Iroha handbook: installation, getting started, API, guides, and troubleshooting

Release

Hyperledger Iroha community

Apr 22, 2020

Table of contents

Welcome! Hyperledger Iroha is a simple blockchain platform you can use to make trusted, secure, and fast applications by bringing the power of permission-based blockchain with Crash fault-tolerant consensus. It's free, open-source, and works on Linux and Mac OS, with a variety of mobile and desktop libraries.

You can download the source code of Hyperledger Iroha and latest releases from GitHub page.

This documentation will guide you through the installation, deployment, and launch of Iroha network, and explain to you how to write an application for it. We will also see which use case scenarios are feasible now, and are going to be implemented in the future.

As Hyperledger Iroha is an open-source project, we will also cover contribution part and explain you a working process.

CHAPTER 1

Overview of Iroha

1.1 What are the key features of Iroha?

- Simple deployment and maintenance
- Variety of libraries for developers
- Role-based access control
- Modular design, driven by command–query separation principle
- Assets and identity management

In our quality model, we focus on and continuously improve:

- Reliability (fault tolerance, recoverability)
- Performance Efficiency (in particular time-behavior and resource utilization)
- Usability (learnability, user error protection, appropriateness recognisability)

1.2 Where can Iroha be used?

Hyperledger Iroha is a general purpose permissioned blockchain system that can be used to manage digital assets, identity, and serialized data. This can be useful for applications such as interbank settlement, central bank digital currencies, payment systems, national IDs, and logistics, among others.

For a detailed description please check our Use Case Scenarios section.

1.3 How is it different from Bitcoin or Ethereum?

Bitcoin and Ethereum are designed to be permissionless ledgers where anyone can join and access all the data. They also have native cryptocurrencies that are required to interact with the systems.

In Iroha, there is no native cryptocurrency. Instead, to meet the needs of enterprises, system interaction is permissioned, meaning that only people with requisite access can interact with the system. Additionally, queries are also permissioned, such that access to all the data can be controlled.

One major difference from Ethereum, in particular, is that Hyperledger Iroha allows users to perform common functions, such as creating and transferring digital assets, by using prebuilt commands that are in the system. This negates the need to write cumbersome and hard to test smart contracts, enabling developers to complete simple tasks faster and with less risk.

1.4 How is it different from the rest of Hyperledger frameworks or other permissioned blockchains?

Iroha has a novel, Crash fault tolerant consensus algorithm (called YAC[1]) that is high-performance and allows for finality of transactions with low latency.

Also, Iroha's built-in commands are a major benefit compared to other platforms, since it is very simple to do common tasks such as create digital assets, register accounts, and transfer assets between accounts. Moreover, it narrows the attack vector, improving overall security of the system, as there are less things to fail.

Finally, Iroha is the only ledger that has a robust permission system, allowing permissions to be set for all commands, queries, and joining of the network.

1.5 How to create applications around Iroha?

In order to bring the power of blockchain into your application, you should think first of how it is going to interface with Iroha peers. A good start is to check Core Concepts section, explaining what exactly is a transaction and query, and how users of your application are supposed to interact with it.

We also have several client libraries which provide tools for developers to form building blocks, such as signatures, commands, send messages to Iroha peers and check the status.

[1] Yet Another Consensus

CHAPTER 2

Getting Started

In this guide, we will create a very basic Iroha network, launch it, create a couple of transactions, and check the data written in the ledger. To keep things simple, we will use Docker.

Note: Ledger is the synonym for a blockchain, and Hyperledger Iroha is known also as Distributed Ledger Technology framework — which in essence is the same as "blockchain framework". You can check the rest of terminology used in the *Core concepts* section.

2.1 Prerequisites

For this guide, you need a machine with `Docker` installed. You can read how to install it on a Docker's website.

Note: Of course you can build Iroha from scratch, modify its code and launch a customized node! If you are curious how to do that — you can check *Building Iroha* section. In this guide we will use a standard distribution of Iroha available as a docker image.

2.2 Starting Iroha Node

2.2.1 Creating a Docker Network

To operate, Iroha requires a `PostgreSQL` database. Let's start with creating a Docker network, so containers for Postgres and Iroha can run on the same virtual network and successfully communicate. In this guide we will call it `iroha-network`, but you can use any name. In your terminal write following command:

```
docker network create iroha-network
```

2.2.2 Starting PostgreSQL Container

Now we need to run `PostgreSQL` in a container, attach it to the network you have created before, and expose ports for communication:

```
docker run --name some-postgres \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=mysecretpassword \
-p 5432:5432 \
--network=iroha-network \
-d postgres:9.5 \
-c 'max_prepared_transactions=100'
```

Note: If you already have Postgres running on a host system on default port (5432), then you should pick another free port that will be occupied. For example, 5433: `-p 5433:5432`

2.2.3 Creating Blockstore

Before we run Iroha container, we may create a persistent volume to store files, storing blocks for the chain. It is done via the following command:

```
docker volume create blockstore
```

2.2.4 Preparing the configuration files

Note: To keep things simple, in this guide we will create a network containing only a single node. To understand how to run several peers, follow *Deploying Iroha*

Now we need to configure our Iroha network. This includes creating a configuration file, generating keypairs for a users, writing a list of peers and creating a genesis block.

Don't be scared away — we have prepared an example configuration for this guide, so you can start testing Iroha node now. In order to get those files, you need to clone the Iroha repository from Github or copy them manually (cloning is faster, though).

```
git clone -b master https://github.com/hyperledger/iroha --depth=1
```

Hint: `--depth=1` option allows us to download only the latest commit and save some time and bandwidth. If you want to get a full commit history, you can omit this option.

There is a guide on how to set up the parameters and tune them with respect to your environment and load expectations: *Configuration*. We don't need to do this at the moment.

2.2.5 Starting Iroha Container

We are almost ready to launch our Iroha container. You just need to know the path to configuration files (from the step above).

Let's start Iroha node in Docker container with the following command:

```
docker run --name iroha \
-d \
-p 50051:50051 \
-v $(pwd)/iroha/example:/opt/iroha_data \
-v blockstore:/tmp/block_store \
--network=iroha-network \
-e KEY='node0' \
hyperledger/iroha:latest
```

If you started the node successfully you would see the container id in the same console where you started the container.

Let's look in details what this command does:

- `docker run --name iroha \` creates a container `iroha`
- `-d \` runs container in the background
- `-p 50051:50051 \` exposes a port for communication with a client (we will use this later)
- `-v YOUR_PATH_TO_CONF_FILES:/opt/iroha_data \` is how we pass our configuration files to docker container. The example directory is indicated in the code block above.
- `-v blockstore:/tmp/block_store \` adds persistent block storage (Docker volume) to a container, so that the blocks aren't lost after we stop the container
- `--network=iroha-network \` adds our container to previously created `iroha-network` for communication with PostgreSQL server
- `-e KEY='node0' \` - here please indicate a key name that will identify the node allowing it to confirm operations. The keys should be placed in the directory with configuration files mentioned above.
- `hyperledger/iroha:latest` is a reference to the image pointing to the latest release

You can check the logs by running `docker logs iroha`.

You can try using one of sample guides in order to send some transactions to Iroha and query its state.

2.3 Try other guides

2.3.1 CLI guide: sending your first transactions and queries

You can interact with Iroha using various ways. You can use our client libraries to write code in various programming languages (e.g. Java, Python, Javascript, Swift) which communicates with Iroha. Alternatively, you can use `iroha-cli` – our command-line tool for interacting with Iroha. As a part of this guide, let's get familiar with `iroha-cli`

> **Attention:** Despite that `iroha-cli` is arguably the simplest way to start working with Iroha, `iroha-cli` covers only some possible commands/queries, so user experience might not be the best. If you want to help us build a better CLI version please let us know!

Open a new terminal (note that Iroha container and `irohad` should be up and running) and attach to an `iroha` docker container:

```
docker exec -it iroha /bin/bash
```

Now you are in the interactive shell of Iroha's container again. We need to launch `iroha-cli` and pass an account name of the desired user. In our example, the account `admin` is already created in the `test` domain. Let's use this account to work with Iroha.

```
iroha-cli -account_name admin@test
```

Note: Full account name has a `@` symbol between name and domain. Note that the keypair has the same name.

Creating the First Transaction

You can see the interface of `iroha-cli` now. Let's create a new asset, add some asset to the admin account and transfer it to other account. To achieve this, please choose option `1. New transaction (tx)` by writing `tx` or `1` to a console.

Now you can see a list of available commands. Let's try creating a new asset. Select `14. Create Asset (crt_ast)`. Now enter a name for your asset, for example `coolcoin`. Next, enter a Domain ID. In our example we already have a domain `test`, so let's use it. Then we need to enter an asset precision – the amount of numbers in a fractional part. Let's set precision to `2`.

Congratulations, you have created your first command and added it to a transaction! You can either send it to Iroha or add some more commands `1. Add one more command to the transaction (add)`. Let's add more commands, so we can do everything in one shot. Type `add`.

Now try adding some `coolcoins` to our account. Select `16. Add Asset Quantity (add_ast_qty)`, enter asset ID – `coolcoin#test`, integer part and `coolcoin#test`, integer part and precision. For example, to add 200.50 precision. For example, to add 200.50 `coolcoins`, we need to enter integer `coolcoins`, we need to enter integer part as `20050` and precision as part as `20050` and precision as `2`, so it becomes `200.50`.

Note: Full asset name has a `#` symbol between name and domain.

Let's transfer 100.50 `coolcoins` from `admin@test` to `test@test` by adding one more command and choosing `5. Transfer Assets (tran_ast)`. Enter Source Account and Destination Account, in our case `admin@test` and `test@test`, Asset ID (`coolcoin#test`), integer part and precision (`10050` and `2` accordingly).

Now we need to send our transaction to Iroha peer (`2. Send to Iroha peer (send)`). Enter peer address (in our case `localhost`) and port (`50051`). Now your transaction is submitted and you can see your transaction hash. You can use it to check transaction's status.

Go back to a terminal where `irohad` is running. You can see logs of your transaction.

Yay! You have submitted your first transaction to Iroha.

Creating the First Query

Now let's check if `coolcoins` were successfully transferred from `admin@test` to `test@test`. Choose `2. New query (qry)`. `8. Get Account's Assets (get_acc_ast)` can help you to check if `test@test` now has `coolcoin`. Form a query in a similar way you did with commands you did with commands and `1. Send to Iroha peer (send)`. Now you can see information about how many `coolcoin` does `test@test` have. It will look similar to this:

```
[2018-03-21 12:33:23.179275525][th:36][info] QueryResponseHandler [Account Assets]
[2018-03-21 12:33:23.179329199][th:36][info] QueryResponseHandler -Account Id:-␣
↪test@test
[2018-03-21 12:33:23.179338394][th:36][info] QueryResponseHandler -Asset Id- coolcoin
↪#test
[2018-03-21 12:33:23.179387969][th:36][info] QueryResponseHandler -Balance- 100.50
```

Isn't that awesome? You have submitted your first query to Iroha and got a response!

Hint: To get information about all available commands and queries please check our API section.

Being Badass

Let's try being badass and cheat Iroha. For example, let's transfer more `coolcoins` than `admin@test` has. Try to transfer 100000.00 `coolcoins` from `admin@test` to `test@test`. Again, proceed to `1. New transaction (tx)`, `5. Transfer Assets (tran_ast)`, enter Source Account and Destination Account, in our case `admin@test` and `test@test`, Asset ID (`coolcoin#test`), integer part and precision (`10000000` and `2` accordingly). Send a transaction to Iroha peer as you did before. Well, it says

```
[2018-03-21 12:58:40.791297963][th:520][info] TransactionResponseHandler Transaction␣
↪successfully sent
Congratulation, your transaction was accepted for processing.
Its hash is fc1c23f2de1b6fccbfe1166805e31697118b57d7bb5b1f583f2d96e78f60c241
```

Your transaction was accepted for processing. Does it mean that we had successfully cheated Iroha? Let's try to see transaction's status. Choose `3. New transaction status request (st)` and enter transaction's hash which you can get in the console after the previous command. Let's send it to Iroha. It replies with:

```
Transaction has not passed stateful validation.
```

Apparently no. Our transaction was not accepted because it did not pass stateful validation and `coolcoins` were not transferred. You can check the status of `admin@test` and `test@test` with queries to be sure (like we did earlier).

2.3.2 Sending transactions with Python library

Prerequisites

Note: The library only works in Python 3 environment (Python 2 is not supported yet).

To use Iroha Python library, you need to get it from the repository or via pip3:

```
pip3 install iroha
```

Now, as we have the library, we can start sending the actual transactions.

Running example transactions

If you only want to try what Iroha transactions would look like, you can simply go to the examples from the repository here. Let's check out the *tx-example.py* file.

Here are Iroha dependencies. Python library generally consists of 3 parts: Iroha, IrohaCrypto and IrohaGrpc which we need to import:

```
from iroha import Iroha, IrohaGrpc
from iroha import IrohaCrypto
```

The line

```
from iroha.primitive_pb2 import can_set_my_account_detail
```

is actually about the permissions you might be using for the transaction. You can find a full list here: Permissions.

In the next block we can see the following:

```
admin_private_key = 'f101537e319568c765b2cc89698325604991dca57b9716b58016b253506cab70'
user_private_key = IrohaCrypto.private_key()
user_public_key = IrohaCrypto.derive_public_key(user_private_key)
iroha = Iroha('admin@test')
net = IrohaGrpc()
```

Here you can see the example account information. It will be used later with the commands. If you change the commands in the transaction, the set of data in this part might also change depending on what you need.

Defining the commands

Let's look at the first of the defined commands:

```
def create_domain_and_asset():
    commands = [
        iroha.command('CreateDomain', domain_id='domain', default_role='user'),
        iroha.command('CreateAsset', asset_name='coin',
                      domain_id='domain', precision=2)
    ]
    tx = IrohaCrypto.sign_transaction(
        iroha.transaction(commands), admin_private_key)
    send_transaction_and_print_status(tx)
```

Here we define a transaction made of 2 commands: CreateDomain and CreateAsset. You can find a full list here: commands. Each of Iroha commands has its own set of parameters. You can check them in command descriptions in iroha-api-reference.

Then we sign the transaction with the parameters defined earlier.

You can define queries the same way.

Running the commands

Last lines

```
create_domain_and_asset()
add_coin_to_admin()
create_account_userone()
...
```

run the commands defined previously.

Now, if you have *irohad* running, you can run the example or your own file by simply opening the .py file in another tab.

CHAPTER 3

Use Case Scenarios

We list a number of use cases and specific advantages that Hyperledger Iroha can introduce to these applications. We hope that the applications and use cases will inspire developers and creators to further innovation with Hyperledger Iroha.

