1 Overview

DashPay is one of the first applications of Dash Platform's *data contracts* . At its core DashPay is a data contract that enables a decentralized application that creates bidirectional *direct settlement payment channels* between *identities*.

> For previews of an updated Dash mobile wallet UI based on the DashPay contract or to join the alpha test program, please visit the DashPay landing page at dash.org.

The DashPay contract enables an improved Dash wallet experience with features including:

- **User Centric Interaction**: DashPay brings users front and center in a cryptocurrency wallet. Instead of sending to an address, a user sends directly to another user. Users will have a username, a display name, an avatar and a quick bio/information message.
- **Easy Payments**: Once two users have exchanged contact requests, each can make payments to the other without manually sharing addresses via emails, texts or BIP21 QR codes. This is because every contact request contains the information (an encrypted extended public key) required to send payments to the originator of the request. When decrypted, this extended public key can be used by the recipient of the contact request to generate payment addresses for the originator of the contact request.
- **Payment History**: When a contact is established, a user can easily track the payments they have sent to another user and the payments that they have received from that other user. A user will have an extended private key to track payments that are received from the other user and an extended public key to track payments that are sent to that other user.
- **Payment Participant Protection**: The extended public keys in contact requests are encrypted in such a way that only the two users involved in a contact's two way relationship can decrypt those keys. This ensures that when any two users make payments in DashPay, only they know the sender and receiver while 3rd parties do not.

1.24.2 Details

The contract defines three document types: `contactRequest`, `profile` and `contactInfo`. ContactRequest documents are the most important. They are used to establish relationships and payment channels between Dash identities. Profile documents are used to store public facing information about Dash identities including avatars and display names. ContactInfo documents can be used to store private information about other Dash identities.

Establishing a Contact

1. Bob installs wallet software that supports DashPay.
2. Bob *registers an identity* and then *creates a username* through *DPNS*.
3. Bob searches for Carol by her username. Behind the scenes this search returns the unique identifier for Carol's identity. An example of doing this can be seen in the *Retrieve a Name tutorial*.
4. Bob sends a contact request containing an encrypted extended public key to Carol. This establishes a one way relationship from Bob to Carol.
5. Carol accepts the request by sending a contact request containing an encrypted extended public key back to Bob. This establishes a one way relationship from Carol to Bob.
6. Bob and Carol are now contacts of one another and can make payments to each other by decrypting the extended public key received from the other party and deriving payment addresses from it. Since both have established one way relationships with each other, they now have a two way relationship. If Bob gets a new device, he can use his recovery phrase from step one and restore his wallet, contacts (including Carol) and payments to and from his contacts.

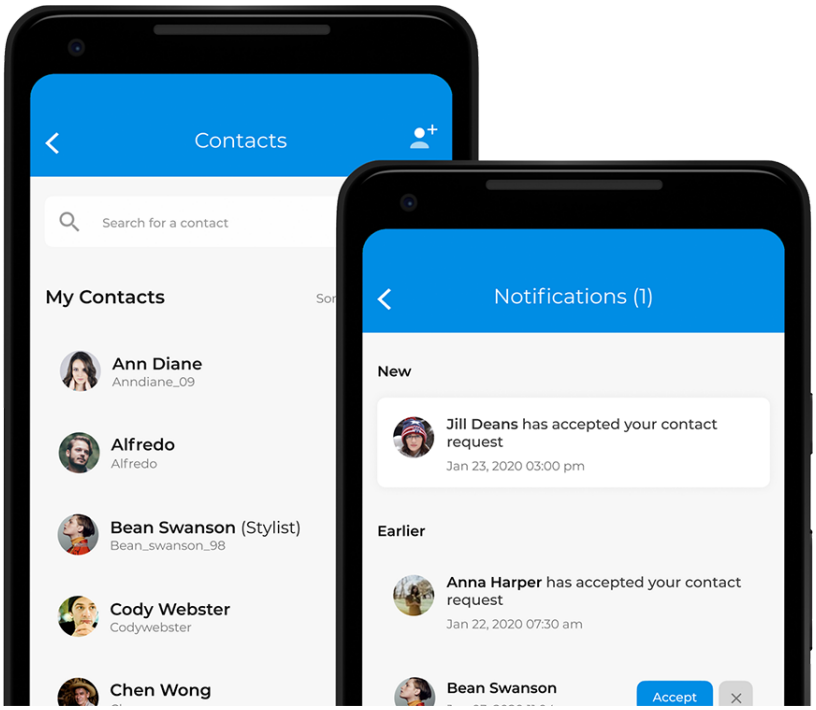

Fig. 8: Contact-based Wallet

Implementation

DashPay has many constraints as defined in the DashPay data contract. Additionally, the DashPay data triggers defined in js-dpp enforce additional validation rules related to the `contactRequest` document.

> DashPay DIP
>
> Please refer to the DashPay Dash Improvement Proposal (DIP) for more extensive background information and complete details about the data contract.

- Contact request details
- Profile details
- Contact Info details

```
{
  "profile": {
    "type": "object",
    "indices": [
      {
        "properties": [
          {
            "$ownerId": "asc"
          }
        ],
        "unique": true
      },
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "$updatedAt": "asc"
          }
        ]
      }
    ],
    "properties": {
      "avatarUrl": {
        "type": "string",
        "format": "url",
        "maxLength": 2048
      },
      "avatarHash": {
        "type": "array",
        "byteArray": true,
        "minItems": 32,
        "maxItems": 32,
        "description": "SHA256 hash of the bytes of the image specified by avatarUrl"
      },
      "avatarFingerprint": {
        "type": "array",
        "byteArray": true,
        "minItems": 8,
```

(continues on next page)

(continued from previous page)

```
        "maxItems": 8,
        "description": "dHash the image specified by avatarUrl"
      },
      "publicMessage": {
        "type": "string",
        "maxLength": 140
      },
      "displayName": {
        "type": "string",
        "maxLength": 25
      }
    },
    "required": [
      "$createdAt",
      "$updatedAt"
    ],
    "additionalProperties": false
  },
  "contactInfo": {
    "type": "object",
    "indices": [
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "rootEncryptionKeyIndex": "asc"
          },
          {
            "derivationEncryptionKeyIndex": "asc"
          }
        ],
        "unique": true
      },
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "$updatedAt": "asc"
          }
        ]
      }
    ],
    "properties": {
      "encToUserId": {
        "type": "array",
        "byteArray": true,
        "minItems": 32,
        "maxItems": 32
```

(continues on next page)

(continued from previous page)

```
      },
      "rootEncryptionKeyIndex": {
        "type": "integer",
        "minimum": 0
      },
      "derivationEncryptionKeyIndex": {
        "type": "integer",
        "minimum": 0
      },
      "privateData": {
        "type": "array",
        "byteArray": true,
        "minItems": 48,
        "maxItems": 2048,
        "description": "This is the encrypted values of aliasName + note + displayHidden␣
↪encoded as an array in cbor"
      }
    },
    "required": [
      "$createdAt",
      "$updatedAt",
      "encToUserId",
      "privateData",
      "rootEncryptionKeyIndex",
      "derivationEncryptionKeyIndex"
    ],
    "additionalProperties": false
  },
  "contactRequest": {
    "type": "object",
    "indices": [
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "toUserId": "asc"
          },
          {
            "accountReference": "asc"
          }
        ],
        "unique": true
      },
      {
        "properties": [
          {
            "$ownerId": "asc"
          },
          {
            "toUserId": "asc"
```

(continues on next page)

(continued from previous page)

```
        }
      ]
    },
    {
      "properties": [
        {
          "toUserId": "asc"
        },
        {
          "$createdAt": "asc"
        }
      ]
    },
    {
      "properties": [
        {
          "$ownerId": "asc"
        },
        {
          "$createdAt": "asc"
        }
      ]
    }
  ],
  "properties": {
    "toUserId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "encryptedPublicKey": {
      "type": "array",
      "byteArray": true,
      "minItems": 96,
      "maxItems": 96
    },
    "senderKeyIndex": {
      "type": "integer",
      "minimum": 0
    },
    "recipientKeyIndex": {
      "type": "integer",
      "minimum": 0
    },
    "accountReference": {
      "type": "integer",
      "minimum": 0
    },
    "encryptedAccountLabel": {
      "type": "array",
```

(continues on next page)

(continued from previous page)

```
        "byteArray": true,
        "minItems": 48,
        "maxItems": 80
      },
      "autoAcceptProof": {
        "type": "array",
        "byteArray": true,
        "minItems": 38,
        "maxItems": 102
      },
      "coreHeightCreatedAt": {
        "type": "integer",
        "minimum": 1
      }
    },
    "required": [
      "$createdAt",
      "toUserId",
      "encryptedPublicKey",
      "senderKeyIndex",
      "recipientKeyIndex",
      "accountReference"
    ],
    "additionalProperties": false
  }
}
```

1.25 Fees

1.25.1 Overview

Since Dash Platform is a decentralized system with inherent costs to its functionality, an adequate fee system is necessary in order to incentive the hosts (masternodes) to maintain it.

Fees on Dash Platform are divided into two main categories:

- Storage fees
- Processing fees

Storage fees cover the costs to store the various types of data throughout the network, while processing fees cover the computational costs incurred by the masternodes to process state transitions. For everyday use, processing fees are minuscule compared to storage fees. However, they are important in the prevention of attacks on the network, in which case they become prohibitively expensive.

> Fee System DIP
>
> Comprehensive details regarding fees will be available in an upcoming *Dash Platform Fee System* DIP.

1.25.2 Costs

The current cost schedule is outlined in the table below:

Operation	Cost (credits)
Permanent storage	40000 / byte
Base processing fee	100000
Write to storage	750 / byte
Load from storage	3500 / byte
Seek storage	2000
Query	75 / byte
Load from memory	20 / byte
Blake3 hash function	400 + 64 / 64-byte block

Credits

Refer to the *Identity explanation* section for information regarding how credits are created.

1.25.3 Fee Multiplier

Given fluctuations of the Dash price, a variable *Fee Multiplier* provides a way to balance the cost of fees with network hosting requirements. All fees are multiplied by the Fee Multiplier:

```
feePaid = initialFee * feeMultiplier
```

The Fee Multiplier is subject to change at any time at the discretion of the masternodes via a voting mechanism. Dash Core Group research indicates maintaining fees at approximately 2x the cost of hosting the network is optimal.

1.25.4 Storage Refund

In an attempt to minimize Dash Platform's storage requirements, users are incentivized to remove data that they no longer want to be stored in the Dash Platform state for a refund. Data storage fees are distributed to masternodes over the data's lifetime which is 50 years for permanent storage. Therefore, at any time before the data's fees are entirely distributed, there will be fees remaining which can be refunded to the user if they decide to delete the data.

1.25.5 User Tip

Wallets will be enabled to give users the option to provide a tip to the block proposer in hopes of incentivizing them to include their state transition in the next block. This feature will be especially useful in times of high traffic.

1.25.6 Formula

The high level formula for a state transition's fee is:

```
fee = storageFee + processingFee - storageRefund + userTip
```

1.26 DAPI Endpoints

DAPI currently provides 2 types of endpoints: JSON-RPC and gRPC. The JSON-RPC endpoints expose some layer 1 information while the gRPC endpoints support layer 2 as well as streaming of events related to blocks and transactions/transitions.

1.26.1 JSON-RPC Endpoints

Layer	Endpoint	Description
1	`getBestBlockHash`	Returns block hash of the chaintip
1	`getBlockHash`	Returns block hash of the requested block
1	`getMnListDiff`	Returns masternode list diff for the provided block hashes

1.26.2 gRPC Endpoints

Core gRPC Service

Layer	Endpoint	
1	`broadcastTransaction`	Broadcasts the provided transaction
1	`getBlock`	Returns information for the requested block
1	`getStatus`	Returns blockchain status information
1	`getTransaction`	Returns details for the requested transaction
1	`subscribeToBlockHeadersWithChainLocks`	Returns the requested block headers along with the associated ChainLocks.*Added in Dash Platform v0.22*
1	`subscribeToTransactionsWithProofs`	Returns transactions matching the provided bloom filter along with the associated `islock` message and merkle block

Platform gRPC Service

In addition to providing the request data, the following endpoints can also provide proofs that the data returned is valid and complete.

Layer	Endpoint	
2	`broadcastStateTransition`	Broadcasts the provided State Transition
2	`getIdentity`	Returns the requested identity
2	`getIdentitiesByPublicKeyHashes`	Returns the identities associated with the provided public key hashes*Added in Dash Platform v0.16*
2	`getDataContract`	Returns the requested data contract
2	`getDocuments`	Returns the requested document(s)
2	`waitForStateTransitionResult`	Responds with the state transition hash and either a proof that the state transition was confirmed in a block or an error

The previous version of documentation can be viewed here.

JSON-RPC Endpoints

Overview

The endpoints described on this page provide access to information from the Core chain (layer 1).

Required Parameters

All valid JSON-RPC requests require the inclusion the parameters listed in the following table.

Name	Type	Description
`method`	String	Name of the endpoint
`id`	Integer	Request id (returned in the response to differentiate results from the same endpoint)
`jsonrpc`	String	JSON-RPC version (“2.0”)

Additional information may be found in the JSON-RPC 2.0 specification.

Endpoint Details

getBestBlockHash

Returns: the block hash of the chaintip
Parameters: none

Example Request and Response

SHELL

```
curl -k --request POST \
  --url https://seed-1.testnet.networks.dash.org:1443/ \
  --header 'content-type: application/json' \
  --data '{
      "method":"getBestBlockHash",
      "id":1,
      "jsonrpc":"2.0",
      "params":{}
    }'
```

JAVASCRIPT

```
var request = require("request");

var options = {
  method: 'POST',
  url: 'https://seed-1.testnet.networks.dash.org:1443',
  headers: {'content-type': 'application/json'},
  body: '{"method":"getBestBlockHash","id":1,"jsonrpc":"2.0"}'
```

(continues on next page)

(continued from previous page)

```
};

request(options, function (error, response, body) {
  if (error) throw new Error(error);

  console.log(body);
});
```

JAVASCRIPT

```
// Node.js
var XMLHttpRequest = require('xhr2');
var data = '{"method":"getBestBlockHash","id":1,"jsonrpc":"2.0"}';

var xhr = new XMLHttpRequest();

xhr.addEventListener("readystatechange", function () {
  if (this.readyState === this.DONE) {
    console.log(this.responseText);
  }
});

xhr.open("POST", "https://seed-1.testnet.networks.dash.org:1443");
xhr.setRequestHeader("content-type", "application/json");

xhr.send(data);
```

PYTHON

```
import requests
import json

url = "https://seed-1.testnet.networks.dash.org:1443/"
headers = {'content-type': 'application/json'}

payload_json = {
    "method": "getBestBlockHash",
    "id": 1,
    "jsonrpc": "2.0",
    "params": {}
}

response = requests.request("POST", url, data=json.dumps(payload_json), headers=headers)

print(response.text)
```

RUBY

```ruby
require 'uri'
require 'net/http'

url = URI("https://seed-1.testnet.networks.dash.org:1443/")
http = Net::HTTP.new(url.host, url.port)

request = Net::HTTP::Post.new(url)
request["content-type"] = 'application/json'

request.body = '{
    "method":"getBestBlockHash",
    "id":1,
    "jsonrpc":"2.0",
    "params":{ }
}'

response = http.request(request)
puts response.read_body
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "0000009fd106a7aa7142925fcd311442790145a3351fa2508d9da2b3462086fd"
}
```

getBlockHash

Returns: the block hash for the given height
Parameters:

Name	Type	Required	Description
`height`	Integer	Yes	Block height

Example Request and Response

SHELL

```shell
curl -k --request POST \
  --url https://seed-1.testnet.networks.dash.org:1443/ \
  --header 'content-type: application/json' \
  --data '{
      "method":"getBlockHash",
      "id":1,
      "jsonrpc":"2.0",
      "params": {
        "height": 1
       }
    }'
```

PYTHON

```python
import requests
import json

url = "https://seed-1.testnet.networks.dash.org:1443/"
headers = {'content-type': 'application/json'}

payload_json = {
    "method": "getBlockHash",
    "id": 1,
    "jsonrpc": "2.0",
    "params": {
        "height": 100
    }
}

response = requests.request("POST", url, data=json.dumps(payload_json), headers=headers)

print(response.text)
```

RUBY

```ruby
require 'uri'
require 'net/http'

url = URI("https://seed-1.testnet.networks.dash.org:1443/")
http = Net::HTTP.new(url.host, url.port)

request = Net::HTTP::Post.new(url)
request["content-type"] = 'application/json'

request.body = '{
    "method":"getBlockHash",
    "id":1,
    "jsonrpc":"2.0",
    "params":{
        "height":100
    }
}'

response = http.request(request)
puts response.read_body
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "0000047d24635e347be3aaaeb66c26be94901a2f962feccd4f95090191f208c1"
}
```

getMnListDiff

Returns: a masternode list diff for the provided block hashes
Parameters:

Name	Type	Required	Description
baseBlockHash	String	Yes	Block hash for the starting block
blockHash	String	Yes	Block hash for the ending block

Example Request and Response

SHELL

```
curl -k --request POST \
  --url https://seed-1.testnet.networks.dash.org:1443/ \
  --header 'content-type: application/json' \
  --data '{
      "method":"getMnListDiff",
      "id":1,
      "jsonrpc":"2.0",
      "params": {
        "baseBlockHash":
↪"00000016b4d13db8395b31d87c76ca88824b26e03e54480d8c9ddf6f11857a7c",
        "blockHash": "0000002266d8e7836eb116fe467597d13d9862c6612e31bbd6161c35b6485493"
      }
    }'
```

PYTHON

```
import requests
import json

url = "https://seed-1.testnet.networks.dash.org:1443/"
headers = {'content-type': 'application/json'}

payload_json = {
    "method":"getMnListDiff",
    "id":1,
    "jsonrpc":"2.0",
    "params": {
        "baseBlockHash":
↪"00000016b4d13db8395b31d87c76ca88824b26e03e54480d8c9ddf6f11857a7c",
        "blockHash": "0000002266d8e7836eb116fe467597d13d9862c6612e31bbd6161c35b6485493"
    }
}

response = requests.request("POST", url, data=json.dumps(payload_json), headers=headers)

print(response.text)
```

RUBY

```
require 'uri'
require 'net/http'

url = URI("https://seed-1.testnet.networks.dash.org:1443/")
http = Net::HTTP.new(url.host, url.port)

request = Net::HTTP::Post.new(url)
request["content-type"] = 'application/json'

request.body = '{
    "method":"getMnListDiff",
    "id":1,
    "jsonrpc":"2.0",
    "params": {
        "baseBlockHash":
↪"00000016b4d13db8395b31d87c76ca88824b26e03e54480d8c9ddf6f11857a7c",
        "blockHash": "0000002266d8e7836eb116fe467597d13d9862c6612e31bbd6161c35b6485493"
    }
}'

response = http.request(request)
puts response.read_body
```


```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "baseBlockHash": "00000016b4d13db8395b31d87c76ca88824b26e03e54480d8c9ddf6f11857a7c",
    "blockHash": "0000002266d8e7836eb116fe467597d13d9862c6612e31bbd6161c35b6485493",
    "cbTxMerkleTree":
↪"0300000003795f4c55c12757f3783a81a804585545d1844660f0b59e25a93d332bdf98a1032552fe1c2eada657f6714b14e1
↪",
    "cbTx":
↪"03000500010000000000000000000000000000000000000000000000000000000000000000ffffffff1202e21b0e2f503250
↪",
    "deletedMNs": [],
    "mnList": [
      {
        "proRegTxHash": "682b3e58e283081c51f2e8e7a7de5c7312a2e8074affaf389fafcc39c4805404
↪",
        "confirmedHash":
↪"00000018c82435520c6a850076c041b533d05cbe481f8187e541d7e2f856def",
        "service": "64.193.62.206:19999",
        "pubKeyOperator":
↪"85f2269374676476f00068b7cb168d124b7b780a92e8564e18edf45d77497abd9debf186ee98001a0c9a6dfccbab7a0a
↪",
        "votingAddress": "yid7uAsVJzvSLrEekHuGNuY3KWCqJopyJ8",
        "isValid": true,
        "nVersion": 2,
        "nType": 0
      },
```


(continues on next page)

(continued from previous page)


```
      {
        "proRegTxHash": "c48a44a9493eae641bea36992bc8c27eaaa33adb1884960f55cd259608d26d2f
↪",
        "confirmedHash":
↪"000000237725f8fe7d78153ae9c11193ee0cda18f8b48141acff8e1ac713da5b",
        "service": "173.61.30.231:19013",
        "pubKeyOperator":
↪"8700add55a28ef22ec042a2f28e25fb4ef04b3024a7c56ad7eed4aebc736f312d18f355370dfb6a5fec9258f464b227e
↪",
        "votingAddress": "yTMDce5yEpiPqmgPrPmTj7yAmQPJERUSVy",
        "isValid": true,
        "nVersion": 2,
        "nType": 0
      },
      {
        "proRegTxHash": "9f4f9f83ecbcd5739d7f1479ee14b508f2414d044a717acba0960566c4e6091d
↪",
        "confirmedHash":
↪"0000000000000000000000000000000000000000000000000000000000000000",
        "service": "45.32.211.155:19999",
        "pubKeyOperator":
↪"88e37b3fcba972fe0c2c0ea15f8285c8bfb262ad4d8a6741a530154f1abc4edd367a22abd0cb1934647f033913cca58a
↪",
        "votingAddress": "ybAZoZ6iybhEwoCfb6utGfU753R1wcQSZT",
        "isValid": true,
        "nVersion": 2,
        "nType": 0
      }
    ],
    "nVersion": 2,
    "deletedQuorums": [],
    "newQuorums": [],
    "merkleRootMNList": "e9bf66fe8884e046ef1c393813a91ac7dfb77dd0fb9abb077ed2259b430420f0
↪"
  }
}
```


Deprecated Endpoints

There are no recently deprecated endpoint, but the previous version of documentation can be viewed here.

Code Reference

Implementation details related to the information on this page can be found in:

- The DAPI repository `lib/rpcServer/commands` folder

gRPC Overview

The gRPC endpoints provide access to information from Dash Platform (layer 2) as well as streaming of events related to blocks and transactions/transitions.

Connecting to gRPC

Auto-generated Clients

Clients for a number of languages are built automatically from the protocol definitions and are available in the `packages/dapi-grpc/clients` folder of the platform repository. The protocol definitions are available in the `protos` folder. Pull requests are welcome to add support for additional languages that are not currently being built.

Command Line Examples

Some examples shown in the endpoint details pages use a command-line tool named gRPCurl that allows interacting with gRPC servers in a similar way as `curl` does for the *JSON-RPCs*. Additional information may be found in the gRPC documentation.

To use gRPCurl as shown in the detailed examples, clone the platform repository and execute the example requests from the `packages/dapi-grpc` directory of that repository as shown in this example:

```
## Clone the dapi-grpc repository
git clone https://github.com/dashpay/platform.git
cd platform/packages/dapi-grpc

## Execute gRPCurl command
grpcurl -plaintext -proto protos/...
```

Data Encoding

The data submitted/received from the gRPC endpoints is encoded using both CBOR and Base64. Data is first encoded with CBOR and the resulting output is then encoded in Base64 before being sent.

Canonical encoding is used for state transitions, identities, data contracts, and documents. This puts the object's data fields in a sorted order to ensure the same hash is produced every time regardless of the actual order received by the encoder. Reproducible hashes are necessary to support validation of request/response data.

Libraries such as `cbor` (JavaScript) and `cbor2` (Python) can be used to encode/decode data for DAPI gRPC endpoints.

