Openchain Documentation

Release 0.7.0

Flavien Charlon

Sep 05, 2017

Contents

1 Overview **3**
- 1.1 Overview of Openchain . . . 3

2 Getting Started **5**
- 2.1 Getting started with the wallet . . . 5
- 2.2 Openchain Server Docker deployment . . . 16

3 General **19**
- 3.1 Running Openchain . . . 19
- 3.2 The transaction stream . . . 21
- 3.3 Anchoring and ledger integrity . . . 21
- 3.4 Openchain Server Configuration . . . 22
- 3.5 Openchain modules . . . 25
- 3.6 Setting the instance info on a new instance . . . 25
- 3.7 Upgrading Openchain server . . . 26
- 3.8 Deploying Openchain in a production environment . . . 26
- 3.9 Troubleshooting . . . 27

4 Public API **29**
- 4.1 Openchain data structures . . . 29
- 4.2 Ledger structure . . . 31
- 4.3 Method calls . . . 33

5 Ledger rules **41**
- 5.1 Default ledger rules . . . 41
- 5.2 Dynamic permissions . . . 44
- 5.3 How to: Configure a ledger to be closed-loop . . . 46

Openchain is an open source distributed ledger technology. It is suited for organizations wishing to issue and manage digital assets in a robust, secure and scalable way.

CHAPTER 1

Overview

Overview of Openchain

What is Openchain?

Openchain is an open source distributed ledger technology. It is suited for organizations wishing to issue and manage digital assets in a robust, secure and scalable way.

Features of Openchain include:

1. Instant confirmation of transactions.
2. No mining fees.
3. Extremely high scalability.
4. Secured through digital signatures.
5. *Immutability*: Commit an anchor in the Bitcoin Blockchain to benefit from the irreversibility of its Proof of Work.
6. *Assign aliases* to users instead of using base-58 addresses.
7. Multiple levels of control:
   - Fully open ledger that can be joined anonymously.
   - *Closed-loop ledger* where participants must be approved by the administrator.
   - A mix of the above where approved users enjoy more rights than anonymous users.
8. *Hierarchical account system* allowing to set permissions at any level.
9. Transparency and auditability of transactions.
10. *Handle loss or theft* of private keys without any loss to the end users.
11. Ability to have multiple Openchain instances *replicating from each other*.

Getting started

To familiarize yourself with Openchain, you can:

- *Try the wallet* against the test endpoint
- *Deploy your own Openchain server*

Frequently Asked Questions

Is Openchain a block chain?

Openchain falls under the umbrella of Blockchain technology. However, if we take the term "block chain" literally, Openchain is not a "block chain", but a close cousin. A block chain is a data structure that orders blocks of transactions and links them cryptographically through hashing.

Openchain doesn't use the concept of blocks. Transactions are directly chained with one another, and they are no longer grouped in blocks. Having to group transactions in blocks introduces a delay. Even if some systems manage to reduce the block time to just a few seconds, a few seconds is still a long time for latency-sensitive applications, such as trading. In Openchain, transactions are linked to the chain as soon as they are submitted to the network. As a result, Openchain is able to offer real-time confirmations.

This means that a more appropriate term for Openchain is a "transaction chain" rather than a "block chain".

Is Openchain a sidechain?

It is possible to use a pegging module that will act as a bridge between a Blockchain (such as Bitcoin) and an Openchain instance. When Bitcoins are sent to a specific address, a proxy for those coins will be created on the Openchain instance. Later on, these proxy tokens can be redeemed to unlock the Bitcoins on the main chain. This setup creates a 2-way peg between Bitcoin and the Openchain instance. In that scenario, the Openchain instance is behaving as a sidechain.

The pegging module is optional, and an instance doesn't have to be setup as a sidechain if that is not required.

Does Openchain support multi-signature?

Multi-signature is supported. Permissions are expressed using a list of public keys, and a number of require signatures. If you provide 3 public keys, and require 2 signatures, you have a 2-of-3 multi-signature account. Read about *dynamic permissions* to learn more about it.

CHAPTER 2

Getting Started

Getting started with the wallet

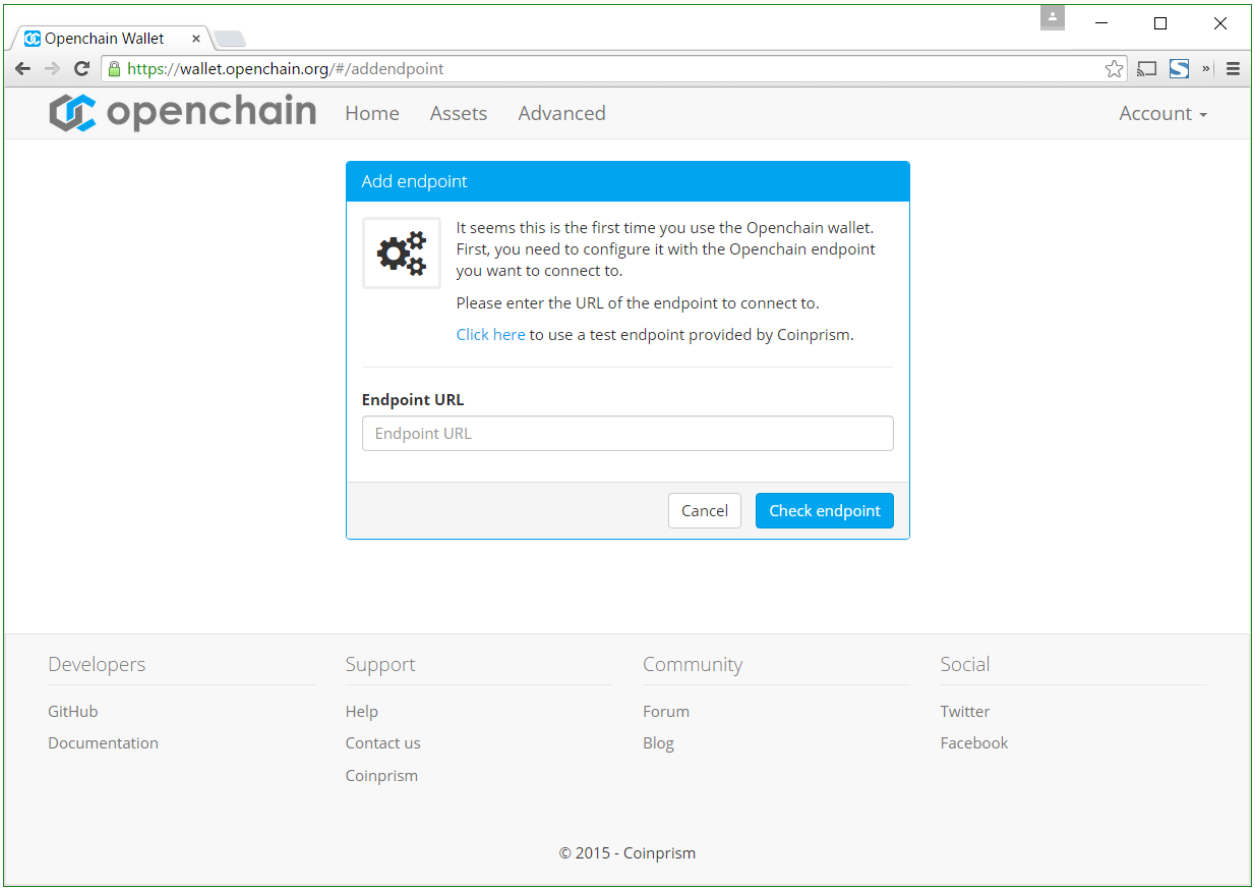
Openchain Server exposes a *public HTTP API*, which can be called by any program capable of making HTTP calls.

To wrap all those operations in a user-friendly user interface, we also provide a client: the Openchain Wallet.

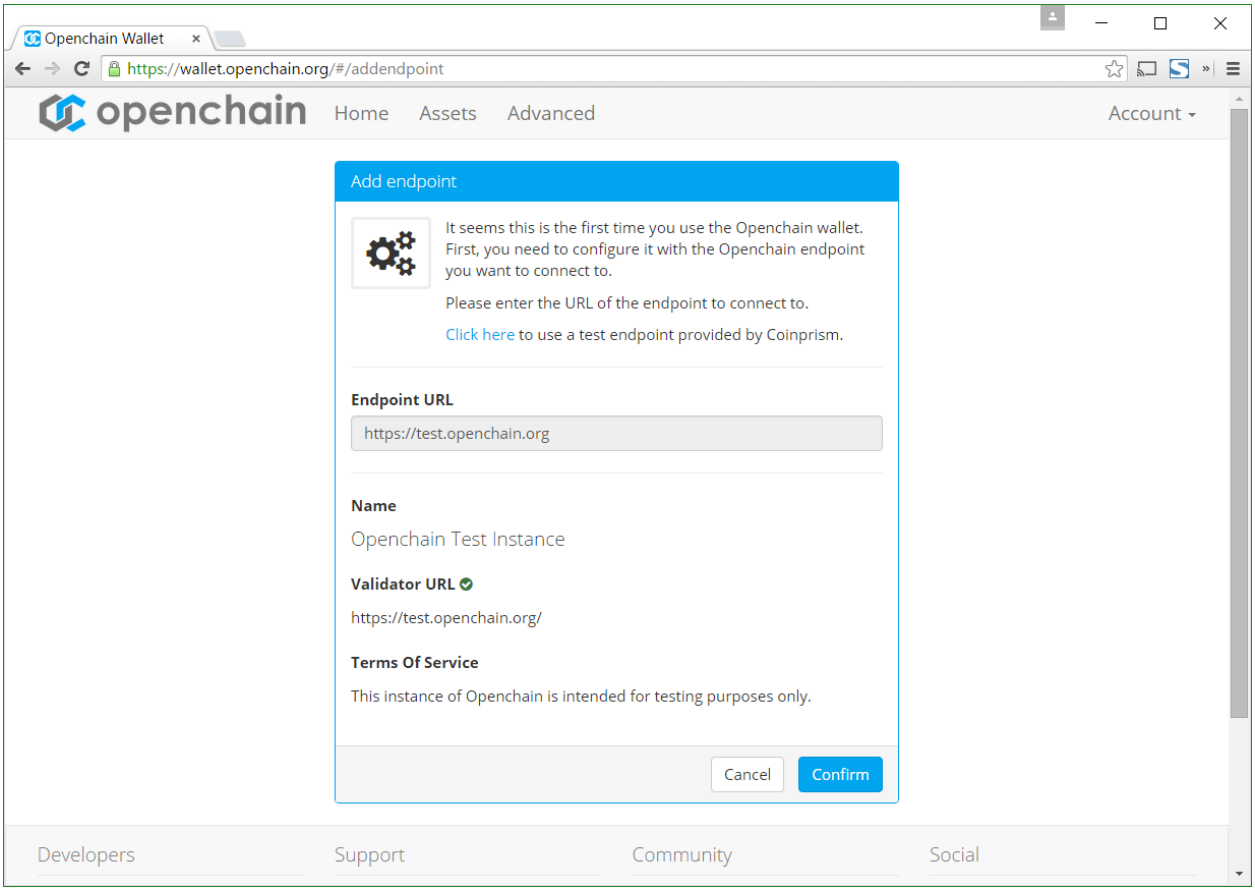
The Openchain Wallet is an open source web based interface, available at wallet.openchain.org.

Connecting to a server

The wallet is a client side application running in the browser, and capable of connecting to any Openchain endpoint. It can connect to multiple endpoints at the same time, and pull information and submit transactions to multiple instances of Openchain, however the first time you use it, you need to connect to at least one endpoint.



The first page invites you to connect to an endpoint. Click the link to use the test endpoint provided by Coinprism, then click "Check endpoint". The wallet will then try to connect to the Openchain instance and retrieve the instance information.