3.1 Certificates in Education, Healthcare

Hyperledger Iroha incorporates into the system multiple certifying authorities such as universities, schools, and medical institutions. Flexible permission model used in Hyperledger Iroha allows building certifying identities, and grant certificates. The storage of explicit and implicit information in users' account allows building various reputation and identity systems.

By using Hyperledger Iroha each education or medical certificate can be verified that it was issued by certain certifying authorities. Immutability and clear validation rules provide transparency to health and education significantly reducing the usage of fake certificates.

3.1.1 Example

Imagine a medical institution registered as a `hospital` domain in Hyperledger Iroha. This domain has certified and registered workers each having some role, e.g. `physician`, `therapist`, `nurse`. Each patient of the hospital has an account with full medical history. Each medical record, like blood test results, is securely and privately stored in the account of the patient as JSON key/values. Rules in `hospital` domain are defined such that only certified medical workers and the user can access the personal information. The medical data returned by a query is verified that it comes from a trusted source.

Hospital is tied to a specific location, following legal rules of that location, like storing personal data of citizens only in specific regions(privacy rules). A multi-domain approach in Hyperledger Iroha allows sharing information across multiple countries not violating legal rules. For example, if the user `makoto@hospital` decides to share personal case history with a medical institution in another country, the user can use `grant` command with permission `can_get_my_acc_detail`.

Similar to a medical institution, a registered university in Hyperledger Iroha has permissions to push information to the graduated students. A diploma or certificate is essentially Proof-of-Graduation with a signature of recognized University. This approach helps to ease hiring process, with an employer making a query to Hyperledger Iroha to get the acquired skills and competence of the potential employee.

3.2 Cross-Border Asset Transfers

Hyperledger Iroha provides fast and clear trade and settlement rules using multi-signature accounts and atomic exchange. Asset management is easy as in centralized systems while providing necessary security guarantees. By simplifying the rules and commands required to create and transfer assets, we lower the barrier to entry, while at the same time maintaining high-security guarantees.

3.2.1 Example

For example[1], a user might want to transfer the ownership of a car. User `haruto` has registered owner-asset relationship with a car of `sora` brand with parameters: `{"id": "343220697320746865206 20616E73776572", "color": "red", "size": "small"}`. This ownership is fixed in an underlying database of the system with copies at each validating peer. To perform the transfer operation user `haruto` creates an offer, i.e. a multi-signature transaction with two commands: `transfer` to user `haru` the car identifier and `transfer` some amount of `usd` tokens from `haru` to `haruto`. Upon receiving the offer `haru` accepts it by signing the multi-signature transaction, in this case, transaction atomically commits to the system.

Hypeledger Iroha has no built-in token, but it supports different assets from various creators. This approach allows building a decentralized exchange market. For example, the system can have central banks from different countries to issue assets.

3.3 Financial Applications

Hyperleger Iroha can be very useful in the auditing process. Each information is validated by business rules and is constantly maintained by distinct network participants. Access control rules along with some encryption maintain desired level of privacy. Access control rules can be defined at different levels: user-level, domain-level or system-level. At the user-level privacy rules for a specific individual are defined. If access rules are determined at domain or system level, they are affecting all users in the domain. In Hyperledger Iroha we provide convenient role-based access control rules, where each role has specific permissions.

Transactions can be traced with a local database. Using Iroha-API auditor can query and perform analytics on the data, execute specific audit software. Hyperledger Iroha supports different scenarios for deploying analytics software: on a local computer, or execute code on specific middleware. This approach allows analyzing Big Data application with Hadoop, Apache, and others. Hypeledger Iroha serves as a guarantor of data integrity and privacy (due to the query permissions restriction).

3.3.1 Example

For example, auditing can be helpful in financial applications. An auditor account has a role of the `auditor` with permissions to access the information of users in the domain without bothering the user. To reduce the probability of account hijacking and prevent the auditor from sending malicious queries, the auditor is typically defined as a multi-signature account, meaning that auditor can make queries only having signatures from multiple separate identities. The auditor can make queries not only to fetch account data and balance but also all transactions of a user, e.g. all

[1] Currently not implemented

transfers of user `haruto` in domain `konoha`. To efficiently analyze data of million users each Iroha node can work in tandem with analytics software.

Multi-signature transactions are a powerful tool of Hyperledger Iroha that can disrupt tax system. Each transaction in a certain domain can be as a multi-signature transaction, where one signature comes from the user (for example asset transfer) and the second signature comes from special taxing nodes. Taxing nodes will have special validation rules written using Iroha-API, e.g. each purchase in the certified stores must pay taxes. In other words, Iroha a valid purchase transaction must contain two commands: money transfer(purchase) to the store and money transfer(tax payment) to the government.

3.4 Identity Management

Hyperledger Iroha has an intrinsic support for identity management. Each user in the system has a uniquely identified account with personal information, and each transaction is signed and associated with a certain user. This makes Hyperledger Iroha perfect for various application with KYC (Know Your Customer) features.

3.4.1 Example

For example, insurance companies can benefit from querying the information of user's transaction without worrying about the information truthfulness. Users can also benefit from storing personal information on a blockchain since authenticated information will reduce the time of claims processing. Imagine a situation where a user wants to make a hard money loan. Currently, pre-qualification is a tedious process of gathering information about income, debts and information verification. Each user in Hyperledger Iroha has an account with verified personal information, such as owning assets, job positions, and debts. User income and debts can be traced using query `GetAccountTransactions`, owning assets using query `GetAccountAssets` and job positions using `GetAccountDetail`. Each query returns verified result reducing the processing time of hard money loan will take only a few seconds. To incentivize users to share personal information, various companies can come up with business processes. For example, insurance companies can create bonus discounts for users making fitness activities. Fitness applications can push private Proof-of-Activity to the system, and the user can decide later to share information with insurance companies using `GrantPermission` with permission `can_get_my_acc_detail`.

3.5 Supply Chain

Governance of a decentralized system and representing legal rules as a system's code is an essential combination of any supply chain system. Certification system used in Hyperledger Iroha allows tokenization of physical items and embedding them into the system. Each item comes with the information about "what, when, where and why".

Permission systems and restricted set of secure core commands narrows the attack vector and provides effortlessly a basic level of privacy. Each transaction is traceable within a system with a hash value, by the credentials or certificates of the creator.

3.5.1 Example

Food supply chain is a shared system with multiple different actors, such as farmers, storehouses, grocery stores, and customers. The goal is to deliver food from a farmer's field to the table of a customer. The product goes through many stages, with each stage recorded in shared space. A customer scans a code of the product via a mobile device, in which an Iroha query is encoded. Iroha query provides a full history with all stages, information about the product and the farmer.

For example, `gangreen` is a registered farmer `tomato` asset creator, he serves as a guarantor tokenizing physical items, i.e. associating each tomato with an Iroha `tomato` item. Asset creation and distribution processes are totally transparent for network participants. Iroha `tomato` goes on a journey through a multitude of vendors to finally come to user `chad`.

We simplified asset creation to just a single command `CreateAsset` without the need to create complex smart contracts. One the major advantages of Hyperledger Iroha is in its ease, that allows developers to focus on the provided value of their applications.

3.6 Fund Management

With the support of multisignature transactions it is possible to maintain a fund by many managers. In that scheme investment can only be made after the confirmation of the quorum participants.

3.6.1 Example

The fund assets should be held at one account. Its signatories should be fund managers, who are dealing with investments and portfolio distributions. That can be added via `AddSignatory` command. All of the assets should be held within one account, which signatories represent the fund managers. Thus the concrete exchanges can be performed with the multisignature transaction so that everyone will decide on a particular financial decision. The one may confirm a deal by sending the original transaction and one of managers' signature. Iroha will maintain the transaction sending so that the deal will not be completed until it receives the required number of confirmation, which is parametrized with the transaction quorum parameter.

3.7 Related Research

(The idea was to show current pioneers of blockchain applications and their works.)

CHAPTER 4

Core concepts

Why Iroha runs in a network? How to understand the objects inside and outside the system? How peers in the network collaborate and decide which data to put into the blockchain? We will look through the basics of Iroha in this section.

4.1 Sections

4.1.1 Account

An Iroha entity that is able to perform specified set of actions. Each account belongs to one of existing *domains*.

An account has some number of *roles* (can be null) — which is a collection of permissions. Only *grantable permissions* are assigned to an account directly.

4.1.2 Asset

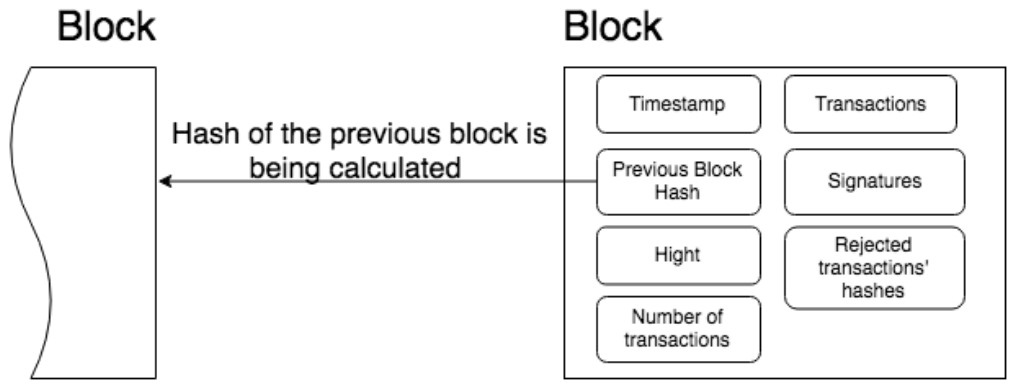
Any countable commodity or value. Each asset is related to one of existing *domains*. For example, an asset can represent any kind of such units - currency unit, a bar of gold, real estate unit, etc.

4.1.3 Block

Transaction data is permanently recorded in files called blocks. Blocks are organized into a linear sequence over time (also known as the block chain)[1].

Blocks are signed with the cryptographic signatures of Iroha *peers*, voting for this block during *consensus*. Signable content is called payload, so the structure of a block looks like this:

[1] https://en.bitcoin.it/wiki/Block



Outside payload

- signatures — signatures of peers, which voted for the block during consensus round

Inside payload

- height — a number of blocks in the chain up to the block
- timestamp — Unix time (in milliseconds) of block forming by a peer
- array of transactions, which successfully passed validation and consensus step
- hash of a previous block in the chain
- rejected transactions hashes — array of transaction hashes, which did not pass stateful validation step; this field is optional

4.1.4 Client

Any application that uses Iroha is treated as a client.

A distinctive feature of Iroha is that all clients are using simple client-server abstractions when they interact with a peer network: they don't use any abstractions which are specific for blockchain-related systems. For example, in Bitcoin clients have to validate blocks, or in HL Fabric they need to poll several peers to make sure that a transaction was written in a block, whereas in HL Iroha a client interacts with any peer similarly to a single server.

4.1.5 Command

A command is an intention to change the *state* of the network. For example, in order to create a new *role* in Iroha you have to issue Create role command.

4.1.6 Consensus

A consensus algorithm is a process in computer science used to achieve agreement on a single data value among distributed processes or systems. Consensus algorithms are designed to achieve reliability in a network involving multiple unreliable nodes. Solving that issue – known as the consensus problem – is important in distributed computing and multi-agent systems.

Consensus, as an algorithm

> An algorithm to achieve agreement on a block among peers in the network. By having it in the system, reliability is increased.

For consensus as Iroha's component, please check this link.

4.1.7 Domain

A named abstraction for grouping *accounts* and *assets*. For example, it can represent an organisation in the group of organisations working with Iroha.

4.1.8 Peer

A node that is a part of Iroha network. It participates in *consensus* process.

4.1.9 Permission

A named rule that gives the privilege to perform a command. Permission **cannot** be granted to an *account* directly, instead, account has roles, which are collections of permissions. Although, there is an exception, see *Grantable Permission*.

List of Iroha permissions.

Grantable Permission

Only grantable permission is given to an *account* directly. An account that holds grantable permission is allowed to perform some particular action on behalf of another account. For example, if account a@domain1 gives the account b@domain2 a permission that it can transfer assets — then b@domain2 can transfer assets of a@domain1 to anyone.

4.1.10 Proposal

A set of *transactions* that have passed only *stateless validation*.

Verified Proposal

A set of transactions that have passed both *stateless* and *stateful* validation, but were not committed yet.

4.1.11 Query

A request to Iroha that does **not** change the state of the network. By performing a query, a client can request data from the state, for example a balance of his account, a history of transactions, etc.

4.1.12 Quorum

In the context of transactions signing, quorum number is a minimum amount of signatures required to consider a transaction signed. The default value is 1. For *MST transactions* you will need to increase that number.

Each account can link additional public keys and increase own quorum number.

4.1.13 Role

A named abstraction that holds a set of *permissions*.

4.1.14 Signatory

Represents an entity that can confirm multisignature transactions for an *account*. It can be attached to account via AddSignatory and detached via RemoveSignatory.

4.1.15 Transaction

An ordered set of *commands*, which is applied to the ledger atomically. Any non-valid command within a transaction leads to rejection of the whole transaction during the validation process.

Transaction Structure

Payload stores all transaction fields, except signatures:

- Time of creation (unix time, in milliseconds)
- Account ID of transaction creator (username@domain)
- Quorum field (indicates required number of signatures)
- Repeated commands which are described in details in commands section
- Batch meta information (optional part). See *Batch of Transactions* for details

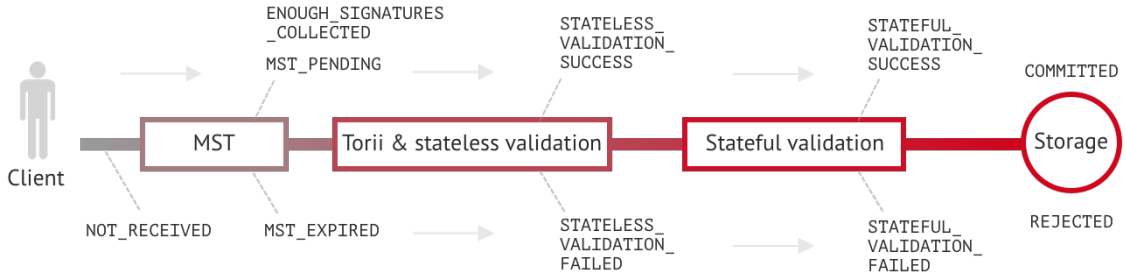
Signatures contain one or many signatures (ed25519 public key + signature)

Reduced Transaction Hash

Reduced hash is calculated over transaction payload excluding batch meta information. Used in *Batch of Transactions*.

Transaction Statuses

Hyperledger Iroha supports both push and pull interaction mode with a client. A client that uses pull mode requests status updates about transactions from Iroha peer by sending transaction hashes and awaiting a response. On the contrary, push interaction is performed by listening of an event stream for each transaction. In any of these modes, the set of transaction statuses is the same:



Transaction Status Set

- NOT_RECEIVED: requested peer does not have this transaction.
- ENOUGH_SIGNATURES_COLLECTED: this is a multisignature transaction which has enough signatures and is going to be validated by the peer.