The examples below use the response from a *`getIdentity` gPRC request* to demonstrate how to both encode data for sending and decode received data:

JAVASCRIPT

```
// NodeJS - Decode Identity
const cbor = require('cbor');

const grpc_identity_response =
↪'o2JpZHgsQ2JZVnlvS25HeGtIYUJydWNDQWhQRUJjcHV6OGoxNWNuWVlpdjFDRUhCTnhkdHlwZQFqcHVibGljS2V5c4GkYmlkAWRk
↪'

const identity_cbor = Buffer.from(grpc_identity_response, 'base64').toJSON();
const identity = cbor.decode(Buffer.from(identity_cbor));

console.log('Identity details');
console.dir(identity);
```

PYTHON

```
# Python - Decode Identity
from base64 import b64decode, b64encode
import json
import cbor2

grpc_identity_response =
↪'o2JpZHgsQ2JZVnlvS25HeGtIYUJydWNDQWhQRUJjcHV6OGoxNWNuWVlpdjFDRUhCTnhkdHlwZQFqcHVibGljS2V5c4GkYmlkAWRk
↪'

identity_cbor = b64decode(grpc_identity_response)
identity = cbor2.loads(identity_cbor)

print('Identity details:\n{}\n'.format(json.dumps(identity, indent=2)))
```

PYTHON

```
# Python - Encode Identity
from base64 import b64decode, b64encode
import json
import cbor2

## Encode an identity
identity = {
  "id": "CbYVyoKnGxkHaBrucCAhPEBcpuz8j15cnYYiv1CEHBNx",
  "type": 1,
  "publicKeys": [
    {
      "id": 1,
      "data": "AmzR2FM4fYwCmZxGZ1N2ta2FfuJ95Ow+LLArZDLuYBjt",
      "type": 1,
      "isEnabled": True
    }
  ]
```

(continues on next page)

(continued from previous page)

```
}

identity_cbor = cbor2.dumps(identity)
identity_grpc = b64encode(identity_cbor)
print('Identity gRPC data: {}'.format(identity_grpc))
```

Core gRPC Endpoints

Please refer to the *gRPC Overview* for details regarding running the examples shown below, encoding/decoding the request/response data, and clients available for several languages.

Endpoint Details

broadcastTransaction

Returns: The transaction id (TXID) if successful
Parameters:

Name	Type	Required	Description
`transaction`	Bytes	Yes	A raw transaction
`allow_high_fees`	Boolean	No	Enables bypassing the high fee check
`bypass_limits`	Boolean	No	

Example Request and Response

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');
const { Transaction } = require('@dashevo/dashcore-lib');

const client = new DAPIClient({
  seeds: [{
    host: 'seed-1.testnet.networks.dash.org',
    port: 1443,
  }],
});

// Replace the transaction hex below with your own transaction prior to running
const tx = Transaction(
↪'02000000022fd1c4583099109524b8216d712373bd837d24a502414fcadd8ae94753c3d87e010000006a4730440220
↪');

client.core.broadcastTransaction(tx.toBuffer())
  .then((response) => console.log(response));
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: {
    CorePromiseClient,
  },
} = require('@dashevo/dapi-grpc');
const { Transaction } = require('@dashevo/dashcore-lib');

const corePromiseClient = new CorePromiseClient('https://seed-1.testnet.networks.dash.
↪org:1443');

// Replace the transaction hex below with your own transaction prior to running
const tx = Transaction(
↪'02000000022fd1c4583099109524b8216d712373bd837d24a502414fcadd8ae94753c3d87e010000006a4730440220
↪');

corePromiseClient.client.broadcastTransaction({ transaction: tx.toBuffer() })
  .then((response) => console.log(response));
```

JSON

```
{
  "transactionId": "552eaf24a60014edcbbb253dbc4dd68766532cab3854b44face051cedcfd578f"
}
```

getStatus

Returns: Status information from the Core chain
Parameters: None

Example Request and Response

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');

const client = new DAPIClient({
  seeds: [{
    host: 'seed-1.testnet.networks.dash.org',
    port: 1443,
  }],
});

client.core.getStatus()
  .then((response) => console.log(response));
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: {
    GetStatusRequest,
    CorePromiseClient,
  },
} = require('@dashevo/dapi-grpc');

const corePromiseClient = new CorePromiseClient('https://seed-1.testnet.networks.dash.
↪org:1443');

corePromiseClient.client.getStatus(new GetStatusRequest())
  .then((response) => console.log(response));
```

SHELL

```
# gRPCurl
# Run in the platform repository's `packages/dapi-grpc/` directory
grpcurl -proto protos/core/v0/core.proto \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Core/getStatus
```

Note: The gRPCurl response `bestBlockHash`, `chainWork`, and `proTxHash` data is Base64 encoded.

JSON

```
// Response (JavaScript)
{
  "version":{
    "protocol":70227,
    "software":190100,
    "agent":"/Dash Core:19.1.0(dcg-masternode-27)/"
  },
  "time":{
    "now":1684860969,
    "offset":0,
    "median":1684860246
  },
  "status":"READY",
  "syncProgress":0.9999992137956843,
  "chain":{
    "name":"test",
    "headersCount":892412,
    "blocksCount":892412,
    "bestBlockHash":"<Buffer 00 00 00 96 7b 75 05 9c ad ff 07 71 89 74 1b 0a 8f f1 77 62␣
↪1d 3e 6e 45 e9 32 02 55 19 fe df a9>",
    "difficulty":0.003254173843543036,
```

(continues on next page)

(continued from previous page)

```
    "chainWork":"<Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00␣
↪00 00 00 02 d6 8d 0c 89 2a 87 47>",
    "isSynced":true,
    "syncProgress":0.9999992137956843
  },
  "masternode":{
    "status":"READY",
    "proTxHash":"<Buffer 3b 27 b5 ea 14 6a d9 d1 ff 6b c7 14 7e f2 5e f7 33 01 df 98 cc␣
↪2c 77 23 67 3c 3a 0f 39 fd b0 7a>",
    "posePenalty":0,
    "isSynced":true,
    "syncProgress":1
  },
  "network":{
    "peersCount":145,
    "fee":{
      "relay":0.00001,
      "incremental":0.00001
    }
  }
}
```

JSON

```
// Response (gRPCurl)
{
  "version": {
    "protocol": 70227,
    "software": 190000,
    "agent": "/Dash Core:19.0.0/"
  },
  "time": {
    "now": 1684357132,
    "median": 1684356285
  },
  "status": "READY",
  "syncProgress": 0.9999996650927735,
  "chain": {
    "name": "test",
    "headersCount": 888853,
    "blocksCount": 888853,
    "bestBlockHash": "AAAAtZ1kS2uIxOX4u1CaHqEPQhOVs23wPK9TjBZnZAI=",
    "difficulty": 0.003153826459898978,
    "chainWork": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtaNAXYoQDE=",
    "isSynced": true,
    "syncProgress": 0.9999996650927735
  },
  "masternode": {
    "status": "READY",
    "proTxHash": "vcAa/9GeHoyawgatmvVCbavRGA3uUtnDigwp7EqRyn0=",
```

(continues on next page)

(continued from previous page)

```
    "isSynced": true,
    "syncProgress": 1
  },
  "network": {
    "peersCount": 147,
    "fee": {
      "relay": 1e-05,
      "incremental": 1e-05
    }
  }
}
```

getBlock

Returns: A raw block
Parameters:

Name	Type	Required	Description
One of the following:			
`hash`	Bytes	No	Return the block matching the block hash provided
`height`	Integer	No	Return the block matching the block height provided

Example Request and Response

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');

const client = new DAPIClient({
  seeds: [{
    host: 'seed-1.testnet.networks.dash.org',
    port: 1443,
  }],
});

client.core.getBlockByHeight(1)
  .then((response) => console.log(response.toString('hex')));
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: {
    CorePromiseClient,
  },
} = require('@dashevo/dapi-grpc');

const corePromiseClient = new CorePromiseClient('https://seed-1.testnet.networks.dash.
↪org:1443');

corePromiseClient.client.getBlock({ height: 1 })
  .then((response) => console.log(response.block.toString('hex')));
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: {
    CorePromiseClient,
  },
} = require('@dashevo/dapi-grpc');

const corePromiseClient = new CorePromiseClient('https://seed-1.testnet.networks.dash.
↪org:1443');

corePromiseClient.client.getBlock({
  hash: '0000047d24635e347be3aaaeb66c26be94901a2f962feccd4f95090191f208c1',
}).then((response) => {
  console.log(response.block.toString('hex'));
});
```

SHELL

```
# gRPCurl
grpcurl -proto protos/core/v0/core.proto \
  -d '{
    "height":1
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Core/getBlock
```

Block Encoding

Note: The gRPCurl response block data is Base64 encoded

SHELL

```
# Response (JavaScript)
020000002cbcf83b62913d56f605c0e581a48872839428c92e5eb76cd7ad94bcaf0b00007f11dcce14075520e8f74cc4ddf092b
```

JSON

```
// Response (gRPCurl)
{
  "block":
↪"AgAAACy8+DtikT1W9gXA5YGkiHKDlCjJLl63bNetlLyvCwAAfxHczhQHVSDo90zE3fCStOJuvSO42GZaGuW/
↪xBtY/bTDqV5T//8PHvN6AAABAQAAAAEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAP////
↪8KUQEBBi9QMlNIL/////8BAHQ7pAsAAAAjIQIBMfOK4+sHFFMdv8P0VJG0Ex0SEeN3cXdjY4i7WnTD5KwAAAAA"
}
```

getTransaction

Returns: A raw transaction
Parameters:

Name	Type	Required	Description
id	String	Yes	A transaction id (TXID)

Example Request and Response

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');

const client = new DAPIClient({
  seeds: [{
    host: 'seed-1.testnet.networks.dash.org',
    port: 1443,
  }],
});

const txid = '4004d3f9f1b688f2babb1f98ea48e1472be51e29712f942fc379c6e996cdd308';
client.core.getTransaction(txid)
  .then((response) => console.dir(response, { length: 0 }));
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: {
    CorePromiseClient,
  },
} = require('@dashevo/dapi-grpc');

const corePromiseClient = new CorePromiseClient('https://seed-1.testnet.networks.dash.
↪org:1443');

const txid = '4004d3f9f1b688f2babb1f98ea48e1472be51e29712f942fc379c6e996cdd308';

corePromiseClient.client.getTransaction({ id: txid })
  .then((response) => console.dir(response, { length: 0 }));
```

SHELL

```
# gRPCurl
grpcurl -proto protos/core/v0/core.proto \
  -d '{
    "id":"4004d3f9f1b688f2babb1f98ea48e1472be51e29712f942fc379c6e996cdd308"
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Core/getTransaction
```

Transaction Encoding

Note: The gRPCurl response `transaction` and `blockHash` data are Base64 encoded

TEXT

```
# Response (JavaScript)
GetTransactionResponse {
  transaction: Buffer(196) [Uint8Array] [
      3,   0,   5,   0,   1,   0,   0,   0,   0,   0,   0,  0,
      0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  0,
      0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  0,
      0, 255, 255, 255, 255,   6,   3, 194,  90,   6,   1,  9,
    255, 255, 255, 255,   2, 238, 252, 207,  49,   0,   0,  0,
      0,  25, 118, 169,  20, 126, 178,  93, 197, 175,  71, 45,
    107, 241, 154, 135, 122, 150, 240, 167,   7, 194, 198, 27,
    118, 136, 172, 101, 251, 183,  74,   0,   0,   0,   0, 25,
    118, 169,  20,  30,
    ... 96 more items
  ],
  blockHash: Buffer(32) [Uint8Array] [
      0,   0,   2,   9, 133, 199, 245,  83,
    191, 120, 191, 203, 109, 166,   9, 115,
    193, 152, 249,  11,   7, 245, 126,  31,
```

(continues on next page)

(continued from previous page)

```
    55,  65,  10, 150, 205, 150, 131, 213
  ],
  height: 416450,
  confirmations: 386421,
  instantLocked: false,
  chainLocked: true
}
```

JSON

```
// Response (gRPCurl)
{
  "transaction": "AwAFAAEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAP////8GA8JaBgEJ/////
↪wLu/M8xAAAAAAB12qRR+sl3Fr0cta/
↪Gah3qW8KcHwsYbdoisZfu3SgAAAAAZdqkUHsXGbpeJxlWuBo01CItAczRf4LCIrAAAAABGAgDCWgYA3zSmucmdu7+CaY+6n4aGHyS
↪",
  "blockHash": "AAACCYXH9VO/eL/LbaYJc8GY+QsH9X4fN0EKls2Wg9U=",
  "height": 416450,
  "confirmations": 472404,
  "isChainLocked": true
}
```

subscribeToBlockHeadersWithChainLocks

This endpoint helps support simplified payment verification (SPV) via DAPI by providing access to block headers which can then be used to verify transactions and simplified masternode lists.

Returns: streams the requested block header information
Parameters:

Name	Type	Required	Description
———			
One of the following:			
`from_block_hash`	Bytes	No	Return records beginning with the block hash provided
`from_block_height`	Integer	No	Return records beginning with the block height provided
———			
`count`	Integer	No	Number of blocks to sync. If set to 0 syncing is continuously sends new data as well (default: 0)

** Example Request and Response **

SHELL

```
# gRPCurl
grpcurl -proto protos/core/v0/core.proto \
  -d '{
  "from_block_height": 1,
  "count": 1
}' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Core/subscribeToBlockHeadersWithChainLocks
```

Note: The gRPCurl response `chainlock` and `headers` data is Base64 encoded

JSON

```
// Response (gRPCurl)
{
  "chainLock":
↪"FZANAAJkZxaMU6888G2zlRNCD6EemlC7+OXEiGtLZJ21AAAAo7qvfeETyNxWVog47Yiyx9j9FSUCVkUWBrn0ZAfIbeU75kiccv4i
↪MuDt7rYnVfmPWIUj03QYWKzQKr/PaMkavTaa+PCOrqQYxcLX/s"
}
{
  "blockHeaders": {
    "headers": [

↪"AgAAACy8+DtikT1W9gXA5YGkiHKDlCjJLl63bNetlLyvCwAAfxHczhQHVSDo90zE3fCStOJuvSO42GZaGuW/
↪xBtY/bTDqV5T//8PHvN6AAA="
    ]
  }
}
```

subscribeToTransactionsWithProofs

Returns: streams the requested transaction information
Parameters:

Name	Type	Required	Description
bloom_filter.v_data	Bytes	Yes	The filter itself is simply a bit field of arbitrary byte-aligned size. The maximum size is 36,000 bytes
bloom_filter.n_hash_funcs	Integer	Yes	The number of hash functions to use in this filter. The maximum value allowed in this field is 50
bloom_filter.n_tweak	Integer	Yes	A random value to add to the seed value in the hash function used by the bloom filter
bloom_filter.n_flags	Integer	Yes	A set of flags that control how matched items are added to the filter
———			
One of the following:			
from_block_hash	Bytes	No	Return records beginning with the block hash provided
from_block_height	Integer	No	Return records beginning with the block height provided
———			
count	Integer	No	Number of blocks to sync. If set to 0 syncing is continuously sends new data as well (default: 0)
send_transaction_hashes*	Boolean	No	

** Example Request and Response **

SHELL

```
# gRPCurl
grpcurl -proto protos/core/v0/core.proto \
  -d '{
  "from_block_height": 1,
  "count": 1,
  "bloom_filter": {
    "n_hash_funcs": 11,
    "v_data": "",
    "n_tweak": 0,
    "n_flags": 0
  }
}' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Core/subscribeToTransactionsWithProofs
```

Note: The gRPCurl response `transactions` and `rawMerkleBlock` data is Base64 encoded

JSON

```
// Response (gRPCurl)
{
  "rawTransactions": {
    "transactions": [
      "AQAAAAEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAP////8KUQEBBi9QMlNIL/////
↪8BAHQ7pAsAAAAjIQIBMfOK4+sHFFMdv8P0VJG0Ex0SEeN3cXdjY4i7WnTD5KwAAAAA"
    ]
  }
}
{
  "rawMerkleBlock":
↪"AgAAACy8+DtikT1W9gXA5YGkiHKDlCjJLl63bNetlLyvCwAAfxHczhQHVSDo90zE3fCStOJuvSO42GZaGuW/
↪xBtY/bTDqV5T//8PHvN6AAABAAAAAX8R3M4UB1Ug6PdMxN3wkrTibr0juNhmWhrlv8QbWP20AQE="
}
```

Deprecated Endpoints

There are no recently deprecated endpoints, but the previous version of documentation can be viewed here.

Code Reference

Implementation details related to the information on this page can be found in:

- The Platform repository `packages/dapi/lib/grpcServer/handlers/core` folder
- The Platform repository `packages/dapi-grpc/protos` folder

Platform gRPC Endpoints

Please refer to the *gRPC Overview* for details regarding running the examples shown below, encoding/decoding the request/response data, and clients available for several languages.

Data Proofs and Metadata

Since Dash Platform 0.20.0, Platform gRPC endpoints can provide proofs so the data returned for a request can be verified as being valid. Full support is not yet available in the JavaScript client, but can be used via the low level dapi-grpc library.

Some additional metadata is also provided with responses:

Metadata field	Description
`height`	Last committed platform chain height
`coreChainLockedHeight`	Height of the most recent ChainLock on the core chain
`timeMs`	Unix timestamp in milliseconds for the response
`protocolVersion`	Platform protocol version

Endpoint Details

broadcastStateTransition

> **Note:** The *`waitForStateTransitionResult` endpoint* should be used in conjunction with this one for instances where proof of block confirmation is required.

Broadcasts a *state transition* to the platform via DAPI to make a change to layer 2 data. The `broadcastStateTransition` call returns once the state transition has been accepted into the mempool.

Returns: Nothing or error

Parameters:

Name	Type	Required	Description
`state_transition`	Bytes (Base64)	Yes	A *state transition*

Response: No response except on error

getIdentity

> Breaking changes
>
> As of Dash Platform 0.24 the `protocolVersion` is no longer included in the CBOR-encoded data. It is instead prepended as a varint to the data following CBOR encoding.

Returns: *Identity* information for the requested identity
Parameters:

Name	Type	Required	Description
`id`	Bytes	Yes	An identity `id`
`prove`	Boolean	No	Set to `true` to receive a proof that contains the requested identity

> **Note**: When requesting proofs, the data requested will be encoded as part of the proof in the response.

** Example Request and Response **

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');
const Identifier = require('@dashevo/dpp/lib/Identifier');
const cbor = require('cbor');
const varint = require('varint');

const client = new DAPIClient();

const identityId = Identifier.from('4EfA9Jrvv3nnCFdSf7fad59851iiTRZ6Wcu6YVJ4iSeF');
client.platform.getIdentity(identityId).then((response) => {
  // Strip off protocol version (leading varint) and decode
```

(continues on next page)

(continued from previous page)

```
  const identityBuffer = Buffer.from(response.getIdentity());
  const protocolVersion = varint.decode(identityBuffer);
  const identity = cbor.decode(
    identityBuffer.slice(varint.encodingLength(protocolVersion), identityBuffer.length),
  );
  console.log(identity);
});
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: { PlatformPromiseClient, GetIdentityRequest },
} = require('@dashevo/dapi-grpc');
const Identifier = require('@dashevo/dpp/lib/Identifier');
const cbor = require('cbor');
const varint = require('varint');

const platformPromiseClient = new PlatformPromiseClient(
  'https://seed-1.testnet.networks.dash.org:1443',
);

const id = Identifier.from('4EfA9Jrvv3nnCFdSf7fad59851iiTRZ6Wcu6YVJ4iSeF');
const idBuffer = Buffer.from(id);
const getIdentityRequest = new GetIdentityRequest();
getIdentityRequest.setId(idBuffer);
getIdentityRequest.setProve(false);

platformPromiseClient.getIdentity(getIdentityRequest)
  .then((response) => {
    // Strip off protocol version (leading varint) and decode
    const identityBuffer = Buffer.from(response.getIdentity());
    const protocolVersion = varint.decode(identityBuffer);
    const decodedIdentity = cbor.decode(
      identityBuffer.slice(varint.encodingLength(protocolVersion), identityBuffer.
↪length),
    );
    console.log(decodedIdentity);
  })
  .catch((e) => console.error(e));
```

SHELL

```
# gRPCurl
# `id` must be represented in base64
grpcurl -proto protos/platform/v0/platform.proto \
  -d '{
    "id":"MBLBm5jsADOt2zbNZLf1EGcPKjUaQwS19plBRChu/aw="
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Platform/getIdentity
```

JSON

```
// Response (JavaScript)
{
  "id": "<Buffer 30 12 c1 9b 98 ec 00 33 ad db 36 cd 64 b7 f5 10 67 0f 2a 35 1a 43 04 b5␣
↪f6 99 41 44 28 6e fd ac>",
  "balance": 5255234422,
  "revision": 0,
  "publicKeys": [
    {
      "id": 0,
      "data": "<Buffer 02 c8 b4 74 7b 52 8c ac 5f dd f7 a6 cc 63 70 2e e0 4e d7 d1 33 29␣
↪04 e0 85 10 34 3e a0 0d ce 54 6a>",
      "type": 0,
      "purpose": 0,
      "readOnly": false,
      "securityLevel": 0
    },
    {
      "id": 1,
      "data": "<Buffer 02 01 ee 28 f8 4f 54 85 39 05 67 e9 39 c2 b5 86 01 0b 63 a6 9e c9␣
↪2c ab 53 5d c9 6a 8c 71 91 36 02>",
      "type": 0,
      "purpose": 0,
      "readOnly": false,
      "securityLevel": 2
    }
  ]
}
```

JSON

```
// Response (gRPCurl)
{
  "identity":
↪"AaRiaWRYIDASwZuY7AAzrds2zWS39RBnDyo1GkMEtfaZQUQobv2sZ2JhbGFuY2UbAAAAATk8g3ZocmV2aXNpb24AanB1YmxpY0tl
↪",
  "metadata": {
    "height": "4217",
```

(continues on next page)

(continued from previous page)

```
    "coreChainLockedHeight": 858833,
    "timeMs": "1688058824358",
    "protocolVersion": 1
  }
}
```

getIdentitiesByPublicKeyHashes

Returns: *Identity* an array of identities associated with the provided public key hashes
Parameters:

Name	Type	Required	Description
public_key_hashes	Bytes	Yes	Public key hashes (sha256-ripemd160) of identity public keys
prove	Boolean	No	Set to true to receive a proof that contains the requested identities

> **Note**: When requesting proofs, the data requested will be encoded as part of the proof in the response.
>
> Public key hash
>
> Note: the hash must be done using all fields of the identity public key object - e.g.