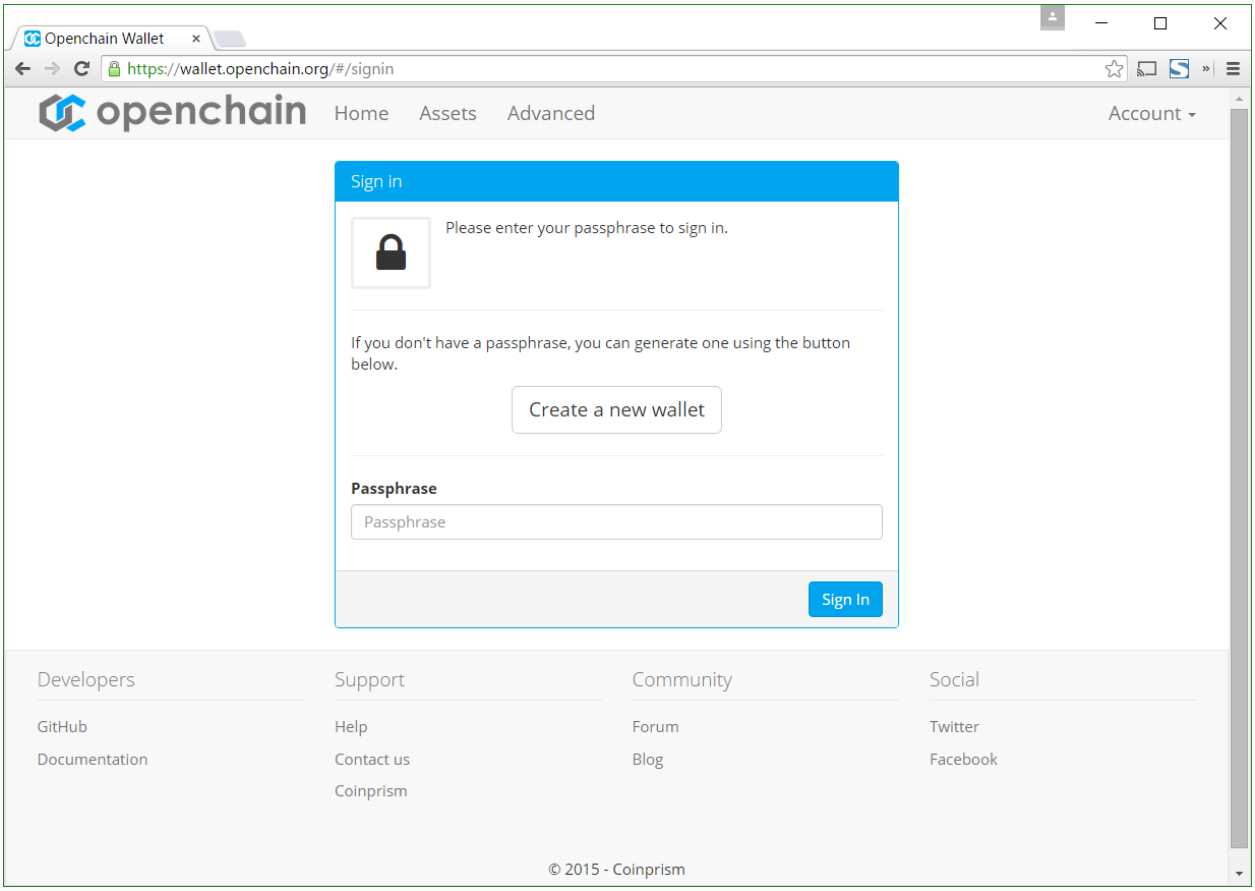


Confirm to connect to this endpoint.

Note: The Openchain wallet will memorize the endpoint you are connecting to, so you will only have to perform this step once.

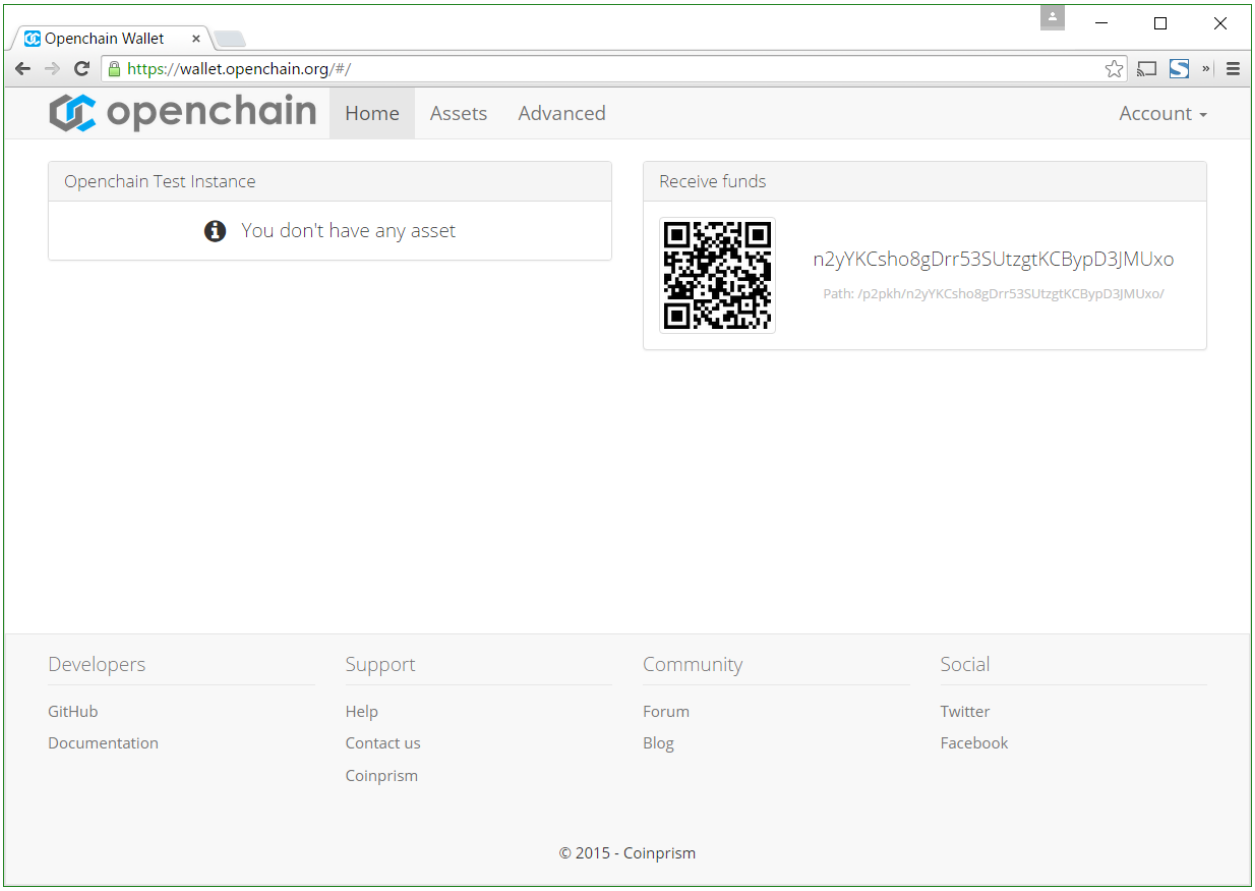
Logging in

The wallet will now ask you to provide a mnemonic seed used to derive your private key and address.



Click "Create a new wallet" if you want to generate a new mnemonic, and reuse one you have already generated. Click "Sign in" to confirm.

After the key has been derived from your seed, you should see your home screen:

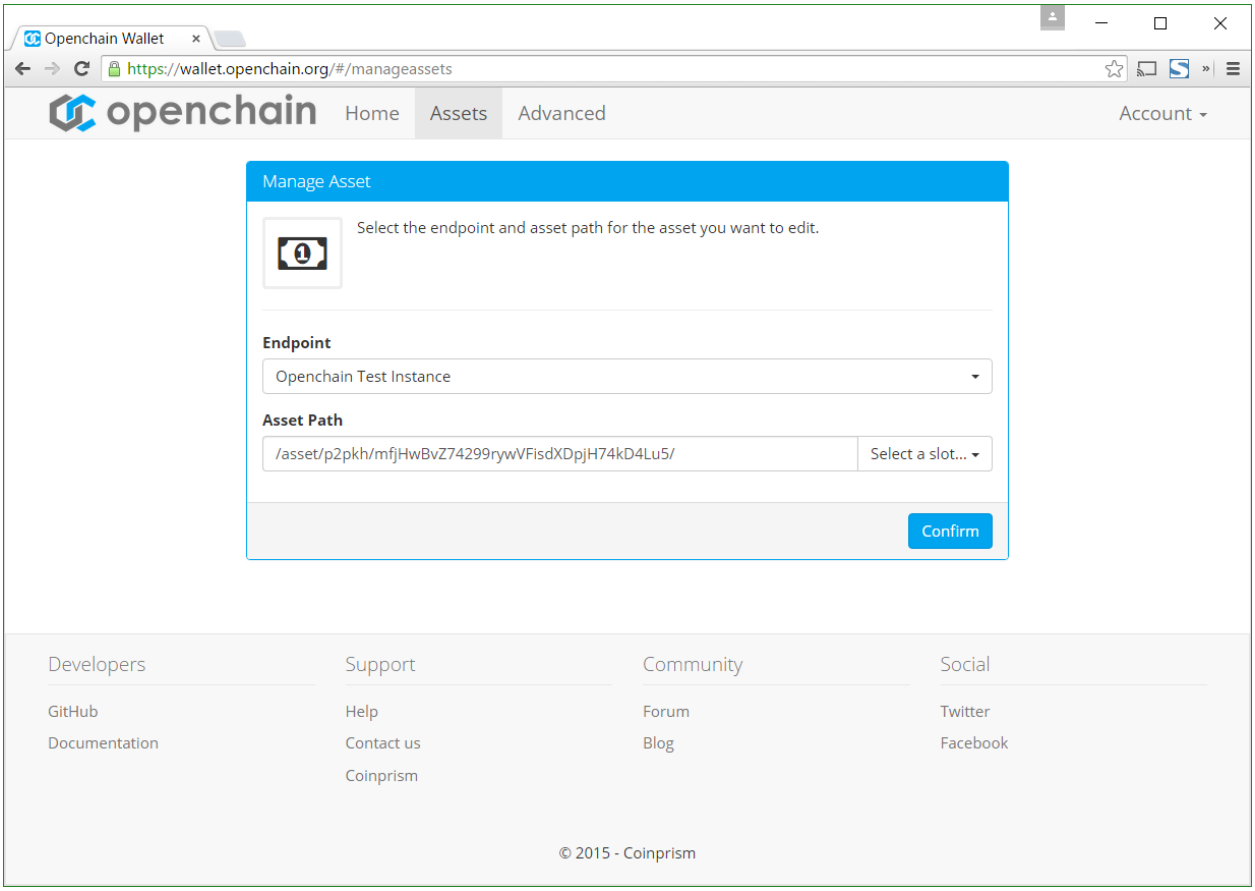


You are now able to receive payments on the Openchain instance by giving your account path to the payer (`/p2pkh/n2yYKCsho8gDrr53SUtzgtKCBypD3JMUxo/` in the example above).