- MST_PENDING: this transaction is a multisignature transaction which has to be signed by more keys (as requested in quorum field).
- MST_EXPIRED: this transaction is a multisignature transaction which is no longer valid and is going to be deleted by this peer.
- STATELESS_VALIDATION_FAILED: the transaction was formed with some fields, not meeting stateless validation constraints. This status is returned to a client, who formed transaction, right after the transaction was sent. It would also return the reason — what rule was violated.
- STATELESS_VALIDATION_SUCCESS: the transaction has successfully passed stateless validation. This status is returned to a client, who formed transaction, right after the transaction was sent.
- STATEFUL_VALIDATION_FAILED: the transaction has commands, which violate validation rules, checking state of the chain (e.g. asset balance, account permissions, etc.). It would also return the reason — what rule was violated.
- STATEFUL_VALIDATION_SUCCESS: the transaction has successfully passed stateful validation.
- COMMITTED: the transaction is the part of a block, which gained enough votes and is in the block store at the moment.
- REJECTED: this exact transaction was rejected by the peer during stateful validation step in previous consensus rounds. Rejected transactions' hashes are stored in *block* store. This is required in order to prevent replay attacks.

Pending Transactions

Any transaction that has lesser signatures at the moment than *quorum* of transaction creator account is considered as pending. Pending transaction will be submitted for *stateful validation* as soon as *multisignature* mechanism will collect required amount of signatures for quorum.

Transaction that already has quorum of signatures can also be considered as pending in cases when the transaction is a part of *batch of transactions* and there is a not fully signed transaction.

4.1.16 Batch of Transactions

Transactions batch is a feature that allows sending several transactions to Iroha at once preserving their order.

Each transaction within a batch includes batch meta information. Batch meta contains batch type identifier (atomic or ordered) and a list of *reduced hashes* of all transactions within a batch. The order of hashes defines transactions sequence.

Batch can contain transactions created by different accounts. Any transaction within a batch can require single or *multiple* signatures (depends on quorum set for an account of transaction creator). At least one transaction inside a batch should have at least one signature to let the batch pass *stateless validation*.

You can read an article about batches on our Contributors' Page on Medium.

Atomic Batch

All the transactions within an atomic batch should pass *stateful validation* for the batch to be applied to a ledger.

Ordered Batch

Ordered batch preserves only the sequence of transactions applying to a ledger. All the transactions that able to pass stateful validation within a batch will be applied to a ledger. Validation failure of one transaction would NOT directly imply the failure of the whole batch.

4.1.17 Multisignature Transactions

A transaction which has the *quorum* greater than one is considered as multisignature (also called mst). To achieve *stateful validity* the confirmation is required by the *signatories* of the creator account. These participants need to send the same transaction with their signature.

4.1.18 Validation

There are two kinds of validation - stateless and stateful.

Stateless Validation

Performed in Torii. Checks if an object is well-formed, including the signatures.

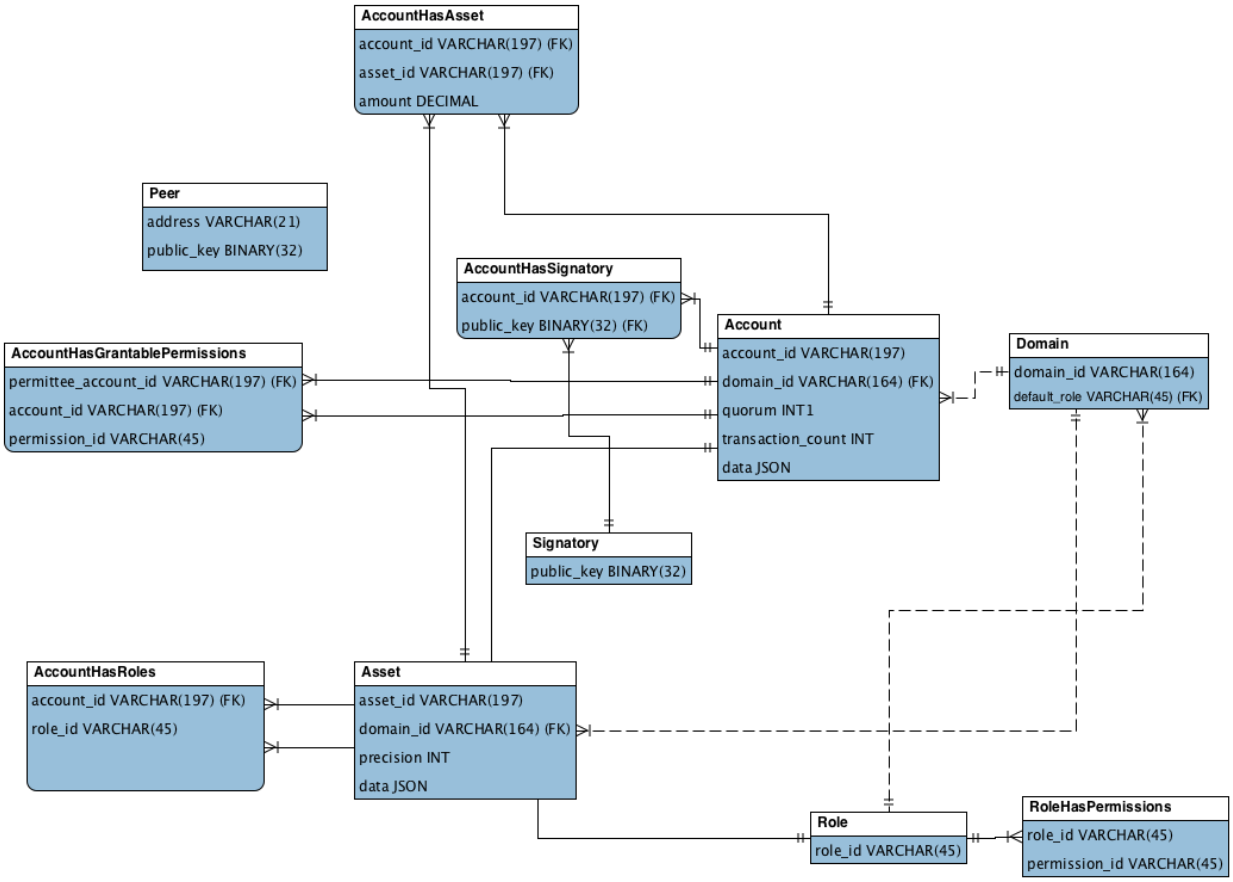
Stateful Validation

Performed in *Verified Proposal Creator*. Validates against World State View.

4.1.19 Entity-relationship model

Each Hyperledger Iroha peer has a state, called "World State View", which is represented by a set of entities and relations among them. To explain which entities exist in the system and what are the relations, this sections includes ER diagram and an explanation of its components.

ER diagram



Peer

- address — network address and internal port, is used for synchronization, consensus, and communication with the ordering service
- public_key — key, which will be used for signing blocks during consensus process

Asset

- asset_id — identifier of asset, formatted as asset_name#domain_id
- domain_id — identifier of domain, where the asset was created, references existing domain
- precision — size of fractional part
- data — JSON with arbitrary structure of asset description

Signatory

- public_key — a public key

Domain

- domain_id — identifier of a domain
- default_role — a default role per user created in the domain, references existing role

Role

- role_id — identifier of role

RoleHasPermissions

- role_id — identifier of role, references existing role
- permission_id — an id of predefined role

Account

- account_id — identifier of account, formatted as account_name@domain_id
- domain_id — identifier of domain where the account was created, references existing domain
- quorum — number of signatories required for creation of valid transaction from this account
- transaction_count – counter of transactions created by this account
- data — key-value storage for any information, related to the account. Size is limited to 268435455 bytes (0x0FFFFFFF) (PostgreSQL JSONB field).

AccountHasSignatory

- account_id — identifier of account, references existing account
- public_key — a public key (which is also called signatory), references existing signatory

AccountHasAsset

- account_id — identifier of account, references existing account
- asset_id — identifier of asset, references existing asset
- amount — an amount of the asset, belonging to the account

AccountHasRoles

- account_id — identifier of account, references existing account
- role_id — identifier of role, references existing role

AccountHasGrantablePermissions

- account_id — identifier of account, references existing account. This account gives grantable permission to perform operation over itself to permittee.
- permittee_account_id — identifier of account, references existing account. This account is given permission to perform operation over account_id.
- permission_id — identifier of grantable_permission

CHAPTER 5

Architecture

5.1 Iroha Execution Model

5.1.1 Introduction

This document describes threads and their interaction inside Iroha daemon. Overall configuration is presented and each thread's responsibilities are described in detail.

This is done to provide understanding in concurrency model of Iroha as well as to find any incorrect behavior in threading model.

5.1.2 On Thread Management

Iroha daemon avoids using raw threads provided by C++ standard library. Instead it relies on two distinct runtimes - grpc and rxcpp. They use threads internally to provide necessary functionality. In this document we treat these libraries as black boxes, meaning we are only stating the fact that the thread from a specific runtime is used, without going into details of where this thread came from.

The only place where a raw thread is used is AsyncGrpcClient implementation, which spawns a separate thread to listen to grpc replies.

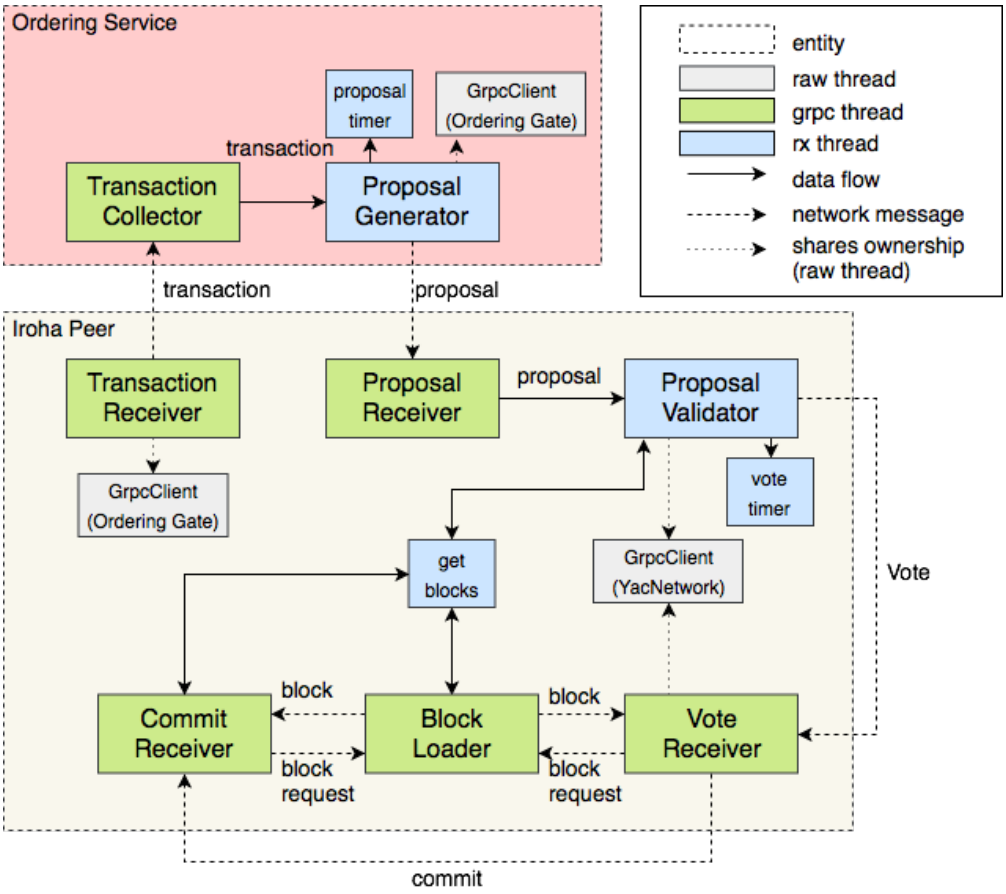
5.1.3 Multisignature Transaction Support

Everything described below is done with MST support turned off.

5.1.4 Transactions Pipeline

Transaction Diagram

The diagram below represents thread configuration when executing transactions on Iroha network.



Each block is a separate thread, and data flow is represented by arrows.

Detailed Description

Transaction Receiver

Used components:

- Torii (tx processor)
- Peer Communication Service
- Ordering Gate
- This is a grpc thread which handles transactions sent by users.

Goal is to pass the transaction to the ordering service.

Transaction Collector

Used components:

- Ordering Service Transport
- Ordering Service
- This grpc thread receives transactions from peers and puts them in a queue.

Queue is processed by Proposal Generator

Proposal Generator

Used components:

- Ordering Service Transport
- Ordering Service
- This is an rx thread which is triggered by two events: transactions added to the queue and proposal timeout.

When queue is big enough or timeout event occurs, it starts proposal generation.

Proposal Timer

To prevent transactions from hanging indefinitely, special timer is started, which works in a separate thread. As soon as timer goes off, proposal is formed from all transactions currently in the queue.

Proposal Receiver

Used components:

- Ordering Service Transport
- Ordering Gate
- This thread receives proposals from ordering service and puts them in a proposal queue to be validated.

Proposal Validator

Used components:

- Ordering Gate
- Simulator
- Stateful Validator
- Yac (gate, order, network)

This thread takes proposal from the proposal queue and sends it for a validation. When proposal is validated we obtain verified proposal. Vote for this proposal is broadcasted to the other peers.

Vote Timer

To optimize vote propagation among peers, we need to wait before sending vote to the next peer in the list. This is done to allow first peer to collect necessary votes to issue a commit.

Vote Receiver

Used components:

- Yac
- Yac Block Storage
- Vote Receiver is a thread which listens for incoming votes.

When enough votes were received, commit message is formed and broadcasted to all the peers.

Block Loader

Used components:

- Block Loader Service
- Block Loader
- This thread listens for requests to load blocks from the peer.

It is called when we receive commit message and want to retrieve commited block.

Commit Receiver

Used components:

- Yac
- Commit receiver is a thread which listens for incoming commits, verifies them and passes to the pipeline to be applied.

Commit Application

Used components:

- Synchronizer
- Chain Validator
- Ametsuchi

Commit is applied in a thread in which commit is obtained. If the peer creates commit, it is applied in Vote Receiver. If the peer receives commit, Commit Receiver applies it. Application of the commit begins with the retrieval of the block which is to be commited, then it is passed to the storage, which executes all transactions in the block.