```
{
  "id": 0,
  "type": 0,
  "purpose": 0,
  "securityLevel": 0,
  "data": "A2GTAJk9eAWkMXVCb+rRKXH99POtR5OaW6zqZl7/yozp",
  "readOnly": false
}
```

> When using the js-dpp library, the hash can be accessed via the IdentityPublicKey object's `hash` method (e.g. `identity.getPublicKeyById(0).hash()`).

** Example Request and Response **

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');
const DashPlatformProtocol = require('@dashevo/dpp');

const client = new DAPIClient();
const dpp = new DashPlatformProtocol();

const publicKeyHash = 'b8d1591aa74d440e0af9c0be16c55bbc141847f7';
const publicKeysBuffer = [Buffer.from(publicKeyHash, 'hex')];

dpp.initialize().then(() => {
  client.platform.getIdentitiesByPublicKeyHashes(publicKeysBuffer)
```

(continues on next page)

(continued from previous page)

```
    .then((response) => {
      const retrievedIdentity = dpp.identity.createFromBuffer(response.identities[0]);
      console.log(retrievedIdentity.toJSON());
    });
});
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: { PlatformPromiseClient, GetIdentitiesByPublicKeyHashesRequest },
} = require('@dashevo/dapi-grpc');
const DashPlatformProtocol = require('@dashevo/dpp');

const dpp = new DashPlatformProtocol();

dpp.initialize()
  .then(() => {
    const platformPromiseClient = new PlatformPromiseClient(
      'https://seed-1.testnet.networks.dash.org:1443',
    );

    const publicKeyHash = 'b8d1591aa74d440e0af9c0be16c55bbc141847f7';
    const publicKeysBuffer = [Buffer.from(publicKeyHash, 'hex')];

    const getIdentitiesByPublicKeyHashesRequest = new␣
↪GetIdentitiesByPublicKeyHashesRequest();
    getIdentitiesByPublicKeyHashesRequest.setPublicKeyHashesList(publicKeysBuffer);

    platformPromiseClient.
↪getIdentitiesByPublicKeyHashes(getIdentitiesByPublicKeyHashesRequest)
      .then((response) => {
        const identitiesResponse = response.getIdentitiesList();
             console.log(dpp.identity.createFromBuffer(Buffer.
↪from(identitiesResponse[0])).toJSON());
      })
      .catch((e) => console.error(e));
         });
```

SHELL

```
# gRPCurl
# `public_key_hashes` must be represented in base64
grpcurl -proto protos/platform/v0/platform.proto \
  -d '{
      "public_key_hashes":"uNFZGqdNRA4K+cC+FsVbvBQYR/c="
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Platform/getIdentitiesByPublicKeyHashes
```

JSON

```
// Response (JavaScript)
{
  "protocolVersion": 1,
  "id": "4EfA9Jrvv3nnCFdSf7fad59851iiTRZ6Wcu6YVJ4iSeF",
  "publicKeys": [
    {
      "id": 0,
      "type": 0,
      "purpose": 0,
      "securityLevel": 0,
      "data": "Asi0dHtSjKxf3femzGNwLuBO19EzKQTghRA0PqANzlRq",
      "readOnly": false
    },
    {
      "id": 1,
      "type": 0,
      "purpose": 0,
      "securityLevel": 2,
      "data": "AgHuKPhPVIU5BWfpOcK1hgELY6aeySyrU13JaoxxkTYC",
      "readOnly": false
    }
  ],
  "balance": 5255234422,
  "revision": 0
}
```

JSON

```
// Response (gRPCurl)
{
  "identities": [

↪"AaRiaWRYIDASwZuY7AAzrds2zWS39RBnDyo1GkMEtfaZQUQobv2sZ2JhbGFuY2UbAAAAATk8g3ZocmV2aXNpb24AanB1YmxpY0tl
↪"
  ],
  "metadata": {
    "height": "4216",
    "coreChainLockedHeight": 858832,
    "timeMs": "1688058626337",
    "protocolVersion": 1
  }
}
```

getDataContract

Returns: *Data Contract* information for the requested data contract
Parameters:

Name	Type	Required	Description
`id`	Bytes	Yes	A data contract `id`
`prove`	Boolean	No	Set to `true` to receive a proof that contains the requested data contract

Note: When requesting proofs, the data requested will be encoded as part of the proof in the response.

** Example Request and Response **

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');
const Identifier = require('@dashevo/dpp/lib/Identifier');
const cbor = require('cbor');
const varint = require('varint');

const client = new DAPIClient();

const contractId = Identifier.from('GWRSAVFMjXx8HpQFaNJMqBV7MBgMK4br5UESsB4S31Ec');
client.platform.getDataContract(contractId).then((response) => {
    // Strip off protocol version (leading varint) and decode
    const contractBuffer = Buffer.from(response.getDataContract());
    const protocolVersion = varint.decode(contractBuffer);
    const contract = cbor.decode(
      contractBuffer.slice(varint.encodingLength(protocolVersion), contractBuffer.
↪length),
    );
  console.dir(contract, { depth: 10 });
});
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: { PlatformPromiseClient, GetDataContractRequest },
} = require('@dashevo/dapi-grpc');
const Identifier = require('@dashevo/dpp/lib/Identifier');
const cbor = require('cbor');
const varint = require('varint');

const platformPromiseClient = new PlatformPromiseClient(
  'https://seed-1.testnet.networks.dash.org:1443',
);
```

(continues on next page)

(continued from previous page)

```
const contractId = Identifier.from('GWRSAVFMjXx8HpQFaNJMqBV7MBgMK4br5UESsB4S31Ec');
const contractIdBuffer = Buffer.from(contractId);
const getDataContractRequest = new GetDataContractRequest();
getDataContractRequest.setId(contractIdBuffer);

platformPromiseClient.getDataContract(getDataContractRequest)
  .then((response) => {
    // Strip off protocol version (leading varint) and decode
    const contractBuffer = Buffer.from(response.getDataContract());
    const protocolVersion = varint.decode(contractBuffer);
    const decodedDataContract = cbor.decode(
      contractBuffer.slice(varint.encodingLength(protocolVersion), contractBuffer.
↪length),
    );
    console.dir(decodedDataContract, { depth: 5 });
  })
  .catch((e) => console.error(e));
```

SHELL

```
# gRPCurl
# `id` must be represented in base64
grpcurl -proto protos/platform/v0/platform.proto \
  -d '{
    "id":"5mjGWa9mruHnLBht3ntbfgodcSoJxA1XIfYiv1PFMVU="
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Platform/getDataContract
```

JSON

```
// Response (JavaScript)
{
  "$id": "Buffer(32) [Uint8Array] [
    230, 104, 198,  89, 175, 102, 174, 225,
    231,  44,  24, 109, 222, 123,  91, 126,
     10,  29, 113,  42,   9, 196,  13,  87,
     33, 246,  34, 191,  83, 197,  49,  85
  ]",
  "$schema": "https://schema.dash.org/dpp-0-4-0/meta/data-contract",
  "ownerId": "Buffer(32) [Uint8Array] [
     48,  18, 193, 155, 152, 236,   0,  51,
    173, 219,  54, 205, 100, 183, 245,  16,
    103,  15,  42,  53,  26,  67,   4, 181,
    246, 153,  65,  68,  40, 110, 253, 172
  ]",
  "version": 1,
  "documents": {
    "domain": {
```

(continues on next page)

(continued from previous page)

```
      "type": "object",
      "indices": [
        {
          "name": "parentNameAndLabel",
          "unique": true,
          "properties": [
            { "normalizedParentDomainName": "asc" },
            { "normalizedLabel": "asc" }
          ]
        },
        {
          "name": "dashIdentityId",
          "unique": true,
          "properties": [ { "records.dashUniqueIdentityId": "asc" } ]
        },
        {
          "name": "dashAlias",
          "properties": [ { "records.dashAliasIdentityId": "asc" } ]
        }
      ],
      "$comment": "In order to register a domain you need to create a preorder. The␣
↪preorder step is needed to prevent man-in-the-middle attacks. normalizedLabel + '.' +␣
↪normalizedParentDomain must not be longer than 253 chars length as defined by RFC 1035.
↪ Domain documents are immutable: modification and deletion are restricted",
      "required": [
        "label",
        "normalizedLabel",
        "normalizedParentDomainName",
        "preorderSalt",
        "records",
        "subdomainRules"
      ],
      "properties": {
        "label": {
          "type": "string",
          "pattern": "^[a-zA-Z0-9][a-zA-Z0-9-]{0,61}[a-zA-Z0-9]$",
          "maxLength": 63,
          "minLength": 3,
          "description": "Domain label. e.g. 'Bob'."
        },
        "records": {
          "type": "object",
          "$comment": "Constraint with max and min properties ensure that only one␣
↪identity record is used - either a `dashUniqueIdentityId` or a `dashAliasIdentityId`",
          "properties": {
            "dashAliasIdentityId": {
              "type": "array",
              "$comment": "Must be equal to the document owner",
              "maxItems": 32,
              "minItems": 32,
              "byteArray": true,
              "description": "Identity ID to be used to create alias names for the␣
```

(continues on next page)

(continued from previous page)

```
↪Identity",
              "contentMediaType": "application/x.dash.dpp.identifier"
            },
            "dashUniqueIdentityId": {
              "type": "array",
              "$comment": "Must be equal to the document owner",
              "maxItems": 32,
              "minItems": 32,
              "byteArray": true,
              "description": "Identity ID to be used to create the primary name the␣
↪Identity",
              "contentMediaType": "application/x.dash.dpp.identifier"
            }
          },
          "maxProperties": 1,
          "minProperties": 1,
          "additionalProperties": false
        },
        "preorderSalt": {
          "type": "array",
          "maxItems": 32,
          "minItems": 32,
          "byteArray": true,
          "description": "Salt used in the preorder document"
        },
        "subdomainRules": {
          "type": "object",
          "required": [ "allowSubdomains" ],
          "properties": {
            "allowSubdomains": {
              "type": "boolean",
              "$comment": "Only the domain owner is allowed to create subdomains for non␣
↪top-level domains",
              "description": "This option defines who can create subdomains: true -␣
↪anyone; false - only the domain owner"
            }
          },
          "description": "Subdomain rules allow domain owners to define rules for␣
↪subdomains",
          "additionalProperties": false
        },
        "normalizedLabel": {
          "type": "string",
          "pattern": "^[a-z0-9][a-z0-9-]{0,61}[a-z0-9]$",
          "$comment": "Must be equal to the label in lowercase. This property will be␣
↪deprecated due to case insensitive indices",
          "maxLength": 63,
          "description": "Domain label in lowercase for case-insensitive uniqueness␣
↪validation. e.g. 'bob'"
        },
        "normalizedParentDomainName": {
          "type": "string",
```

(continues on next page)

(continued from previous page)

```
          "pattern": "^$|^[a-z0-9][a-z0-9-\\.]{0,61}[a-z0-9]$",
          "$comment": "Must either be equal to an existing domain or empty to create a␣
↪top level domain. Only the data contract owner can create top level domains.",
          "maxLength": 63,
          "minLength": 0,
          "description": "A full parent domain name in lowercase for case-insensitive␣
↪uniqueness validation. e.g. 'dash'"
        }
      },
      "additionalProperties": false
    },
    "preorder": {
      "type": "object",
      "indices": [
        {
          "name": "saltedHash",
          "unique": true,
          "properties": [ { "saltedDomainHash": "asc" } ]
        }
      ],
      "$comment": "Preorder documents are immutable: modification and deletion are␣
↪restricted",
      "required": [ "saltedDomainHash" ],
      "properties": {
        "saltedDomainHash": {
          "type": "array",
          "maxItems": 32,
          "minItems": 32,
          "byteArray": true,
          "description": "Double sha-256 of the concatenation of a 32 byte random salt␣
↪and a normalized domain name"
        }
      },
      "additionalProperties": false
    }
  }
}
```

JSON

```
// Response (gRPCurl)
{
  "dataContract": "AaVjJGlkWCDmaMZZr2au4ecsGG3ee1t+Ch1xKgnEDVch9iK/
↪U8UxVWckc2NoZW1heDRodHRwczovL3NjaGVtYS5kYXNoLm9yZy9kcHAtMC00LTAvbWV0YS9kYXRhLWNvbnRyYWN0Z293bmVySWRYI
↪RscHJlb3JkZXJTYWx0pWR0eXBlZWFycmF5aG1heEl0ZW1zGCBobWluSXRlbXMYIGlieXRlQXJyYXn1a2Rlc2NyaXB0aW9ueCJTYWx
↪aW1pbkxlbmd0aABrZGVzY3JpcHRpb254XkEgZnVsbCBwYXJlbnQgZG9tYWluIG5hbWUgaW4gbG93ZXJjYXNlIGZvciBjYXNlLWluc
↪Q=",
  "metadata": {
    "height": "4253",
    "coreChainLockedHeight": 889435,
```

(continues on next page)

(continued from previous page)

```
    "timeMs": "1684440772828",
    "protocolVersion": 1
  }
}
```

getDocuments

Returns: *Document* information for the requested document(s)
Parameters:

- Parameter constraints

The `where`, `order_by`, `limit`, `start_at`, and `start_after` parameters must comply with the limits defined on the *Query Syntax* page.

Additionally, note that `where` and `order_by` must be CBOR encoded.

Name	Type	Required	Description
`data_contract_id`	Bytes	Yes	A data contract `id`
`document_type`	String	Yes	A document type defined by the data contract (e.g. `preorder` or `domain` for the DPNS contract)
`where` *	Bytes	No	Where clause to filter the results (**must be CBOR encoded**)
`order_by` *	Bytes	No	Sort records by the field(s) provided (**must be CBOR encoded**)
`limit`	Integer	No	Maximum number of results to return
——————-			
One of the following:			
`start_at`	Integer	No	Return records beginning with the index provided
`start_after`	Integer	No	Return records beginning after the index provided
——————-			
`prove`	Boolean	No	Set to `true` to receive a proof that contains the requested document(s)

Note: When requesting proofs, the data requested will be encoded as part of the proof in the response.

** Example Request and Response **

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');
const Identifier = require('@dashevo/dpp/lib/Identifier');
const cbor = require('cbor');
const varint = require('varint');

const client = new DAPIClient();

const contractId = Identifier.from('GWRSAVFMjXx8HpQFaNJMqBV7MBgMK4br5UESsB4S31Ec');
client.platform.getDocuments(contractId, 'domain', { limit: 10 }).then((response) => {
  for (const rawData of response.documents) {
    // Strip off protocol version (leading varint) and decode
    const documentBuffer = Buffer.from(rawData);
    const protocolVersion = varint.decode(documentBuffer);
    const document = cbor.decode(
      documentBuffer.slice(varint.encodingLength(protocolVersion), documentBuffer.
↪length),
    );
    console.log(document);
  }
});
```

JAVASCRIPT

```
// JavaScript (dapi-grpc)
const {
  v0: { PlatformPromiseClient, GetDocumentsRequest },
} = require('@dashevo/dapi-grpc');
const cbor = require('cbor');
const Identifier = require('@dashevo/dpp/lib/Identifier');
const varint = require('varint');

const platformPromiseClient = new PlatformPromiseClient(
  'https://seed-1.testnet.networks.dash.org:1443',
);

const contractId = Identifier.from('GWRSAVFMjXx8HpQFaNJMqBV7MBgMK4br5UESsB4S31Ec');
const contractIdBuffer = Buffer.from(contractId);
const getDocumentsRequest = new GetDocumentsRequest();
const type = 'domain';
const limit = 10;

getDocumentsRequest.setDataContractId(contractIdBuffer);
getDocumentsRequest.setDocumentType(type);
// getDocumentsRequest.setWhere(whereSerialized);
// getDocumentsRequest.setOrderBy(orderBySerialized);
getDocumentsRequest.setLimit(limit);
// getDocumentsRequest.setStartAfter(startAfter);
// getDocumentsRequest.setStartAt(startAt);
```

(continues on next page)

(continued from previous page)

```
platformPromiseClient.getDocuments(getDocumentsRequest)
  .then((response) => {
    for (const document of response.getDocumentsList()) {
      // Strip off protocol version (leading varint) and decode
      const documentBuffer = Buffer.from(document);
      const protocolVersion = varint.decode(documentBuffer);
      const decodedDocument = cbor.decode(
        documentBuffer.slice(varint.encodingLength(protocolVersion), documentBuffer.
↪length),
      );
      console.log(decodedDocument);
    }
  })
  .catch((e) => console.error(e));
```

SHELL

```
# gRPCurl
# Request documents
# `id` must be represented in base64
grpcurl -proto protos/platform/v0/platform.proto \
  -d '{
    "data_contract_id":"5mjGWa9mruHnLBht3ntbfgodcSoJxA1XIfYiv1PFMVU=",
    "document_type":"domain",
    "limit":1
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Platform/getDocuments
```

JSON

```
// Response (JavaScript)
{
  "$id": "<Buffer 01 a0 7c 69 43 82 cf fe 93 97 be c9 f4 be cd 67 81 8f 60 d2 a7 56 48␣
↪08 11 80 49 84 0b 2e 2c 5d>",
  "$type": "domain",
  "label": "Dash01",
  "records": {
    "dashUniqueIdentityId": "<Buffer f5 50 ed 37 1a 12 3f 54 00 59 31 84 f7 f7 37 f1 f4␣
↪b1 5d 05 6f 9c a8 0e 5f 00 52 82 08 77 7c 4a>"
  },
  "$ownerId": "<Buffer f5 50 ed 37 1a 12 3f 54 00 59 31 84 f7 f7 37 f1 f4 b1 5d 05 6f 9c␣
↪a8 0e 5f 00 52 82 08 77 7c 4a>",
  "$revision": 1,
  "preorderSalt": "<Buffer 2c b4 1b e9 f4 40 03 9b 47 2f 31 74 46 df 7f 4f 43 fe 14 80␣
↪be ca 84 0d 63 0f a6 65 23 b9 9c a1>",
  "subdomainRules": { "allowSubdomains": false },
  "$dataContractId": "<Buffer e6 68 c6 59 af 66 ae e1 e7 2c 18 6d de 7b 5b 7e 0a 1d 71␣
```

(continues on next page)

(continued from previous page)

```
↪2a 09 c4 0d 57 21 f6 22 bf 53 c5 31 55>",
  "normalizedLabel": "dash01",
  "normalizedParentDomainName": "dash"
}
```

JSON

```
// Response (gRPCurl)
{
  "documents": [

↪"AatjJGlkWCACod79ik2tILNnybx5VepoaX2cceXDSogwSgxdWi9zYmUkdHlwZWZkb21haW5lbGFiZWx0Yzg4OWMyM2FiY2ZkYzU3
↪axoJG93bmVySWRYIDASwZuY7AAzrds2zWS39RBnDyo1GkMEtfaZQUQobv2saSRyZXZpc2lvbgFscHJlb3JkZXJTYWx0WCAkJyav6i
↪NdvejXt6aSG5zdWJkb21haW5SdWxlc6FvYWxsb3dTdWJkb21haW5z9W8kZGF0YUNvbnRyYWN0SWRYIOZoxlmvZq7h5ywYbd57W34K
↪"
  ],
  "metadata": {
    "height": "4254",
    "coreChainLockedHeight": 889435,
    "timeMs": "1684440970270",
    "protocolVersion": 1
  }
}
```

waitForStateTransitionResult

Returns: The state transition hash and either a proof that the state transition was confirmed in a block or an error.
Parameters:

Name	Type	Required	Description
`state_transition_hash`	Bytes	Yes	Hash of the state transition
`prove`	Boolean	Yes	Set to `true` to request a proof

Note: When requesting proofs, the data requested will be encoded as part of the proof in the response.

** Example Request**

JAVASCRIPT

```
// JavaScript (dapi-client)
const DAPIClient = require('@dashevo/dapi-client');

const client = new DAPIClient();

// Replace <YOUR_STATE_TRANSITION_HASH> with your actual hash
const hash = <YOUR_STATE_TRANSITION_HASH>;
client.platform.waitForStateTransitionResult(hash, { prove: true })
```

(continues on next page)

(continued from previous page)

```
  .then((response) => {
    console.log(response);
  });
```

SHELL

```
# gRPCurl
# Replace `your_state_transition_hash` with your own before running
# `your_state_transition_hash` must be represented in base64
#    Example: wEiwFu9WvAtylrwTph5v0uXQm743N+75C+C9DhmZBkw=
grpcurl -proto protos/platform/v0/platform.proto \
  -d '{
    "state_transition_hash":your_state_transition_hash,
    "prove": "true"
    }' \
  seed-1.testnet.networks.dash.org:1443 \
  org.dash.platform.dapi.v0.Platform/waitForStateTransitionResult
```

Deprecated Endpoints

No endpoints were deprecated in Dash Platform v0.24, but the previous version of documentation can be viewed here.

Code Reference

Implementation details related to the information on this page can be found in:

- The Platform repository `packages/dapi/lib/grpcServer/handlers/core` folder
- The Platform repository `packages/dapi-grpc/protos` folder

1.27 Query Syntax

1.27.1 Overview

Generally queries will consist of a `where` clause plus optional *modifiers* controlling the specific subset of results returned.

> Query limitations
>
> Dash Platform v0.22 introduced a number of limitations due to the switch to using GroveDB. See details in pull requests 77 and 230 that implemented these changes.
>
> Query validation details may be found here along with the associated validation tests.

1.27.2 Where Clause

The Where clause must be a non-empty array containing not more than 10 conditions. For some operators, `value` will be an array. See the following general syntax example:

> As of Dash Platform v0.22, *all fields* referenced in a query's where clause must be defined in the *same index*. This includes `$createdAt` and `$updatedAt`.

```
{
  where: [
    [<fieldName>, <operator>, <value>],
    [<fieldName>, <array operator>, [<value1>, <value2>]]
  ]
}
```

Fields

Valid fields consist of the indices defined for the document being queried. For example, the DPNS data contract defines three indices:

Index Field(s)	Index Type	Unique
normalizedParentDomainName, normalizedLabel	Compound	Yes
records.dashUniqueIdentityId	Single Field	Yes
records.dashAliasIdentityId	Single Field	No

Comparison Operators

Equal

Name	Description
==	Matches values that are equal to a specified value

Range

> Dash Platform v0.22 notes
>
> - Only one range operator is allowed in a query (except for between behavior)
> - The `in` operator is only allowed for last two indexed properties
> - Range operators are only allowed after == and `in` operators
> - Range operators are only allowed for the last two fields used in the where condition
> - Queries using range operators must also include an `orderBy` statement

Name	Description
<	Matches values that are less than a specified value
<=	Matches values that are less than or equal to a specified value
>=	Matches values that are greater than or equal to a specified value
>	Matches values that are greater than a specified value
in	Matches all document(s) where the value of the field equals any value in the specified array Array may include up to 100 (unique) elements

Array Operators

Name	Description
length	**Not available in Dash Platform v0.22**Selects documents if the array field is a specified size (integer)
contains	**Not available in Dash Platform v0.22**- Matches arrays that contain all elements specified in the query condition array - 100 element maximum
elementMatch	**Not available in Dash Platform v0.22**- Matches documents that contain an array field with at least one element that matches all the criteria in the query condition array - Two or more conditions must be provided

Evaluation Operators

Name	Description
startsWith	Selects documents where the value of a field begins with the specified characters (string, <= 255 characters). Must include an `orderBy` statement.

Operator Examples

```
{
  where: [
    ['nameHash', '<', '56116861626961756e6176657a382e64617368'],
  ],
}
```

```
{
  where: [
      ['normalizedParentDomainName', '==', 'dash'],
      // Return all matching names from the provided array
      ['normalizedLabel', 'in', ['alice', 'bob']],
    ]
}
```

```
{
  where: [
      ['normalizedParentDomainName', '==', 'dash'],
      // Return any names beginning with "al" (e.g. alice, alfred)
      ['normalizedLabel', 'startsWith', 'al'],
    ]
}
```

```
// Not available in Dash Platform v0.22
// See https://github.com/dashevo/platform/pull/77
{
  where: [
      // Return documents that have 5 values in their `items` array
      ['items', 'length', 5],
    ]
}
```

```
// Not available in Dash Platform v0.22
// See https://github.com/dashevo/platform/pull/77
{
  where: [
      // Return documents that have both "red" and "blue"
      // in the `colors` array
      ['colors', 'contains', ['red', 'blue']],
    ]
}
```

```
// Not available in Dash Platform v0.22
// See https://github.com/dashevo/platform/pull/77
{
  where: [
    // Return `scores` documents where the results contain
    // elements in the range 80-90
    ['scores', 'elementMatch',
      [
        ['results', '>=', '80'],
        ['results', '<=', '90']
      ],
    ],
  ]
}
```

1.27.3 Query Modifiers

The query modifiers described here determine how query results will be sorted and what subset of data matching the query will be returned.

> Breaking changes
>
> Starting with Dash Platform v0.22, `startAt` and `startAfter` must be provided with a document ID rather than an integer.

Modifier	Effect	Example
`limit`	Restricts the number of results returned (maximum: 100)	`limit: 10`
`orderBy`	Returns records sorted by the field(s) provided (maximum: 2). Sorting must be by the last indexed property. Can only be used with >, <, >=, <=, and `startsWith` queries.	`orderBy: [['normalizedLabel', 'asc']]`
`startAt`	Returns records beginning with the document ID provided	`startAt: Buffer.from(Identifier.from(<document ID>))`
`startAfter`	Returns records beginning after the document ID provided	`startAfter: Buffer.from(Identifier.from(<document ID>))`

Compound Index Constraints

For indices composed of multiple fields (example from the DPNS data contract), the sort order in an `orderBy` must either match the order defined in the data contract OR be the inverse order.