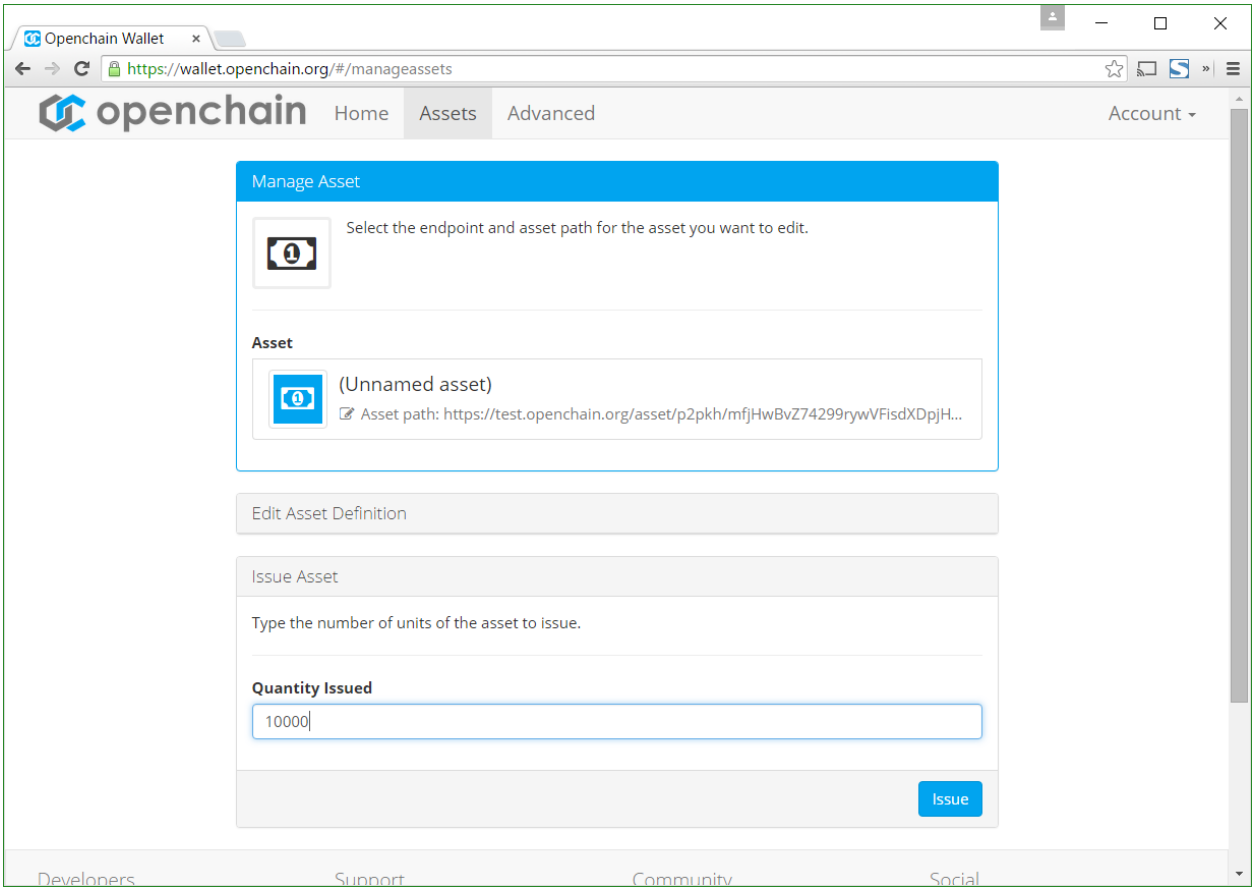
Issue an asset

The test endpoint provided by Coinprism has third party asset issuance enabled, so we can now issue an asset.

To do this, click the “Assets” tab.

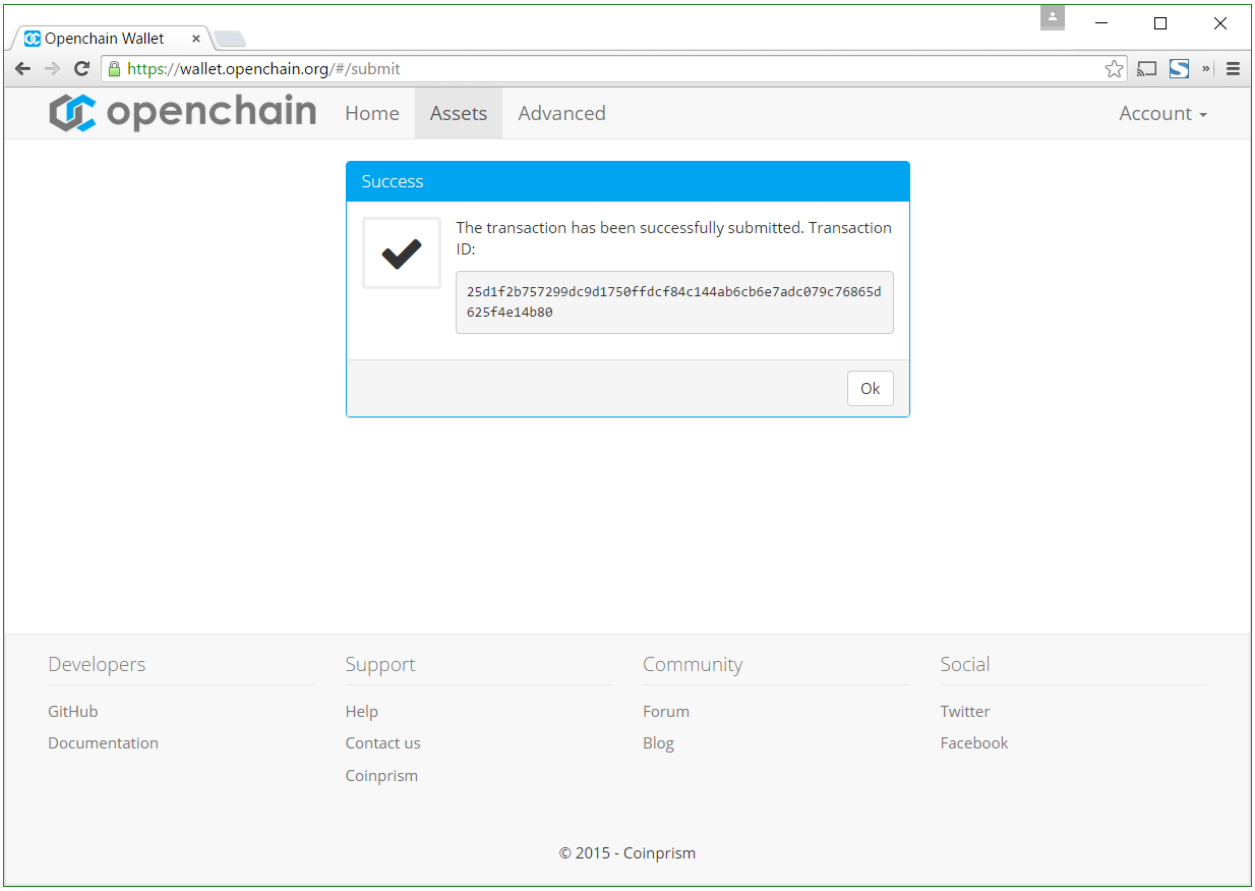


Select the endpoint and the first slot, and click “Confirm”.

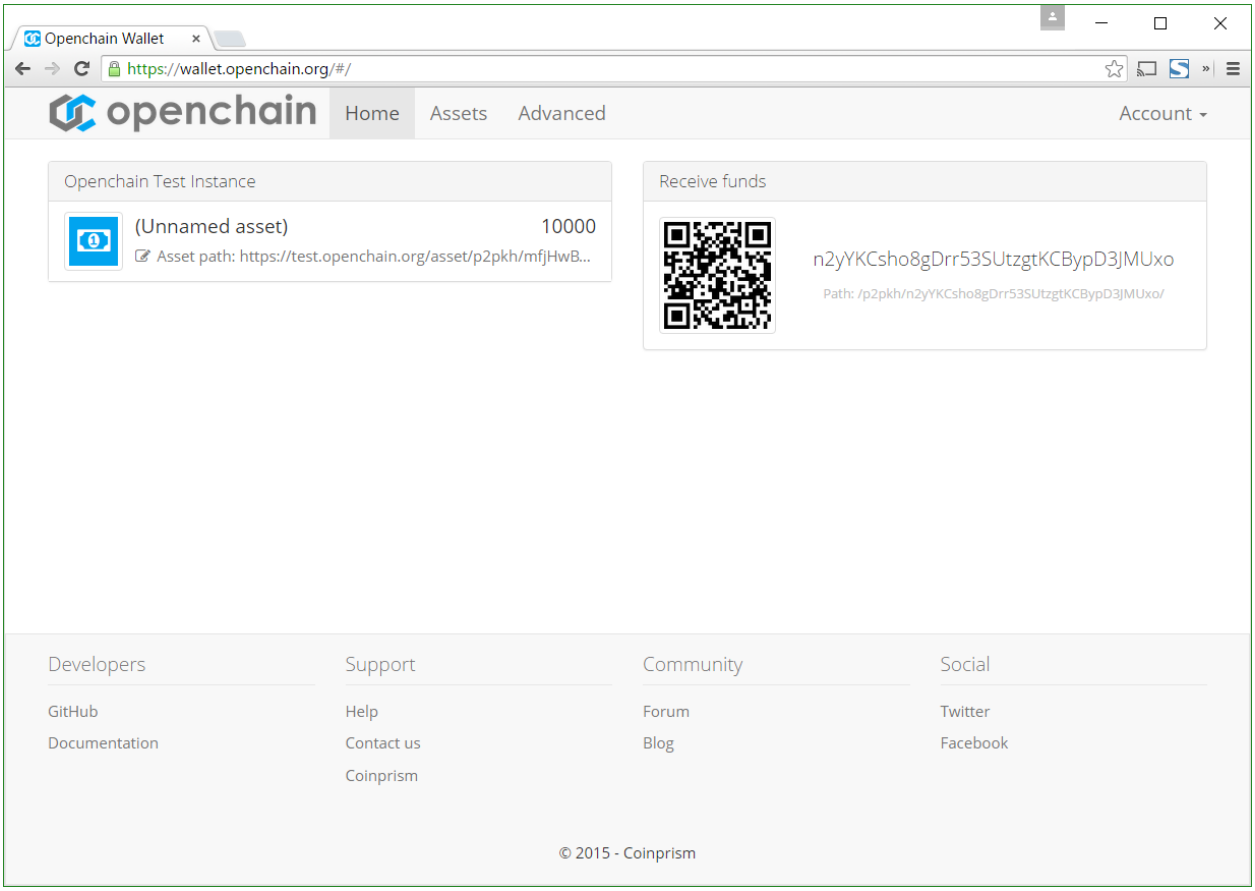


Click “Issue Asset” and type an amount to issue (10000 for example). Press “Issue”.

You should then see a confirmation of the transaction.



Your account should have been updated with the newly issued asset.