Get Blocks

In a lot of cases GetTopBlocks function is called on a separate thread, it can be a different thread each time, and this is not really needed. Consider this a bug. List of components which call getTopBlocks in a separate thread:

- Simulator
- Storage
- Block Loader

GRPC Client

Only raw thread, it listens for replies from grpc calls. Each component which sends grpc calls uses this client, and thus occupies a thread. Transaction Receiver sends transactions to ordering service and owns its own client (ordering gate transport). Vote Receiver sends votes, and Proposal Validator sends commits. Both these actions are done via YacNetwork, so both threads share ownership of YacNetwork client.

5.1.5 Query Pipeline

Query execution in iroha is a lot simpler than transaction pipeline. When request arrives from the network it is handled by one of the grpc threads, which synchronously retrieves needed information from storage. Everything is done in a single thread.

5.2 Iroha Shared Objects Description

This document describes access to shared objects in a multithreaded scenario, whether or not is properly synchronized.

5.2.1 Document Structure

Each component in this document has its own diagram describing all members, and whether or not access to them is synchronized or not:

- Synchronized
- Not Synchronized

5.2.2 Query Service

Query Service
Cache
Query Processor

Query service is a grpc endpoint for queries from clients. It has cache, from which it receives responses. Access to the cache is protected by an internal mutex. Query processor actually fetches data from the storage. It is not synchronized here since all synchronization is internal.

5.2.3 Query Processor

Query Processor
Query Response Observable
Block Query Observable
Storage

5.2.4 Transaction Service

Transaction Service
Response Cache
Transaction Processor
Storage

Transaction Service uses response cache to send transaction statuses. Also, transaction processor is synchronized in status streaming.

5.2.5 Transaction Processor

Transaction Processor
Peer Communication Service
Transaction Status Notifier
Proposal Set
Candidate Set

5.2.6 Simulator

Simulator
Proposal Notifier
Block Notifier
Stateful Validator
Temporary Factory
Block Query

5.2.7 Synchronizer

Synchronizer
Chain Validator
Mutable Factory
Block Loader

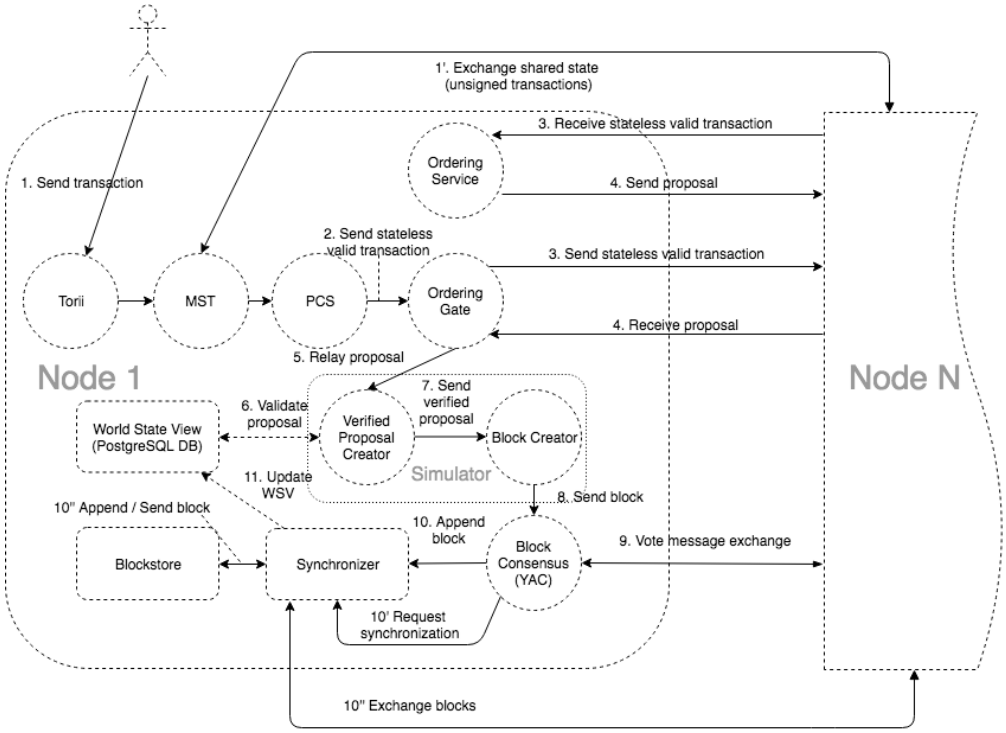
5.2.8 Yac Gate

Yac Gate
Hash Gate
Peer Orderer
Hash Provider
Block Creator
Block Loader

5.2.9 Yac

Yac
Yac Vote Storage
Yac Network

HL Iroha network consists of several essential components that provide the communication between the nodes. You can learn about them below.



5.3 Torii

Entry point for clients. Uses gRPC as a transport. In order to interact with Iroha anyone can use gRPC endpoints, described in Commands and Queries sections, or use client libraries.

5.4 MST Processor

Multisignature Transactions Processor

It is an internal gRPC service that sends and receives messages from other peers through Gossip protocol. Its mission is to send out multisignature transactions that have not received enough signatures to reach the quorum until it is reached.

5.5 Peer Communication Service

Internal component of Iroha - an intermediary that transmits transaction from *Torii* through *MstProcessor* to *Ordering Gate*. The main goal of PCS is to hide the complexity of interaction with consensus implementation.

5.6 Ordering Gate

It is an internal Iroha component (gRPC client) that relays transactions from *Peer Communication Service* to *Ordering Service*. Ordering Gate recieves proposals (potential blocks in the chain) from Ordering Service and sends them to *Simulator* for stateful validation. It also requests proposal from the Ordering Service based on the consensus round.

5.7 Ordering Service

Internal Iroha component (gRPC server) that receives messages from other peers and combines several transactions that have been passed stateless validation into a proposal. Each node has its own ordering service. Proposal creation could be triggered by one of the following events:

1. Time limit dedicated to transactions collection has expired.
2. Ordering service has received the maximum amount of transactions allowed for a single proposal.

Both parameters (timeout and maximum size of proposal) are configurable (check environment-specific parameters page).

A common precondition for both triggers is that at least one transaction should reach the ordering service. Otherwise, no proposal will be formed.

Ordering service also performs preliminary validation of the proposals (e.g. clearing out statelessly rejected transactions from the proposal).

5.8 Verified Proposal Creator

Internal Iroha component that performs stateful validation of transactions contained in received proposal from the *Ordering Service*. On the basis of transactions that have passed stateful validation **verified proposal** will be created and passed to *Block Creator*. All the transactions that have not passed stateful validation will be dropped and not included in a verified proposal.

5.9 Block Creator

System component that forms a block from a set of transactions that have passed stateless and stateful validation for further propagation to *consensus*.

Block creator, together with the *Verified Proposal Creator* form a component called Simulator.

5.10 Block Consensus (YAC)

Consensus, as a component

Consensus is the heart of the blockchain - it preserves a consistent state among the peers within a peer network. Iroha uses own consensus algorithm called Yet Another Consensus (aka YAC).

You can check out a video where HL Iroha maintainer thoroughly explains the principles of consensus and YAC in particular here.

Distinctive features of YAC algorithm are its scalability, performance and Crash fault tolerance.

To ensure consistency in the network, if there are missing blocks, they will be downloaded from another peer via *Synchronizer*. Committed blocks are stored in *Ametsuchi* block storage.

For general definition of the consensus, please check this link.

5.11 Synchronizer

Is a part of *consensus*. Adds missing blocks to peers' chains (downloads them from other peers to preserve consistency).

5.12 Ametsuchi Blockstore

Iroha storage component, which stores blocks and a state generated from blocks, called *World State View*. There is no way for the client to directly interact with Ametsuchi.

5.13 World State View

WSV reflects the current state of the system, can be considered as a snapshot. For example, WSV holds information about an amount of assets that an account has at the moment but does not contain any info history of transaction flow.

CHAPTER 6

Guides and how-tos

6.1 Building Iroha

In this guide we will learn how to install all dependencies, required to build Iroha and how to actually build it.

There will be 3 steps:

1. Installing environment prerequisites
2. Installing Iroha dependencies (will be performed automatically for Docker)
3. Building Iroha

Note: You don't need to build Iroha to start using it. Instead, you can download prepared Docker image from the Hub, this process explained in details in the *Getting Started* page of this documentation.

6.1.1 Prerequisites

In order to successfully build Iroha, we need to configure the environment. There are several ways to do it and we will describe all of them.

Currently, we support Unix-like systems (we are basically targeting popular Linux distros and MacOS). If you happen to have Windows or you don't want to spend time installing all dependencies you might want to consider using Docker environment. Also, Windows users might consider using WSL

Technically Iroha can be built under Windows natively in experimental mode. This guide covers that way too. All the stages related to native Windows build are separated from the main flow due to its significant differences.

Please choose your preferred platform below for a quick access:

- *Docker*
- *Linux*
- *MacOS*

- *Windows*

Hint: Having troubles? Check FAQ section or communicate to us directly, in case you were stuck on something. We don't expect this to happen, but some issues with an environment are possible.

Docker

First of all, you need to install `docker` and `docker-compose`. You can read how to install it on the Docker's website

---

Note: Please, use the latest available docker daemon and docker-compose.

Then you should clone the Iroha repository to the directory of your choice:

```
git clone -b master https://github.com/hyperledger/iroha --depth=1
```

Hint: `--depth=1` option allows us to download only latest commit and save some time and bandwidth. If you want to get a full commit history, you can omit this option.

When it is done, you need to run the development environment. Run the `scripts/run-iroha-dev.sh` script:

```
bash scripts/run-iroha-dev.sh
```

Hint: Please make sure that Docker is running before executing the script. MacOS users could find a Docker icon in system tray, Linux users can use `systemctl start docker`

After you execute this script, the following things will happen:

#. The script will check whether you have containers with Iroha already running. Successful completion finishes with the new container shell.

1. The script will download `hyperledger/iroha:develop-build` and `postgres` images. `hyperledger/iroha:develop-build` image contains all development dependencies and is based on top of `ubuntu:18.04`. `postgres` image is required for starting and running Iroha.
2. Two containers are created and launched.

#. The user is attached to the interactive environment for development and testing with `iroha` folder mounted from the host machine. Iroha folder is mounted to `/opt/iroha` in Docker container.

Now your are ready to build Iroha! Please go directly to *Building Iroha* section.

Linux

To build Iroha, you will need the following packages:

`build-essential` `git` `ca-certificates` `tar` `ninja-build` `curl` `unzip` `cmake`

Use this code to install environment dependencies on Debian-based Linux distro.

```
apt-get update; \
apt-get -y --no-install-recommends install \
build-essential ninja-build \
git ca-certificates tar curl unzip cmake
```

Note: If you are willing to actively develop Iroha and to build shared libraries, please consider installing the latest release of CMake.

Now you are ready to *install Iroha dependencies*.

MacOS

If you want to build Iroha from scratch and actively develop it, please use the following code to install all environment dependencies with Homebrew:

```
xcode-select --install
brew install cmake ninja git gcc@7
```

Hint: To install the Homebrew itself please run

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/homebrew/install/
master/install)"
```

Now you are ready to *install Iroha dependencies*.

Windows

Note: All the listed commands are designed for building 64-bit version of Iroha.

Chocolatey Package Manager

First of all you need Chocolatey package manager installed. Please refer the guide for chocolatey installation.

Building the Toolset

Install CMake, Git, Microsoft compilers via chocolatey being in Administrative mode of command prompt:

```
choco install cmake git visualstudio2019-workload-vctools ninja
```

PostgreSQL is not a build dependency, but it is recommended to install it now for the testing later:

```
choco install postgresql
# Don't forget the password you set!
```

Now you are ready to *install Iroha dependencies*.

6.1.2 Installing dependencies with Vcpkg Dependency Manager

Currently we use Vcpkg as a dependency manager for all platforms - Linux, Windows and MacOS. We use a fixed version of Vcpkg to ensure the patches we need will work.

That stable version can only be found inside the Iroha repository, so we will need to clone Iroha. The whole process is pretty similar for all platforms but the exact commands are slightly different.

Linux and MacOS

Run in terminal:

```
git clone https://github.com/hyperledger/iroha.git
iroha/vcpkg/build_iroha_deps.sh
vcpkg/vcpkg integrate install
```

After the installation of vcpkg you will be provided with a CMake build parameter like `-DCMAKE_TOOLCHAIN_FILE=/path/to/vcpkg/scripts/buildsystems/vcpkg.cmake`. Save it somewhere for later use and move to *Building Iroha* section.

Windows

Execute from Power Shell:

```
git clone https://github.com/hyperledger/iroha.git
powershell -ExecutionPolicy ByPass -File .\iroha\.packer\win\scripts\vcpkg.ps1 .
↪\vcpkg .\iroha\vcpkg
```

After the installation of vcpkg you will be provided with a CMake build parameter like `-DCMAKE_TOOLCHAIN_FILE=C:/path/to/vcpkg/scripts/buildsystems/vcpkg.cmake`. Save it somewhere for later use and move to *Building Iroha* section.

Note: If you plan to build 32-bit version of Iroha - you will need to install all the mentioned librares above prefixed with `x86` term instead of `x64`.

6.1.3 Build Process

Cloning the Repository

This step is currently unnecessary since you have already cloned Iroha in the previous step. But if you want, you can clone the Iroha repository to the directory of your choice.