Please refer to pull request 230 for additional information related to compound index constraints in Platform v0.22.

1.27.4 Example query

The following query combines both a where clause and query modifiers.

```
import Dash from "dash"

const { Essentials: { Buffer }, PlatformProtocol: { Identifier } } = Dash;

const query = {
  limit: 5,
  startAt: Buffer.from(Identifier.from('4Qp3menV9QjE92hc3BzkUCusAmHLxh1AU6gsVsPF4L2q')),
  where: [
    ['normalizedParentDomainName', '==', 'dash'],
    ['normalizedLabel', 'startsWith', 'test'],
  ],
  orderBy: [
    ['normalizedLabel', 'asc'],
  ],
}
```

1.28 Data Contracts

1.28.1 Overview

Data contracts define the schema (structure) of data an application will store on Dash Platform. Contracts are described using JSON Schema which allows the platform to validate the contract-related data submitted to it.

The following sections provide details that developers need to construct valid contracts: *documents* and *definitions*. All data contracts must define one or more documents, whereas definitions are optional and may not be used for simple contracts.

1.28.2 Documents

The `documents` object defines each type of document required by the data contract. At a minimum, a document must consist of 1 or more properties. Documents may also define *indices* and a list of *required properties*. The `additionalProperties` properties keyword must be included as described in the *constraints* section.

The following example shows a minimal `documents` object defining a single document (`note`) that has one property (`message`).

```
{
  "note": {
    "properties": {
      "message": {
        "type": "string"
      }
    },
    "additionalProperties": false
  }
}
```

Document Properties

The `properties` object defines each field that will be used by a document. Each field consists of an object that, at a minimum, must define its data `type` (`string`, `number`, `integer`, `boolean`, `array`, `object`).

Fields may also apply a variety of optional JSON Schema constraints related to the format, range, length, etc. of the data. A full explanation of the capabilities of JSON Schema is beyond the scope of this document. For more information regarding its data types and the constraints that can be applied, please refer to the JSON Schema reference documentation.

Special requirements for `object` properties

The `object` type is required to have properties defined either directly or via the data contract's *$defs*. For example, the `body` property shown below is an object containing a single string property (`objectProperty`):

```
const contractDocuments = {
  message: {
    type: "object",
    properties: {
      body: {
        type: "object",
       properties: {
          objectProperty: {
            type: "string"
          },
        },
        additionalProperties: false,
      },
      header: {
        type: "string"
      }
    },
```

(continues on next page)

(continued from previous page)

```
    additionalProperties: false
  }
};
```

Property Constraints

There are a variety of constraints currently defined for performance and security reasons.

Description	Value
Minimum number of properties	1
Maximum number of properties	100
Minimum property name length	1 (Note: minimum length was 3 prior to v0.23)
Maximum property name length	64
Property name characters	Alphanumeric (`A-Z`, `a-z`, `0-9`)Hyphen (`-`) Underscore (`_`)

Prior to Dash Platform v0.23 there were stricter limitations on minimum property name length and the characters that could be used in property names.

Required Properties (Optional)

Each document may have some fields that are required for the document to be valid and other fields that are optional. Required fields are defined via the `required` array which consists of a list of the field names from the document that must be present. The `required` object should be excluded for documents without any required properties.

```
"required": [
  "<field name a>",
  "<field name b>"
]
```

Example
The following example (excerpt from the DPNS contract's `domain` document) demonstrates a document that has 6 required fields:

```
"required": [
  "nameHash",
  "label",
  "normalizedLabel",
  "normalizedParentDomainName",
  "preorderSalt",
  "records"
],
```

Document Indices

Document indices may be defined if indexing on document fields is required. The `indices` object should be excluded for documents that do not require indices.

The `indices` array consists of:

- One or more objects that each contain:
  - A unique `name` for the index
  - A `properties` array composed of a <`field name: sort order`> object for each document field that is part of the index (sort order: `asc` only for Dash Platform v0.23)
  - An (optional) `unique` element that determines if duplicate values are allowed for the document

Compound Indices

When defining an index with multiple properties (i.e a compound index), the order in which the properties are listed is important. Refer to the mongoDB documentation for details regarding the significance of the order as it relates to querying capabilities. Dash uses GroveDB which works similarly but does requiring listing all the index's fields in query order by statements.

```
"indices": [
  {
    "properties": [
      { "<field name a>": "<asc"|"desc>" },
      { "<field name b>": "<asc"|"desc>" }
    ],
    "unique": true|false
  },
  {
    "properties": [
      { "<field name c>": "<asc"|"desc>" },
    ],
  }
]
```

Index Constraints

For performance and security reasons, indices have the following constraints. These constraints are subject to change over time.

Description	Value
Minimum/maximum length of index `name`	1 / 32
Maximum number of indices	10
Maximum number of unique indices	3
Maximum number of properties in a single index	10
Maximum length of indexed string property	63
Note: Dash Platform v0.22+. does not allow indices for arrays Maximum length of indexed byte array property	255
Note: Dash Platform v0.22+. does not allow indices for arrays Maximum number of indexed array items	1024
Usage of `$id` in an index disallowed	N/A

Example

The following example (excerpt from the DPNS contract's `preorder` document) creates an index on `saltedDomainHash` that also enforces uniqueness across all documents of that type:

```
"indices": [
  {
    "properties": [
      { "saltedDomainHash": "asc" }
    ],
    "unique": true
  }
],
```

Full Document Syntax

This example syntax shows the structure of a documents object that defines two documents, an index, and a required field.

```
{
  "<document name a>": {
    "type": "object",
    "properties": {
      "<field name b>": {
        "type": "<field data type>"
      },
      "<field name c>": {
        "type": "<field data type>"
      },
    },
    "indices": [
      {
        "name": "<index name>",
        "properties": [
          {
            "<field name c>": "asc"
          }
        ],
        "unique": true|false
      },
    ],
    "required": [
      "<field name c>"
    ]
    "additionalProperties": false
  },
  "<document name x>": {
    "type": "object",
    "properties": {
      "<property name y>": {
        "type": "<property data type>"
      },
      "<property name z>": {
        "type": "<property data type>"
```

(continues on next page)

(continued from previous page)

```
      },
    },
    "additionalProperties": false
  },
}
```

1.28.3 Definitions

> Definitions are currently unavailable

The optional `$defs` object enables definition of aspects of a schema that are used in multiple places. This is done using the JSON Schema support for reuse.

Items defined in `$defs` may then be referenced when defining `documents` through use of the `$ref` keyword. Properties defined in the `$defs` object must meet the same criteria as those defined in the `documents` object. Data contracts can only use the `$ref` keyword to reference their own `$defs`. Referencing external definitions is not supported by the platform protocol.

Example
The following example shows a definition for a `message` object consisting of two properties:

```
{
  // Preceeding content truncated ...
  "$defs": {
    "message": {
      "type": "object",
      "properties": {
        "timestamp": {
          "type": "number"
        },
        "description": {
          "type": "string"
        }
      },
      "additionalProperties": false
    }
  }
}
```

General Constraints

There are a variety of constraints currently defined for performance and security reasons. The following constraints are applicable to all aspects of data contracts. Unless otherwise noted, these constraints are defined in the platform's JSON Schema rules (e.g. rs-dpp data contract meta schema).

Keyword

The `$ref` keyword has been disabled since Platform v0.22.

Keyword	Constraint
`default`	Restricted - cannot be used (defined in DPP logic)
`propertyNames`	Restricted - cannot be used (defined in DPP logic)
`uniqueItems: true`	`maxItems` must be defined (maximum: 100000)
`pattern: <something>`	`maxLength` must be defined (maximum: 50000)
`format: <something>`	`maxLength` must be defined (maximum: 50000)
`$ref: <something>`	**Temporarily disabled** `$ref` can only reference `$defs` - remote references not supported
`if`, `then`, `else`, `allOf`, `anyOf`, `oneOf`, `not`	Disabled for data contracts
`dependencies`	Not supported. Use `dependentRequired` and `dependentSchema` instead
`additionalItems`	Not supported. Use `items: false` and `prefixItems` instead
`patternProperties`	Restricted - cannot be used for data contracts
`pattern`	Accept only RE2 compatible regular expressions (defined in DPP logic)

Data Size

Note: These constraints are defined in the Dash Platform Protocol logic (not in JSON Schema).

All serialized data (including state transitions) is limited to a maximum size of 16 KB.

Additional Properties

Although JSON Schema allows additional, undefined properties by default, they are not allowed in Dash Platform data contracts. Data contract validation will fail if they are not explicitly forbidden using the `additionalProperties` keyword anywhere `properties` are defined (including within document properties of type `object`).

Include the following at the same level as the `properties` keyword to ensure proper validation:

```
"additionalProperties": false
```

1.29 Glossary

1.29.1 Application

The combination of Application Identity, Data Contract, and Application State that together represent a Dash Platform Application

1.29.2 Application State

The collection of documents created by users during their use of an application

1.29.3 Block

One or more transactions prefaced by a block header and protected by proof of work. Blocks are the data stored on the *core blockchain*

1.29.4 Block Reward

The amount that miners may claim as a reward for creating a block. Equal to the sum of the block subsidy (newly available duffs) plus the transactions fees paid by transactions included in the block

1.29.5 ChainLock

Defined in DIP8, ChainLocks are a method of using an LLMQ to threshold sign a block immediately after it is propagated by the miner in order to enforce the first-seen rule. This powerful method of mitigating 51% mining attacks results in near-instant consensus on the valid chain.

1.29.6 Classical Transactions

Standard Dash transactions moving Dash on the core blockchain ledger

1.29.7 Coinbase Transaction

The first transaction in a block. Always created by a miner, it includes a single coinbase.

1.29.8 Core Chain

Layer 1 blockchain used for payments, governance, and providing the foundation for tier 2 masternode infrastructure (LLMQs, DML, PoSe, etc.)

1.29.9 Credits

Means of paying fees on the layer 2 platform

1.29.10 DAPI

Dash's decentralized API for interacting with the core blockchain (layer 1) and the platform (layer 2)

1.29.11 DAPI Client

An HTTP Client that connects to DAPI to enable users to read and write data to the Dash platform

1.29.12 DashPay

Dash Platform based wallet supporting payments via usernames

1.29.13 DashPay Contact Request

A platform document that defines a one way relationship between a sender and a recipient. It includes an encrypted extended public key which will allow the sender to pay the recipient using addresses that other users have no knowledge of. The sender creates and publishes this document. When two users have both sent contact requests to each other, then each is considered a fully established contact with the other.

1.29.14 DashPay Contact Info

A platform document containing an identity's set of private information related to other identities that are contacts.

1.29.15 DashPay Profile

A platform document containing a set of public information for an identity that includes a display name, a public message (bio/status) and an avatar URL. The display name and avatar help complement the identity's username from DPNS to better visually identify an identity in a user interface. An identity can only have a single DashPay profile.

1.29.16 Dash Core

Layer 1 core blockchain reference client

1.29.17 Data Contract

The database schema a developer submits in order to start using Dash Platform as a back end for their application

1.29.18 Dash Platform Application

A client application that leverages Dash Platform services

1.29.19 Dash Platform Naming Service (DPNS)

A service used to register names on the Dash Platform. Can be extended to work in a DNS-like mode. Implemented as an application on top of the platform that leverages platform capabilities

1.29.20 Dash Platform Protocol (DPP)

Describes data structures and validation rules for the data structures used by the platform (e.g. Data Contract, Document, and State Transition). Data structures are defined using JSON-Schema based format

1.29.21 Decentralized Autonomous Organization (DAO)

An organization where decision making is governed according to a set of rules that is transparent, controlled by organization members, and lacking any central authority. Financial records are tracked using a blockchain, which provides the transparency and trust required by organization members.

1.29.22 Devnet

A development environment in which developers can obtain and spend Dash that has no real-world value on a network that is very similar to the Dash *mainnet*. Multiple independent devnets can coexist without interference. Devnets can be either public or private networks. See the Testing Applications page for a more detailed description of network types.

1.29.23 Direct Settlement Payment Channel (DSPC)

In DashPay, established contacts have address spaces to send and receive from each other. When these are present either in one way or bi-directional we will call this a direct settlement payment channel.

1.29.24 Distributed Key Generation (DKG)

Distributed key generation (DKG) is a cryptographic process in which multiple parties contribute to the calculation of a shared public and private key set. In Dash, DKG is used to generate a BLS key pair for use in a *long-living masternode quorum* (LLMQ) to perform threshold signing on network messages. Further detail can be found in DIP-6 Long-Living Masternode Quorums.

1.29.25 Document

A data entry, similar to a document in a document-oriented database. Represented as a JSON. An atomic entity used by the platform to store the user-submitted data

1.29.26 Drive

Layer 2 platform storage

1.29.27 Layer (1, 2, 3)

- Layer 1: Core blockchain and *Dash Core*
- Layer2: Drive and DAPI
- Layer 3: DAPI clients

1.29.28 Local network

A configuration unique to dashmate that uses Dash Core's *regtest* network type to create a multi-node network on a single computer. This configuration allows developers to work independently on their own network for testing and development.

1.29.29 Long Living Masternode Quorum (LLMQ)

Deterministic subset of the global deterministic masternode list used to perform threshold signing of consensus-related messages

1.29.30 Mainnet

The original and main network for Dash transactions, where transaction have real economic value.

1.29.31 Masternode

2nd-tier collateralized Node in the Dash P2P network, performing additional functions and forming a provision layer

1.29.32 Platform Chain

Layer 2 blockchain that propagates platform data among masternodes, propagates platform blocks among masternodes, applies Layer 2 consensus, authoritatively orders state transitions, and controls platform state consistency

1.29.33 Platform State

All layer 2 data including contracts, documents (user data), credit balance, identity (username)

1.29.34 practical Byzantine Fault Tolerance (pBFT)

A consensus algorithm designed to work efficiently in asynchronous environments while assuming the presence of adversarial actors. Advantages of pBFT include energy efficiency, transaction finality, and low reward variance.

1.29.35 Proof of Service (PoSe)

Ability to trustlessly prove that a *masternode* provided the required service to the network in order to earn a reward

1.29.36 Proof of Work (PoW)

Ability to trustlessly prove that a node completed a certain amount of work during the process of confirming a new block to the blockchain.

1.29.37 Quorum

Group of masternodes signing some action, formation of the group determined by via some determination algorithm

1.29.38 Quorum Signature

BLS signature resulting from some agreement within a masternode quorum

1.29.39 Regtest

A local regression testing environment in which developers can almost instantly generate blocks on demand for testing events, and can create private Dash with no real-world value. See the Testing Applications page for a more detailed description of network types.

1.29.40 Simple Payment Verification

A method for verifying if transactions are part of a block without downloading the whole block. This is useful for lightweight clients which don't run continuously and which don't have the storage space or bandwidth for a full copy of the blockchain.

1.29.41 Special Transactions

Transactions containing an extra payload using the format defined by DIP-2

1.29.42 State Machine

The application that validates state transitions and updates state in Drive

1.29.43 State Transition

The change a user does to the application and platforms states. Consists of an array of documents *or* one data contract, the id of the application to which the change is made, and a user signature

1.29.44 Tenderdash

Dash fork of Tendermint modified for use in Dash Platform. See *Platform Consensus* for more information.

1.29.45 Testnet

A global testing environment in which developers can obtain and spend Dash that has no real-world value on a network that is very similar to the Dash *mainnet*. See the Testing Applications page for a more detailed description of network types.

See: *Intro to Testnet* for more information

1.29.46 Validator Set

The group of masternodes responsible for the layer 2 blockchain (platform chain) consensus at a given time. They vote on the content of each platform chain block and are analogous to miners on the layer 1's core blockchain

1.30 Frequently Asked Questions

1.30.1 What is Evolution?

"Evolution" is a codename used to reference various products. It includes "Dash Platform," a FireBase-like platform for developing backends for websites and applications, hosted on the masternode network.

Also, the term " Evolution" refers to several other products that we are going to develop on top of the platform. An example of such an app is DashPay - an easy to use payment solution with usernames and contact lists.

1.30.2 How does a DAPI client discover the IP address of masternodes hosting DAPI endpoints?

The DNS seed will provide a deterministic masternode list (DML) to the client. More on the deterministic MN list can be found here:

- DML spec: https://github.com/dashpay/dips/blob/master/dip-0003.md
- DML verification: https://github.com/dashpay/dips/blob/master/dip-0004.md

1.30.3 Why can't I connect to DAPI from a page served over HTTPS?

Modern browsers block connections to insecure content when the main page is loaded securely. At the moment, there are technical obstacles to serving DAPI content over HTTPS. Until then, the only way to test DAPI from a web page is to serve the web page insecurely. Dash Core team is evaluating different ways to work around this browser restriction and have a trustworthy connection to DAPI.

1.30.4 Will it be possible to use apps with only an identity, or will a DPNS name have to be registered first?

Apps can interact with an identity whether or not it has a DPNS name registered. Someone may create an app that requires one, but it's not a platform restriction.

1.30.5 Should it be possible to create multiple identities using a single private key?

It may not be a very good practice, but this is not restricted.

1.30.6 Will DAPI RPCs always be free? How will DoS attacks be mitigated?

Right now there's only IP based rate limits. Generally Core team wants platform data to be available for everyone, so there are no plans today to have paid queries.

1.30.7 When I try to load the Dash javascript library, why is there is a syntax error "Invalid regular expression"?

This can be caused by loading the script with the wrong character encoding. The `dash` npm package uses UTF-8 encoding. Try this: `<script src="https://unpkg.com/dash" encoding="UTF-8"></script>`

1.31 Overview

1.31.1 Introduction

The Dash Platform Protocol (DPP) defines a protocol for the data objects (e.g. *identities*, data contracts, documents, state transitions) that can be stored on *Dash's layer 2 platform*. All data stored on Dash Platform is governed by DPP to ensure data consistency and integrity is maintained.

Dash Platform data objects consist of JSON and are validated using the JSON Schema specification via pre-defined JSON Schemas and meta-schemas described in these sections. The meta-schemas allow for creation of DPP-compliant schemas which define fields for third-party Dash Platform applications.

In addition to ensuring data complies with predefined JSON Schemas, DPP also defines rules for hashing and serialization of these objects.

1.31.2 Reference Implementation

The current reference implementation is the (Rust) rs-dpp library. The schemas and meta-schemas referred to in this specification can be found here in the reference implementation: https://github.com/dashpay/platform/tree/master/packages/rs-dpp/src/schema.

1.31.3 Release Notes

Release notes for past versions are located on the dashpay/platform GitHub release page. They provide information about breaking changes, features, and fixes.

1.31.4 Topics

1.32 Identity

1.32.1 Identity Overview

Identities are a low-level construct that provide the foundation for user-facing functionality on the platform. An identity is a public key (or set of public keys) recorded on the platform chain that can be used to prove ownership of data. Please see the Identity DIP for additional information.

Identities consist of three components that are described in further detail in the following sections:

Field	Type	Description
protocolVersion	integer	The protocol version
id	array of bytes	The identity id (32 bytes)
publicKeys	array of keys	Public key(s) associated with the identity
balance	integer	Credit balance associated with the identity
revision	integer	Identity update revision

Each identity must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "id": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "publicKeys": {
      "type": "array",
      "minItems": 1,
      "maxItems": 32,
      "uniqueItems": true
    },
    "balance": {
      "type": "integer",
      "minimum": 0
    },
    "revision": {
      "type": "integer",
      "minimum": 0,
      "description": "Identity update revision"
  }
},
  "required": [
    "protocolVersion",
    "id",
    "publicKeys",
    "balance",
    "revision"
  ]
}
```

Example Identity

```
{
  "protocolVersion":1,
```

(continues on next page)

(continued from previous page)


```
  "id":"6YfP6tT9AK8HPVXMK7CQrhpc8VMg7frjEnXinSPvUmZC",
  "publicKeys":[
    {
      "id":0,
      "type":0,
      "purpose":0,
      "securityLevel":0,
      "data":"AkWRfl3DJiyyy6YPUDQnNx5KERRnR8CoTiFUvfdaYSDS",
      "readOnly":false
    }
  ],
  "balance":0,
  "revision":0
}
```


Identity id

The identity `id` is calculated by Base58 encoding the double sha256 hash of the outpoint used to fund the identity creation.

`id = base58(sha256(sha256(<identity create funding output>)))`

Note: The identity `id` uses the Dash Platform specific `application/x.dash.dpp.identifier` content media type. For additional information, please refer to the js-dpp PR 252 that introduced it and identifier.rs.

Identity publicKeys

The identity `publicKeys` array stores information regarding each public key associated with the identity. Multiple identities may use the same public key.

Note: Since v0.23, each identity must have at least two public keys: a primary key (security level `0`) that is only used when updating the identity and an additional one (security level `2`) used to sign state transitions.

Each item in the `publicKeys` array consists of an object containing:

Field	Type	Description
id	integer	The key id (all public keys must be unique)
type	integer	Type of key (default: 0 - ECDSA)
data	array of bytes	Public key (0 - ECDSA: 33 bytes, 1 - BLS: 48 bytes, 2 - ECDSA Hash160: 20 bytes, 3 - BIP13 Hash160: 20 bytes)
purpose	integer	Public key purpose (0 - Authentication, 1 - Encryption, 2 - Decryption, 3 - Withdraw)
securityLevel	integer	Public key security level (0 - Master, 1 - Critical, 2 - High, 3 - Medium)
readonly	boolean	Identity public key can't be modified with `readOnly` set to `true`. This can't be changed after adding a key.
disabledAt	integer	Timestamp indicating that the key was disabled at a specified time

Keys for some purposes must meet certain security level criteria as detailed below:

Key Purpose	Allowed Security Level(s)
Authentication	Any security level
Encryption	Medium
Decryption	Medium
Withdraw	Critical

Each identity public key must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "id": {
      "type": "integer",
      "minimum": 0,
      "description": "Public key ID",
      "$comment": "Must be unique for the identity. It can't be changed after adding a␣
↪key. Included when signing state transitions to indicate which identity key was used␣
↪to sign."
    },
    "type": {
      "type": "integer",
      "enum": [
        0,
        1,
        2,
        3
      ],
      "description": "Public key type. 0 - ECDSA Secp256k1, 1 - BLS 12-381, 2 - ECDSA␣
↪Secp256k1 Hash160, 3 - BIP 13 Hash160",
      "$comment": "It can't be changed after adding a key"
    },
    "purpose": {
      "type": "integer",
      "enum": [
        0,
        1,
        2,
        3
      ],
      "description": "Public key purpose. 0 - Authentication, 1 - Encryption, 2 -␣
↪Decryption, 3 - Withdraw",
      "$comment": "It can't be changed after adding a key"
    },
    "securityLevel": {
      "type": "integer",
      "enum": [
        0,
        1,
        2,
        3
      ],
```

(continues on next page)

(continued from previous page)

```
      "description": "Public key security level. 0 - Master, 1 - Critical, 2 - High, 3 -␣
↪Medium",
      "$comment": "It can't be changed after adding a key"
    },
    "data": true,
    "readOnly": {
      "type": "boolean",
      "description": "Read only",
      "$comment": "Identity public key can't be modified with readOnly set to true. It␣
↪can't be changed after adding a key"
    },
    "disabledAt": {
      "type": "integer",
      "description": "Timestamp indicating that the key was disabled at a specified time
↪",
      "minimum": 0
    }
  },
  "allOf": [
    {
      "if": {
        "properties": {
          "type": {
            "const": 0
          }
        }
      },
      "then": {
        "properties": {
          "data": {
            "type": "array",
            "byteArray": true,
            "minItems": 33,
            "maxItems": 33,
            "description": "Raw ECDSA public key",
            "$comment": "It must be a valid key of the specified type and unique for the␣
↪identity. It can't be changed after adding a key"
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "type": {
            "const": 1
          }
        }
      },
      "then": {
        "properties": {
          "data": {
```

(continues on next page)

(continued from previous page)

```
            "type": "array",
            "byteArray": true,
            "minItems": 48,
            "maxItems": 48,
            "description": "Raw BLS public key",
            "$comment": "It must be a valid key of the specified type and unique for the␣
↪identity. It can't be changed after adding a key"
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "type": {
            "const": 2
          }
        }
      },
      "then": {
        "properties": {
          "data": {
            "type": "array",
            "byteArray": true,
            "minItems": 20,
            "maxItems": 20,
            "description": "ECDSA Secp256k1 public key Hash160",
            "$comment": "It must be a valid key hash of the specified type and unique␣
↪for the identity. It can't be changed after adding a key"
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "type": {
            "const": 3
          }
        }
      },
      "then": {
        "properties": {
          "data": {
            "type": "array",
            "byteArray": true,
            "minItems": 20,
            "maxItems": 20,
            "description": "BIP13 script public key",
            "$comment": "It must be a valid script hash of the specified type and unique␣
↪for the identity"
          }
```

(continues on next page)

(continued from previous page)

```
        }
      }
    }
  ],
  "required": [
    "id",
    "type",
    "data",
    "purpose",
    "securityLevel"
  ],
  "additionalProperties": false
}
```

Public Key `id`

Each public key in an identity's `publicKeys` array must be assigned a unique index number (`id`).