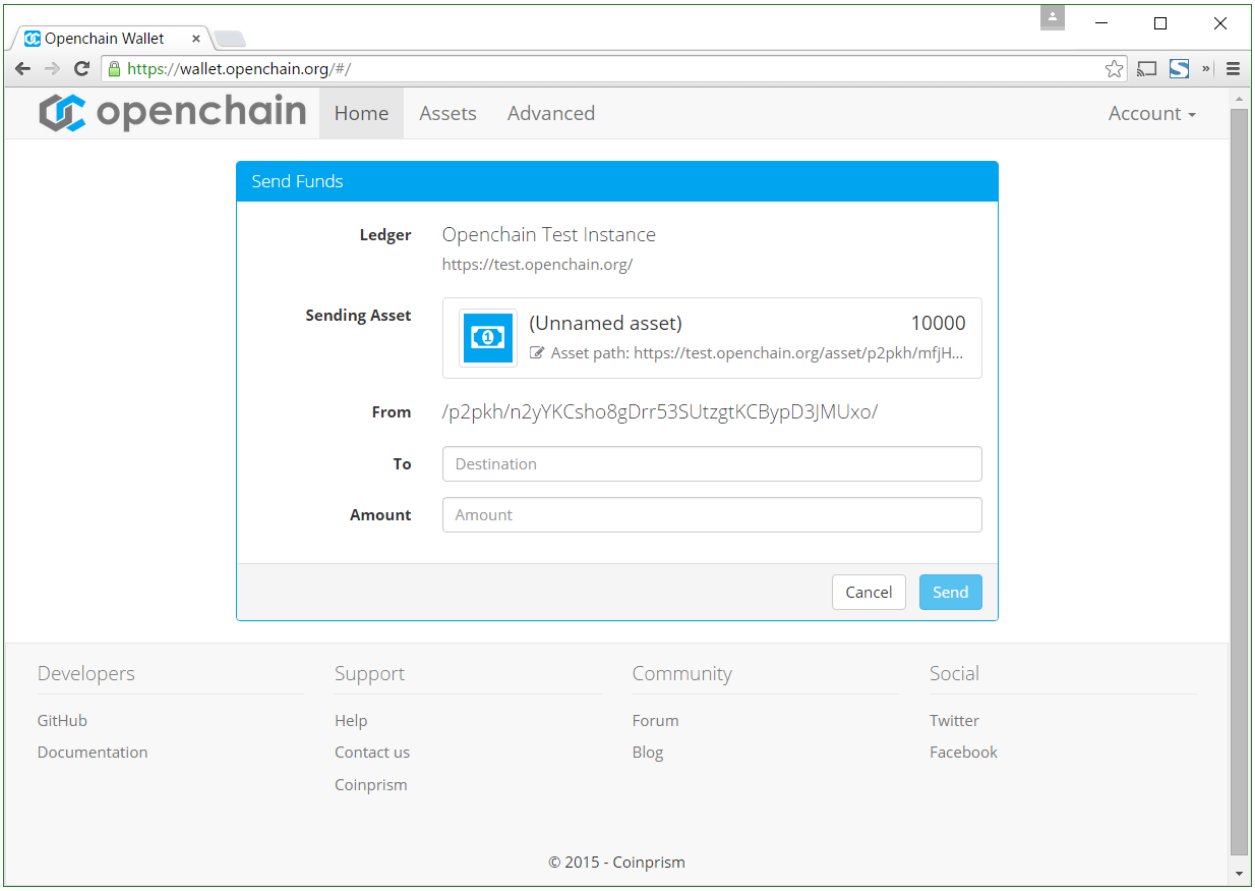


Tip: You can use the "Edit Asset Definition" box in the asset issuance page to define *metadata* about your asset, such as a name and icon.

Send a payment

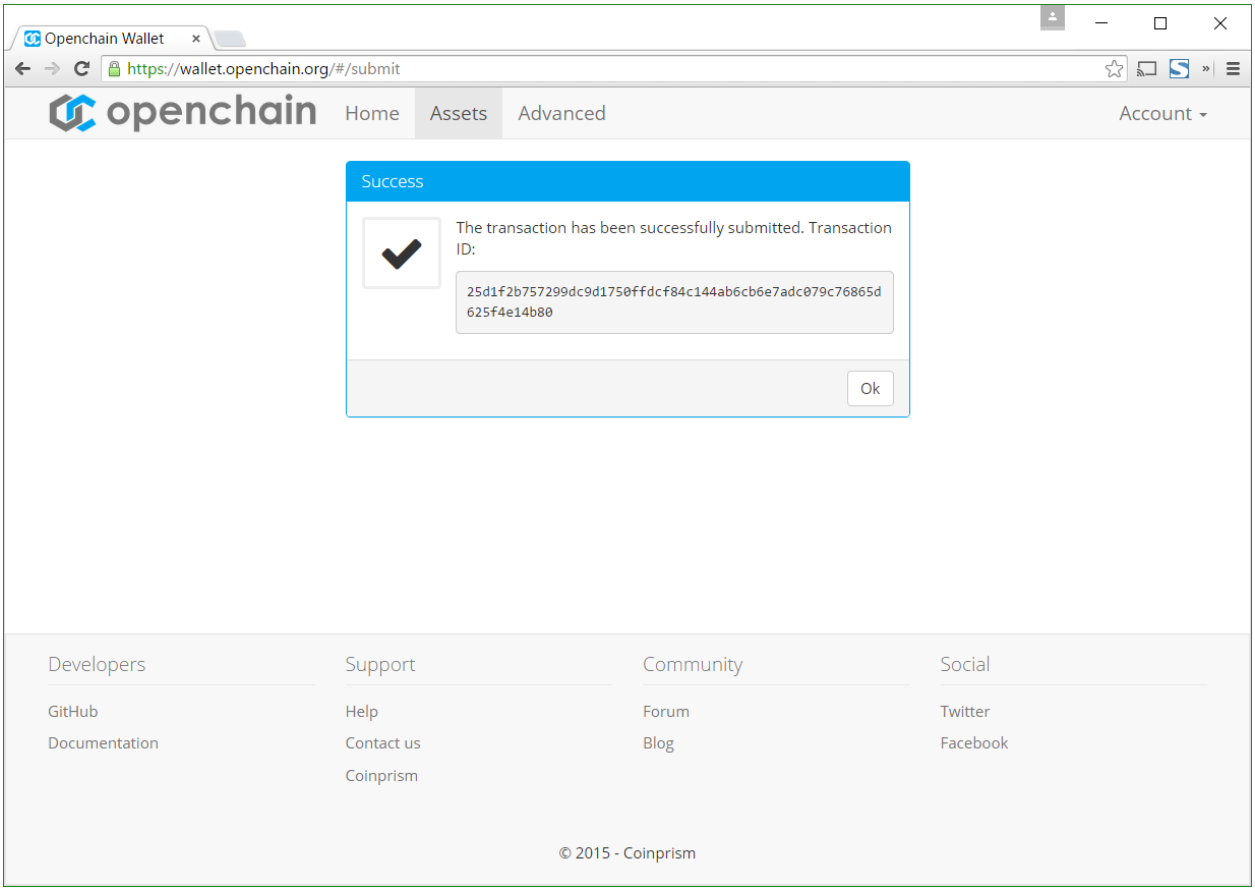
Now that we have funds, we can send them.

Click the newly issued asset to be taken to the "Send" page.



Type a valid destination, such as `/p2pkh/mfiCwNxuFYMtb5ytCacgzDAineD2GNCnYo/`, and a valid amount.

Press "Send" to confirm. If the transaction went through successfully, you should see the transaction confirmation screen.

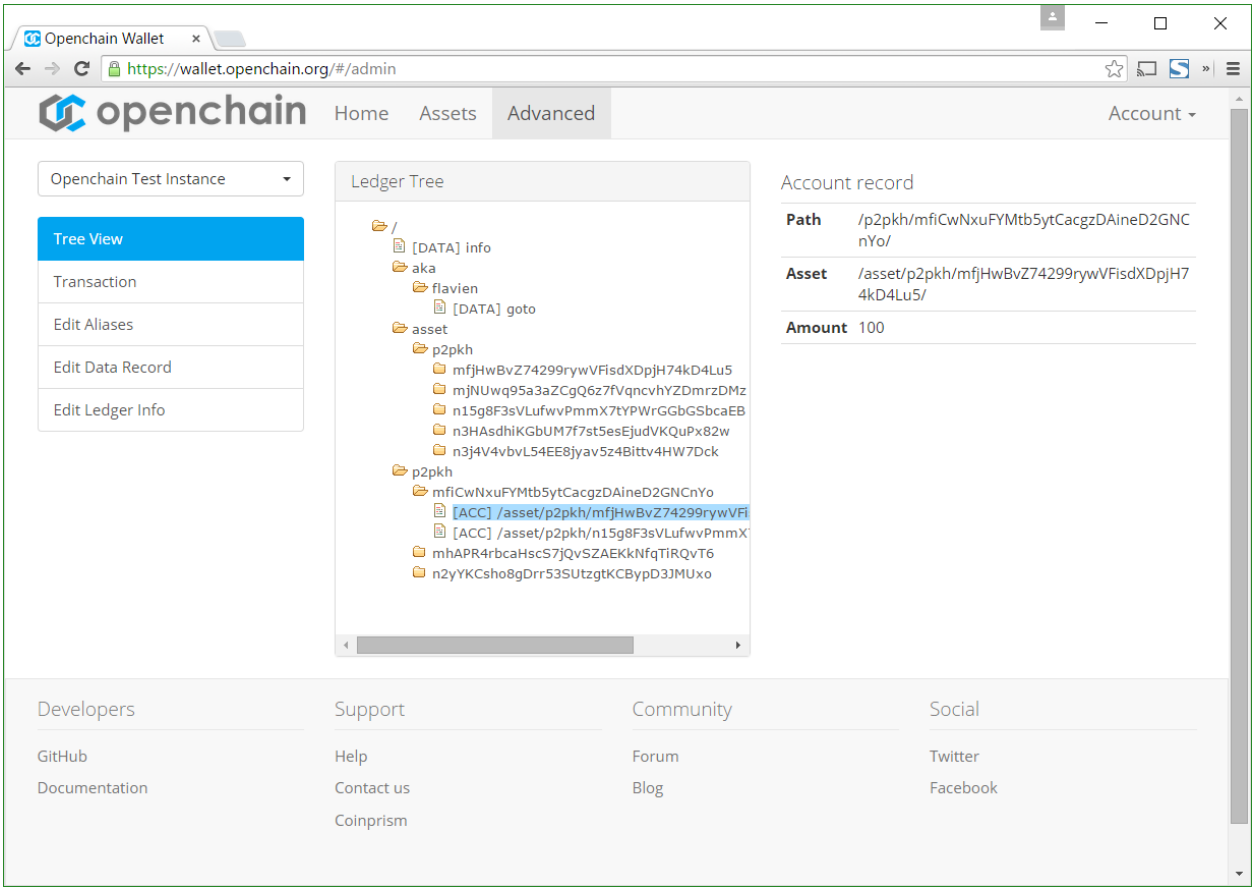


Admin tools

The wallet also has admin tools built-in.

Ledger tree view

The ledger tree view displays a visual representation of the *account hierarchy*. The details of the record selected on the left will be showed on the right hand side.



Alias editor

The alias editor lets you configure *aliases* for specific paths. After an alias has been set, it is possible to send funds to the alias directly using the `@` prefix. The wallet will automatically resolve the alias.

Note: In the default permission layout, aliases can only be modified by an administrator.

Openchain Server Docker deployment

Openchain Server is cross platform and can be deployed as a DNX application on Windows, OS X and Linux. However, to simplify dependency management and homogenize deployment of Openchain, we are shipping it as a Docker image.

This document explains the few steps necessary to have the Openchain server running. Refer to the next section to deploy Openchain directly.

Install Docker

Note: This assumes you are running Linux. Use these instructions if you are running Windows, and these instructions if you are running OS X.

First, install Docker if you don't have it:

```
wget -qO- https://get.docker.com/ | sh
```

Then install Docker Compose:

```
apt-get install python-pip
pip install -U docker-compose
```

Install Openchain Server

Clone the openchain/docker repository from GitHub, and copy the configuration files from the templates provided.

```
git clone https://github.com/openchain/docker.git openchain
cd openchain
cp templates/docker-compose-direct.yml docker-compose.yml
mkdir data
cp templates/config.json data/config.json
```

Now, edit the configuration file (`data/config.json`):

```
nano data/config.json
```

Set the `instance_seed` setting to a random (non-empty) string.

```
 [...]
   // Define transaction validation parameters
   "validator_mode": {
     // Required: A random string used to generate the chain namespace
     "instance_seed": "",
     "validator": {
 [...]
```

Note: By default, the Openchain server will run on port 8080. You can edit `docker-compose.yml` if you want to run on a non-default port.