```
git clone -b master https://github.com/hyperledger/iroha
cd iroha
```

Hint: If you have installed the prerequisites with Docker, you don't need to clone Iroha again, because when you run `run-iroha-dev.sh` it attaches to Iroha source code folder. Feel free to edit source code files with your host environment and build it within docker container.

Building Iroha

To build Iroha, use these commands:

```
cmake -H. -Bbuild -DCMAKE_TOOLCHAIN_FILE=/path/to/vcpkg/scripts/buildsystems/vcpkg.
↪cmake -G "Ninja"
cmake --build build --target irohad -- -j<number of threads>
```

Note: On Docker the path to a toolchain file is `/opt/dependencies/scripts/buildsystems/vcpkg.cmake`. In other environment please use the path you have got in previous steps.

Number of threads will be defined differently depending on the platform: - On Linux: via `nproc`. - On MacOS: with `sysctl -n hw.ncpu`. - On Windows: use `echo %NUMBER_OF_PROCESSORS%`.

Note: When building on Windows do not execute this from the Power Shell. Better use x64 Native tools command prompt.

Now Iroha is built. Although, if you like, you can build it with additional parameters described below.

CMake Parameters

We use CMake to generate platform-dependent build files. It has numerous flags for configuring the final build. Note that besides the listed parameters cmake's variables can be useful as well. Also as long as this page can be deprecated (or just not complete) you can browse custom flags via `cmake -L`, `cmake-gui`, or `ccmake`.

Hint: You can specify parameters at the cmake configuring stage (e.g cmake -DTESTING=ON).

Main Parameters

Parameter	Possible values	Default	Description
TESTING	ON/OFF	ON	Enables or disables build of the tests
BENCHMARKING		OFF	Enables or disables build of the Google Benchmarks library
COVERAGE		OFF	Enables or disables lcov setting for code coverage generation

Packaging Specific Parameters

Parameter	Possible values	Default	Description
ENABLE_LIBS_PACKAGING	ON/OFF	ON	Enables or disables all types of packaging
PACKAGE_ZIP		OFF	Enables or disables zip packaging
PACKAGE_TGZ		OFF	Enables or disables tar.gz packaging
PACKAGE_RPM		OFF	Enables or disables rpm packaging
PACKAGE_DEB		OFF	Enables or disables deb packaging

Running Tests (optional)

After building Iroha, it is a good idea to run tests to check the operability of the daemon. You can run tests with this code:

```
cmake --build build --target test
```

Alternatively, you can run the following command in the `build` folder

```
cd build
ctest . --output-on-failure
```

Note: Some of the tests will fail without PostgreSQL storage running, so if you are not using `scripts/run-iroha-dev.sh` script please run Docker container or create a local connection with following parameters:

```
docker run --name some-postgres \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=mysecretpassword \
-p 5432:5432 \
-d postgres:9.5 \
-c 'max_prepared_transactions=100'
```

6.2 Configuration

In this section we will understand how to configure Iroha. Let's take a look at `example/config.sample`

```
{
  "block_store_path": "/tmp/block_store/",
  "torii_port": 50051,
  "internal_port": 10001,
  "pg_opt": "host=localhost port=5432 user=postgres password=mysecretpassword␣
↪dbname=iroha",
  "database": {
    "host": "localhost",
    "port": 5432,
    "user": "postgres",
    "password": "mysecretpassword",
    "working database": "iroha_data",
    "maintenance database": "postgres"
  }
  "max_proposal_size": 10,
  "proposal_delay": 5000,
  "vote_delay": 5000,
  "mst_enable" : false,
  "mst_expiration_time" : 1440,
  "mst_stale_threshold_milliseconds" : 1800000,
  "max_rounds_delay": 3000,
  "stale_stream_max_rounds": 2
}
```

As you can see, configuration file is a valid `json` structure. Let's go line-by-line and understand what every parameter means.

6.2.1 Deployment-specific parameters

- `block_store_path` sets path to the folder where blocks are stored.
- `torii_port` sets the port for external communications. Queries and transactions are sent here.
- `internal_port` sets the port for internal communications: ordering service, consensus and block loader.
- `database` (optional) is used to set the database configuration (see below)
- `pg_opt` (optional) is a deprecated way of setting credentials of PostgreSQL: hostname, port, username, password and database name. All data except the database name are mandatory. If database name is not provided, the default one gets used, which is `iroha_default`.
- `log` is an optional parameter controlling log output verbosity and format (see below).

> **Warning:** Configuration field `pg_opt` is deprecated, please use `database` section!
>
> The `database` section overrides `pg_opt` when both are provided in configuration.
>
> Both `pg_opt` and `database` fields are optional, but at least one must be specified.

The `database` section fields:

- `host` the host to use for PostgreSQL connection
- `port` the port to use for PostgreSQL connection
- `user` the user to use for PostgreSQL connection
- `password` the password to use for PostgreSQL connection
- `working database` is the name of database that will be used to store the world state view and optionally blocks.
- `maintenance database` is the name of databse that will be used to maintain the working database. For example, when iroha needs to create or drop its working database, it must use another database to connect to PostgreSQL.

6.2.2 Environment-specific parameters

- `max_proposal_size` is the maximum amount of transactions that can be in one proposal, and as a result in a single block as well. So, by changing this value you define the size of potential block. For a starter you can stick to `10`. However, we recommend to increase this number if you have a lot of transactions per second.
- `proposal_delay` is a timeout in milliseconds that a peer waits a response from the orderding service with a proposal.
- `vote_delay` is a waiting time in milliseconds before sending vote to the next peer. Optimal value depends heavily on the amount of Iroha peers in the network (higher amount of nodes requires longer `vote_delay`). We recommend to start with 100-1000 milliseconds.
- `mst_enable` enables or disables multisignature transaction network transport in Iroha. Note that MST engine always works for any peer even when the flag is set to `false`. The flag only allows sharing information about MST transactions among the peers.
- `mst_expiration_time` is an optional parameter specifying the time period in which a not fully signed transaction (or a batch) is considered expired (in minutes). The default value is 1440.

- `max_rounds_delay` is an optional parameter specifying the maximum delay between two consensus rounds (in milliseconds). The default value is 3000. When Iroha is idle, it gradually increases the delay to reduce CPU, network and logging load. However too long delay may be unwanted when first transactions arrive after a long idle time. This parameter allows users to find an optimal value in a tradeoff between resource consumption and the delay of getting back to work after an idle period.
- `mst_stale_threshold_milliseconds` is an optional parameter that sets the time after which the node that sent the MST transaction in the first place would try to send it again if no final status is received. It is needed in case, say, some nodes or network fail and restart or if there are network problems and other nodes do not receive the transaction. Default value is 1800000 (30 minutes). Please take into consideration that decreasing the value too much will lead to using processing resources constantly connecting the nodes. Increasing it too much will cause delays in receiving MST transactions in case of failed nodes.
- `stale_stream_max_rounds` is an optional parameter specifying the maximum amount of rounds to keep an open status stream while no status update is reported. The default value is 2. Increasing this value reduces the amount of times a client must reconnect to track a transaction if for some reason it is not updated with new rounds. However large values increase the average number of connected clients during each round.
- `"initial_peers` is an optional parameter specifying list of peers a node will use after startup instead of peers from genesis block. It could be useful when you add a new node to the network where the most of initial peers may become malicious. Peers list should be provided as a JSON array:

  ```
  "initial_peers" :  [{"address":"127.0.0.1:10001", "public_key":
  "bddd58404d1315e0eb27902c5d7c8eb0602c16238f005773df406bc191308929"}]
  ```

6.2.3 Logging

In Iroha logging can be adjusted as granularly as you want. Each component has its own logging configuration with properties inherited from its parent, able to be overridden through config file. This means all the component loggers are organized in a tree with a single root. The relevant section of the configuration file contains the overriding values:

```
"log": {
  "level": "info",
  "patterns": {
    "debug": "don't panic, it's %v.",
    "error": "MAMA MIA! %v!!!"
  },
  "children": {
    "KeysManager": {
      "level": "trace"
    },
    "Irohad": {
      "children": {
        "Storage": {
          "level": "trace",
          "patterns": {
            "debug": "thread %t: %v."
          }
        }
      }
    }
  }
}
```

Every part of this config section is optional.

- `level` sets the verbosity. Available values are (in decreasing verbosity order):

  - `trace` - print everything
  - `debug`
  - `info`
  - `warning`
  - `error`
  - `critical` - print only critical messages
- `patterns` controls the formatting of each log string for different verbosity levels. Each value overrides the less verbose levels too. So in the example above, the "don't panic" pattern also applies to info and warning levels, and the trace level pattern is the only one that is not initialized in the config (it will be set to default hardcoded value).
- `children` describes the overrides of child nodes. The keys are the names of the components, and the values have the same syntax and semantics as the root log configuration.

6.3 Deploying Iroha

Hyperledger Iroha can be deployed in different ways, depending on the perspective and the purpose. There can be either a single node deployed, or multiple nodes running in several containers on a local machine or spread across the network — so pick any case you need. This page describes different scenarios and is intended to act as a how-to guide for users, primarily trying out Iroha for the first time.

6.3.1 Running single instance

Generally, people want to run Iroha locally in order to try out the API and explore the capabilities. This can be done in local or container environment (Docker). We will explore both possible cases, but in order to simplify peer components deployment, *it is advised to have Docker installed on your machine*.

Local environment

By local environment, it is meant to have daemon process and Postgres deployed without any containers. This might be helpful in cases when messing up with Docker is not preferred — generally a quick exploration of the features.

Run postgres server

In order to run postgres server locally, you should check postgres website and follow their description. Generally, postgres server runs automatically when the system starts, but this should be checked in the configuration of the system.

Run iroha daemon (irohad)

There is a list of preconditions which you should meet before proceeding:

- Postgres server is up and running
- *irohad* Iroha daemon binary is built and accessible in your system
- The genesis block and configuration files were created

- Config file uses valid postgres connection settings
- A keypair for the peer is generated
- This is the first time you run the Iroha on this peer and you want to create new chain

Hint: Have you got something that is not the same as in the list of assumptions? Please, refer to the section below the document, titled as *Dealing with troubles*.

In case of valid assumptions, the only thing that remains is to launch the daemon process with following parameters:

Parameter	Meaning
config	configuration file, containing postgres connection and values to tune the system
genesis_block	initial block in the ledger
keypair_name	private and public key file names without file extension, used by peer to sign the blocks

Attention: Specifying a new genesis block using *–genesis_block* with blocks already present in ledger requires *–overwrite_ledger* flag to be set. The daemon will fail otherwise.

An example of shell command, running Iroha daemon is

```
irohad --config example/config.sample --genesis_block example/genesis.block --keypair_
↪name example/node0
```

Attention: If you have stopped the daemon and want to use existing chain — you should not pass the genesis block parameter.

Docker

In order to run Iroha peer as a single instance in Docker, you should pull the image for Iroha first:

```
docker pull hyperledger/iroha:latest
```

Hint: Use *latest* tag for latest stable release, and *develop* for latest development version

Then, you have to create an enviroment for the image to run without problems:

Create docker network

Containers for Postgres and Iroha should run in the same virtual network, in order to be available to each other. Create a network, by typing following command (you can use any name for the network, but in the example, we use *iroha-network* name):

```
docker network create iroha-network
```

Run Postgresql in a container

Similarly, run postgres server, attaching it to the network you have created before, and exposing ports for communication:

```
docker run --name some-postgres \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=mysecretpassword \
-p 5432:5432 \
--network=iroha-network \
-d postgres:9.5
```

Create volume for block storage

Before we run iroha daemon in the container, we should create persistent volume to store files, storing blocks for the chain. It is done via the following command:

```
docker volume create blockstore
```

Running iroha daemon in docker container

There is a list of assumptions which you should review before proceeding:

- Postgres server is running on the same docker network
- There is a folder, containing config file and keypair for a single node
- This is the first time you run the Iroha on this peer and you want to create new chain

If they are met, you can move forward with the following command:

```
docker run --name iroha \
# External port
-p 50051:50051 \
# Folder with configuration files
-v ~/Developer/iroha/example:/opt/iroha_data \
# Blockstore volume
-v blockstore:/tmp/block_store \
# Postgres settings
-e POSTGRES_HOST='some-postgres' \
-e POSTGRES_PORT='5432' \
-e POSTGRES_PASSWORD='mysecretpassword' \
-e POSTGRES_USER='postgres' \
# Node keypair name
-e KEY='node0' \
# Docker network name
--network=iroha-network \
hyperledger/iroha:latest
```

6.3.2 Running multiple instances (peer network)

In order to set up a peer network, one should follow routines, described in this section. In this version, we support manual deployment and automated by Ansible Playbook. Choose an option, that meets your security criteria and other needs.

Manually

By manual deployment, we mean that Iroha peer network is set up without automated assistance. It is similar to the process of running a single local instance, although the difference is the genesis block includes more than a single peer. In order to form a block, which includes more than a single peer, or requires customization for your needs, please take a look at *Dealing with troubles* section.

Automated

Follow this guide

6.3.3 Dealing with troubles

—"Please, help me, because I. . . "

Do not have Iroha daemon binary

You can build Iroha daemon binary from sources. You can get binaries here

Do not have a config file

Check how to create a configuration file by following this link

Do not have a genesis block

Create genesis block by generating it via *iroha-cli* or manually, using the example and checking out permissions

Do not have a keypair for a peer

In order to create a keypair for an account or a peer, use iroha-cli binary by passing the name of the peer with *–new_account* option. For example:

```
./iroha-cli --account_name newuser@test --new_account
```

6.4 Client Libraries

6.4.1 Java Library

Client library of Iroha written completely in Java 8, which includes:

- SDK to work with Iroha API
- async wrapper over Iroha API
- *testcontainers* wrapper for convenient integration testing with Iroha
- examples in Java and Groovy

Both options are described in the following sections. Please check readme file in project's repo.

How to use

- JitPack
- GitHub

Example code

```
import iroha.protocol.BlockOuterClass;
import iroha.protocol.Primitive.RolePermission;
import java.math.BigDecimal;
import java.security.KeyPair;
import java.util.Arrays;
import jp.co.soramitsu.crypto.ed25519.Ed25519Sha3;
import jp.co.soramitsu.iroha.testcontainers.IrohaContainer;
import jp.co.soramitsu.iroha.testcontainers.PeerConfig;
import jp.co.soramitsu.iroha.testcontainers.detail.GenesisBlockBuilder;
import lombok.val;

public class Example1 {

  private static final String bankDomain = "bank";
  private static final String userRole = "user";
  private static final String usdName = "usd";

  private static final Ed25519Sha3 crypto = new Ed25519Sha3();

  private static final KeyPair peerKeypair = crypto.generateKeypair();

  private static final KeyPair useraKeypair = crypto.generateKeypair();
  private static final KeyPair userbKeypair = crypto.generateKeypair();

  private static String user(String name) {
    return String.format("%s@%s", name, bankDomain);
  }

  private static final String usd = String.format("%s#%s", usdName, bankDomain);

  /**
   * <pre>
   * Our initial state cosists of:
   * - domain "bank", with default role "user" - can transfer assets and can query␣
↪their amount
   * - asset usd#bank with precision 2
   * - user_a@bank, which has 100 usd
   * - user_b@bank, which has 0 usd
   * </pre>
   */
  private static BlockOuterClass.Block getGenesisBlock() {
    return new GenesisBlockBuilder()
        // first transaction
        .addTransaction(
            // transactions in genesis block can have no creator
            Transaction.builder(null)
                // by default peer is listening on port 10001
                .addPeer("0.0.0.0:10001", peerKeypair.getPublic())
                // create default "user" role
```

```
                .createRole(userRole,
                    Arrays.asList(
                        RolePermission.can_transfer,
                        RolePermission.can_get_my_acc_ast,
                        RolePermission.can_get_my_txs,
                        RolePermission.can_receive
                    )
                )
                .createDomain(bankDomain, userRole)
                // create user A
                .createAccount("user_a", bankDomain, useraKeypair.getPublic())
                // create user B
                .createAccount("user_b", bankDomain, userbKeypair.getPublic())
                // create usd#bank with precision 2
                .createAsset(usdName, bankDomain, 2)
                // transactions in genesis block can be unsigned
                .build() // returns ipj model Transaction
                .build() // returns unsigned protobuf Transaction
        )
        // we want to increase user_a balance by 100 usd
        .addTransaction(
            Transaction.builder(user("user_a"))
                .addAssetQuantity(usd, new BigDecimal("100"))
                .build()
                .build()
        )
        .build();
}

public static PeerConfig getPeerConfig() {
  PeerConfig config = PeerConfig.builder()
      .genesisBlock(getGenesisBlock())
      .build();