Public Key `type`

The `type` field indicates the algorithm used to derive the key.

Type	Description
0	ECDSA Secp256k1 (default)
1	BLS 12-381
2	ECDSA Secp256k1 Hash160
3	BIP13 pay-to-script-hash public key

Public Key `data`

The `data` field contains the compressed public key.

Public Key `purpose`

The `purpose` field describes which operations are supported by the key. Please refer to DIP11 - Identities for additional information regarding this.

Type	Description
0	Authentication
1	Encryption
2	Decryption
3	Withdraw

Public Key `securityLevel`

The `securityLevel` field indicates how securely the key should be stored by clients. Please refer to DIP11 - Identities for additional information regarding this.

Level	Description	Security Practice
0	Master	Should always require a user to authenticate when signing a transition. Can only be used to update an identity.
1	Critical	Should always require a user to authenticate when signing a transition
2	High	Should be available as long as the user has authenticated at least once during a session. Typically used to sign state transitions, but cannot be used for identity update transitions.
3	Medium	Should not require user authentication but must require access to the client device

Public Key `readOnly`

The `readOnly` field indicates that the public key can't be modified if it is set to `true`. The value of this field cannot be changed after adding the key.

Public Key `disabledAt`

The `disabledAt` field indicates that the key has been disabled. Its value equals the timestamp when the key was disabled.

Identity balance

Each identity has a balance of credits established by value locked via a layer 1 lock transaction. This credit balance is used to pay the fees associated with state transitions.

1.32.2 Identity State Transition Details

There are three identity-related state transitions: *identity create*, *identity topup*, and *identity update*. Details are provided in this section including information about *asset locking* and *signing* required for these state transitions.

Identity Creation

Identities are created on the platform by submitting the identity information in an identity create state transition.

Field	Type	Description
protocolVersion	integer	The protocol version (currently 1)
type	integer	State transition type (2 for identity create)
assetLockProof	object	*Asset lock proof object* proving the layer 1 locking transaction exists and is locked
publicKeys	array of keys	*Public key(s)* associated with the identity
signature	array of bytes	Signature of state transition data by the single-use key from the asset lock (65 bytes)

Each identity must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "type": {
      "type": "integer",
      "const": 2
    },
    "assetLockProof": {
      "type": "object"
    },
    "publicKeys": {
      "type": "array",
      "minItems": 1,
      "maxItems": 10,
      "uniqueItems": true
    },
    "signature": {
      "type": "array",
      "byteArray": true,
      "minItems": 65,
      "maxItems": 65,
      "description": "Signature made by AssetLock one time ECDSA key"
    }
  },
  "additionalProperties": false,
  "required": [
    "protocolVersion",
    "type",
    "assetLockProof",
    "publicKeys",
    "signature"
  ]
}
```

Example State Transition

```
{
  "protocolVersion":1,
  "type":2,
  "signature":"IBTTgge+/VDa/9+n2q3pb4tAqZYI48AX8X3H/uedRLH5dN8Ekh/
↪sxRRQQS9LaOPwZSCVED6XIYD+vravF2dhYOE=",
  "assetLockProof":{
    "type":0,
    "instantLock":"AQHDHQdekbFZJOQFEe1FnRjoDemL/oPF/v9IME/qphjt5gEAAAB/
↪OlZB9p8vPzPE55MlegR7nwhXRpZC4d5sYnOIypNgzfdDRsW01v8UtlRoORokjoDJ9hA/
↪XFMK65iYTrQ8AAAAGI4q8GxtK9LHOT1JipnIfwiiv8zW+C/sbokbMhi/
↪BsEl51dpoeBQEUAYWT7KRiJ4Atx49zIrqsKvmU1mJQza0Y1YbBSS/b/IPO8StX04bItPpDuTp6zlh/
```

(continues on next page)

(continued from previous page)

```
↪G7YOGzlEoe",
    "transaction":
↪"0300000001c31d075e91b15924e40511ed459d18e80de98bfe83c5feff48304feaa618ede6010000006b483045022100dd0e
↪",
    "outputIndex":0
  },
  "publicKeys":[
    {
      "id":0,
      "type":0,
      "purpose":0,
      "securityLevel":0,
      "data":"AkWRfl3DJiyyy6YPUDQnNx5KERRnR8CoTiFUvfdaYSDS",
      "readOnly":false
    }
  ]
}
```

Identity TopUp

Identity credit balances are increased by submitting the topup information in an identity topup state transition.

Field	Type	Description
protocolVersion	integer	The protocol version (currently 1)
type	integer	State transition type (3 for identity topup)
assetLockProof	object	*Asset lock proof object* proving the layer 1 locking transaction exists and is locked
identityId	array of bytes	An *Identity ID* for the identity receiving the topup (can be any identity) (32 bytes)
signature	array of bytes	Signature of state transition data by the single-use key from the asset lock (65 bytes)

Each identity must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "type": {
      "type": "integer",
      "const": 3
    },
    "assetLockProof": {
      "type": "object"
    },
```

(continues on next page)

(continued from previous page)

```
    "identityId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "signature": {
      "type": "array",
      "byteArray": true,
      "minItems": 65,
      "maxItems": 65,
      "description": "Signature made by AssetLock one time ECDSA key"
    }
  },
  "additionalProperties": false,
  "required": [
    "protocolVersion",
    "type",
    "assetLockProof",
    "identityId",
    "signature"
  ]
}
```

Example State Transition

```
{
  "protocolVersion":1,
  "type":3,
  "signature":
↪"IEqOV4DsbVa+nPipva0UrT0z0ZwubwgP9UdlpwBwXbFSWb7Mxkwqzv1HoEDICJ8GtmUSVjp4Hr2x0cVWe7+yUGc=
↪",
  "identityId":"6YfP6tT9AK8HPVXMK7CQrhpc8VMg7frjEnXinSPvUmZC",
  "assetLockProof":{
    "type":0,
    "instantLock":"AQF/
↪OlZB9p8vPzPE55MlegR7nwhXRpZC4d5sYnOIypNgzQEAAAAm8edm9p8URNEE9PBo0lEzZ2s9nf4u1SV0MaZyB0JTRasiXu8QtTmfq
↪2fV+0Ffi3AAAAhA77E0aScf+5PTYzgV5WR6VJ/EnjvXyAMmAcu222JyvA7M+5OoCzVF/
↪IQs2IWaPOFsRl1n5C+dMxdvrxhpVLT8QfZJSl19wzybWrHbGRaHDw4iWHvfYdwyXN+vP8UwDz",
    "transaction":
↪"03000000017f3a5641f69f2f3f33c4e793257a047b9f0857469642e1de6c627388ca9360cd010000006b4830450221 00d8c3
↪",
    "outputIndex":0
  }
}
```

Identity Update

Identities are updated on the platform by submitting the identity information in an identity update state transition. This state transition requires either a set of one or more new public keys to add to the identity or a list of existing keys to disable.

Field	Type	Description
protocolVersion	integer	The protocol version (currently 1)
type	integer	State transition type (5 for identity update)
identityId	array of bytes	The identity id (32 bytes)
signature	array of bytes	Signature of state transition data (65 bytes)
revision	integer	Identity update revision
publicKeysDisabledAt	integer	(Optional) Timestamp when keys were disabled. Required if `disablePublicKeys` is present.
addPublicKeys	array of public keys	(Optional) Array of up to 10 new public keys to add to the identity. Required if adding keys.
disablePublicKeys	array of integers	(Optional) Array of up to 10 existing identity public key ID(s) to disable for the identity. Required if disabling keys.
signaturePublicKeyId	integer	The ID of public key used to sign the state transition

Each identity must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "type": {
      "type": "integer",
      "const": 5
    },
    "identityId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "signature": {
      "type": "array",
      "byteArray": true,
      "minItems": 65,
      "maxItems": 96
    },
    "revision": {
      "type": "integer",
      "minimum": 0,
      "description": "Identity update revision"
```

(continues on next page)

(continued from previous page)

```
    },
    "publicKeysDisabledAt": {
      "type": "integer",
      "minimum": 0
    },
    "addPublicKeys": {
      "type": "array",
      "minItems": 1,
      "maxItems": 10,
      "uniqueItems": true
    },
    "disablePublicKeys": {
      "type": "array",
      "minItems": 1,
      "maxItems": 10,
      "uniqueItems": true,
      "items": {
        "type": "integer",
        "minimum": 0
      }
    },
    "signaturePublicKeyId": {
      "type": "integer",
      "minimum": 0
    }
  },
  "dependentRequired" : {
    "disablePublicKeys": ["publicKeysDisabledAt"],
    "publicKeysDisabledAt": ["disablePublicKeys"]
  },
  "anyOf": [
    {
      "type": "object",
      "required": ["addPublicKeys"],
      "properties": {
        "addPublicKeys": true
      }
    },
    {
      "type": "object",
      "required": ["disablePublicKeys"],
      "properties": {
        "disablePublicKeys": true
      }
    }
  ],
  "additionalProperties": false,
  "required": [
    "protocolVersion",
    "type",
    "identityId",
    "signature",
```

(continues on next page)

(continued from previous page)

```
    "revision",
    "signaturePublicKeyId"
  ]
}
```

Asset Lock

The *identity create* and *identity topup* state transitions both include an asset lock proof object. This object references the layer 1 lock transaction and includes proof that the transaction is locked.

Currently there are two types of asset lock proofs: InstantSend and ChainLock. Transactions almost always receive InstantSend locks, so the InstantSend asset lock proof is the predominate type.

InstantSend Asset Lock Proof

The InstantSend asset lock proof is used for transactions that have received an InstantSend lock.

Field	Type	Description
type	integer	The asset lock proof type (`0` for InstantSend locks)
instantLock	array of bytes	The InstantSend lock (`islock`)
transaction	array of bytes	The asset lock transaction
outputIndex	integer	Index of the transaction output to be used

Asset locks using an InstantSend lock as proof must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "type": {
      "type": "integer",
      "const": 0
    },
    "instantLock": {
      "type": "array",
      "byteArray": true,
      "minItems": 165,
      "maxItems": 100000
    },
    "transaction": {
      "type": "array",
      "byteArray": true,
      "minItems": 1,
      "maxItems": 100000
    },
    "outputIndex": {
      "type": "integer",
      "minimum": 0
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "additionalProperties": false,
  "required": [
    "type",
    "instantLock",
    "transaction",
    "outputIndex"
  ]
}
```

ChainLock Asset Lock Proof

The ChainLock asset lock proof is used for transactions that have note received an InstantSend lock, but have been included in a block that has received a ChainLock.

Field	Type	Description
type	array of bytes	The type of asset lock proof (1 for ChainLocks)
coreChainLockedHeight	integer	Height of the ChainLocked Core block containing the transaction
outPoint	object	The outpoint being used as the asset lock

Asset locks using a ChainLock as proof must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "type": {
      "type": "integer",
      "const": 1
    },
    "coreChainLockedHeight":  {
      "type": "integer",
      "minimum": 1,
      "maximum": 4294967295
    },
    "outPoint": {
      "type": "array",
      "byteArray": true,
      "minItems": 36,
      "maxItems": 36
    }
  },
  "additionalProperties": false,
  "required": [
    "type",
    "coreChainLockedHeight",
    "outPoint"
  ]
}
```

Identity State Transition Signing

Note: The identity create and topup state transition signatures are unique in that they must be signed by the private key used in the layer 1 locking transaction. All other state transitions will be signed by a private key of the identity submitting them.

The process to sign an identity create state transition consists of the following steps:

1. Canonical CBOR encode the state transition data - this include all ST fields except the `signature`
2. Sign the encoded data with private key associated with a lock transaction public key
3. Set the state transition `signature` to the value of the signature created in the previous step

Code snipits related to signing

```
/// From rs-dpp
/// abstract_state_transition.rs
/// Signs data with the private key
fn sign_by_private_key(
    &mut self,
    private_key: &[u8],
    key_type: KeyType,
    bls: &impl BlsModule,
) -> Result<(), ProtocolError> {
    let data = self.to_buffer(true)?;
    match key_type {
        KeyType::BLS12_381 => self.set_signature(bls.sign(&data, private_key)?.into()),

        // https://github.com/dashevo/platform/blob/
↪9c8e6a3b6afbc330a6ab551a689de8ccd63f9120/packages/js-dpp/lib/stateTransition/
↪AbstractStateTransition.js#L169
        KeyType::ECDSA_SECP256K1 | KeyType::ECDSA_HASH160 => {
            let signature = signer::sign(&data, private_key)?;
            self.set_signature(signature.to_vec().into());
        }

        // the default behavior from
        // https://github.com/dashevo/platform/blob/
↪6b02b26e5cd3a7c877c5fdfe40c4a4385a8dda15/packages/js-dpp/lib/stateTransition/
↪AbstractStateTransition.js#L187
        // is to return the error for the BIP13_SCRIPT_HASH
        KeyType::BIP13_SCRIPT_HASH => {
            return Err(ProtocolError::InvalidIdentityPublicKeyTypeError(
                InvalidIdentityPublicKeyTypeError::new(key_type),
            ))
        }
    };
    Ok(())
}


/// From rust-dashcore
/// signer.rs
```

(continues on next page)

(continued from previous page)

```rust
/// sign and get the ECDSA signature
pub fn sign(data: &[u8], private_key: &[u8]) -> Result<[u8; 65], anyhow::Error> {
    let data_hash = double_sha(data);
    sign_hash(&data_hash, private_key)
}

/// signs the hash of data and get the ECDSA signature
pub fn sign_hash(data_hash: &[u8], private_key: &[u8]) -> Result<[u8; 65], anyhow::Error>
↪ {
    let pk = SecretKey::from_slice(private_key)
        .map_err(|e| anyhow!("Invalid ECDSA private key: {}", e))?;

    let secp = Secp256k1::new();
    let msg = Message::from_slice(data_hash).map_err(anyhow::Error::msg)?;

    let signature = secp
        .sign_ecdsa_recoverable(&msg, &pk)
        .to_compact_signature(true);
    Ok(signature)
}
```

1.33 Data Contract

1.33.1 Data Contract Overview

Data contracts define the schema (structure) of data an application will store on Dash Platform. Contracts are described using JSON Schema which allows the platform to validate the contract-related data submitted to it.

The following sections provide details that developers need to construct valid contracts: *documents* and *definitions*. All data contracts must define one or more documents, whereas definitions are optional and may not be used for simple contracts.

General Constraints

There are a variety of constraints currently defined for performance and security reasons. The following constraints are applicable to all aspects of data contracts. Unless otherwise noted, these constraints are defined in the platform's JSON Schema rules (e.g. rs-dpp data contract meta schema).

Keyword

The `$ref` keyword has been disabled since Platform v0.22.

Keyword	Constraint
`default`	Restricted - cannot be used (defined in DPP logic)
`propertyNames`	Restricted - cannot be used (defined in DPP logic)
`uniqueItems: true`	`maxItems` must be defined (maximum: 100000)
`pattern: <something>`	`maxLength` must be defined (maximum: 50000)
`format: <something>`	`maxLength` must be defined (maximum: 50000)
`$ref: <something>`	**Temporarily disabled** `$ref` can only reference `$defs` - remote references not supported
`if`, `then`, `else`, `allOf`, `anyOf`, `oneOf`, `not`	Disabled for data contracts
`dependencies`	Not supported. Use `dependentRequired` and `dependentSchema` instead
`additionalItems`	Not supported. Use `items: false` and `prefixItems` instead
`patternProperties`	Restricted - cannot be used for data contracts
`pattern`	Accept only RE2 compatible regular expressions (defined in DPP logic)

Data Size

Note: These constraints are defined in the Dash Platform Protocol logic (not in JSON Schema).

All serialized data (including state transitions) is limited to a maximum size of 16 KB.

Additional Properties

Although JSON Schema allows additional, undefined properties by default, they are not allowed in Dash Platform data contracts. Data contract validation will fail if they are not explicitly forbidden using the `additionalProperties` keyword anywhere `properties` are defined (including within document properties of type `object`).

Include the following at the same level as the `properties` keyword to ensure proper validation:

```
"additionalProperties": false
```

1.33.2 Data Contract Object

The data contract object consists of the following fields as defined in the JavaScript reference client (rs-dpp):

Property	Type	Required	Description
protocolVersion	integer	Yes	The platform protocol version (currently 1)
$schema	string	Yes	A valid URL (default: https://schema.dash.org/dpp-0-4-0/meta/data-contract)
$id	array of bytes	Yes	Contract ID generated from `ownerId` and entropy (32 bytes; content media type: `application/x.dash.dpp.identifier`)
version	integer	Yes	The data contract version
ownerId	array of bytes	Yes	*Identity* that registered the data contract defining the document (32 bytes; content media type: `application/x.dash.dpp.identifier`
documents	object	Yes	Document definitions (see *Documents* for details)
$defs	object	No	Definitions for `$ref` references used in the `documents` object (if present, must be a non-empty object with <= 100 valid properties)

Data Contract Schema

Details regarding the data contract object may be found in the rs-dpp data contract meta schema. A truncated version is shown below for reference:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://schema.dash.org/dpp-0-4-0/meta/data-contract",
  "type": "object",
  "$defs": {
    // Truncated ...
  },
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "minimum": 0,
      "$comment": "Maximum is the latest protocol version"
    },
    "$schema": {
      "type": "string",
      "const": "https://schema.dash.org/dpp-0-4-0/meta/data-contract"
    },
    "$id": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "version": {
      "type": "integer",
      "minimum": 1
    },
    "ownerId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "documents": {
      "type": "object",
      "propertyNames": {
        "pattern": "^[a-zA-Z0-9-_]{1,64}$"
      },
      "additionalProperties": {
        "type": "object",
        "allOf": [
          {
            "properties": {
              "indices": {
                "type": "array",
                "items": {
```

(continues on next page)

(continued from previous page)

```
                "type": "object",
                "properties": {
                  "name": {
                    "type": "string",
                    "minLength": 1,
                    "maxLength": 32
                  },
                  "properties": {
                    "type": "array",
                    "items": {
                      "type": "object",
                      "propertyNames": {
                        "maxLength": 256
                      },
                      "additionalProperties": {
                        "type": "string",
                        "enum": [
                          "asc"
                        ]
                      },
                      "minProperties": 1,
                      "maxProperties": 1
                    },
                    "minItems": 1,
                    "maxItems": 10
                  },
                  "unique": {
                    "type": "boolean"
                  }
                },
                "required": [
                  "properties",
                  "name"
                ],
                "additionalProperties": false
              },
              "minItems": 1,
              "maxItems": 10
            },
            "type": {
              "const": "object"
            },
            "signatureSecurityLevelRequirement": {
              "type": "integer",
              "enum": [
                0,
                1,
                2,
                3
              ],
              "description": "Public key security level. 0 - Master, 1 - Critical, 2 -␣
↪High, 3 - Medium. If none specified, High level is used"
```

(continues on next page)

(continued from previous page)

```
              }
            }
          },
          {
            "$ref": "#/$defs/documentSchema"
          }
        ],
        "unevaluatedProperties": false
      },
      "minProperties": 1,
      "maxProperties": 100
    },
    "$defs": {
      "$ref": "#/$defs/documentProperties"
    }
  },
  "required": [
    "protocolVersion",
    "$schema",
    "$id",
    "version",
    "ownerId",
    "documents"
  ],
  "additionalProperties": false
}
```

Example

```
{
  "id": "AoDzJxWSb1gUi2dSmvFeUFpSsjZQRJaqCpn7vCLkwwJj",
  "ownerId": "7NUbPf231ixt1kVBQsBvSMMBxd7AgPad8KtdtfFGhXDP",
  "schema": "https://schema.dash.org/dpp-0-4-0/meta/data-contract",
  "documents": {
    "note": {
      "properties": {
        "message": {
          "type": "string"
        }
      },
      "additionalProperties": false
    }
  }
}
```

Data Contract id

The data contract `$id` is a hash of the `ownerId` and entropy as shown here.

```
// From the Rust reference implementation (rs-dpp)
// generate_data_contract.rs
/// Generate data contract id based on owner id and entropy
pub fn generate_data_contract_id(owner_id: impl AsRef<[u8]>, entropy: impl AsRef<[u8]>) -
↪> Vec<u8> {
    let mut b: Vec<u8> = vec![];
    let _ = b.write(owner_id.as_ref());
    let _ = b.write(entropy.as_ref());
    hash(b)
}
```

Data Contract version

The data contract `version` is an integer representing the current version of the contract. This property must be incremented if the contract is updated.

Data Contract Documents

The `documents` object defines each type of document required by the data contract. At a minimum, a document must consist of 1 or more properties. Documents may also define *indices* and a list of *required properties*. The `additionalProperties` properties keyword must be included as described in the *constraints* section.

The following example shows a minimal `documents` object defining a single document (`note`) that has one property (`message`).

```
{
  "note": {
    "type": "object",
    "properties": {
      "message": {
        "type": "string"
      }
    },
    "additionalProperties": false
  }
}
```

Document Properties

The `properties` object defines each field that will be used by a document. Each field consists of an object that, at a minimum, must define its data `type` (`string`, `number`, `integer`, `boolean`, `array`, `object`). Fields may also apply a variety of optional JSON Schema constraints related to the format, range, length, etc. of the data.

Note: The `object` type is required to have properties defined either directly or via the data contract's *$defs*. For example, the body property shown below is an object containing a single string property (objectProperty):

```
const contractDocuments = {
  message: {
    "type": "object",
    properties: {
      body: {
        type: "object",
        properties: {
          objectProperty: {
            type: "string"
          },
        },
        additionalProperties: false,
      },
      header: {
        type: "string"
      }
    },
    additionalProperties: false
  }
};
```

Note: A full explanation of the capabilities of JSON Schema is beyond the scope of this document. For more information regarding its data types and the constraints that can be applied, please refer to the JSON Schema reference documentation.

Property Constraints

There are a variety of constraints currently defined for performance and security reasons.

Description	Value
Minimum number of properties	1
Maximum number of properties	100
Minimum property name length	1 (Note: minimum length was 3 prior to v0.23)
Maximum property name length	64
Property name characters	Alphanumeric (`A-Z`, `a-z`, `0-9`)Hyphen (`-`) Underscore (`_`)

Prior to Dash Platform v0.23 there were stricter limitations on minimum property name length and the characters that could be used in property names.

Required Properties (Optional)

Each document may have some fields that are required for the document to be valid and other fields that are optional. Required fields are defined via the `required` array which consists of a list of the field names from the document that must be present. The `required` object should be excluded for documents without any required properties.

```
"required": [
  "<field name a>",
  "<field name b>"
]
```

Example
The following example (excerpt from the DPNS contract's `domain` document) demonstrates a document that has 6 required fields:

```
"required": [
  "label",
  "normalizedLabel",
  "normalizedParentDomainName",
  "preorderSalt",
  "records",
  "subdomainRules"
]
```

Document Indices

Document indices may be defined if indexing on document fields is required.