You can now start the server:

```
docker-compose up -d
```

This will start the Openchain server in the background. To check that the server is running properly, check the docker logs:

```
docker logs openchain-server
```

You should not see any error:

```
info: General[0]
    [2016-07-10 18:20:10Z] Starting Openchain v0.7.0
info: General[0]
```

```
    [2016-07-10 18:20:11Z]
info: General[0]
    [2016-07-10 18:20:13Z] Stream subscriber disabled
info: General[0]
    [2016-07-10 18:20:13Z] Anchoring disabled
Hosting environment: Production
Content root path: /openchain
Now listening on: http://0.0.0.0:8080
Application started. Press Ctrl+C to shut down.
```

Tip: You can also run the Openchain Docker container in the foreground by running `docker-compose up` and omitting the `-d` switch.

Now that you have a server running, you can connect to the server with a *client*.

Configuring admin keys

Use the *client* to generate a seed, and derive it into an address. Once you have an address, you can use it as an admin address on your server instance. To do so, update `data/config.json` and add it to the `admin_addresses` list:

```
// ...
"admin_addresses": [
  "<your_address_here>"
],
// ...
```

Tip: Follow *these steps* to configure the `info` record on your new instance. The `info` record is used by clients connecting to the instance to receive additional information about the instance they are connecting to.

Controlling the server

To restart the server, use:

```
docker-compose restart
```

To stop it, use:

```
docker-compose stop
```

CHAPTER 3

General

Running Openchain

Deploying Openchain server can be done *through Docker*.

This document explains how to deploy Openchain directly on a machine without using docker.

Prerequisites

Install the .NET Command Line Interface . This is cross-platform and runs on Windows, Linux and OS X.

Download the project files

Download the `project.json`, `Program.cs` and `config.json` files from GitHub, then restore the NuGet dependencies. On Linux:

```
$ wget https://raw.githubusercontent.com/openchain/openchain/v0.6.2/src/Openchain/
↪project.json
$ wget https://raw.githubusercontent.com/openchain/openchain/v0.6.2/src/Openchain/
↪Program.cs
$ wget https://raw.githubusercontent.com/openchain/openchain/v0.6.2/src/Openchain/
↪data/config.json -P data
$ dotnet restore
```

Note: On Windows, simply download the files manually using your browser, then run `dotnet restore`.

Run Openchain Server

Run openchain server using the following command:

```
$ dotnet run
```

Configuration

The dependencies section of the `project.json` file references the external providers pulled from NuGet:

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Openchain.Server": "0.6.2",

  "Openchain.Anchoring.Blockchain": "0.6.2",
  "Openchain.Sqlite": "0.6.2",
  "Openchain.SqlServer": "0.6.2",
  "Openchain.Validation.PermissionBased": "0.6.2"
},
```

By defaut, this imports the Sqlite storage engine (`Openchain.Sqlite`), the SQL Server storage engine (`Openchain.SqlServer`), the permission-based validation module (`Openchain.Validation.PermissionBased`), and the Blockchain anchoring module (`Openchain.Anchoring.Blockchain`). Update this list with the modules (and versions) you want to import.

You can then edit the `data/config.json` file to reference the *providers you want to use*.

Tip: For example, if you want to use the `SQLite` provider as a storage engine, you will need to make sure the `Openchain.Sqlite` module is listed in the dependencies.

Make sure you run `dotnet restore` again after modifying project.json.

Note: The Openchain.Server dependency is the only one that is always required. The version of the `Openchain.Server` package is the version of Openchain you will be running.

Updating the target platform

The frameworks section of the project.json file lists the available target frameworks:

```
"frameworks": {
  "netcoreapp1.0": {},
  "net451": {}
}
```

By default .NET Core (cross-platform) and the .NET Framework (Windows only) are both targeted. Some providers run only on a subset of frameworks. In that case, remove the unsupported frameworks from the list to ensure the project runs.

The transaction stream

Openchain server exposes a websocket endpoint (`/stream`) called the transaction stream. The transaction stream provides a live stream of transactions as they get committed into the ledger.

Note: See the *documentation* about the `/stream` endpoint for more details.

Validator nodes

The Openchain Server node can function in two different modes: **validator mode** and **observer mode**.

In validator mode, the node accepts transactions and validates them. Rules that make a transaction valid or invalid are customizable. They can be defined by the administrator of the validator node, and are a combination of *implicit rules*, and explicit permissions.

When a transaction is deemed valid, it gets committed into the ledger.

Observer nodes

Observer nodes are nodes connecting to an upstream node, and downloading all transactions in real time using the transaction stream. The validator node is always the most upstream node. When it verifies a transaction, the transaction trickles down to its observers. All the observers should have an exact copy of the state held by the verifying node.

It is not possible to submit a transaction for validation to an observer node, as it only has a read-only view of the ledger.

Observer nodes have the ability to verify the integrity of their copy of the ledger through *anchors*.

Configuration

To configure a node to be in observer mode, the `observer_mode` section needs to exist in the *configuration file*, and the `upstream_url` must be set to the root URL of the upstream node.

Anchoring and ledger integrity

Openchain is capable of immutability by committing a hash of the entire ledger (the **cumulative hash**) onto a non-reversible Blockchain such as Bitcoin.

Note: In the current version, the only anchoring mode available is the `blockchain` mode, based on the Bitcoin blockchain. Different anchoring modes will be available in the future, such as anchoring in a central repository.

With the Bitcoin anchoring mode, one transaction is committed in every Bitcoin block, and contains the cumulative hash at the current time.

By doing this, even if Openchain is processing thousands of transactions per second, only one transaction gets sent to the Bitcoin blockchain every 10 minutes. There are multiple benefits to this approach:

- The irreversibility of the Openchain ledger is ensured by the Bitcoin miners, therefore Openchain enjoys the same level of irreversibility as Bitcoin itself.

- At the maximum resolution (one anchor per block), no more than 4,320 transactions per month (in average) will be committed into the blockchain, which will cost about $10 per month (as of October 2015), regardless of the number of transactions processed.
- The resolution can be tuned to further reduce that cost.
- Openchain can process thousands of transactions per second while remaining very cost-efficient.

"Observer nodes" replicating all the verified transactions locally have the ability to compute their own version of the cumulative hash and compare it to the anchor in the Bitcoin blockchain.

Calculating the cumulative hash

The cumulative hash is updated every time a new transaction is added to the ledger.

The cumulative hash at a given height is calculated using the previous cumulative hash and the hash of the new transaction being added to the ledger:

```
cumulative_hash = SHA256( SHA256( previous_cumulative_hash + new_transaction_hash ) )
```

- `previous_cumulative_hash` (32 bytes) is the cumulative hash at the previous height. At height 0 (when the ledger has no transaction), a 32 bytes buffer filled with zeroes is used.
- `new_transaction_hash` (32 bytes) is the double SHA-256 hash of the *raw transaction* being added to the ledger.

Both values are concatenated to form a 64 bytes array, then hashed using double SHA-256.

Blockchain anchor format

The Blockchain anchor is stored in the blockchain using an OP_RETURN operator, followed by a pushdata containing the anchor.

```
OP_RETURN <anchor (42 bytes)>
```

The anchor is constructed in the following way:

```
0x4f 0x43 <transaction count (8 bytes)> <cumulative hash (32 bytes)>
```

- The first two bytes indicates that the output represents an Openchain anchor.
- The transaction count is the number of transactions being represented by the cumulative hash (the height). It's an unsigned 64 bits integer, encoded in big endian.
- The cumulative hash is full cumulative hash (256 bits) as calculated in the previous section.

Openchain Server Configuration

The configuration of Openchain server is handled through a JSON file named *config.json*. The file is stored under the `data` folder.

It is possible to override a configuration value through environment variables. The name of the variable should be the concatenation of all the components of the path, separated by the character `:`. For example: `validator_mode:validator:allow_third_party_assets`.

config.json

Here is the default file:

```
{
  "enable_transaction_stream": true,

  "storage": {
    "provider": "SQLite",
    "path": "ledger.db"
  },

  // Define transaction validation parameters
  "validator_mode": {
    // Required: A random string used to generate the chain namespace
    "instance_seed": "",
    "validator": {
      "provider": "PermissionBased",
      // Enable /p2pkh/<address>/ accounts
      "allow_p2pkh_accounts": true,
      // Enable /asset/p2pkh/<address>/ accounts
      "allow_third_party_assets": true,
      // Base-58 addresses that must have admin rights
      "admin_addresses": [
      ],
      "version_byte": 76
    }
  },

  // Uncomment this and comment the "validator_mode" section to enable observer mode
  // "observer_mode": {
  //   "upstream_url": ""
  // },

  "anchoring": {
    "provider": "Blockchain",
    // The key used to publish anchors in the Blockchain
    "key": "",
    "bitcoin_api_url": "https://testnet.api.coinprism.com/v1/",
    "network_byte": 111,
    "fees": 5000,
    "storage": {
      "provider": "SQLite",
      "path": "anchors.db"
    }
  }
}
```

Root section

- `enable_transaction_stream`: Boolean indicating whether the transaction stream websocket should be enabled on this instance.

`storage` section

`provider` defines which storage engine to use. The two built-in values are `SQLite` and `MSSQL`.

`SQLite` storage engine

If the storage provider is set to `SQLite`, the chain is stored locally using SQLite. In that case, the following setting is used:

- `path`: The path of the Sqlite database, relative to the `wwwroot/App_Data` folder. Absolute paths are also allowed, however, make sure the user under which the DNX process is running has write access to the file.

`MSSQL` storage engine

If the storage provider is set to `MSSQL`, the chain is stored using Microsoft SQL Server. In that case, the following setting is used:

- `connection_string`: The connection string to the SQL Server database.

Note: Third party storage engines can be build and used by Openchain. The `provider` setting is used to identify at runtime which storage engine should be instantiated.

`validator_mode` and `observer_mode` sections

These two sections are mutually exclusive. Depending whether the instance is setup in validator mode or observer mode, either the `validator_mode` section or `observer_mode` section should be present.

In the case of validator mode:

- `validator_mode:instance_seed`: A random string that should be unique to that instance. It is hashed to obtain a `namespace` specific to that instance.
- `validator_mode:validator:provider`: The type of validation performed by the Openchain instance when transactions are submitted. The only supported values currently are `PermissionBased`, `PermitAll` and `DenyAll`.
  - `PermitAll` indicates that all transactions are valid, regardless of who signed them. Use this mostly for testing.
  - `DenyAll` indicates that all transactions are invalid, regardless of who signed them. Use this to set the chain in read-only mode.
  - See *this section* for more details about the implicit rules of the `PermissionBased` mode. The relevant configuration settings with the `PermissionBased` mode are the following:
    - `validator_mode:validator:allow_p2pkh_accounts`: Boolean indicating whether *P2PKH accounts* (`/p2pkh/<address>/`) are enabled.
    - `validator_mode:validator:allow_third_party_assets`: Boolean indicating whether *thrid party issuance accounts* (`/asset/p2pkh/<address>/`) are enabled.
    - `validator_mode:validator:admin_addresses`: List of strings representing all addresses with admin rights.
    - `validator_mode:validator:version_byte`: The version byte to use when representing a public key using its Bitcoin address representation.

In the case of observer mode:

- `observer_mode:upstream_url`: The endpoint URL of the upstream instance to connect to. Transactions will be replicated using this endpoint.

`anchoring` section

This section contains configuration settings relative to publishing an anchor to preserve data integrity.

- `provider`: Value defining which anchoring mode to use. Currently, the only supported value is `Blockchain`, and publishes a cumulative hash of the database onto a Bitcoin-compatible blockchain.
- `key`: The private key to use (in WIF format) as the signing address for the proof of publication transactions.
- `bitcoin_api_url`: The Coinprism API endpoint to use to list unspent outputs and broadcast the signed transaction. Valid values include:
  - https://api.coinprism.com/v1/ (Bitcoin mainnet)
  - https://testnet.api.coinprism.com/v1/ (Bitcoin testnet)
- `network_byte`: The network byte corresponding to the network on which the anchor transaction is published.
- `storage:provider`: Value defining how to cache anchors locally. Currently, the only supported value is `SQLite` and caches data locally in a SQLite database.
- `storage:path`: The path of the local anchor cache database, relative to the `wwwroot/App_Data` folder.

Openchain modules

Openchain uses an extensible architecture where modules can be swapped in and out depending on the functionality needed. Modules are selected by:

- Referencing the appropriate package in the `project.json` file. Packages are then pull automatically from NuGet.
- Referencing the module in `config.json`.

This document lists the available modules, and relevant packages.

Storage engines

Storage engines are core components responsible for storing the transaction chain and records.

Provider	Module	Description	Maintainer
`SQLite`	Openchain.Sqlite	Stores the chain in a local Sqlite database.	Coinprism
`MsSQL`	Openchain.SqlServer	Stores the chain in a SQL Server database.	Coinprism
`MongoDB`	Openchain.MongoDb	Stores the chain in a MongoDB database.	@fluce

Validation engines

Anchoring media

Setting the instance info on a new instance

The *ledger info record* exposes meta-information about the ledger itself. It is used by clients that connect to the instance to retrieve informations such as the name of the instance, and the associated terms of service.

After you have deployed a new instance, it is a good idea to create the info record. This can be done from the web interface.

1. First, follow *these steps* to connect to the instance and log in. Make sure you log in with a seed that has admin access on this instance as the `info` record can only be modified by an administrator.
2. Go to the **Advanced** tab and click **Edit Ledger Info** on the left. The screen will show you a form that will let you edit the ledger name and other fields stored in the `info` record.

Important: Make sure that the **Validator Root URL** is set to the same value as the `root_url` setting in the configuration file.

Upgrading Openchain server

To upgrade an Openchain deployment done *through Docker*, run the following commands:

```
git reset --hard
git pull
cp templates/docker-compose-direct.yml docker-compose.yml
docker-compose build
docker-compose restart
```

Note: If the new version you are upgrading to includes a configuration file schema change, don't forget to update the configuration file before restarting Openchain.

Deploying Openchain in a production environment

In production, it is recommended to proxy the Openchain server behind a reverse proxy server such as Nginx. This architecture enables a number of possibilities:

- Expose Openchain through SSL/TLS
- Host multiple Openchain server instances on the same port
- Change the URL path under which the Openchain server is being exposed
- Route requests to different Openchain instances depending on the host name used

This document explain the few steps necessary to expose Openchain through Nginx.

Install Docker

Refer to the *base Docker deployment documentation* to find out how to install Docker and Docker Compose.

Pull the Docker images through Docker Compose

Clone the openchain/docker repository from GitHub, and copy the configuration files from the templates provided.

```
git clone https://github.com/openchain/docker.git openchain
cd openchain
cp templates/docker-compose-proxy.yml docker-compose.yml
cp templates/nginx.conf nginx/nginx.conf
```

```
mkdir data
cp templates/config.json data/config.json
```

Edit the configuration file (`data/config.json`) as described in the *base Docker deployment documentation*.

You can now start the server:

```
docker-compose up -d
```

Note: By default, Nginx will run on port 80.

Troubleshooting

Error "The namespace used in the transaction is invalid"

You might receive this error message when submitting a transaction. You will get this error if the `root_url` set in the configuration doesn't match the namespace set by the client in the transaction. Clients will always use the URL they are connected to as the namespace.

This ensures that a transaction is only valid for one specific instance of Openchain, and that it is not possible to reuse a signed transaction on multiple ledgers.

Solution

To solve this, make sure the URL *set in your configuration file* (`validator_mode:root_url`) matches the URL that clients use to connect to your Openchain instance. All the components of the URL must match:

- The scheme, e.g.: `http://endpoint.com/` vs `https://endpoint.com/`
- The hostname, e.g.: `http://127.0.0.1/` vs `http://localhost/`
- The port, e.g.: `http://endpoint.com:80/` vs `http://endpoint.com/`
- The path, e.g.: `http://endpoint.com/path/` vs `http://endpoint.com/`

Important: Make sure you don't forget the trailing slash, as clients will always include it in the namespace. E.g.: `https://endpoint.com/` instead of `https://endpoint.com`.

CHAPTER 4

Public API

Openchain data structures

Openchain relies on several data structures for communication between clients and servers. These data structures are a key part of the Openchain API.

These data structures are serialized and deserialized using Protocol Buffers.

Schema

The full schema is the following:

```
syntax = "proto3";

package Openchain;

message RecordValue {
    bytes data = 1;
}

message Record {
    bytes key = 1;
    RecordValue value = 2;
    bytes version = 3;
}

message Mutation {
    bytes namespace = 1;
    repeated Record records = 2;
    bytes metadata = 3;
}

message Transaction {
    bytes mutation = 1;
```

```
    int64 timestamp = 2;
    bytes transaction_metadata = 3;
}
```

Note: The schema uses the version 3 of Protocol Buffers.

Record

A record object represents the intent to modify the value of a record in the data store. The key, value and version of a record can be any arbitrary byte string.

```
message Record {
    bytes key = 1;
    RecordValue value = 2;
    bytes version = 3;
}
```

- `key`: A value that uniquely identifies the record to modify.
- `value`: The new value that the record should have after update. If it is unspecified, the record version is checked, but no update is made to the value.
- `version`: The last version of the record being updated. Every modification of the record will cause the version to change. If the version specified doesn't match the actual version in the data store, then the update fails.

A record that has never been set has a `value` and `version` both equal to an empty byte string.

Check-only records

If a record object has a null `value` field, the record object is called a **check-only record**, and does not cause a mutation to the record. It however expresses the requirement that the record (as represented by the `key` field) must have the version specified in the `version` field of the record object. If the versions don't match, the whole mutation fails to apply.

This provides a way to ensure that a given record has not been modified between the moment the transaction was created and the moment it gets validated, even if the record doesn't have to be modified.

Mutation

A mutation is a set of records atomically modifying the state of the data. They are typically generated by a client, signed, then sent to the validator along with the signatures.

```
message Mutation {
    bytes namespace = 1;
    repeated Record records = 2;
    bytes metadata = 3;
}
```

- `namespace`: The namespace under which the records live. Generally, each instance of Openchain has its own namespace.

- `records`: A set of records to be modified atomically by this mutation. Each record is identified by its key. The version of each record in the mutation has to match the versions at the current time. If any version mismatches, then the entire mutation fails to apply. Records with an unspecified value don't cause updates, but their versions still have to match for the mutation to succeed.
- `metadata`: Arbitrary metadata to be stored in the mutation.

The version of all updated records after a mutation becomes the hash of that mutation.

Transaction

A transaction is a wapper around a mutation.

```
message Transaction {
    bytes mutation = 1;
    int64 timestamp = 2;
    bytes transaction_metadata = 3;
}
```

- `mutation`: The mutation applied by the transaction. It is represented as a byte string but deserialized according to the *Mutation schema*.
- `timestamp`: A timestamp for the transaction.
- `transaction_metadata`: Arbitrary metadata to be stored in the mutation. This will typically contain a digital signature of the mutation by the required parties.

Ledger structure

At the core, an Openchain ledger is a key-value store, represented by *records*. At the data store level, record keys can be any arbitrary byte string, however Openchain Ledger expects a well defined structure for the record keys.

Record keys

Record keys are UTF8-encoded strings. They are structured in three parts, separated by colons (`:`).

1. **The record path**: A path in the *account hierarchy* indicating where the record is situated.
2. **The record type**: A value indicating the *type* of the record.
3. **The record name**: The name of the record.

The combination of these three values uniquely identify a record.

Example 1

```
/p2pkh/mfiCwNxuFYMtb5ytCacgzDAineD2GNCnYo/:ACC:/asset/p2pkh/
↪n15g8F3sVLufwvPmmX7tYPWrGGbGSbcaEB/
```

The path is `/p2pkh/mfiCwNxuFYMtb5ytCacgzDAineD2GNCnYo/`, the record type is `ACC` and the record name is `/asset/p2pkh/n15g8F3sVLufwvPmmX7tYPWrGGbGSbcaEB/`.

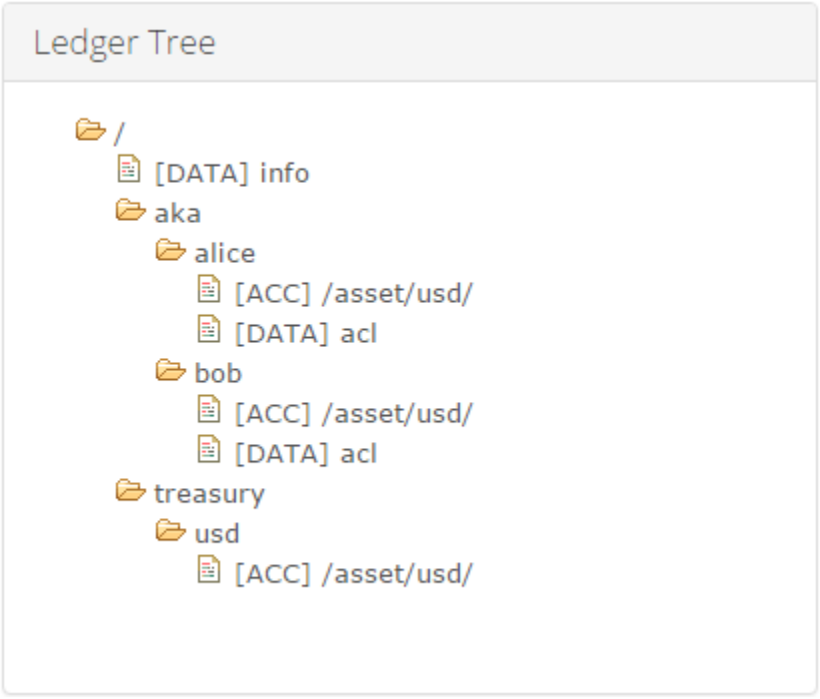
Example 2

```
/:DATA:info
```

The path is `/` (root path), the record type is `DATA` and the record name is `info`.

Account hierarchy

Openchain uses a hierarchy of accounts, similar to a file system. This adds a lot of interesting management options that systems like Bitcoin don't have.



Accounts are identified by a path.

Account paths

The syntax for an account path follows a number of rules:

- Account paths start with the character `/`.
- Account paths end with the character `/`.
- Sections of an account path are separated by the character `/`.
- Sections of an account path may only contain alphanumeric characters and characters from the following set: `$-_.+!*'(),`.

Record types

There are two valid record types as of this version of Openchain.

ACC record

The ACC record is used for representing a balance for a given asset type. The name of the record must be a path that represents the asset type. The value must be a 64-bits signed integer encoded in big endian. The value represents the current balance for the given account and the given asset type.

DATA record

The DATA record is used to store arbitrary text data. The record name can be any valid UTF-8 string. It can be used to store things such as *asset metadata*, *symbolic links* within the accounting system, *permissions*, or any other important piece of arbitrary data that needs to be cryptographically secure.

Method calls

The Openchain server exposes an HTTP API that can be used to interact with the data. The URL of an operation is constructed from the base URL of the endpoint, and concatenating it with the relative path of the operation being called.

For example, if the base URL is `https://www.openchain.org/endpoint/`, for calling the `/record` operation (query a record), the full URL should be `https://www.openchain.org/endpoint/record`.

Submit a transaction (`/submit`)

Submits a transaction for validation.

Method: POST

Inputs

The input is a JSON document passed as part of the body of the request.

The format of the JSON document is the following:

```
{
    "mutation": "<string>",
    "signatures": [
        {
            "pub_key": "<string>",
            "signature": "<string>"
        }
    ]
}
```

Description of the payload:

- `mutation`: The hex-encoded mutation. The mutation is serialized using the *Mutation Protocol Buffers schema*.
- `signatures`: An array of documents with two properties, `pub_key` and `signature`.
  - `pub_key`: The hex-encoded public key used to sign.
  - `signature`: The hex-encoded signature of the hash of the mutation.

Signing Process

For producing the signatures:

1. Serialize the mutation using the *Mutation Protocol Buffers schema*.
2. Hash the mutation byte string using double SHA256.
3. Sign it with the relevant private key using Secp256k1. The matching public key must be submitted along with the signature.

Important: You must submit the exact byte string as obtained after step 1. If it modified, the hash won't match and the signature will then be invalid.

Outputs

The output is a JSON document passed as part of the body of the response.