  // don't forget to add peer keypair to config
  config.withPeerKeyPair(peerKeypair);

  return config;
}

/**
 * Custom facade over GRPC Query
 */
public static int getBalance(IrohaAPI api, String userId, KeyPair keyPair) {
  // build protobuf query, sign it
  val q = Query.builder(userId, 1)
      .getAccountAssets(userId)
      .buildSigned(keyPair);

  // execute query, get response
  val res = api.query(q);

  // get list of assets from our response
  val assets = res.getAccountAssetsResponse().getAccountAssetsList();

  // find usd asset
  val assetUsdOptional = assets
      .stream()
```

```
        .filter(a -> a.getAssetId().equals(usd))
        .findFirst();

    // numbers are small, so we use int here for simplicity
    return assetUsdOptional
        .map(a -> Integer.parseInt(a.getBalance()))
        .orElse(0);
  }

  public static void main(String[] args) {
    // for simplicity, we will create Iroha peer in place
    IrohaContainer iroha = new IrohaContainer()
        .withPeerConfig(getPeerConfig());

    // start the peer. blocking call
    iroha.start();

    // create API wrapper
    IrohaAPI api = new IrohaAPI(iroha.getToriiAddress());

    // transfer 100 usd from user_a to user_b
    val tx = Transaction.builder("user_a@bank")
        .transferAsset("user_a@bank", "user_b@bank", usd, "For pizza", "10")
        .sign(useraKeypair)
        .build();

    // create transaction observer
    // here you can specify any kind of handlers on transaction statuses
    val observer = TransactionStatusObserver.builder()
        // executed when stateless or stateful validation is failed
        .onTransactionFailed(t -> System.out.println(String.format(
            "transaction %s failed with msg: %s",
            t.getTxHash(),
            t.getErrOrCmdName()
        )))
        // executed when got any exception in handlers or grpc
        .onError(e -> System.out.println("Failed with exception: " + e))
        // executed when we receive "committed" status
        .onTransactionCommitted((t) -> System.out.println("Committed :)"))
        // executed when transfer is complete (failed or succeed) and observable is␣
↪closed
        .onComplete(() -> System.out.println("Complete"))
        .build();

    // blocking send.
    // use .subscribe() for async sending
    api.transaction(tx)
        .blockingSubscribe(observer);

    /// now lets query balances
    val balanceUserA = getBalance(api, user("user_a"), useraKeypair);
    val balanceUserB = getBalance(api, user("user_b"), userbKeypair);

    // ensure we got correct balances
    assert balanceUserA == 90;
    assert balanceUserB == 10;
  }
}
```

6.4.2 Javascript library

This library provides functions which will help you to interact with Hyperledger Iroha from your JS program.

Installation

Via npm

```
$ npm i iroha-helpers
```

Via yarn

```
$ yarn add iroha-helpers
```

Commands

For usage of any command you need to provide `commandOptions` as a first argument.

```
const commandOptions = {
  privateKeys: ['f101537e319568c765b2cc89698325604991dca57b9716b58016b253506cab70'], /
↪/ Array of private keys in hex format
  creatorAccountId: '', // Account id, ex. admin@test
  quorum: 1,
  commandService: null
}
```

As second argument you need to provide object that contains properties for required command.

```
// Example usage of setAccountDetail

const commandService = new CommandService_v1Client(
  '127.0.0.1:50051',
  grpc.credentials.createInsecure()
)

const adminPriv = 'f101537e319568c765b2cc89698325604991dca57b9716b58016b253506cab70'

commands.setAccountDetail({
  privateKeys: [adminPriv],
  creatorAccountId: 'admin@test',
  quorum: 1,
  commandService
}, {
  accountId: 'admin@test',
  key: 'jason',
  value: 'statham'
})
```

Queries

For usage of any query you need to provide `queryOptions` as a first argument.

```
const queryOptions = {
  privateKey: 'f101537e319568c765b2cc89698325604991dca57b9716b58016b253506cab70', //␣
↪Private key in hex format
  creatorAccountId: '', // Account id, ex. admin@test
  queryService: null
}
```

As second argument you need to provide object that contains properties for required query.

```
// Example usage of getAccountDetail

const queryService = new QueryService_v1Client(
  '127.0.0.1:50051',
  grpc.credentials.createInsecure()
)

const adminPriv = 'f101537e319568c765b2cc89698325604991dca57b9716b58016b253506cab70'

queries.getAccountDetail({
  privateKey: adminPriv,
  creatorAccountId: 'admin@test',
  queryService
}, {
  accountId: 'admin@test'
})
```

Example code

```
import grpc from 'grpc'
import {
  QueryService_v1Client,
  CommandService_v1Client
} from '../iroha-helpers/lib/proto/endpoint_grpc_pb'
import { commands, queries } from 'iroha-helpers'

const IROHA_ADDRESS = 'localhost:50051'
const adminPriv =
  'f101537e319568c765b2cc89698325604991dca57b9716b58016b253506cab70'

const commandService = new CommandService_v1Client(
  IROHA_ADDRESS,
  grpc.credentials.createInsecure()
)

const queryService = new QueryService_v1Client(
  IROHA_ADDRESS,
  grpc.credentials.createInsecure()
)

Promise.all([
  commands.setAccountDetail({
    privateKeys: [adminPriv],
    creatorAccountId: 'admin@test',
    quorum: 1,
    commandService
  }, {
```

```
    accountId: 'admin@test',
    key: 'jason',
    value: 'statham'
  }),
  queries.getAccountDetail({
    privateKey: adminPriv,
    creatorAccountId: 'admin@test',
    queryService
  }, {
    accountId: 'admin@test'
  })
])
  .then(a => console.log(a))
  .catch(e => console.error(e))
```

6.4.3 Python Library

Where to Get

A supported python library for Iroha is available at its own Hyperledger iroha-python repo. Python 3+ is supported.

You can also install Python library via pip:

```
pip install iroha
```

Example Code

```
from iroha import Iroha, IrohaCrypto, IrohaGrpc

iroha = Iroha('alice@test')
net = IrohaGrpc('127.0.0.1:50051')

alice_key = IrohaCrypto.private_key()
alice_tx = iroha.transaction(
 [iroha.command(
     'TransferAsset',
     src_account_id='alice@test',
     dest_account_id='bob@test',
     asset_id='bitcoin#test',
     description='test',
     amount='1'
 )]
)

IrohaCrypto.sign_transaction(alice_tx, alice_key)
    net.send_tx(alice_tx)

for status in net.tx_status_stream(alice_tx):
    print(status)
```

6.4.4 iOS Swift Library

The library was created to provide convenient interface for iOS applications to communicate with Iroha blockchain including sending transactions/query, streaming transaction statuses and block commits.

Where to get

Iroha iOS library is available through CocoaPods. To install it, simply add the following line to your Podfile:

```
pod 'IrohaCommunication'
```

Also you can download the source code for the library in its repo

How to use

For new Iroha users we recommend to checkout iOS example project. It tries to establish connection with Iroha peer which should be also run locally on your computer to create new account and send some asset quantity to it. To run the project, please, go through steps below:

- Follow instructions from Iroha documentation to setup and run iroha peer in Docker container.
- Clone iroha-ios repository.
- cd Example directory and run pod install.
- Open IrohaCommunication.xcworkspace in XCode
- Build and Run IrohaExample target.
- Consider logs to see if the scenario completed successfully.

Feel free to experiment with example project and don't hesitate to ask any questions in Rocket.Chat.

6.5 Installing Dependencies

This page contains references and guides about installation of various tools you may need during build of different targets of Iroha project.

Note: Please note that most likely you do not need to install all the listed tools. Some of them are required only for building specific versions of Iroha Client Library.

6.5.1 Automake

Installation on Ubuntu

```
sudo apt install automake
automake --version
# automake (GNU automake) 1.15
```

6.5.2 Bison

Installation on Ubuntu

```
sudo apt install bison
bison --version
# bison (GNU Bison) 3.0.4
```

6.5.3 CMake

Minimum required version is 3.11.4, but we recommend to install the latest available version (3.12.0 at the moment).

Installation on Ubuntu

Since Ubuntu repositories contain unsuitable version of CMake, you need to install the new one manually. Here is how to build and install CMake from sources.

```
wget https://cmake.org/files/v3.11/cmake-3.11.4.tar.gz
tar -xvzf cmake-3.11.4.tar.gz
cd cmake-3.11.4/
./configure
make
sudo make install
cmake --version
# cmake version 3.11.4
```

Installation on macOS

```
brew install cmake
cmake --version
# cmake version 3.12.1
```

6.5.4 Git

Installation on Ubuntu

```
sudo apt install git
git --version
# git version 2.7.4
```

6.5.5 Python

Installation on Ubuntu

For Python 2:

```
sudo apt install python-dev
python --version
# Python 2.7.12
```

For Python 3:

```
sudo apt install python3-dev
python3 --version
# Python 3.5.2
```

Installation on macOS

For Python 2:

```
brew install python
python --version
# Python 2.7.12
```

For Python 3:

```
brew install python3
python3 --version
# Python 3.5.2
```

6.5.6 PIP

Installation on Ubuntu

For Python 2:

```
sudo apt install python-pip
pip --version
# pip 8.1.1 from /usr/lib/python2.7/dist-packages (python 2.7)
```

For Python 3:

```
sudo apt install python3-pip
pip3 --version
# pip 8.1.1 from /usr/lib/python3/dist-packages (python 3.5)
```

Installation on macOS

For Python 2:

```
sudo easy_install pip
pip --version
# pip 9.0.3 from /usr/local/lib/python2.7/site-packages (python 2.7)
```

For Python 3:

```
wget https://bootstrap.pypa.io/get-pip.py
sudo python3 get-pip.py
python3 -m pip --version
# pip 9.0.3 from /usr/local/Cellar/python/3.6.4_4/Frameworks/Python.framework/
↪Versions/3.6/lib/python3.6/site-packages (python 3.6)
```

6.5.7 Boost

Installation on Ubuntu

```
git clone https://github.com/boostorg/boost /tmp/boost;
(cd /tmp/boost ; git submodule update --init --recursive);
(cd /tmp/boost ; /tmp/boost/bootstrap.sh);
(cd /tmp/boost ; /tmp/boost/b2 headers);
(cd /tmp/boost ; /tmp/boost/b2 cxxflags="-std=c++14" install);
ldconfig;
rm -rf /tmp/boost
```

Installation on macOS

```
brew install boost
```

6.5.8 Protobuf

Installation on macOS

```
CMAKE_BUILD_TYPE="Release"
git clone https://github.com/google/protobuf /tmp/protobuf;
(cd /tmp/protobuf ; git checkout 106ffc04be1abf3ff3399f54ccf149815b287dd9);
cmake \
    -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE} \
    -Dprotobuf_BUILD_TESTS=OFF \
    -Dprotobuf_BUILD_SHARED_LIBS=ON \
    -H/tmp/protobuf/cmake \
    -B/tmp/protobuf/.build;
cmake --build /tmp/protobuf/.build --target install;
ldconfig;
rm -rf /tmp/protobuf
```

6.6 Deploying Iroha on Kubernetes cluster

> **Warning:** Some parts of this guide are deprecated. Proceed at your own discretion.

By following this guide you will be able to deploy a Kubernetes cluster from scratch on AWS cloud using Terraform and Kubespray, and deploy a network of Iroha nodes on it.

6.6.1 Prerequisites

- machine running Linux (tested on Ubuntu 16.04) or MacOS
- Python 3.3+
- boto3
- Ansible 2.4+
- *ed25519-cli* utility for key generation. Statically linked binary (for x86_64 platform) can be found in deploy/ansible/playbooks/iroha-k8s/scripts directory. You may need to compile it yourself.

You do not need the items below if you already have a working Kubernetes (k8s) cluster. You can skip to *Generating Iroha configs* chapter.

- Terraform 0.11.8+
- AWS account for deploying a k8s cluster on EC2

6.6.2 Preparation

You need to obtain AWS key for managing resources. We recommend to create a separate IAM user for that. Go to your AWS console, head to "My Security Credentials" menu and create a user in "Users" section. Assign "AmazonEC2FullAccess" and "AmazonVPCFullAccess" policies to that user. Click "Create access key" on Security credentials tab. Take a note for values of Access key ID and Secret key. Set these values as environment variables in your console:

```
export AWS_ACCESS_KEY_ID='<The value of Access key ID>'
export AWS_SECRET_ACCESS_KEY='<The value of Secret key>'
```

Checkout the source tree from Github:

```
git clone https://github.com/hyperledger/iroha && cd iroha
```

6.6.3 Setting up cloud infrastructure

We use Hashicorp's Terraform infrastructure management tool for automated deployment of AWS EC2 nodes in multiple regions. Kubespray Ansible module is used for setting up a production-grade k8s cluster.

Terraform module creates 3 AWS instances in 3 different regions: eu-west-1, eu-west-2, eu-west-3 by default. Instance type is *c5.large*. There is a separate VPC created in every region. All created VPCs are then connected using VPC peering connection. That is to create a seamless network for k8s cluster.

There are several configurable options: number of nodes in each region and its role in k8s cluster (kube-master or kube-node). They can be set either in *variables.tf* file or via environment variables (using the same variable name but prefixed with TF_VAR. See more in Terraform docs). More options can be configured by tuning parameters in module's *variables.tf* file.

You must set up SSH key in *deploy/tf/k8s/variables.tf* as well. Replace public key with your own. It will added on each created EC2 instance.

Navigate to *deploy/tf/k8s* directory. Terraform needs to download required modules first:

```
pushd deploy/tf/k8s && terraform init
```

Then run module execution:

```
terraform apply && popd
```

Review the execution plan and type *yes* to approve. Upon completion you should see an output similar to this:

```
Apply complete! Resources: 39 added, 0 changed, 0 destroyed.
```

We are now ready to deploy k8s cluster. Wait a couple of minutes before instances are initialized.

6.6.4 Setting up k8s cluster

There is an Ansible role for setting up k8s cluster. It is an external module called Kubespray. It is stored as a submodule in Hyperledger Iroha repository. This means it needs to be initialized first:

```
git submodule init && git submodule update
```

This command will download Kubespray from master repository.

Install required dependencies:

```
pip3 install -r deploy/ansible/kubespray/requirements.txt
```

Proceed to actual cluster deployment. Make sure you replaced *key-file* parameter with an actual path to SSH private key that was used previously during Terraform configuration. *REGIONS* variable corresponds to default list of regions used on a previous step. Modify it accordingly in case you added or removed any. Inventory file is a Python script that returns Ansible-compatible list of hosts filtered by tag.