Note: Dash Platform v0.23 only allows ascending default ordering for indices.

The `indices` array consists of:

- One or more objects that each contain:
  - A unique `name` for the index
  - A `properties` array composed of a <`field name: sort order`> object for each document field that is part of the index (sort order: `asc` only for Dash Platform v0.23)
  - An (optional) `unique` element that determines if duplicate values are allowed for the document type

Note:

- The `indices` object should be excluded for documents that do not require indices.
- When defining an index with multiple properties (i.e a compound index), the order in which the properties are listed is important. Refer to the mongoDB documentation for details regarding the significance of the order as it relates to querying capabilities. Dash uses GroveDB which works similarly but does requiring listing *all* the index's fields in query order by statements.

```
"indices": [
  {
    "name": "Index1",
    "properties": [
      { "<field name a>": "asc" },
      { "<field name b>": "asc" }
    ],
    "unique": true|false
  },
  {
    "name": "Index2",
    "properties": [
      { "<field name c>": "asc" },
    ],
  }
]
```

Index Constraints

For performance and security reasons, indices have the following constraints. These constraints are subject to change over time.

Description	Value
Minimum/maximum length of index `name`	1 / 32
Maximum number of indices	10
Maximum number of unique indices	3
Maximum number of properties in a single index	10
Maximum length of indexed string property	63
Note: Dash Platform v0.22+. does not allow indices for arrays Maximum length of indexed byte array property	255
Note: Dash Platform v0.22+. does not allow indices for arrays Maximum number of indexed array items	1024
Usage of `$id` in an index disallowed	N/A

Example
The following example (excerpt from the DPNS contract's `preorder` document) creates an index named `saltedHash` on the `saltedDomainHash` property that also enforces uniqueness across all documents of that type:

```
"indices": [
  {
    "name": "saltedHash",
    "properties": [
      {
        "saltedDomainHash": "asc"
      }
    ],
    "unique": true
  }
]
```

Full Document Syntax

This example syntax shows the structure of a documents object that defines two documents, an index, and a required field.

```
{
  "<document name a>": {
    "type": "object",
    "properties": {
      "<field name b>": {
        "type": "<field data type>"
      },
      "<field name c>": {
        "type": "<field data type>"
      },
    },
    "indices": [
      {
        "name": "<index name>",
```

(continues on next page)

(continued from previous page)

```
        "properties": [
          {
            "<field name c>": "asc"
          }
        ],
        "unique": true|false
      },
    ],
    "required": [
      "<field name c>"
    ]
    "additionalProperties": false
  },
  "<document name x>": {
    "type": "object",
    "properties": {
      "<property name y>": {
        "type": "<property data type>"
      },
      "<property name z>": {
        "type": "<property data type>"
      },
    },
    "additionalProperties": false
  },
}
```

Document Schema

Full document schema details may be found in this section of the rs-dpp data contract meta schema.

Data Contract Definitions

> Definitions are currently unavailable

The optional `$defs` object enables definition of aspects of a schema that are used in multiple places. This is done using the JSON Schema support for reuse. Items defined in `$defs` may then be referenced when defining `documents` through use of the `$ref` keyword.

Note:

- Properties defined in the `$defs` object must meet the same criteria as those defined in the `documents` object (e.g. the `additionalProperties` properties keyword must be included as described in the *constraints* section).
- Data contracts can only use the `$ref` keyword to reference their own `$defs`. Referencing external definitions is not supported by the platform protocol.

Example
The following example shows a definition for a `message` object consisting of two properties:

```
{
  // Preceding content truncated ...
```

(continues on next page)

(continued from previous page)

```
  "$defs": {
    "message": {
      "type": "object",
      "properties": {
        "timestamp": {
          "type": "number"
        },
        "description": {
          "type": "string"
        }
      },
      "additionalProperties": false
    }
  }
}
```

1.33.3 Data Contract State Transition Details

There are two data contract-related state transitions: *data contract create* and *data contract update*. Details are provided in this section.

Data Contract Creation

Data contracts are created on the platform by submitting the *data contract object* in a data contract create state transition consisting of:

Field	Type	Description
protocolVersion	integer	The platform protocol version (currently 1)
type	integer	State transition type (`0` for data contract create)
dataContract	*data contract object*	Object containing the data contract details
entropy	array of bytes	Entropy used to generate the data contract ID. Generated as *shown here*. (32 bytes)
signaturePublicKeyId	number	The `id` of the *identity public key* that signed the state transition
signature	array of bytes	Signature of state transition data (65 or 96 bytes)

Each data contract state transition must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "type": {
      "type": "integer",
```

(continues on next page)

(continued from previous page)

```
      "const": 0
    },
    "dataContract": {
      "type": "object"
    },
    "entropy": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32
    },
    "signaturePublicKeyId": {
      "type": "integer",
      "minimum": 0
    },
    "signature": {
      "type": "array",
      "byteArray": true,
      "minItems": 65,
      "maxItems": 96
    }
  },
  "additionalProperties": false,
  "required": [
    "protocolVersion",
    "type",
    "dataContract",
    "entropy",
    "signaturePublicKeyId",
    "signature"
  ]
}
```

Example State Transition

```
{
  "protocolVersion":1,
  "type":0,
  "signature":"IFmEb/OwyYG0yn33U4/
↪kieH4JL63Ft25GAun+XqWOalkbDrpL9z+OH+Sb03xsyMNzoILC2T1Q8yV1q7kCmr0HuQ=",
  "signaturePublicKeyId":0,
  "dataContract":{
    "protocolVersion":1,
    "$id":"44dvUnSdVtvPPeVy6mS4vRzJ4zfABCt33VvqTWMM8VG6",
    "$schema":"https://schema.dash.org/dpp-0-4-0/meta/data-contract",
    "version":1,
    "ownerId":"6YfP6tT9AK8HPVXMK7CQrhpc8VMg7frjEnXinSPvUmZC",
    "documents":{
      "note":{
        "type":"object",
        "properties":{
          "message":{
```

(continues on next page)

(continued from previous page)

```
            "type":"string"
          }
        },
        "additionalProperties":false
      }
    }
  },
  "entropy":"J2Sl/Ka9T1paYUv6f2ec5MzaaACs9lcUvOskBU0SMlo="
}
```

Data Contract Update

Existing data contracts can be updated in certain backwards-compatible ways. The following aspects of a data contract can be updated:

- Adding a new document
- Adding a new optional property to an existing document
- Adding non-unique indices for properties added in the update

Data contracts are updated on the platform by submitting the modified *data contract object* in a data contract update state transition consisting of:

Field	Type	Description
protocolVersion	integer	The platform protocol version (currently 1)
type	integer	State transition type (4 for data contract update)
dataContract	*data contract object*	Object containing the updated data contract details**Note:** the data contract's `version` *property* must be incremented with each update
signaturePublicKeyId	number	The `id` of the *identity public key* that signed the state transition
signature	array of bytes	Signature of state transition data (65 or 96 bytes)

Each data contract state transition must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "type": {
      "type": "integer",
      "const": 4
    },
    "dataContract": {
      "type": "object"
```

(continues on next page)

(continued from previous page)

```
    },
    "signaturePublicKeyId": {
      "type": "integer",
      "minimum": 0
    },
    "signature": {
      "type": "array",
      "byteArray": true,
      "minItems": 65,
      "maxItems": 96
    }
  },
  "additionalProperties": false,
  "required": [
    "protocolVersion",
    "type",
    "dataContract",
    "signaturePublicKeyId",
    "signature"
  ]
}
```

Example State Transition

```
{
  "protocolVersion":1,
  "type":4,
  "signature":"IBboAbqbGBiWzyJDyhwzs1GujR6Gb4m5Gt/QCugLV2EYcsBaQKTM/
↪Stq7iyIm2YyqkV8VlWqOfGebW2w5Pjnfak=",
  "signaturePublicKeyId":0,
  "dataContract":{
    "protocolVersion":1,
    "$id":"44dvUnSdVtvPPeVy6mS4vRzJ4zfABCt33VvqTWMM8VG6",
    "$schema":"https://schema.dash.org/dpp-0-4-0/meta/data-contract",
    "version":2,
    "ownerId":"6YfP6tT9AK8HPVXMK7CQrhpc8VMg7frjEnXinSPvUmZC",
    "documents":{
      "note":{
        "type":"object",
        "properties":{
          "message":{
            "type":"string"
          },
          "author":{
            "type":"string"
          }
        },
        "additionalProperties":false
      }
    }
  }
}
```

Data Contract State Transition Signing

Data contract state transitions must be signed by a private key associated with the contract owner's identity.

The process to sign a data contract state transition consists of the following steps:

1. Canonical CBOR encode the state transition data - this include all ST fields except the `signature` and `signaturePublicKeyId`
2. Sign the encoded data with a private key associated with the `ownerId`
3. Set the state transition `signature` to the value of the signature created in the previous step
4. Set the state transition`signaturePublicKeyId` to the *public key* `id` corresponding to the key used to sign

1.34 State Transition

1.34.1 State Transition Overview

State transitions are the means for submitting data that creates, updates, or deletes platform data and results in a change to a new state. Each one must contain:

- *Common fields* present in all state transitions
- Additional fields specific to the type of action the state transition provides (e.g. *creating an identity*)

Fees

State transition fees are paid via the credits established when an identity is created. Credits are created at a rate of 1000 credits/satoshi. Fees for actions vary based on parameters related to storage and computational effort that are defined in rs-dpp.

Size

All serialized data (including state transitions) is limited to a maximum size of 16 KB.

Common Fields

All state transitions include the following fields:

Field	Type	Description
protocolVersion	integer	The platform protocol version (currently `1`)
type	integer	State transition type:`0` - *data contract create*`1` - *documents batch*`2` - *identity create*`3` - *identity topup*`4` - *data contract update*`5` - *identity update*
signature	array of bytes	Signature of state transition data (65 bytes)

Additionally, all state transitions except the identity create and topup state transitions include:

Field	Type	Description
signaturePublicKeyId	integer	The `id` of the *identity public key* that signed the state transition (=> `0`)

1.34.2 State Transition Types

Data Contract Create

Field	Type	Description
dataContract	*data contract object*	Object containing valid *data contract* details
entropy	array of bytes	Entropy used to generate the data contract ID (32 bytes)

More detailed information about the `dataContract` object can be found in the *data contract section*.

Entropy Generation

Entropy is included in *Data Contracts* and *Documents*.

```
// From the Rust reference implementation (rs-dpp)
// entropyGenerator.js
fn generate(&self) -> anyhow::Result<[u8; 32]> {
  let mut buffer = [0u8; 32];
  getrandom(&mut buffer).context("generating entropy failed")?;
  Ok(buffer)
}
```

Data Contract Update

Field	Type	Description
dataContract	*data contract object*	Object containing valid *data contract* details

More detailed information about the `dataContract` object can be found in the *data contract section*.

Documents Batch

Field	Type	Description
ownerId	array of bytes	*Identity* submitting the document(s) (32 bytes)
transitions	array of transition objects	Document `create`, `replace`, or `delete` transitions (up to 10 objects)

More detailed information about the `transitions` array can be found in the *document section*.

Identity Create

Field	Type	Description
assetLockProof	array of bytes	Lock outpoint from the layer 1 locking transaction (36 bytes)
publicKeys	array of keys	*Public key(s)* associated with the identity (maximum number of keys: `10`)

More detailed information about the `publicKeys` object can be found in the *identity section*.

Identity TopUp

Field	Type	Description
assetLock-Proof	array of bytes	Lock outpoint from the layer 1 locking transaction (36 bytes)
identityId	array of bytes	An *Identity ID* for the identity receiving the topup (can be any identity) (32 bytes)

Identity Update

Field	Type	Description
identityId	array of bytes	The *Identity ID* for the identity being updated (32 bytes)
revision	integer	Identity update revision. Used for optimistic concurrency control. Incremented by one with each new update so that the update will fail if the underlying data is modified between reading and writing.
addPub-licKeys	array of public keys	(Optional) Array of up to 10 new public keys to add to the identity. Required if adding keys.
dis-ablePub-licKeys	array of integers	(Optional) Array of up to 10 existing identity public key ID(s) to disable for the identity. Required if disabling keys.
publicK-eysDis-abledAt	integer	(Optional) Timestamp when keys were disabled. Required if `disablePublicKeys` is present.

1.34.3 State Transition Signing

State transitions must be signed by a private key associated with the identity creating the state transition. Since v0.23, each identity must have at least two keys: a primary key (security level `0`) that is only used when signing identity update state transitions and an additional key (security level `2`) that is used to sign all other state transitions.

The process to sign a state transition consists of the following steps:

1. Canonical CBOR encode the state transition data - this include all ST fields except the `signature` and `signaturePublicKeyId`
2. Sign the encoded data with a private key associated with the identity creating the state transition
3. Set the state transition `signature` to the value of the signature created in the previous step
4. For all state transitions *other than identity create or topup*, set the state transition`signaturePublicKeyId` to the *public key* `id` corresponding to the key used to sign

Signature Validation

The `signature` validation (see js-dpp) verifies that:

1. The identity exists
2. The identity has a public key
3. The identity's public key is of type `ECDSA`
4. The state transition signature is valid

The example test output below shows the necessary criteria:

```
validateStateTransitionIdentitySignatureFactory
  ✓ should pass properly signed state transition
  ✓ should return invalid result if owner id doesn't exist
  ✓ should return MissingPublicKeyError if the identity doesn't have a matching public␣
↪key
  ✓ should return InvalidIdentityPublicKeyTypeError if type is not exist
  ✓ should return InvalidStateTransitionSignatureError if signature is invalid
  Consensus errors
    ✓ should return InvalidSignaturePublicKeySecurityLevelConsensusError if␣
↪InvalidSignaturePublicKeySecurityLevelError was thrown
    ✓ should return PublicKeySecurityLevelNotMetConsensusError if␣
↪PublicKeySecurityLevelNotMetError was thrown
    ✓ should return WrongPublicKeyPurposeConsensusError if WrongPublicKeyPurposeError␣
↪was thrown
    ✓ should return PublicKeyIsDisabledConsensusError if PublicKeyIsDisabledError was␣
↪thrown
    ✓ should return InvalidStateTransitionSignatureError if DPPError was thrown
    ✓ should throw unknown error
    ✓ should not verify signature on dry run
```

1.35 Document

1.35.1 Document Submission

Documents are sent to the platform by submitting the them in a document batch state transition consisting of:

Field	Type	Description
protocolVersion	integer	The platform protocol version (currently 1)
type	integer	State transition type (1 for document batch)
ownerId	array	*Identity* submitting the document(s) (32 bytes)
transitions	array of transition objects	Document `create`, `replace`, or `delete` transitions (up to 10 objects)
signaturePublicKeyId	number	The `id` of the *identity public key* that signed the state transition
signature	array	Signature of state transition data (65 or 96 bytes)

Each document batch state transition must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "type": {
      "type": "integer",
      "const": 1
    },
    "ownerId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "transitions": {
      "type": "array",
      "items": {
        "type": "object"
      },
      "minItems": 1,
      "maxItems": 10
    },
    "signaturePublicKeyId": {
      "type": "integer",
      "minimum": 0
    },
    "signature": {
      "type": "array",
      "byteArray": true,
      "minItems": 65,
      "maxItems": 96
    }
  },
  "additionalProperties": false,
  "required": [
    "protocolVersion",
    "type",
    "ownerId",
    "transitions",
    "signaturePublicKeyId",
    "signature"
  ]
}
```

Document Base Transition

All document transitions in a document batch state transition are built on the base schema and include the following fields:

Field	Type	Description
$id	array	The *document ID* (32 bytes)
$type	string	Name of a document type found in the data contract associated with the `dataContractId` (1-64 characters)
$action	array of integers	*Action* the platform should take for the associated document
$dataContractId	array	Data contract ID *generated* from the data contract's `ownerId` and `entropy` (32 bytes)

Each document transition must comply with the document transition base schema:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "$id": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "$type": {
      "type": "string"
    },
    "$action": {
      "type": "integer",
      "enum": [0, 1, 3]
    },
    "$dataContractId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    }
  },
  "required": [
    "$id",
    "$type",
    "$action",
    "$dataContractId"
  ],
  "additionalProperties": false
}
```

Document id

The document `$id` is created by hashing the document's `dataContractId`, `ownerId`, `type`, and `entropy` as shown in rs-dpp.

```
// From the Rust reference implementation (rs-dpp)
// generate_document_id.rs
pub fn generate_document_id(
    contract_id: &Identifier,
    owner_id: &Identifier,
    document_type: &str,
    entropy: &[u8],
) -> Identifier {
    let mut buf: Vec<u8> = vec![];

    buf.extend_from_slice(&contract_id.to_buffer());
    buf.extend_from_slice(&owner_id.to_buffer());
    buf.extend_from_slice(document_type.as_bytes());
    buf.extend_from_slice(entropy);

    Identifier::from_bytes(&hash(&buf)).unwrap()
}
```

Document Transition Action

Action	Name	Description
0	Create	Create a new document with the provided data
1	Replace	Replace an existing document with the provided data
2	RESERVED	Unused action
3	Delete	Delete the referenced document

Document Create Transition

The document create transition extends the base schema to include the following additional fields:

Field	Type	Description
$entropy	array	Entropy used in creating the *document ID*. Generated as *shown here*. (32 bytes)
$createdAt	integer	(Optional)
$updatedAt	integer	(Optional)

Each document create transition must comply with this JSON-Schema definition established in rs-dpp (in addition to the document transition base schema) that is required for all document transitions):

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "$entropy": {
      "type": "array",
```

(continues on next page)

(continued from previous page)

```
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32
    },
    "$createdAt": {
      "type": "integer",
      "minimum": 0
    },
    "$updatedAt": {
      "type": "integer",
      "minimum": 0
    }
  },
  "required": [
    "$entropy"
  ],
  "additionalProperties": false
}
```

Note: The document create transition must also include all required properties of the document as defined in the data contract.

The following example document create transition and subsequent table demonstrate how the document transition base, document create transition, and data contract document definitions are assembled into a complete transition for inclusion in a *state transition*:

```
{
  "$action": 0,
  "$dataContractId": "5wpZAEWndYcTeuwZpkmSa8s49cHXU5q2DhdibesxFSu8",
  "$id": "6oCKUeLVgjr7VZCyn1LdGbrepqKLmoabaff5WQqyTKYP",
  "$type": "note",
  "$entropy": "yfo6LnZfJ5koT2YUwtd8PdJa8SXzfQMVDz",
  "message": "Tutorial Test @ Mon, 27 Apr 2020 20:23:35 GMT"
}
```

Field	Required By
$action	Document *base transition*
$dataContractId	Document *base transition*
$id	Document *base transition*
$type	Document *base transition*
$entropy	Document *create transition*
message	Data Contract (the `message` document defined in the referenced data contract - `5wpZAEWndYcTeuwZpkmSa8s49cHXU5q2DhdibesxFSu8`)

Document Replace Transition

The document replace transition extends the base schema to include the following additional fields:

Field	Type	Description
$revision	integer	Document revision (=> 1)
$updatedAt	integer	(Optional)

Each document replace transition must comply with this JSON-Schema definition established in rs-dpp (in addition to the document transition base schema) that is required for all document transitions):

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "$revision": {
      "type": "integer",
      "minimum": 1
    },
    "$updatedAt": {
      "type": "integer",
      "minimum": 0
    }
  },
  "required": [
    "$revision"
  ],
  "additionalProperties": false
}
```

Note: The document create transition must also include all required properties of the document as defined in the data contract.

The following example document create transition and subsequent table demonstrate how the document transition base, document create transition, and data contract document definitions are assembled into a complete transition for inclusion in a *state transition*:

```
{
  "$action": 1,
  "$dataContractId": "5wpZAEWndYcTeuwZpkmSa8s49cHXU5q2DhdibesxFSu8",
  "$id": "6oCKUeLVgjr7VZCyn1LdGbrepqKLmoabaff5WQqyTKYP",
  "$type": "note",
  "$revision": 1,
  "message": "Tutorial Test @ Mon, 27 Apr 2020 20:23:35 GMT"
}
```

Field	Required By
$action	Document *base transition*
$dataContractId	Document *base transition*
$id	Document *base transition*
$type	Document *base transition*
$revision	Document revision
message	Data Contract (the `message` document defined in the referenced data contract - `5wpZAEWndYcTeuwZpkmSa8s49cHXU5q2DhdibesxFSu8`)

Document Delete Transition

The document delete transition only requires the fields found in the *base document transition*.