```
{
    "transaction_hash": "<string>"
}
```

The `transaction_hash` field contains the hex-encoded hash of the full transaction.

Query a record (`/record`)

Query the value and version of a record given its key.

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`key`	The hex-encoded key of the record being queried.

Output

The output is a JSON document passed as part of the body of the response.

The format of the JSON document is the following:

```
{
    "key": "<string>",
    "value": "<string>",
    "version": "<string>"
}
```

The fields are the following:

- `key`: The hex-encoded key of the record.
- `value`: The hex-encoded value of the record.
- `version`: The hex-encoded version of the record.

Transaction stream (`/stream`)

Method: GET

This endpoint is a WebSocket endpoint. It can be used to receive all the newly confirmed transaction in real-time.

Inputs

Inputs are passed through the query string as URL encoded parameters.

`from`	**(optional)** The hex-encoded hash of the last transaction to resume from. If omitted, it will start from the first transaction.

Output

The output is a WebSocket binary stream.

Each message in the stream is the *serialized transaction*.

Retrieve the chain info (`/info`)

Get information about the Openchain instance.

Method: GET

Inputs

This method has no input parameters.

Output

The output is a JSON array passed as part of the body of the response.

The format of the JSON array is the following:

```
{
    "namespace": "<string>"
}
```

`namespace` is the hex representation of the namespace expected in transactions submitted to the Openchain instance.

Query an account (`/query/account`)

Query all the ACC records at a given path (non-recursively).

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`account`	The path to query for.

Output

The output is a JSON array passed as part of the body of the response.

The format of the JSON array is the following:

```
[
    {
        "account": "<string>",
        "asset": "<string>",
        "balance": "<string>",
        "version": "<string>"
    }
]
```

The fields of each item of the array are the following:

- `account`: The path of the record.
- `value`: The asset ID of the record (the record name).
- `balance`: The balance for that asset ID at that path.
- `version`: The hex-encoded version of the record.

Query a transaction (`/query/transaction`)

Retrieve a transaction given the hash of the mutation.

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`mutation_hash`	The hex-encoded hash of the mutation represented by the transaction.
`format`	The output format (`raw` or `json`).

Output

The output is a JSON document passed as part of the body of the response.

The format of the JSON document depends on the `format` argument:

1. `raw` output format (default):

```
{
    "raw": "<string>"
}
```

The `raw` property contains the serialized transaction.

2. `json` output format

```
{
    "transaction_hash": "<string>",
    "mutation_hash": "<string>",
    "mutation": {
```

```
        "namespace": "<string>",
        "records": [
            {
                "key": "<string>",
                "value": "<string>",
                "version": "<string>"
            }
        ]
    },
    "timestamp": "<string>",
    "transaction_metadata": "<string>"
}
```

Query a specific version of a record (`/query/recordversion`)

Retrieve a specific version of a record.

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`key`	The hex-encoded record key.

Output

The output is a JSON document passed as part of the body of the response.

The format of the JSON document is the following:

```
{
    "key": "<string>",
    "value": "<string>",
    "version": "<string>"
}
```

The fields are the following:

- `key`: The hex-encoded key of the record.
- `value`: The hex-encoded value of the record.
- `version`: The hex-encoded version of the record.

If the record version doesn't exist, HTTP code 404 will be returned by the server.

Query all mutations that have affected a record (`/query/recordmutations`)

Retrieve all the mutations that have affected a given record.

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`key`	The key of the record of which mutations are being retrieved.

Output

The output is a JSON document passed as part of the body of the response.

The format of the JSON document is the following:

```
[
    {
        "mutation_hash": "<string>"
    }
]
```

The output is a list representing all the mutation hashes of the mutations that have affected the key represented by the `key` argument.

Query records in an account and its subaccounts (`/query/subaccounts`)

Retrieve all the record under a given path (includes sub-paths).

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`account`	The path being queried.

Output

The output is a JSON document passed as part of the body of the response.

The format of the JSON document is the following:

```
[
    {
        "key": "<string>",
        "value": "<string>",
        "version": "<string>"
    }
]
```

The fields are the following:

- `key`: The hex-encoded key of the record.
- `value`: The hex-encoded value of the record.
- `version`: The hex-encoded version of the record.

Query all records with a given type and name (`/query/recordsbyname`)

Retrieve all records with a given type and name

Method: GET

Inputs

Inputs are passed through the query string as URL encoded parameters.

`name`	The name of the records being queried.
`type`	The type of the records being queried.

Output

The output is a JSON document passed as part of the body of the response.

The format of the JSON document is the following:

```
[
    {
        "key": "<string>",
        "value": "<string>",
        "version": "<string>"
    }
]
```

The fields are the following:

- `key`: The hex-encoded key of the record.
- `value`: The hex-encoded value of the record.
- `version`: The hex-encoded version of the record.

CHAPTER 5

Ledger rules

Default ledger rules

Global rules

A transaction is made of multiple *record mutations*. `ACC` record mutations are subject to a balancing rule. The balancing rule works as follow:

1. For every `ACC` record, the delta between the previous balance and the new proposed balance is calculated.
2. The sum of all deltas **per asset type** is calculated.
3. For every asset type, the sum must be equal to zero.

This ensures every asset creation and destruction is recorded through an account in the system. This means however that at least one account must be able to have a negative balance. Usually, a special account is used to do so, and the ability to create a negative balance on an account requires special permissions.

Tip: *Third-party asset issuance accounts* are allowed to have negative balances.

Aliases (`/aka/<name>/`)

Openchain has the ability to define aliases for accounts, this simplify the user experience as users no longer have to remember a base-58 random string of characters.

To do so, clients should understand the following syntax as a valid account path: `@<name>`, and turn it internally into `/aka/<name>/`.

Example

If a user wants to send funds to the following account:

```
@bank
```

The client application should convert it internally into:

```
/aka/bank/
```

Goto records (`goto`)

Goto records are special *DATA records* instructing the client application to use a different account.

Goto records must have the special name `goto`.

When a client application sends funds to a path, it must first look for a `DATA` record named `goto`. If it exists, the client application must use the path defined as the value of the record instead.

Example

If a user wants to send funds to the following account:

```
/account/alpha/
```

The client must first check the existance of a record with the following key:

```
/account/alpha/:DATA:goto
```

If the record doesn't exist, nothing happens and funds are sent to `/account/alpha/`. If the record exists, assuming its value is:

```
/account/beta/
```

Then funds are sent instead to `/account/beta/`.

Note: It is possible and recommended for security reasons that the client application uses a *check-only record* with the goto record to make sure the value of the goto record is still valid and hasn't changed when the transaction is validated.

Asset definition record (`asdef`)

It is important to be able to associate information with an asset type so that users have the right expectations about it.

The asset definition record can be used to record this information. The asset definition record is a `DATA` record with the special name `asdef`. In addition, it must be placed under the same path as the asset it is attached to.

Example

In order to associate information with the asset represented by path `/asset/gold/`, the following record must be set:

```
/asset/gold/:DATA:asdef
```

The value of the record is a UTF-8 string representing a JSON document with the following schema:

```
{
    name: '<string>',
    name_short: '<string>',
    icon_url: '<string>'
}
```

The definition of these fields are the following:

- `name`: The full name of the asset (e.g.: `U.S. Dollar`, `Gold Ounce`).
- `name_short`: The short name of the asset. This is used to denominate amounts (e.g.: `USD`, `XAU`)
- `icon_url`: The URL to an icon representing the asset.

Ledger info record (`info`)

Each Openchain instance can store a *DATA record* named `info` at the root path (`/`). In other words, the record key should be `/:DATA:info`.

The info record exposes meta-information about the ledger itself. The value must be a JSON document with the following schema:

```
{
    name: '<string>',
    validator_url: '<string>',
    tos: '<string>',
    webpage_url: '<string>'
}
```

The definition of these fields are the following:

- `name`: The name of the Openchain instance.
- `validator_url`: The URL of the main validator for this Openchain instance.
- `tos`: The terms of service of the Openchain instance.
- `webpage_url`: A link to user-readable content where users can get more information about this Openchain instance.

Pay-To-Pubkey-Hash accounts (`/p2pkh/<address>/`)

Pay-To-Pubkey-Hash accounts are special accounts with implicit permissions. Signing a transaction spending funds from this account or any sub-account requires the private key corresponding to `<address>`.

This automatically works with any account of that format, where `<address>` is a valid base-58 address.

Note: `<address>` is a base-58 address constructed in the same way a Bitcoin address for the same private and public key would be.

Third-party asset issuance accounts (`/asset/p2pkh/<address>/`)

Third-party asset issuance accounts are special accounts with implicit permissions. The owner of the private key corresponding to <address> can sign transactions spending funds from this account. Funds have to be of the asset type

`/asset/p2pkh/<address>`. Also, this address is authorized to have a negative balance. This means it is possible to use this address as the issuance source of asset type `/asset/p2pkh/<address>`.

This automatically works with any account of that format, where `<address>` is a valid base-58 address.

Note: `<address>` is a base-58 address constructed in the same way a Bitcoin address for the same private and public key would be.

Dynamic permissions

Openchain supports an implicit permission layout through *P2PKH accounts* (`/p2pkh/<address>/`) and *thrid party issuance accounts* (`/asset/p2pkh/<address>/`). It is also possible to dynamically define permissions by submitting transactions modifying a special record: the `acl` record.

Access Control Lists

Permissions are applied to a specific path. To apply an access control list to a path, set the `acl` record under that path. It must be a `DATA` record. The value is a JSON file.

For example, when trying to set the permissions to the path `/users/alice/`, the following record must be set: `/users/alice/:DATA:acl`.

Schema

The schema of the JSON file that the record contains is the following:

```
[
    {
        "subjects": [
            {
                "addresses": [ "<string>" ],
                "required": <integer>
            }
        ],
        "recursive": <boolean>,
        "record_name": "<string>",
        "record_name_matching": "<record-matching-type>",
        "permissions": {
            "account_negative": "<permission>",
            "account_spend": "<permission>",
            "account_modify": "<permission>",
            "account_create": "<permission>",
            "data_modify": "<permission>"
        }
    }
]
```

The contents is an array containing all the applicable permissions. When the `acl` record does not exist, this is equivalent to having an empty array.

The meaning of the fields within a permission object are the following:

- `subjects`: An array of subjects for which this permission object applies.

  - `addresses`: An array of strings representing the addresses for which signatures are expected.
  - `required`: The number of required signatures from the `addresses` array. If the `addresses` array contains 3 addresses, and `required` is set to 2, that means that for the permission to apply, at least 2 signatures from the 3 addresses specified must be present. This is known as a n-of-m multi-signature scheme.

- `recursive`: (Default: true) A boolean indicating whether the permission applies recursively to the sub accounts.

Note: With recursion, lower level permissions overrule higher level permissions.

- `record_name`: (Default: empty string) The pattern to use for record name matching.
- `record_name_matching`: (Default: `Prefix`) The type of record name matching to use. There are two possible values:
  - `Exact` means that the record name must be exactly equal to the value of the `record_name` field for the permission to apply.
  - `Prefix` means that the record name must start with the value of the `record_name` field for the permission to apply. Using `Prefix` with an empty `record_name` means that the permission applies to all records.

Hint: The record name of an `ACC` record is the asset path.

- `permissions`: Contains the permissions being applied if this permission object is a match. The meaning of the various permissions is explained in the next section. The value must be set to `Permit` for the permission to be granted, or `Deny` for the permission to be denied. If it is unset, the inherited value is used.

Permissions

account_negative

This permission indicates the right to affect the balance of `ACC` records, both to increase it (receive funds) and decrease it (send funds) with no restriction on the final balance. If this permission is granted, the `ACC` record balance can be made negative.

This permission is typically granted to the users allowed to issue an asset.

account_spend

This permission indicates the right to affect the balance of `ACC` records, both to increase it (receive funds) and decrease it (send funds) with the restriction that the final balance must remain positive or zero.

account_modify

This permission is required to affect the balance of `ACC` records that have already been modified before (the record version is non-empty).

account_create

This permission is required to affect the balance of `ACC` records that have never been modified before (the record version is empty).

Note: A user can only send funds from an account if she has the `account_negative` or `account_spend` rights plus the `account_modify` or `account_create` rights. Sending to an account requires `account_modify` or `account_create` on the destination account.

A closed loop ledger can be created by denying `account_modify` and `account_create` by default, and selectively granting these for some accounts. By doing this, only approved accounts can receive funds.

---

data_modify

This permission is required to modify a `DATA` record.

How to: Configure a ledger to be closed-loop

Financial institutions and companies letting their users transfer value often have to comply with regulations that require them to "know their customers" (KYC).

It is possible to use Openchain in this configuration with little effort. This section describes the necessary steps.

The goal of this walkthrough is to configure Openchain so that:

1. Users go through an external registration process where they have their identity verified by the company administrating the ledger, and associate their identity with a public key.
2. Only public keys matching a registered user can be used to send funds.
3. Funds can only be sent to registered users.

This way, funds can only circulate amongst "known" users.

Initial configuration

The Openchain instance must be configured with both *P2PKH accounts* and *thrid party issuance accounts* disabled. The settings `validator_mode:validator:allow_p2pkh_accounts` and `validator_mode:validator:allow_third_party_assets` must both be set to false to achieve this. See *this section* for more details.

With this configuration, by default, users have no rights, while administrators have all rights. It is not possible for any normal user to either send or receive tokens.

Onboarding process

The second step is to build an onboarding workflow for the users. For example, this could be a mobile application where the user creates a username and password, enters her email address and submits a proof of identity (photo of her passport).

As part of the process, a **private key** is generated and stored on the user's device. The matching **public key** is sent along with the other pieces of information. This part can be entirely invisible to the user.

Creating the access rights

Once the company has validated the identity of the user, it can create an account on Openchain for that user, and associate her username with her public key.

Aliases are based on a special path (`/aka/<alias>/`). Assuming that the username of the user is `alice`, we need to:

1. Allow other users to send funds to `/aka/alice/` (and subaccounts).
2. Allow Alice's public key to be used to spend funds on `/aka/alice/` (and subaccounts).

This can be achieved by creating an `acl` record under `/aka/alice/`.

Tip: See the documentation about *dynamic permissions* for more details.

The record `/aka/alice/:DATA:acl` must be created and set to:

```
[
    {
        "subjects": [ { "addresses": [ ], "required": 0 } ],
        "permissions": { "account_modify": "Permit", "account_create": "Permit" }
    },
    {
        "subjects": [ { "addresses": [ "<alices-address>" ], "required": 1 } ],
        "permissions": { "account_spend": "Permit" }
    }
]
```

Important: Since only an administrator will have the right to modify this record, the mutation creating this record must be signed using an administrator key.

Alice's address is the base-58 representation of the hash of her public key. It is constructed the same way it would be for a Bitcoin address.

By tweaking the access control list, it is possible to:

1. Handle multiple devices (with different keys) per user.
2. Implement multisignature schemes, for joint accounts for example.

Credit the user's account

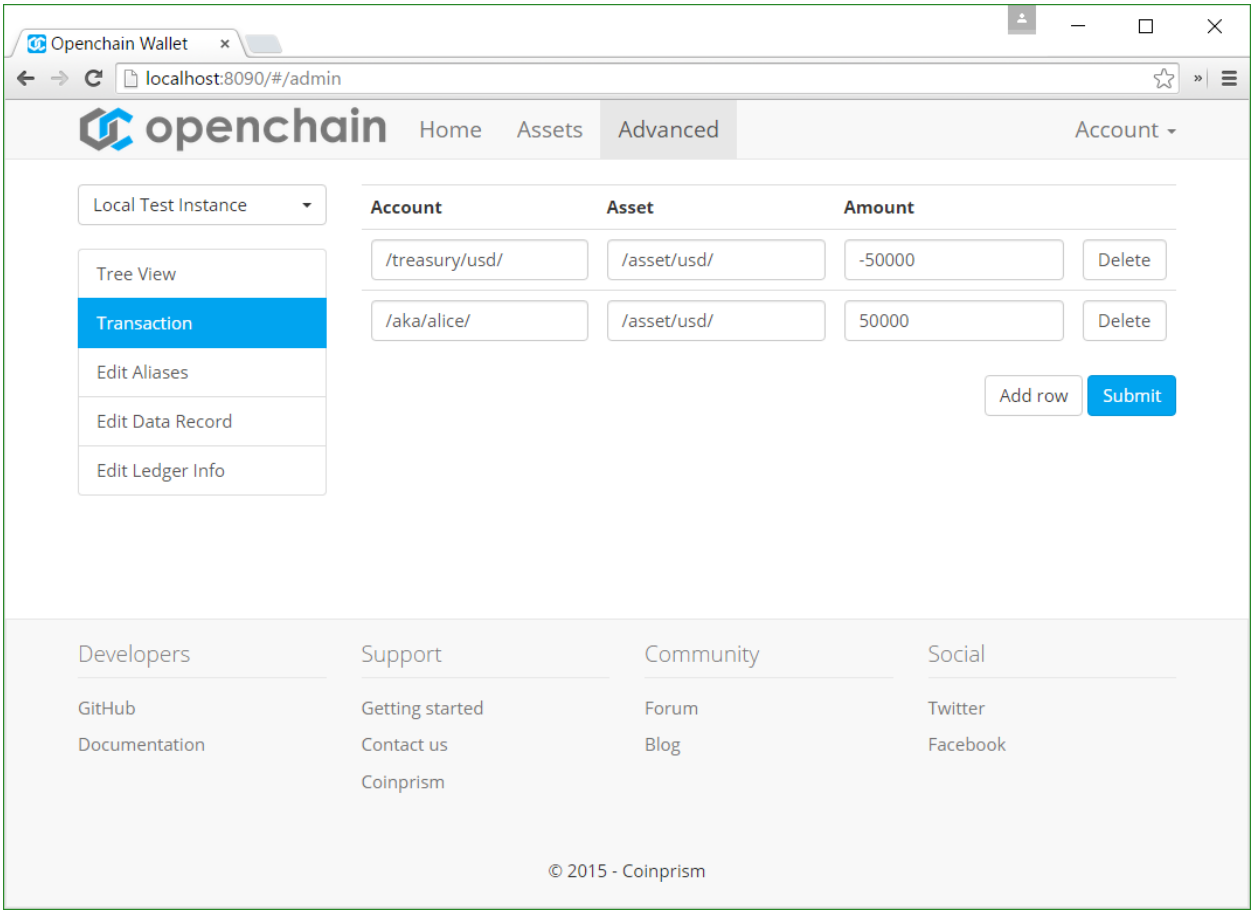
Now that the user has an account she can use, she will want to fund it. There are many possible configurations for this:

- A treasury is initially created by the company and credits are send from that treasury.
- Tokens are issued dynamically whenever the user purchases them through an external payment method.

Assuming the following:

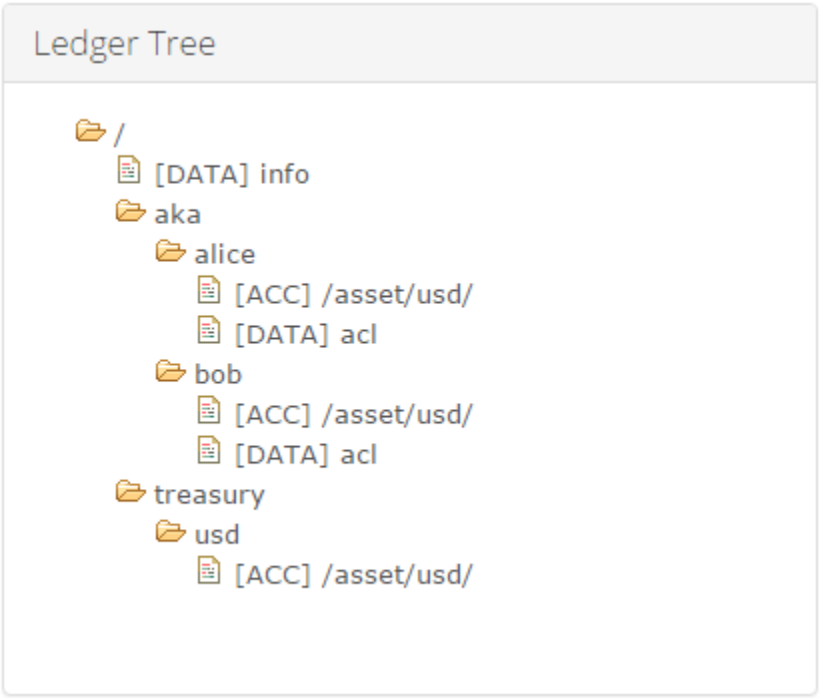
- The asset path for the tokens is `/asset/usd/` (this can be arbitrarily chosen).
- The tokens are dynamically issued from the account `/treasury/usd/`.

A funding transaction will simply take the form of a transaction sending X units of the asset `/asset/usd/` from the account `/treasury/usd/` to the account `/aka/alice/`.



The transaction should be signed by an administrator only an administrator has access to `/treasury/usd/`. The balance on `/treasury/usd/` will be negative, and reflect the total amount of tokens that have been issued on the ledger. Again, the administrator is allowed to make the balance negative.

The final ledger tree should look as follow:



Addressing loss and theft of the private keys

Inevitably, some users will lose the device on which their private key is stored.

When this happens, they should report it to the company administering the Openchain instance. The company will first perform identity checks, then ask the user to generate a new key on a new device.

The administrator can then simply update the relevant `acl` record to change the previous address into the new address, corresponding to the new key.

Handling fraudulent transactions

If fraudulent transactions have happened in the meantime, the administrator can commit a new transaction representing the opposite transfer.

For example, if 10 units have been sent fraudulently from `/aka/alice/` to `/aka/oscar/`, then the administrator can simply submit a new transaction sending 10 units from `/aka/oscar/` to `/aka/alice/`, thus reverting the effects of the fraudulent transaction. The ledger being immutable, both transactions will remain visible in the ledger, with the fact that the second transaction transferring funds back from `/aka/oscar/` is not signed by Oscar's key, but instead signed by the administrator's key.

Note: It bears mentioning that in a setup where all the users have to go through an identity verification process, it is unlikely that Oscar steals funds from Alice in the first place, since the company running the ledger has all the information about Oscar, and could press charges against him.

Conclusion

With this setup, users are able to send tokens to each other, however, they are not able to send funds to addresses that are not associated to a registered user.

This represents just one way to implement a closed-loop ledger, and there are many other possible configurations depending on the requirements.