```
pushd deploy/ansible && REGIONS="eu-west-1,eu-west-2,eu-west-3" VPC_VISIBILITY="public
↪" ansible-playbook -u ubuntu -b --ssh-extra-args="-o IdentitiesOnly=yes" --key-file=
↪<PATH_TO_SSH_KEY> -i inventory/kubespray-aws-inventory.py kubespray/cluster.yml
popd
```

Upon successful completion you will have working k8s cluster.

6.6.5 Generating Iroha configs

In order for Iroha to work properly it requires to generate a key pair for each node, genesis block and configuration file. This is usually a tedious and error-prone procedure, especially for a large number of nodes. We automated it with Ansible role. You can skip to *Deploying Iroha on the cluster* chapter if you want to quick start using default configs for k8s cluster with 4 Iroha replicas.

Generate configuration files for *N* Iroha nodes. *replicas* variable controls the number of *N*:

```
pushd deploy/ansible && ansible-playbook -e 'replicas=7' playbooks/iroha-k8s/iroha-
↪deploy.yml
popd
```

You should find files created in *deploy/ansible/roles/iroha-k8s/files/conf*.

6.6.6 Deploying Iroha on the cluster

Make sure you have configuration files in *deploy/ansible/roles/iroha-k8s/files*. Specifically, non-empty *conf* directory and *k8s-iroha.yaml* file.

There are two options for managing k8s cluster: logging into either of master node and executing commands there or configure remote management. We will cover the second option here as the first one is trivial.

In case you set up cluster using Kubespray, you can find *admin.conf* file on either of master node in */etc/kubernetes* directory. Copy this file on the control machine (the one you will be running *kubectl* command from). Make sure *server* parameter in this file points to external IP address or DNS name of a master node. Usually, there is a private IP address of the node (in case of AWS). Make sure *kubectl* utility is installed (check out the docs for instructions).

Replace the default *kubectl* configuration:

```
export KUBECONFIG=<PATH_TO_admin.conf>
```

We can now control the remote k8s cluster

k8s-iroha.yaml pod specification file requires the creation of a *config-map* first. This is a special resource that is mounted in the init container of each pod, and contains the configuration and genesis block files required to run Iroha.

```
kubectl create configmap iroha-config --from-file=deploy/ansible/roles/iroha-k8s/
↪files/conf/
```

Each peer will have their public and private keys stored in a Kubernetes secret which is mounted in the init container and copied over for Iroha to use. Peers will only be able read their assigned secret when running Iroha.

```
kubectl create -f deploy/ansible/roles/iroha-k8s/files/k8s-peer-keys.yaml
```

Deploy Iroha network pod specification:

```
kubectl create -f deploy/ansible/roles/iroha-k8s/files/k8s-iroha.yaml
```

Wait a moment before each node downloads and starts Docker containers. Executing *kubectl get pods* command should eventually return a list of deployed pods each in *Running* state.

Hint: Pods do not expose ports externally. You need to connect to Iroha instance by its hostname (iroha-0, iroha-1, etc). For that you have to have a running pod in the same network.

6.7 Iroha installation security tips

This guide is intended to secure Iroha installation. Most of the steps from this guide may seem obvious but it helps to avoid possible security problems in the future.

6.7.1 Physical security

In case the servers are located locally (physically accessible), a number of security measures have to be applied. Skip these steps if cloud hosting is used.

Establish organisational policy and/or access control system such that only authorized personnel has access to the server room. Next, set BIOS/firmware password and configure boot order to prevent unauthorized booting from alternate media. Make sure the bootloader is password protected if there is such a functionality. Also, it is good to have a CCTV monitoring in place.

6.7.2 Deployment

First, verify that official repository is used for downloading source code and Docker images. Change any default passwords that are used during installation, e.g., password for connecting to postgres. Iroha repository contains examples of private and public keys - never use it in production. Moreover, verify that new keypairs are generated in a safe environment and only administrator has access to those keypairs (or at least minimise the number of people). After deploying keys to Iroha peers delete private keys from the host that was used to perform deployment, i.e. private keys should reside only inside Iroha peers. Create an encrypted backup of private keys before deleting them and limit the access to it.

6.7.3 Network configuration

Iroha listens on ports 50051 and 10001. Firewall settings must allow incoming/outgoing connections to/from these ports. If possible, disable or remove any other network services with listening ports (FTP, DNS, LDAP, SMB, DHCP, NFS, SNMP, etc). Ideally, Iroha should be as much isolated as possible in terms of networking.

Currently, there is no traffic encryption in Iroha, we strongly recommend using VPN or Calico for setting up Docker overlay network, i.e. any mechanism that allows encrypting communication between peers. Docker swarm encrypts communications by default, but remember to open necessary ports in the firewall configuration. In case VPN is used, verify that VPN key is unavailable to other users.

If SSH is used, disable root login. Apart from that, disable password authentication and use only keys. It might be helpful to set up SSH log level to INFO as well.

If IPv6 is not used, it might be a good idea to disable it.

6.7.4 Updates

Install the latest operating system security patches and update it regularly. If Iroha is running in Docker containers, update Docker regularly. While being optional, it is considered a good practice to test updates on a separate server before installing to production.

6.7.5 Logging and monitoring

- Collect and ship logs to a dedicated machine using an agent (e.g., Filebeat).
- Collect logs from all Iroha peers in a central point (e.g., Logstash).
- Transfer logging and monitoring information via an encrypted channel (e.g., https).
- Set up an authentication mechanism to prevent third parties from accessing logs.
- Set up an authentication mechanism to prevent third parties from submitting logs.
- Log all administrator access.

6.7.6 OS hardening

The following steps assume Docker is used for running Iroha.

- Enable and configure Docker Content Trust.
- Allow only trusted users to control Docker daemon.
- Set up a limit for Docker container resources.

CHAPTER 7

Iroha API reference

In API section we will take a look at building blocks of an application interacting with Iroha. We will overview commands and queries that the system has, and the set of client libraries encompassing transport and application layer logic.

Iroha API follows command-query separation principle.

Communication between Iroha peer and a client application is maintained via gRPC framework. Client applications should follow described protocol and form transactions accordingly to their description.

7.1 Commands

A command changes the state, called World State View, by performing an action over an entity (asset, account) in the system. Any command should be included in a transaction to perform an action.

7.1.1 Add asset quantity

Purpose

The purpose of add asset quantity command is to increase the quantity of an asset on account of transaction creator. Use case scenario is to increase the number of a mutable asset in the system, which can act as a claim on a commodity (e.g. money, gold, etc.)

Schema

```
message AddAssetQuantity {
    string asset_id = 1;
    string amount = 2;
}
```

Note: Please note that due to a known issue you would not get any exception if you pass invalid precision value. Valid range is: 0 <= precision <= 255

Structure

Field	Description	Constraint	Example
Asset ID	id of the asset	<asset_name>#<domain_id>	usd#morgan
Amount	positive amount of the asset to add	> 0	200.02

Validation

1. Asset and account should exist
2. Added quantity precision should be equal to asset precision
3. Creator of a transaction should have a role which has permissions for issuing assets

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not add asset quantity	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to add asset quantity	Grant the necessary permission
3	No such asset	Cannot find asset with such name or such precision	Make sure asset id and precision are correct
4	Summation overflow	Resulting amount of asset is greater than the system can support	Make sure that resulting amount is less than 2^256

7.1.2 Add peer

Purpose

The purpose of add peer command is to write into ledger the fact of peer addition into the peer network. After a transaction with AddPeer has been committed, consensus and synchronization components will start using it.

Schema

```
message Peer {
    string address = 1;
    bytes peer_key = 2; // hex string
}

message AddPeer {
    Peer peer = 1;
}
```

Structure

Field	Description	Constraint	Example
Address	resolvable address in network (IPv4, IPv6, domain name, etc.)	should be resolvable	192.168.1.1:50541
Peer key	peer public key, which is used in consensus algorithm to sign-off vote, commit, reject messages	ed25519 public key	292a8714694095edce6be799398ed5d6244cd7be37eb813

Validation

1. Peer key is unique (there is no other peer with such public key)
2. Creator of the transaction has a role which has CanAddPeer permission
3. Such network address has not been already added

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not add peer	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to add peer	Grant the necessary permission

7.1.3 Add signatory

Purpose

The purpose of add signatory command is to add an identifier to the account. Such identifier is a public key of another device or a public key of another user.

Schema

```
message AddSignatory {
    string account_id = 1;
    bytes public_key = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	Account to which to add signatory	<account_name>@<domain_id>	makoto@soramitsu
Public key	Signatory to add to account	ed25519 public key	359f925e4eeecfdd6aa1abc0b79a6a121

Validation

Two cases:

Case 1. Transaction creator wants to add a signatory to his or her account, having permission CanAddSignatory

Case 2. CanAddSignatory was granted to transaction creator

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not add signatory	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to add signatory	Grant the necessary permission
3	No such account	Cannot find account to add signatory to	Make sure account id is correct
4	Signatory already exists	Account already has such signatory attached	Choose another signatory

7.1.4 Append role

Purpose

The purpose of append role command is to promote an account to some created role in the system, where a role is a set of permissions account has to perform an action (command or query).

Schema

```
message AppendRole {
   string account_id = 1;
   string role_name = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	id or account to append role to	already existent	makoto@soramitsu
Role name	name of already created role	already existent	MoneyCreator

Validation

1. The role should exist in the system
2. Transaction creator should have permissions to append role (CanAppendRole)
3. Account, which appends role, has set of permissions in his roles that is a superset of appended role (in other words no-one can append role that is more powerful than what transaction creator is)

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not append role	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to append role	Grant the necessary permission
3	No such account	Cannot find account to append role to	Make sure account id is correct
4	No such role	Cannot find role with such name	Make sure role id is correct

7.1.5 Create account

Purpose

The purpose of create account command is to make entity in the system, capable of sending transactions or queries, storing signatories, personal data and identifiers.

Schema

```
message CreateAccount {
    string account_name = 1;
    string domain_id = 2;
    bytes public_key = 3;
}
```

Structure

Field	Description	Constraint	Example
Account name	domain-unique name for account	*[a-z_0-9]{1,32}*	morgan_stanley
Domain ID	target domain to make relation with	should be created before the account	america
Public key	first public key to add to the account	ed25519 public key	407e57f50ca48969b08ba948171bb243

Validation

1. Transaction creator has permission to create an account
2. Domain, passed as domain_id, has already been created in the system
3. Such public key has not been added before as first public key of account or added to a multi-signature account

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not create account	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator either does not have permission to create account or tries to create account in a more privileged domain, than the one creator is in	Grant the necessary permission or choose another domain
3	No such domain	Cannot find domain with such name	Make sure domain id is correct
4	Account already exists	Account with such name already exists in that domain	Choose another name

7.1.6 Create asset

Purpose

The purpose of reate asset command is to create a new type of asset, unique in a domain. An asset is a countable representation of a commodity.

Schema

```
message CreateAsset {
    string asset_name = 1;
    string domain_id = 2;
    uint32 precision = 3;
}
```

Note: Please note that due to a known issue you would not get any exception if you pass invalid precision value. Valid range is: 0 <= precision <= 255

Structure

Field	Description	Constraint	Example
Asset name	domain-unique name for asset	*[a-z_0-9]{1,32}*	soracoin
Domain ID	target domain to make relation with	RFC1035[1], RFC1123[2]	japan
Precision	number of digits after comma/dot	0 <= precision <= 255	2

Validation

1. Transaction creator has permission to create assets

[1] https://www.ietf.org/rfc/rfc1035.txt
[2] https://www.ietf.org/rfc/rfc1123.txt

2. Asset name is unique in domain

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not create asset	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to create asset	Grant the necessary permission
3	No such domain	Cannot find domain with such name	Make sure domain id is correct
4	Asset already exists	Asset with such name already exists	Choose another name

7.1.7 Create domain

Purpose

The purpose of create domain command is to make new domain in Iroha network, which is a group of accounts.

Schema

```
message CreateDomain {
    string domain_id = 1;
    string default_role = 2;
}
```

Structure

Field	Description	Constraint	Example
Domain ID	ID for created domain	unique, RFC1035[1], RFC1123[2]	japan05
Default role	role for any created user in the domain	one of the existing roles	User

Validation

1. Domain ID is unique
2. Account, who sends this command in transaction, has role with permission to create domain
3. Role, which will be assigned to created user by default, exists in the system

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not create domain	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to create domain	Grant the necessary permission
3	Domain already exists	Domain with such name already exists	Choose another domain name
4	No default role found	Role, which is provided as a default one for the domain, is not found	Make sure the role you provided exists or create it

7.1.8 Create role

Purpose

The purpose of create role command is to create a new role in the system from the set of permissions. Combining different permissions into roles, maintainers of Iroha peer network can create customized security model.

Schema

```
message CreateRole {
    string role_name = 1;
    repeated RolePermission permissions = 2;
}
```

Structure

Field	Description	Constraint	Example
Role name	name of role to create	*[a-z_0-9]{1,32}*	User
RolePermission	array of already existent permissions	set of passed permissions is fully included into set of existing permissions	{can_receive, can_transfer}

Validation

1. Set of passed permissions is fully included into set of existing permissions
2. Set of the permissions is not empty

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not create role	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to create role	Grant the necessary permission
3	Role already exists	Role with such name already exists	Choose another role name

7.1.9 Detach role

Purpose

The purpose of detach role command is to detach a role from the set of roles of an account. By executing this command it is possible to decrease the number of possible actions in the system for the user.

Schema

```
message DetachRole {
    string account_id = 1;
    string role_name = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	ID of account where role will be deleted	already existent	makoto@soramitsu
Role name	a detached role name	existing role	User

Validation

1. The role exists in the system
2. The account has such role

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not detach role	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to detach role	Grant the necessary permission
3	No such account	Cannot find account to detach role from	Make sure account id is correct
4	No such role in account's roles	Account with such id does not have role with such name	Make sure account-role pair is correct
5	No such role	Role with such name does not exist	Make sure role id is correct

7.1.10 Grant permission

Purpose

The purpose of grant permission command is to give another account rights to perform actions on the account of transaction sender (give someone right to do something with my account).