Example Document Batch State Transition

```
{
  "protocolVersion": 1,
  "type": 1,
  "signature": "ICu/H7MoqxNUzznP9P2aTVEo91VVy0T8M3QWCH/
↪7dg2UVokG98TbD4DQB4E8SD4GzHoRrBMycJ75SbT2AaF9hFc=",
  "signaturePublicKeyId": 0,
  "ownerId": "4ZJsE1Yg8AosmC4hAeo3GJgso4N9pCoa6eCTDeXsvdhn",
  "transitions": [
    {
      "$id": "8jm8iHsYE6ENENvFVeFVFMCwfgEqo5P1iR2q4KAYgpbS",
      "$type": "note",
      "$action": 1,
      "$dataContractId": "AnmBaYH13RyiuvBkBD6qkdc36H5DKt6ToMrkqgUnnywz",
      "message": "Updated document @ Mon, 26 Oct 2020 14:58:31 GMT",
      "$revision": 2
    }
  ]
}
```

1.35.2 Document Object

The document object represents the data provided by the platform in response to a query. Responses consist of an array of these objects containing the following fields as defined in the Rust reference client (rs-dpp):

Property	Type	Required	Description
protocolVersion	integer	Yes	The platform protocol version (currently 1)
$id	array	Yes	The *document ID* (32 bytes)
$type	string	Yes	Document type defined in the referenced contract (1-64 characters)
$revision	integer	No	Document revision (=>1)
$dataContractId	array	Yes	Data contract ID *generated* from the data contract's `ownerId` and `entropy` (32 bytes)
$ownerId	array	Yes	*Identity* of the user submitting the document (32 bytes)
$createdAt	integer	(Optional)	Time (in milliseconds) the document was created
$updatedAt	integer	(Optional)	Time (in milliseconds) the document was last updated

Each document object must comply with this JSON-Schema definition established in rs-dpp:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "$protocolVersion": {
      "type": "integer",
      "$comment": "Maximum is the latest protocol version"
    },
    "$id": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "$type": {
      "type": "string"
    },
    "$revision": {
      "type": "integer",
      "minimum": 1
    },
    "$dataContractId": {
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "$ownerId": {
```

(continues on next page)

(continued from previous page)

```
      "type": "array",
      "byteArray": true,
      "minItems": 32,
      "maxItems": 32,
      "contentMediaType": "application/x.dash.dpp.identifier"
    },
    "$createdAt": {
      "type": "integer",
      "minimum": 0
    },
    "$updatedAt": {
      "type": "integer",
      "minimum": 0
    }
  },
  "required": [
    "$protocolVersion",
    "$id",
    "$type",
    "$revision",
    "$dataContractId",
    "$ownerId"
  ],
  "additionalProperties": false
}
```

Example Document Object

```
{
  "$protocolVersion": 1,
  "$id": "4mWnFcDDzCpeLExJqE8v7pfN4EERC8NE2xn4hw3VKriU",
  "$type": "note",
  "$dataContractId": "63au7XVDt8aHtPrsYKoHx2bnRTSenwH62pDN1BQ5n5m9",
  "$ownerId": "7TkaE5uhG3T9AhyEkAvYCqZvRH4pyBibhjuSYPReNfME",
  "$revision": 1,
  "message": "Tutorial Test @ Mon, 26 Oct 2020 15:54:35 GMT",
  "$createdAt": 1603727675072,
  "$updatedAt": 1603727675072
}
```

1.36 Data Trigger

1.36.1 Data Trigger Overview

Although *data contracts* provide much needed constraints on the structure of the data being stored on Dash Platform, there are limits to what they can do. Certain system data contracts may require server-side validation logic to operate effectively. For example, *DPNS* must enforce some rules to ensure names remain DNS compatible. Dash Platform Protocol (DPP) supports this application-specific custom logic using Data Triggers.

1.36.2 Details

Since all application data is submitted in the form of documents, data triggers are defined in the context of documents. To provide even more granularity, they also incorporate the *document transition action* so separate triggers can be created for the CREATE, REPLACE, or DELETE actions.

When document state transitions are received, DPP checks if there is a trigger associated with the document transition type and action. If there is, it then executes the trigger logic.

Note: Successful execution of the trigger logic is necessary for the document to be accepted and applied to the platform state.

Example

As an example, DPP contains several data triggers for DPNS as defined in the data triggers factory. The `domain` document has added constraints for creation. All DPNS document types have constraints on replacing or deleting:

Data Contract	Document	Action(s)	Trigger Description
DPNS	`domain`	`CREATE`	Enforces DNS compatibility, validates provided hashes, and restricts top-level domain (TLD) registration
—-	—-	—-	—-
DPNS	All Document Types	`REPLACE`	Prevents updates to existing documents
DPNS	All Document Types	`DELETE`	Prevents deletion of existing documents

DPNS Trigger Constraints

The following table details the DPNS constraints applied via data triggers. These constraints are in addition to the ones applied directly by the DPNS data contract.

Document	Action	Constraint
`domain`	`CREATE`	Full domain length <= 253 characters
`domain`	`CREATE`	`normalizedLabel` matches lowercase `label`
`domain`	`CREATE`	`ownerId` matches `records.dashUniqueIdentityId` or `dashAliasIdentityId` (whichever one is present)
`domain`	`CREATE`	Only creating a top-level domain with an authorized identity
`domain`	`CREATE`	Referenced `normalizedParentDomainName` must be an existing parent domain
`domain`	`CREATE`	Subdomain registration for non top level domains prevented if `subdomainRules.allowSubdomains` is true
`domain`	`CREATE`	Subdomain registration only allowed by the parent domain owner if `subdomainRules.allowSubdomains` is false
`domain`	`CREATE`	Referenced `preorder` document must exist
`domain`	`REPLACE`	Action not allowed
`domain`	`DELETE`	Action not allowed
`preorder`	`REPLACE`	Action not allowed
`preorder`	`DELETE`	Action not allowed

1.37 Consensus Errors

1.37.1 Platform Error Codes

A comprehensive set of consensus error codes were introduced in Dash Platform v0.21. The tables below follow the codes found in code.js of the consensus source code.

The error codes are organized into four categories that each span 1000 error codes. Each category may be further divided into sub-categories. The four categories and their error code ranges are:

Category	Code range	Description
Basic	1000 - 1999	Errors encountered while validating structure and data
Signature	2000 - 2999	Errors encountered while validating identity existence and state transition signature
Fee	3000 - 3999	Errors encountered while validating an identity's balance is sufficient to pay fees
State	4000 - 4999	Errors encounter while validating state transitions against the platform state

1.37.2 Basic

Basic errors occupy the codes ranging from 1000 to 1999. This range is divided into several categories for clarity.

Decoding Errors

Code	Error Description	Comment
1000	ProtocolVersionParsingError	
1001	SerializedObjectParsingError	
1002	UnsupportedProtocolVersionError	
1003	IncompatibleProtocolVersionError	

Structure Errors

Code	Error Description	Comment
1004	JsonSchemaCompilationError	
1005	JsonSchemaError	
1006	InvalidIdentifierError	
1060	ValueError	**Added in v0.24**

Data Contract Errors

Code	Error Description	Comment
1007	DataContractMaxDepthExceedError	
1008	DuplicateIndexError	
1009	IncompatibleRe2PatternError	
1010	InvalidCompoundIndexError	
1011	InvalidDataContractIDError	
1012	InvalidIndexedPropertyConstraintError	
1013	InvalidIndexPropertyTypeError	
1014	InvalidJsonSchemaRefError	
1015	SystemPropertyIndexAlreadyPresentError	
1016	UndefinedIndexPropertyError	
1017	UniqueIndicesLimitReachedError	
1048	DuplicateIndexNameError	Added in v0.22
1050	InvalidDataContractVersionError	4013 prior to v0.23
1051	IncompatibleDataContractSchemaError	4014 prior to v0.23
1052	DataContractImmutablePropertiesUpdateError	4015 prior to v0.23
1053	DataContractUniqueIndicesChangedError	4016 prior to v0.23
1054	DataContractInvalidIndexDefinitionUpdateError	Added in v0.23
1055	DataContractHaveNewUniqueIndexError	Added in v0.23

Document Errors

Code	Error Description	Comment
1018	DataContractNotPresentError	
1019	DuplicateDocumentTransitionsWithIDsError	
1020	DuplicateDocumentTransitionsWithIndicesError	
1021	InconsistentCompoundIndexDataError	
1022	InvalidDocumentTransitionActionError	
1023	InvalidDocumentTransitionIDError	
1024	InvalidDocumentTypeError	
1025	MissingDataContractIDBasicError	
1026	MissingDocumentTransitionActionError	
1027	MissingDocumentTransitionTypeError	
1028	MissingDocumentTypeError	

Identity Errors

Code	Error Description	Comment
1029	DuplicatedIdentityPublicKeyBasicError	
1030	DuplicatedIdentityPublicKeyIDBasicError	
1031	IdentityAssetLockProofLockedTransactionMismatchError	
1032	IdentityAssetLockTransactionIsNotFoundError	
1033	IdentityAssetLockTransactionOutputAlreadyExistsError	
1034	IdentityAssetLockTransactionOutputNotFoundError	
1035	InvalidAssetLockProofCoreChainHeightError	
1036	InvalidAssetLockProofTransactionHeightError	
1037	InvalidAssetLockTransactionOutputReturnSizeError	
1038	InvalidIdentityAssetLockTransactionError	
1039	InvalidIdentityAssetLockTransactionOutputError	
1040	InvalidIdentityPublicKeyDataError	
1041	InvalidInstantAssetLockProofError	
1042	InvalidInstantAssetLockProofSignatureError	
1046	MissingMasterPublicKeyError	Added in v0.22
1047	InvalidIdentityPublicKeySecurityLevelError	Added in v0.22
1056	InvalidIdentityKeySignatureError	Added in v0.23
1057	InvalidIdentityCreditWithdrawalTransitionOutputScriptError	**Added in v0.24**
1058	InvalidIdentityCreditWithdrawalTransitionCoreFeeError	**Added in v0.24**
1059	NotImplementedIdentityCreditWithdrawalTransitionPoolingError	**Added in v0.24**

State Transition Errors

Code	Error Description	Comment
1043	InvalidStateTransitionTypeError	
1044	MissingStateTransitionTypeError	
1045	StateTransitionMaxSizeExceededError	

1.37.3 Signature Errors

Signature errors occupy the codes ranging from 2000 to 2999.

Code	Error Description	Comment
2000	IdentityNotFoundError	
2001	InvalidIdentityPublicKeyTypeError	
2002	InvalidStateTransitionSignatureError	
2003	MissingPublicKeyError	
2004	InvalidSignaturePublicKeySecurityLevelError	Added in v0.23
2005	WrongPublicKeyPurposeError	Added in v0.23
2006	PublicKeyIsDisabledError	Added in v0.23
2007	PublicKeySecurityLevelNotMetError	Added in v0.23

1.37.4 Fee Errors

Fee errors occupy the codes ranging from 3000 to 3999.

Code	Error Description	Comment
3000	BalanceIsNotEnoughError	Current credits balance is insufficient to pay fee

1.37.5 State

State errors occupy the codes ranging from 4000 to 4999. This range is divided into several categories for clarity.

Data Contract Errors

Code	Error Description	Comment
4000	DataContractAlreadyPresentError	

Document Errors

Code	Error Description	Comment
4004	DocumentAlreadyPresentError	
4005	DocumentNotFoundError	
4006	DocumentOwnerIdMismatchError	
4007	DocumentTimestampsMismatchError	
4008	DocumentTimestampWindowViolationError	
4009	DuplicateUniqueIndexError	
4010	InvalidDocumentRevisionError	

Identity Errors

Code	Error Description	Comment
4011	IdentityAlreadyExistsError	
4012	IdentityPublicKeyDisabledAtWindowViolationError	Added in v0.23
4017	IdentityPublicKeyIsReadOnlyError	Added in v0.23
4018	InvalidIdentityPublicKeyIdError	Added in v0.23
4019	InvalidIdentityRevisionError	Added in v0.23
4020	StateMaxIdentityPublicKeyLimitReachedError	Added in v0.23
4021	DuplicatedIdentityPublicKeyStateError	Added in v0.23
4022	DuplicatedIdentityPublicKeyIdStateError	Added in v0.23
4023	IdentityPublicKeyIsDisabledError	Added in v0.23
4024	IdentityInsufficientBalanceError	**Added in v0.24**

Data Trigger Errors

Code	Error Description	Comment
4001	DataTriggerConditionError	
4002	DataTriggerExecutionError	
4003	DataTriggerInvalidResultError	

1.38 Repository Overview

Change to monorepo

Dash Platform v0.21 migrated to a monorepo structure to streamline continuous integration builds and testing. A number of the libraries below were previously independent repositories but now are aggregated into the `packages` directory of the monorepo (https://github.com/dashpay/platform/).

1.38.1 js-dash-sdk

Dash client-side JavaScript library for application development and wallet payment/signing. Uses wallet-lib, dapi-client, and dashcore-lib to expose layer-1 and layer-2 functionality. Main user is app developers.

npm: `dash`
Repository

1.38.2 js-dapi-client

Client library for accessing *DAPI Endpoints* . Enables interaction with Dash platform through the *DAPI* hosted on masternodes. Provides automatic masternode discovery starting from any initial masternode.

npm: `@dashevo/dapi-client`
Repository

1.38.3 dapi

A decentralized API for the Dash network. Exposes endpoints for interacting with the layer 1 blockchain and layer 2 platform services.

Repository

1.38.4 js-dpp

JavaScript implementation of *Dash Platform Protocol*. Performs validation of all data submitted to the platform.

npm: `@dashevo/dpp`
Repository

1.38.5 Supporting Repositories

drive

Manages the platform state and provides decentralized application storage on the Dash network.

Repository

dashcore-lib

A JavaScript Dash library

npm: `@dashevo/dashcore-lib`
Repository: https://github.com/dashpay/dashcore-lib

grove-db

A hierarchical authenticated data structure. The construction is based on Database Outsourcing with Hierarchical Authenticated Data Structures.

Repository

wallet-lib

An extensible JavaScript Wallet Library for Dash. Provides layer 1 SPV wallet functionality.

npm: `@dashevo/wallet-lib`
Repository

dapi-grpc

Decentralized API gRPC definition files and generated clients. Used by clients (e.g. dapi-client) to interact with DAPI endpoints.

npm: `@dashevo/dapi-grpc`
Repository

dash-network-deploy

Tool for assisting Dash devnet network deployment and testing.

https://github.com/dashpay/dash-network-deploy

platform-test-suite

Test suite for end-to-end testing of Dash Platform by running some real-life scenarios against a Dash Network.

Repository

rs-drive

Implements secondary indices for Platform in conjunction with GroveDB.

Repository

dashmate

A distribution package for Dash masternode installation.

Repository

1.38.6 Contract Repositories

dashpay-contract

DashPay contract documents JSON Schema

Repository

dpns-contract

DPNS contract documents JSON Schema

Repository

1.39 Source Code

Source code produced by Dash Core Group is located in two GitHub organizations:

- Dashpay - Dash Core Blockchain software and documention
- Dashevo - Dash Platform software

1.40 Overview

Dash library for JavaScript/TypeScript ecosystem (Wallet, DAPI, Primitives, BLS, ...)

Dash library provides access via *DAPI* to use both the Dash Core network and Dash Platform on supported networks. The Dash Core network can be used to broadcast and receive payments. Dash Platform can be used to manage identities, register data contracts for applications, and submit or retrieve application data via documents.

1.40.1 Install

From NPM

In order to use this library, you will need to add our NPM package to your project.

Having NodeJS installed, just type:

```
npm install dash
```

From unpkg

```
<script src="https://unpkg.com/dash"></script>
```

Usage examples

- *Generate a mnemonic*
- *Receive money and display balance*
- *Pay to another address*
- *Use another BIP44 account*

Dash Platform Tutorials

See the *Tutorial section* of the Dash Platform documentation for examples.

1.40.2 Licence

MIT © Dash Core Group, Inc.

1.41 Examples

1.41.1 Fetching an identity from its name

Assuming you have created an identity and attached a name to it (see how to *register an identity* and how to *attach it to a name*), you will then be able to directly recover an identity from its names. See below:

```
const client = new Dash.Client({
  wallet: {
    mnemonic: '', // Your app mnemonic, which holds the identity
  },
});

// This is the name previously registered in DPNS.
const identityName = 'alice';

const nameDocument = await client.platform.names.resolve(`${identityName}.dash`);
const identity = await client.platform.identities.get(nameDocument.ownerId);
```

1.41.2 Generate a new mnemonic

In order to be able to keep your private keys private, we encourage to create your own mnemonic instead of using those from the examples (that might be empty). Below, you will be proposed two options allowing you to create a new mnemonic, depending on the level of customisation you need.

Dash.Client

By passing `null` to the mnemonic value of the wallet options, you can get Wallet-lib to generate a new mnemonic for you.

```
const Dash = require("dash");
const client = new Dash.Client({
  network: "testnet",
  wallet: {
    mnemonic: null,
  },
});
const mnemonic = client.wallet.exportWallet();
console.log({mnemonic});
```

Dash.Mnemonic

```
const Dash = require("dash");
const {Mnemonic} = Dash.Core;

const mnemnonic = new Mnemonic().toString()
```

Language selection

```
const {Mnemonic} = Dash.Core;
const {CHINESE, ENGLISH, FRENCH, ITALIAN, JAPANESE, SPANISH} = Mnemonic.Words;
console.log(new Mnemonic(Mnemonic.Words.FRENCH).toString())
```

Entropy size

By default, the value for mnemonic is `128` (12 words), but you can generate a 24 words (or other) :

```
const {Mnemonic} = Dash.Core;
console.log(new Mnemonic(256).toString())
```

You can even replace the word list by your own, providing a list of 2048 unique words.

1.41.3 Paying to another address

In order to pay, you need to have an *existing balance*.
The below code will allow you to pay to a single address a specific amount of satoshis.

```
const Dash = require('dash');

const mnemonic = ''; // your mnemonic here.
const client = new Dash.Client({
  wallet: {
    mnemonic,
  },
});

async function payToRecipient(account) {
  const transaction = account.createTransaction({
    recipient: 'yNPbcFfabtNmmxKdGwhHomdYfVs6gikbPf',
    satoshis: 10000,
  });
  const transactionId = await account.broadcastTransaction(transaction);
}

client.wallet.getAccount().then(payToRecipient);
```

See more on create transaction options here.

1.41.4 Receive money and display balance

Initialize client

Initialize the SDK Client with your *generated mnemonic* passed as an option.

```
const Dash = require("dash");
const mnemonic = ''// your mnemonic here.
const client = new Dash.Client({
  wallet: {
    mnemonic,
  }
});

async function showBalance() {
  const account = await client.wallet.getAccount();
  const totalBalance = account.getTotalBalance();
  console.log(`Account's total balance: ${totalBalance} duffs`);
}
```

Having your `client` instance set up, you will be able to access the `account` and `wallet` instance generated from your mnemonic.

By default `getAccount()` returns the first BIP44 account.
You can read more on *how to use a different account*.

Generate a receiving address

Dash wallet supports two different types of addresses:

- `external` addresses used for receiving funds from other addresses
- `internal` addresses used for change outputs of outgoing transactions
- For your privacy, you might want to generate a new address for each payment:

```
async function generateUnusedAddress() {
  const account = await client.wallet.getAccount();
  const { address } = account.getUnusedAddress();
  console.log(`Unused external address: ${address}`);
}
```

This above code will generate a new unique (never used) address.

Displaying your balance

Dash Wallet returns the balance in duffs (1 Dash is equal to 100.000.000 duffs)

`getTotalBalance()` function takes into account `confirmed` and `unconfirmed` transactions (not included in a block). It is recommended to check the confirmed balance before making a payment:

```
async function showBalance() {
  const account = await client.wallet.getAccount();
  const totalBalance = account.getTotalBalance();
  const confirmedBalance = account.getConfirmedBalance();
  const unconfirmedBalance = account.getUnconfirmedBalance();
  console.log(`Account balance:
    Confirmed: ${confirmedBalance}
    Unconfirmed: ${unconfirmedBalance}
    Total: ${totalBalance}
  `);
}
```

Listen for event on received transaction

When a new unconfirmed transaction is received, you can receive an event, and then validate the address or perform an action if needed.

```
// FETCHED/UNCONFIRMED_TRANSACTION event is currently disabled

async function listenUnconfirmedTransaction() {
  const account = await client.wallet.getAccount();
  account.on('FETCHED/UNCONFIRMED_TRANSACTION', (data) => {
    console.dir(data);
  });
}
```

Get address at specific index

In case you want to retrieve an address at specific index:

```
async function getAddressAtIndex() {
  const account = await client.wallet.getAccount();
  const { address: externalAddress } = account.getAddress(2);
  const { address: internalAddress } = account.getAddress(2, 'internal');
}
```

1.41.5 Sign and verify messages

Dash SDK exports the Message constructor inside the Core namespace `new Dash.Core.Message`

```
const Dash = require('dash');

const mnemonic = '';

const client = new Dash.Client({
  wallet: {
    mnemonic,
  },
});

async function signAndVerify() {
  const account = await client.wallet.getAccount();

  const pk = new Dash.Core.PrivateKey();
  const message = new Dash.Core.Message('hello, world');
  const signed = account.sign(message, pk);
  const verified = message.verify(pk.toAddress().toString(), signed.toString());
}
```

1.41.6 Using a different account

Clients initialized with a mnemonic support multiple accounts as defined in BIP44.

By default `client.wallet.getAccount()` returns the account at index `0`.

To access other accounts, pass the `index` option:

```
const secondAccount = await client.wallet.getAccount({ index: 1 })
```

1.42 Getting started

1.42.1 About Schemas

Schemas represents the application data structure, a JSON Schema language based set of rules that allows the creation of a Data Contract.

You can read more in the *Dash Platform Documentation - Data contract section*.

1.42.2 Core concepts

The Dash Core Developer Guide will answer most of questions about the fundamentals of Dash. However, some elements provided by the SDK need to be grasped, so we will quickly cover some of those.

Wallet

At the core of Dash is the Payment Chain. In order to be able to transact on it, one needs to have a set of UTXOs that are controlled by a Wallet instance.

In order to access your UTXO, you will have to provide a valid mnemonic that will unlock the Wallet and automatically fetch the associated UTXOs.

When an SDK instance is created, you can access your wallet via the `client.wallet` variable. (Check wallet-lib documentation for more details)

Account

Since the introduction of deterministic wallets (BIP44), a wallet is represented by multiple accounts.

It is the instance you will use most of the time for receiving or broadcasting payments.

You can access your account with `client.getWalletAccount()`. See *how to use a different account* if you need to get an account at a specific index.

App Schema and Contracts

The Dash Platform Chain provides developers with the ability to create applications.
Each application requires a set of rules and conditions described as a portable document in the form of a JSON Schema.

When registered, those applications schemas are called contracts and contains a contractId (namespace : `client.platform.contracts`).

By default, this library supports Dash Platform Name Service (DPNS) (to attach a name to an identity), under the namespace `client.platform.names` for testnet.

See: *how to use multiple apps*

1.42.3 Dash Platform applications

DPNS

DPNS is handled in the Dash SDK Client under the namespace `client.platform.names.*'`. *Read more here*

DashPay

The DashPay contract is registered on testnet under contract id `Bwr4WHCPz5rFVAD87RqTs3izo4zpzwsEdKPWUT1NS1C7`. Its functionality is not incorporated with the Dash SDK at this time.

1.42.4 Working with multiple apps

When working with other registered contracts, you will need to know their `contractId` and reference it in the SDK constructor.

Assuming a contract DashPay has the following `contractId: "77w8Xqn25HwJhjodrHW133aXhjuTsTv9ozQaYpSHACE3"`. You can then pass it as an option.

```
const client = new Dash.Client({
  apps: {
    dashpay: {
      contractId: '77w8Xqn25HwJhjodrHW133aXhjuTsTv9ozQaYpSHACE3'
    }
  }
});
```

This allow the method `client.platform.documents.get` to provide you field selection. Therefore, if the contract has a `profile` field that you wish to access, the SDK will allow you to use dot-syntax for access :

```
const bobProfile = await client.platform.documents.get('dashpay.profile', { name: 'bob' }
↪);
```

1.42.5 Quick start

In order to use this library, you will need to add our NPM package to your project.

Having NodeJS installed, just type :

```
npm install dash
```

Initialization

Let's create a Dash SDK client instance specifying both our mnemonic and the schema we wish to work with.

```
const Dash = require('dash');
const opts = {
  wallet: {
    mnemonic: "arena light cheap control apple buffalo indicate rare motor valid␣
↪accident isolate",
  },
```

(continues on next page)

(continued from previous page)

```
};
const client = new Dash.Client(opts);
client.wallet.getAccount().then(async (account) => {
  // Do something
})
```

Quick note: If no `mnemonic` is provided or `mnemonic: null` is passed inside the `wallet` option, a new mnemonic will be generated.

Make a payment

```
client.wallet.getAccount().then(async (account) => {
  const transaction = account.createTransaction({
    recipient: 'yixnmigzC236WmTXp9SBZ42csyp9By6Hw8',
    amount: 0.12,
  });
  await account.broadcastTransaction(transaction);
});
```

Interact with Dash Platform

See the *Tutorial section* of the Dash Platform documentation for examples.

1.42.6 TypeScript

In order to use Dash SDK with TypeScript.

Create an index.ts file

```
import Dash from 'dash';
const clientOpts = {
  wallet: {
    mnemonic: null, // Will generate a new address, you should keep it.
  },
};
const client = new Dash.Client(clientOpts);

const initializeAccount = async () => {
  const account = await client.wallet.getAccount();
  const balance = account.getTotalBalance();
  console.log(`Account balance: ${balance}`)
}
```

Have a following `tsconfig.json` file

```
{
  "compilerOptions": {
    "module": "commonjs",
    "moduleResolution": "node",
    "esModuleInterop": true
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Compile: `tsc -p tsconfig.json`
Run: `node index.js`

1.43 Platform

The Dash Platform provides a technology stack on the top of Dash Network that allows creation of feature-rich decentralized applications.

You can learn more from the *Dash Platform Documentation - What is Dash Platform?*

1.43.1 Platform components

- DAPI: A decentralized API that runs on all Masternodes and offers gRPC endpoints for retrieving payment chain metadata (blocks, transactions), as well as application data (documents, contracts, identities).
- Drive: Application chain storage layer where the data defined by Data Contracts is stored and managed.
- DPNS: Naming service provided by a Dash Platform App

Contracts

What is a contract

Contracts are registered sets of rules defined in a *JSON Application Schema*.

See the Dash Platform documentation for more information about *Data Contracts*.

Create

Usage: `client.platform.contracts.create(contractDefinitions, identity)`
Description: This method will return a Contract object initialized with the parameters defined and apply to the used identity.

Parameters:

parameters	type	required	Description
contractDefinitions	JSONDataContract	yes	The defined *JSON Application Schema*
identity	Identity	yes	A valid *registered* `application` *identity*

Example:

```
const identityId = '';// Your identity identifier.

// Your valid json contract definitions
const contractDefinitions = {
  note: {
```

(continues on next page)

(continued from previous page)

```
    properties: {
      message: {
        type: "string"
      }
    },
    additionalProperties: false
  }
};
const identity = await client.platform.identities.get(identityId);
const contract = client.platform.contracts.create(contractDefinitions, identity);

// You can use the validate method from DPP to validate the created contract
const validationResult = client.platform.dpp.dataContract.validate(contract);
```

Note: When your contract is created, it will only exist locally. Use the *publish* method to register it.

Returns: Contract.

Get

Usage: `client.platform.contracts.get(contractId)`
Description: This method will allow you to fetch back a contract from its id.

Parameters:

parameters	type	required	Description
identifier	string	yes	Will fetch back the contract matching the identifier

Example: `await client.platform.contracts.get('77w8Xqn25HwJhjodrHW133aXhjuTsTv9ozQaYpSHACE3')`

Returns: Contract (or `null` if it's not a registered contract).

Publish

Usage: `client.platform.contracts.publish(contract, identity)`
Description: This method will sign and broadcast any valid contract.

Parameters:

parameters	type	required	Description
contract	Contract	yes	A valid *created contract*
identity	Identity	yes	A valid *registered* `application` *identity*

Example:

```
const identityId = '';// Your identity identifier.
const identity = await client.platform.identities.get(identityId);
// See the contract.create documentation for more on how to create a dataContract
const contract = await client.platform.contracts.create(contractDefinitions, identity);
await client.platform.contracts.publish(contract, identity);
```

Returns : DataContractCreateTransition.

Update

Usage: `client.platform.contracts.update(contract, identity)`
Description: This method will sign and broadcast an updated valid contract.

Parameters:

parameters	type	required	Description
contract	Contract	yes	A valid *created contract*
identity	Identity	yes	A valid *registered* `application` *identity*

Returns: DataContractUpdateTransition.

Documents

What is a document

Documents in Dash Platform are similar to those in standard document-oriented databases (MongoDB,...).
They represent a record consisting of one, or multiples field-value pairs and should respect the structure of the data-Contract on which they are submitted in.

See more on the Dash Platform documentation about *Data Contract*.

Broadcast

Usage: `client.platform.document.broadcast(documents, identity)`
Description: This method will broadcast the document on the Application Chain

Parameters:

parameters	type	required	Description
documents	Object	yes	
documents.create	ExtendedDocument[]	no	array of valid *created document* to create
documents.replace	ExtendedDocument[]	no	array of valid *created document* to replace
documents.delete	ExtendedDocument[]	no	array of valid *created document* to delete
identity	Identity	yes	A valid *registered identity*

Example:

```
const identityId = '';// Your identity identifier
const identity = await client.platform.identities.get(identityId);

const helloWorldDocument = await client.platform.documents.create(
    // Assuming a contract tutorialContract is registered with a field note
    'tutorialContract.note',
    identity,
    { message: 'Hello World'},
);

await client.platform.documents.broadcast({ create: [helloWorldDocument] }, identity);
```

Returns: documents.

Create

Usage: `client.platform.documents.create(typeLocator, identity, documentOpts)`
Description: This method will return a ExtendedDocument object initialized with the parameters defined and apply to the used identity.

Parameters:

parameters	type	required	Description
dotLocator	string	yes	Field of a specific application, under the form `appName.fieldName`
identity	Identity	yes	A valid *registered identity*
docOpts	Object	yes	A valid data that match the data contract structure

Example:

```
const identityId = '';// Your identity identifier
const identity = await client.platform.identities.get(identityId);

const helloWorldDocument = await client.platform.documents.create(
      // Assume a contract helloWorldContract is registered with a field note
      'helloWorldContract.note',
      identity,
     { message: 'Hello World'},
  );
```

Note: When your document is created, it will only exist locally, use the *broadcast* method to register it.

Returns: ExtendedDocument

Get

Usage: `client.platform.documents.get(typeLocator, opts)`
Description: This method will allow you to fetch back documents matching the provided parameters.

Parameters:

parameters	type	required	Description
typeLocator	string	yes	Field of a specific application, under the form `appName.fieldName`
opts	object	no (default: {})	Query options of the request

Queries options:

parameters	type	required	Description
where	array	no	Mongo-like where query
orderBy	array	no	Mongo-like orderBy query
limit	integer	no	how many objects to fetch
startAt	integer	no	number of objects to skip
startAfter	integer	no	exclusive skip

Learn more about query syntax.

Example:

```
  const queryOpts = {
        where: [
            ['normalizedLabel', '==', 'alice'],
            ['normalizedParentDomainName', '==', 'dash'],
        ],
    };
 await client.platform.documents.get('dpns.domain', queryOpts);
```

Identities

What is an identity

An Identity is a blockchain-based identifier for individuals (users) and applications.
Identities are the atomic element that, when linked with additional applications, can be extended to provide new functionality.

Read more on the Dash Platform documentation about *Identity*.
You might also want to consult the usage for the *DPNS Name Service* in order to attach a name to your created identity.

Credits

Each identity contains a credit balance. The ratio is 1 duff = 1000 credits.

Get

Usage: `client.platform.identities.get(identityId)`
Description: This method will allow you to fetch back an identity from its id.

Parameters:

parameters	type	required	Description
identifier	string	yes	Will fetch back the identity matching the identifier

Example: `await client.platform.identities.get('3GegupTgRfdN9JMS8R6QXF3B2VbZtiw63eyudh1oMJAk')`

Returns: Identity (or `null` if it does not exist).

Register

Usage: `client.platform.identities.register()`
Description: This method will register a new identity for you.

Parameters:

parameters	type	required	Description
fundingAmount	number	no	Defaults: 10000. Allow to set a funding amount in duffs (satoshis).

Example: `await client.platform.identities.register()`

Note: The created identity will be associated to the active account. You might want to know more about how to *change your active account*.

Returns: Identity.

Topup

Usage: `client.platform.identities.topUp(identity, amount)`
Description: This method will topup the provided identity's balance.

The identity balance might slightly vary from the topped up amount because of the transaction fee estimation.

Parameters:

parameters	type	required	Description
identity	Identity	yes	A valid *registered identity*
amount	number	yes	A duffs (satoshis) value corresponding to the amount you want to top up to the identity.

Example:

```
const identityId = '';// Your identity identifier
const identity = await client.platform.identities.get(identityId);
await client.platform.identities.topUp(identity.getId(), 10000);

console.log(`New identity balance: ${identity.balance}`)
```

Returns: Boolean.

Names

What is DPNS

DPNS is a special Dash Platform Application that is intended to provide a naming service for the Application Chain.

Decoupling name from the blockchain identity enables a unique user experience coupled with the highest security while remaining compatible with Decentralized Identifiers.

Limitation: max length of 63 characters on charset `0-9`,`A-Z`(case insensitive), `-`.

Domain names are linked to an Identity.

Register

Usage: `client.platform.names.register(name, records, identity)`
Description: This method will create a DPNS record matching your identity to the user or appname defined.

Parameters:

parameters	type	required	Description
name	String	yes	An alphanumeric (1-63 character) value used for human-identification (can contain - but not as the first or last character). If a name with no parent domain is entered, '.dash' is used.
records	Object	yes	records object having only one of the following items
records.dashUniqueIdentityId	String	no	Unique Identity ID for this name record
records.dashAliasIdentityId	String	no	Used to signify that this name is the alias for another id
identity	Identity	yes	A valid *registered identity*

Example: `await client.platform.names.register('alice', { dashUniqueIdentityId: identity.getId() }, identity)`

Returns: the created domain document

Resolve

Usage: `client.platform.names.resolve('<name>.dash')`
Description: This method will allow you to resolve a DPNS record from its humanized name.

Parameters:

parameters	type	required	Description
name	String	yes	An alphanumeric (2-63) value used for human-identification (can contains -)

Example: `await client.platform.names.resolve('alice.dash')`

Returns : ExtendedDocument (or `null` if do not exist).

ResolveByRecord

Usage: `client.platform.names.resolveByRecord(record, value)`
Description: This method will allow you to resolve a DPNS record by identity ID.

Parameters:

parameters	type	required	Description
record	String	yes	Type of the record (`dashUniqueIdentityId` or `dashAliasIdentityId`)
value	String	yes	Identifier value for the record

Example:

This example will describe how to resolve names by the dash unique identity id.

```
const identityId = '3ge4yjGinQDhxh2aVpyLTQaoka45BkijkoybfAkDepoN';
const document = await client.platform.names.resolveByRecord('dashUniqueIdentityId',␣
↪identityId);
```

Returns: array of ExtendedDocument.

Search

Usage: `client.platform.names.search(labelPrefix, parentDomain)`
Description: This method will allow you to search all records matching the label prefix on the specified parent domain.

Parameters:

parameters	type	required	Description
labelPrefix	String	yes	label prefix to search for
parentDomain	String	yes	parent domain name on which to perform the search

Example:

This example will describe how to search all names on the parent domain `dash` that starts with the label prefix `al`.
It will resolves names documents such as `alice`, `alex` etc...

```
const labelPrefix = 'al';
const parentDomain = 'dash';
const document = await client.platform.names.search(labelPrefix, parentDomain);
```

Returns: Documents matching the label prefix on the parent domain.

1.44 Usage

1.44.1 DAPI

About DAPI

DAPI (Decentralized API) is a distributed and decentralized endpoints provided by the Masternode Network.

Get the DAPI-Client instance

```
 const dapiClient = client.getDAPIClient();
```

The usage is then *described here*.

1.44.2 Dashcore Lib primitives

All Dashcore lib primitives are exposed via the `Core` namespace.

```
const Dash = require('dash');
const {
  Core: {
    Block,
    Transaction,
    Address,
    // ...
  }
} = Dash;
```

Transaction

The Transaction primitive allows creating and manipulating transactions. It also allows signing transactions with a private key.
Supports fee control and input/output access (which allows passing a specific script).

```
const { Transaction } = Dash.Core;
const tx = new Transaction(txProps)
```

Access the Transaction documentation on dashpay/dashcore-lib

Address

Standardized representation of a Dash Address. Address can be instantiated from a String, PrivateKey, PublicKey, HDPrivateKey or HdPublicKey.
Pay-to-script-hash (P2SH) multi-signature addresses from an array of PublicKeys are also supported.

```
const { Address } = Dash.Core;
```

Access the Address documentation on dashpay/dashcore-lib

Block

Given a binary representation of the block as input, the Block class allows you to have a deserialized representation of a Block or its header. It also allows validating the transactions in the block against the header merkle root.

The block's transactions can also be explored by iterating over elements in array (`block.transactions`).

```
const { Block } = Dash.Core;
```

Access the Block documentation on dashpay/dashcore-lib

UnspentOutput

Representation of an UnspentOutput (also called UTXO as in Unspent Transaction Output).
Mostly useful in association with a Transaction and for Scripts.

```
const { UnspentOutput } = Dash.Core.Transaction;
```

Access the UnspentOutput documentation on dashpay/dashcore-lib

HDPublicKey

Hierarchical Deterministic (HD) version of the PublicKey.
Used internally by Wallet-lib and for exchange between peers (DashPay)

const { HDPublicKey } = Dash.Core;`

Access the HDKeys documentation on dashpay/dashcore-lib

HDPrivateKey

Hierarchical Deterministic (HD) version of the PrivateKey.
Used internally by Wallet-lib.

```
const { HDPrivateKey } = Dash.Core;
```

Access the HDKeys documentation on dashpay/dashcore-lib

PublicKey

```
const { PublicKey } = Dash.Core;
```

Access the PublicKey documentation on dashpay/dashcore-lib

PrivateKey

```
const { PrivateKey } = Dash.Core;
```

Access the PrivateKey documentation on dashpay/dashcore-lib

Mnemonic

Implementation of BIP39 Mnemonic code for generative deterministic keys.
Generates a random mnemonic with the chosen language, validates a mnemonic or returns the associated HDPrivateKey.

```
const { Mnemonic } = Dash.Core;
```

Access the Mnemonic documentation on dashpay/dashcore-lib

Network

A representation of the internal parameters relative to the selected network. By default, all primitives works with 'livenet'.

```
const { Network } = Dash.Core;
```

Access the Network documentation on dashpay/dashcore-lib

Script

```
const { Script } = Dash.Core.Transaction;
```

Access the Script documentation on dashpay/dashcore-lib

Input

`const { Input } = Dash.Core.Transaction;`

Access the Transaction documentation on dashpay/dashcore-lib

Output

`const { Output } = Dash.Core.Transaction;`

Access the Transaction documentation on dashpay/dashcore-lib

1.45 Wallet

1.45.1 About Wallet-lib

When Dash.Client is initiated with a `mnemonic` property, a wallet instance becomes accessible via `client.wallet` property.

To initialize the wallet account and synchronize with the network, use `client.wallet.getAccount()`.

Find out more about the Wallet in its complete documentation

Accounts

Getting an account

When Wallet is initialized with `mnemonic`, it holds multiple Accounts according to BIP44.
Each Account holds the keys needed to make a payments from it.

Wallet's `getAccount` method used to access an account:

```
const client = new Dash.Client({
  wallet: {
    mnemonic: "maximum blast eight orchard waste wood gospel siren parent deer athlete␣
↪impact",
  },
});

const account = await client.wallet.getAccount()
// Do something with account
```

As optional parameter, an integer representing the account `index` can be passed as parameter. By default, index account on call is 0.

```
client.wallet.getAccount({ index: 1 })
```

Awaiting for the `getAccount()` promise is necessary to ensure the wallet is synced-up with the network and make sure that the UTXO set is ready to be used for payment/signing.

Signing and encryption

Obtain account

```
const account = await client.wallet.getAccount();
```

Sign a Transaction

```
const tx = new Dash.Core.Transaction({
  // ...txOpts
});
const signedTx = account.sign(tx);
```

Encrypt a message

```
  const message = 'Something';
  const signedMessage = account.encrypt('AES', message, 'secret');
```

Decrypt a message

```
const encrypted = 'U2FsdGVkX19JLa+1UpbMcut1/QFWLMlKUS+iqz+7Wl4=';
const message = account.decrypt('AES', encrypted, 'secret');
```

1.46 Overview

1.46.1 DAPI-Client

Client library used to access Dash DAPI endpoints

This library enables HTTP-based interaction with the Dash blockchain and Dash Platform via the decentralized API (DAPI) hosted on Dash masternodes.

- `DAPI-Client` provides automatic server (masternode) discovery using either a default seed node or a user-supplied one
- `DAPI-Client` maps to DAPI's RPC and gRPC endpoints

Install

ES5/ES6 via NPM

In order to use this library in Node, you will need to add it to your project as a dependency.

Having NodeJS installed, just type in your terminal :

```
npm install @dashevo/dapi-client
```

CDN Standalone

For browser usage, you can also directly rely on unpkg :

```
<script src="https://unpkg.com/@dashevo/dapi-client"></script>
```

1.46.2 Licence

MIT © Dash Core Group, Inc.

1.47 Quick start

1.47.1 ES5/ES6 via NPM

In order to use this library in Node, you will need to add it to your project as a dependency.

Having NodeJS installed, just type in your terminal :

```
npm install @dashevo/dapi-client
```

1.47.2 CDN Standalone

For browser usage, you can also directly rely on unpkg :

```
<script src="https://unpkg.com/@dashevo/dapi-client"></script>
```

You can see an example usage here .

1.47.3 Initialization

```
const DAPIClient = require('@dashevo/dapi-client');
const client = new DAPIClient();

(async () => {
  const bestBlockHash = await client.core.getBestBlockHash();
  console.log(bestBlockHash);
})();
```

1.47.4 Quicknotes

This package allows you to fetch & send information from both the payment chain (layer 1) and the application chain (layer 2, a.k.a Platform chain).

1.48 Usage

1.48.1 DAPIClient

Usage: `new DAPIClient(options)`
Description: This method creates a new DAPIClient instance.

Parameters:

parameters	type	required[def value]	Description
options	Object		
options.dapiAddressProvider	DAPIAddressProvider	no[ListDAPIAddressProvider]	Allow to override the default dapiAddressProvider (do not allow seeds or dapiAddresses params)
options.seeds	string[]	no[seeds]	Allow to override default seeds (to connect to specific node)
options.network	string	no[=evonet]	Allow to setup the network to be used (livenet, testnet, evonet,..)
options.timeout	number	no[=2000]	Used to specify the timeout time in milliseconds.
options.retries	number	no[=3]	Used to specify the number of retries before aborting and erroring a request.
options.baseBanTime	number	no[=6000]	

Returns : DAPIClient instance.

```
const DAPIClient = require('@dashevo/dapi-client');
const client = new DAPIClient({
  timeout: 5000,
  retries: 3,
  network: 'livenet'
});
```

1.48.2 Core

broadcastTransaction

Usage: `await client.core.broadcastTransaction(transaction)`
Description: Allow to broadcast a valid **signed** transaction to the network.

Parameters:

parameters	type	required	Description
transaction	Buffer	yes	A valid Buffer representation of a transaction
options	Object		
options.allowHighFees	Boolean	no[=false]	As safety measure, "absurd" fees are rejected when considered to high. This allow to overwrite that comportement
options.bypassLimits	Boolean	no[=false]	Allow to bypass default transaction policy rules limitation

Returns : transactionId (string).

N.B : The TransactionID provided is subject to transaction malleability, and is not a source of truth (the transaction might be included in a block with a different txid).

generateToAddress

Usage: `await client.core.generateToAddress(blockMumber, address, options)`
Description: Allow to broadcast a valid **signed** transaction to the network.
Notes: Will only works on regtest.

Parameters:

parameters	type	required	Description
blocksNumber	Number	yes	A number of block to see generated on the regtest network
address	String	yes	The address that will receive the newly generated Dash
options	DAPIClientOptions	no	

Returns : {Promise<string[]>} - a set of generated blockhashes.

getBestBlockHash

Usage: `await client.core.getBestBlockHash(options)`
Description: Allow to fetch the best (highest/latest block hash) from the network

Parameters:

parameters	type	required	Description
options	DAPIClientOptions	no	

Returns : {Promise} - The best block hash

getBlockByHash

Usage: `await client.core.getBlockByHash(hash, options)`
Description: Allow to fetch a specific block by its hash

Parameters:

parameters	type	required	Description
hash	String	yes	A valid block hash
options	DAPIClientOptions	no	

Returns : {Promise<null|Buffer>} - The specified bufferized block

getBlockByHeight

Usage: `await client.core.getBlockByHeight(height, options)`
Description: Allow to fetch a specific block by its height

Parameters:

parameters	type	required	Description
height	Number	yes	A valid block height
options	DAPIClientOptions	no	

Returns : {Promise<null|Buffer>} - The specified bufferized block

getBlockHash

Usage: `await client.core.getBlockHash(height, options)`
Description: Allow to fetch a specific block hash from its height

Parameters:

parameters	type	required	Description
height	Number	yes	A valid block height
options	DAPIClientOptions	no	

Returns : {Promise<null|string>} - the corresponding block hash

getMnListDiff

Usage: `await client.core.getMnListDiff(baseBlockHash, blockHash, options)`
Description: Allow to fetch a specific block hash from its height

Parameters:

parameters	type	required	Description
baseBlockHash	String	yes	hash or height of start block
blockHash	String	yes	hash or height of end block
options	DAPIClientOptions	no	

Returns : {Promise} - The Masternode List Diff of the specified period

getStatus

Usage: `await client.core.getStatus(options)`
Description: Allow to fetch a specific block hash from its height

Parameters:

parameters	type	required	Description
options	DAPIClientOptions	no	

Returns : {Promise} - Status object

```
const status = await client.core.getStatus()
/**
{
  coreVersion: 150000,
  protocolVersion: 70216,
  blocks: 10630,
  timeOffset: 0,
  connections: 58,
  proxy: '',
  difficulty: 0.001745769130443678,
  testnet: false,
  relayFee: 0.00001,
  errors: '',
  network: 'testnet'
}
**/
```

getTransaction

Usage: `await client.core.getTransaction(id, options)`
Description: Allow to fetch a transaction by ID

Parameters:

parameters	type	required	Description
id	string	yes	A valid transaction id to fetch
options	DAPIClientOptions	no	

Returns : {Promise<null|Buffer>} - The bufferized transaction

subscribeToTransactionsWithProofs

Usage: `await client.core.subscribeToTransactionsWithProofs(bloomFilter, options = { count: 0 })`
Description: For any provided bloomfilter, it will return a ClientReadableStream streaming the transaction matching the filter.

Parameters:

parameters	type	required	Description
bloomFilter.vData	Uint8Array/Array	yes	The filter itself is simply a bit field of arbitrary byte-aligned size. The maximum size is 36,000 bytes.
bloomFilter.nHashFuncs	Number	yes	The number of hash functions to use in this filter. The maximum value allowed in this field is 50.
bloomFilter.nTweak	Number	yes	A random value to add to the seed value in the hash function used by the bloom filter.
bloomFilter.nFlags	Number	yes	A set of flags that control how matched items are added to the filter.
options.fromBlockHash	String	yes	Specifies block hash to start syncing from
options.fromBlockHeight	Number	yes	Specifies block height to start syncing from
options.count	Number	no (default: 0)	Number of blocks to sync, if set to 0 syncing is continuously sends new data as well

Returns : Promise|!grpc.web.ClientReadableStream<!TransactionsWithProofsResponse>

Example :

```
const filter; // A BloomFilter object
const stream = await client.subscribeToTransactionsWithProofs(filter, { fromBlockHeight:␣
↪0 });

stream
      .on('data', (response) => {
        const merkleBlock = response.getRawMerkleBlock();
        const transactions = response.getRawTransactions();

        if (merkleBlock) {
          const merkleBlockHex = Buffer.from(merkleBlock).toString('hex');
        }

        if (transactions) {
          transactions.getTransactionsList()
              .forEach((tx) => {
                  // tx are probabilistic, so you will have to verify it's yours
                  const tx = new Transaction(Buffer.from(tx));
              });
        }
      })
    .on('error', (err) => {
        // do something with err
      });
```

1.48.3 Platform

broadcastStateTransition

Usage: `async client.platform.broadcastStateTransition(stateTransition, options)`
Description: Send State Transition to machine

Parameters:

parameters	type	required	Description
stateTransition	Buffer	yes	A valid bufferized state transition
options	DAPIClientOptions	no	A valid state transition

Returns : Promise<!BroadcastStateTransitionResponse>

getDataContract

Usage: `async client.platform.getDataContract(contractId)`
Description: Fetch Data Contract by id

Parameters:

parameters	type	required	Description
contractId	String	yes	A valid registered contractId

Returns : Promise

getDocuments

Usage: `async client.platform.getDocuments(contractId, type, options)`
Description: Fetch Documents from Drive

Parameters:

parameters	type	required	Description
contractId	String	yes	A valid registered contractId
type	String	yes	DAP object type to fetch (e.g: 'preorder' in DPNS)
options.where	Object	yes	Mongo-like query
options.orderBy	Object	yes	Mongo-like sort field
options.limit	Number	yes	Limit the number of object to fetch
options.startAt	Number	yes	number of objects to skip
options.startAfter	Number	yes	exclusive skip

Returns : Promise<Buffer[]>

getIdentity

Usage: `async client.platform.getIdentity(id)`
Description: Fetch the identity by id

Parameters:

parameters	type	required	Description
id	String	yes	A valid registered identity

Returns : Promise<!Buffer|null>

getIdentityByFirstPublicKey

Usage: `async client.platform.getIdentityByFirstPublicKey(publicKeyHash)`
Description: Fetch the identity using the public key hash of the identity's first key

Parameters:

parameters	type	required	Description
publicKeyHash	String	yes	A valid public key hash

Returns : Promise<!Buffer|null>

getIdentityIdByFirstPublicKey

Usage: `async client.platform.getIdentityIdByFirstPublicKey(publicKeyHash)`
Description: Fetch the identity ID using the public key hash of the identity's first key

Parameters:

parameters	type	required	Description
publicKeyHash	String	yes	A valid public key hash

Returns : Promise<!Buffer|null>