Schema

```
message GrantPermission {
    string account_id = 1;
    GrantablePermission permission = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	id of the account to which the rights are granted	already existent	makoto@soramitsu
GrantablePermission name	name of grantable permission	permission is defined	CanTransferAssets

Validation

1. Account exists
2. Transaction creator is allowed to grant this permission

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not grant permission	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to grant permission	Grant the necessary permission
3	No such account	Cannot find account to grant permission to	Make sure account id is correct

7.1.11 Remove peer

Purpose

The purpose of remove peer command is to write into ledger the fact of peer removal from the network. After a transaction with RemovePeer has been committed, consensus and synchronization components will start using it.

Schema

```
message RemovePeer {
    bytes public_key = 1; // hex string
}
```

Structure

Field	Description	Constraint	Example
Public key	peer public key, which is used in consensus algorithm to sign vote messages	ed25519 public key	292a8714694095edce6be799398ed5d6244cd7be37eb813

Validation

1. There is more than one peer in the network
2. Creator of the transaction has a role which has CanRemovePeer permission
3. Peer should have been previously added to the network

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not remove peer	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to remove peer	Grant the necessary permission
3	No such peer	Cannot find peer with such public key	Make sure that the public key is correct
4	Network size does not allow to remove peer	After removing the peer the network would be empty	Make sure that the network has at least two peers

7.1.12 Remove signatory

Purpose

Purpose of remove signatory command is to remove a public key, associated with an identity, from an account

Schema

```
message RemoveSignatory {
    string account_id = 1;
    bytes public_key = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	id of the account to which the rights are granted	already existent	makoto@soramitsu
Public key	Signatory to delete	ed25519 public key	407e57f50ca48969b08ba948171bb243

Validation

1. When signatory is deleted, we should check if invariant of **size(signatories) >= quorum** holds
2. Signatory should have been previously added to the account

Two cases:

> Case 1. When transaction creator wants to remove signatory from their account and he or she has permission CanRemoveSignatory
>
> Case 2. CanRemoveSignatory was granted to transaction creator

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not remove signatory	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to remove signatory from his account	Grant the necessary permission
3	No such account	Cannot find account to remove signatory from	Make sure account id is correct
4	No such signatory	Cannot find signatory with such public key	Make sure public key is correct
5	Quorum does not allow to remove signatory	After removing the signatory account will be left with less signatories, than its quorum allows	Reduce the quorum

7.1.13 Revoke permission

Purpose

The purpose of revoke permission command is to revoke or dismiss given granted permission from another account in the network.

Schema

```
message RevokePermission {
    string account_id = 1;
    GrantablePermission permission = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	id of the account to which the rights are granted	already existent	makoto@soramitsu
GrantablePermission name	name of grantable permission	permission was granted	CanTransferAssets

Validation

Transaction creator should have previously granted this permission to a target account

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not revoke permission	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to revoke permission	Grant the necessary permission
3	No such account	Cannot find account to revoke permission from	Make sure account id is correct

7.1.14 Set account detail

Purpose

Purpose of set account detail command is to set key-value information for a given account

> **Warning:** If there was a value for a given key already in the storage then it will be replaced with the new value

Schema

```
message SetAccountDetail{
    string account_id = 1;
    string key = 2;
    string value = 3;
}
```

Structure

Field	Description	Constraint	Example
Account ID	id of the account to which the key-value information was set	already existent	makoto@soramitsu
Key	key of information being set	*[A-Za-z0-9_]{1,64}*	Name
Value	value of corresponding key	4096	Makoto

Validation

Two cases:

Case 1. When transaction creator wants to set account detail to his/her account and he or she has permission CanSetAccountInfo

Case 2. CanSetAccountInfo was granted to transaction creator

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not set account detail	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to set account detail for another account	Grant the necessary permission
3	No such account	Cannot find account to set account detail to	Make sure account id is correct

7.1.15 Set account quorum

Purpose

The purpose of set account quorum command is to set the number of signatories required to confirm the identity of a user, who creates the transaction. Use case scenario is to set the number of different users, utilizing single account, to sign off the transaction.

Schema

```
message SetAccountQuorum {
    string account_id = 1;
    uint32 quorum = 2;
}
```

Structure

Field	Description	Constraint	Example
Account ID	ID of account to set quorum	already existent	makoto@soramitsu
Quorum	number of signatories needed to be included within a transaction from this account	0 < quorum public-key set up to account 128	5

Validation

When quorum is set, it is checked if invariant of **size(signatories) >= quorum** holds.

Two cases:

Case 1. When transaction creator wants to set quorum for his/her account and he or she has permission CanRemoveSignatory

Case 2. CanRemoveSignatory was granted to transaction creator

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not set quorum	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to set quorum for his account	Grant the necessary permission
3	No such account	Cannot find account to set quorum to	Make sure account id is correct
4	No signatories on account	Cannot find any signatories attached to the account	Add some signatories before setting quorum
5	New quorum is incorrect	New quorum size is less than account's signatories amount	Choose another value or add more signatories

7.1.16 Subtract asset quantity

Purpose

The purpose of subtract asset quantity command is the opposite of AddAssetQuantity commands — to decrease the number of assets on account of transaction creator.

Schema

```
message SubtractAssetQuantity {
    string asset_id = 1;
    string amount = 2;
}
```

Note: Please note that due to a known issue you would not get any exception if you pass invalid precision value. Valid range is: 0 <= precision <= 255

Structure

Field	Description	Constraint	Example
Asset ID	id of the asset	<asset_name>#<domain_id>	usd#morgan
Amount	positive amount of the asset to subtract	> 0	200

Validation

1. Asset and account should exist
2. Added quantity precision should be equal to asset precision

3. Creator of the transaction should have a role which has permissions for subtraction of assets

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not subtract asset quantity	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to subtract asset quantity	Grant the necessary permission
3	No such asset found	Cannot find asset with such name or precision in account's assets	Make sure asset name and precision are correct
4	Not enough balance	Account's balance is too low to perform the operation	Add asset to account or choose lower value to subtract

7.1.17 Transfer asset

Purpose

The purpose of transfer asset command is to share assets within the account in peer network: in the way that source account transfers assets to the target account.

Schema

```
message TransferAsset {
    string src_account_id = 1;
    string dest_account_id = 2;
    string asset_id = 3;
    string description = 4;
    string amount = 5;
}
```

Structure

Field	Description	Constraint	Example
Source account ID	ID of the account to withdraw the asset from	already existent	makoto@soramitsu
Destination account ID	ID of the account to send the asset to	already existent	alex@california
Asset ID	ID of the asset to transfer	already existent	usd#usa
Description	Message to attach to the transfer	Max length is 64	here's my money take it
Amount	amount of the asset to transfer	0 <= precision <= 255	200.20

Validation

1. Source account has this asset in its AccountHasAsset relation[1]
2. An amount is a positive number and asset precision is consistent with the asset definition

3. Source account has enough amount of asset to transfer and is not zero
4. Source account can transfer money, and destination account can receive money (their roles have these permissions)

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not transfer asset	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to transfer asset from his account	Grant the necessary permission
3	No such source account	Cannot find account with such id to transfer money from	Make sure source account id is correct
4	No such destination account	Cannot find account with such id to transfer money to	Make sure destination account id is correct
5	No such asset found	Cannot find such asset	Make sure asset name and precision are correct
6	Not enough balance	Source account's balance is too low to perform the operation	Add asset to account or choose lower value to subtract
7	Too much asset to transfer	Resulting value of asset amount overflows destination account's amount	Make sure final value is less than 2^256

7.1.18 Compare and Set Account Detail

Purpose

Purpose of compare and set account detail command is to set key-value information for a given account if the old value matches the value passed.

Schema

```
message CompareAndSetAccountDetail{
    string account_id = 1;
    string key = 2;
    string value = 3;
    oneof opt_old_value {
        string old_value = 4;
    }
}
```

Note: Pay attention, that old_value field is optional. This is due to the fact that the key-value pair might not exist.

Structure

Field	Description	Constraint	Example
Account ID	id of the account to which the key-value information was set. If key-value pair doesnot exist , then it will be created	an existing account	artyom@soramitsu
Key	key of information being set	*[A-Za-z0-9_]{1,64}*	Name
Value	new value for the corresponding key	length of value 4096	Artyom
Old value	current value for the corresponding key	length of value 4096	Artem

Validation

Three cases:

Case 1. When transaction creator wants to set account detail to his/her account and he or she has permission GetMyAccountDetail / GetAllAccountsDetail / GetDomainAccountDetail

Case 2. When transaction creator wants to set account detail to another account and he or she has permissions SetAccountDetail and GetAllAccountsDetail / GetDomainAccountDetail

Case 3. SetAccountDetail permission was granted to transaction creator and he or she has permission GetAllAccountsDetail / GetDomainAccountDetail

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not compare and set account detail	Internal error happened	Try again or contact developers
2	No such permissions	Command's creator does not have permission to set and read account detail for this account	Grant the necessary permission
3	No such account	Cannot find account to set account detail to	Make sure account id is correct
4	No match values	Old values do not match	Make sure old value is correct

7.2 Queries

A query is a request related to certain part of World State View — the latest state of blockchain. Query cannot modify the contents of the chain and a response is returned to any client immediately after receiving peer has processed a query.

7.2.1 Validation

The validation for all queries includes:

- timestamp — shouldn't be from the past (24 hours prior to the peer time) or from the future (range of 5 minutes added to the peer time)
- signature of query creator — used for checking the identity of query creator
- query counter — checked to be incremented with every subsequent query from query creator

- roles — depending on the query creator's role: the range of state available to query can relate to to the same account, account in the domain, to the whole chain, or not allowed at all

7.2.2 Get Account

Purpose

Purpose of get account query is to get the state of an account.

Request Schema

```
message GetAccount {
    string account_id = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Account ID	account id to request its state	<account_name>@<domain_id>	alex@morgan

Response Schema

```
message AccountResponse {
    Account account = 1;
    repeated string account_roles = 2;
}

message Account {
    string account_id = 1;
    string domain_id = 2;
    uint32 quorum = 3;
    string json_data = 4;
}
```

Response Structure

Field	Description	Constraint	Example
Account ID	account id	<account_name>@<domain_id>	alex@morgan
Domain ID	domain where the account was created	RFC1035[1], RFC1123[2]	morgan
Quorum	number of signatories needed to sign the transaction to make it valid	0 < quorum 128	5
JSON data	key-value account information	JSON	{ genesis: {name: alex} }

[1] https://www.ietf.org/rfc/rfc1035.txt
[2] https://www.ietf.org/rfc/rfc1123.txt

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get account	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get account	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

7.2.3 Get Block

Purpose

Purpose of get block query is to get a specific block, using its height as an identifier

Request Schema

```
message GetBlock {
  uint64 height = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Height	height of the block to be retrieved	0 < height < 2^64	42

Response Schema

```
message BlockResponse {
  Block block = 1;
}
```

Response Structure

Field	Description	Constraint	Example
Block	the retrieved block	block structure	block

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get block	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have a permission to get block	Grant the necessary permission
3	Invalid height	Supplied height is not uint_64 or greater than the ledger's height	Check the height and try again

7.2.4 Get Signatories

Purpose

Purpose of get signatories query is to get signatories, which act as an identity of the account.

Request Schema

```
message GetSignatories {
    string account_id = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Account ID	account id to request signatories	<account_name>@<domain_id>	alex@morgan

Response Schema

```
message SignatoriesResponse {
    repeated bytes keys = 1;
}
```

Response Structure

Field	Description	Constraint	Example
Keys	an array of public keys	ed25519	292a8714694095edce6be799398ed5d6

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get signatories	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get signatories	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

7.2.5 Get Transactions

Purpose

GetTransactions is used for retrieving information about transactions, based on their hashes. .. note:: This query is valid if and only if all the requested hashes are correct: corresponding transactions exist, and the user has a permission to retrieve them

Request Schema

```
message GetTransactions {
    repeated bytes tx_hashes = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Transactions hashes	an array of hashes	array with 32 byte hashes	{hash1, hash2. . . }

Response Schema

```
message TransactionsResponse {
    repeated Transaction transactions = 1;
}
```

Response Structure

Field	Description	Constraint	Example
Transactions	an array of transactions	Committed transactions	{tx1, tx2. . . }

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get transactions	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get transactions	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories
4	Invalid hash	At least one of the supplied hashes either does not exist in user's transaction list or creator of the query does not have permissions to see it	Check the supplied hashes and try again

7.2.6 Get Pending Transactions

Purpose

GetPendingTransactions is used for retrieving a list of pending (not fully signed) multisignature transactions or batches of transactions issued by account of query creator.

Note: This query uses pagination for quicker and more convenient query responses.

Request Schema

```
message TxPaginationMeta {
    uint32 page_size = 1;
    oneof opt_first_tx_hash {
        string first_tx_hash = 2;
    }
}

message GetPendingTransactions {
    TxPaginationMeta pagination_meta = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Page size	maximum amount of transactions returned in the response	page_size > 0	5
First tx hash	optional - hash of the first transaction in the starting batch	hash in hex format	bddd58404d1315e0eb27902c5d7c8eb0

All the user's semi-signed multisignature (pending) transactions can be queried. Maximum amount of transactions contained in a response can be limited by **page_size** field. All the pending transactions are stored till they have collected enough signatures or get expired. The mutual order of pending transactions or batches of transactions is preserved for a user. That allows a user to query all transactions sequentially - page by page. Each response may contain a reference to the next batch or transaction that can be queried. A page size can be greater than the size of the following batch (in transactions). In that case, several batches or transactions will be returned. During navigating over pages, the following batch can collect the missing signatures before it gets queried. This will result in stateful failed query response due to a missing hash of the batch.

Example

If there are two pending batches with three transactions each and a user queries pending transactions with page size 5, then the transactions of the first batch will be in the response and a reference (first transaction hash and batch size, even if it is a single transaction in fact) to the second batch will be specified too. Transactions of the second batch are not included in the first response because the batch cannot be devided into several parts and only complete batches can be contained in a response.

Response Schema

```
message PendingTransactionsPageResponse {
    message BatchInfo {
        string first_tx_hash = 1;
        uint32 batch_size = 2;
    }
    repeated Transaction transactions = 1;
    uint32 all_transactions_size = 2;
    BatchInfo next_batch_info = 3;
}
```

Response Structure

The response contains a list of pending transactions, the amount of all stored pending transactions for the user and the information required to query the subsequent page (if exists).

Field	Description	Constraint	Example
Transactions	an array of pending transactions	Pending transactions	{tx1, tx2...}
All transactions size	the number of stored transactions	all_transactions_size >= 0	0
Next batch info	A reference to the next page - the message might be not set in a response		
First tx hash	hash of the first transaction in the next batch	hash in hex format	bddd58404d1315e0eb27902c5d7c8eb0
Batch size	Minimum page size required to fetch the next batch	batch_size > 0	3

7.2.7 Get Pending Transactions (deprecated)

Warning: The query without parameters is deprecated now and will be removed in the following major Iroha release (2.0). Please use the new query version instead: *Get Pending Transactions*.

Purpose

GetPendingTransactions is used for retrieving a list of pending (not fully signed) multisignature transactions or batches of transactions issued by account of query creator.

Request Schema

```
message GetPendingTransactions {
}
```

Response Schema

```
message TransactionsResponse {
    repeated Transaction transactions = 1;
}
```

Response Structure

The response contains a list of pending transactions.

Field	Description	Constraint	Example
Transactions	an array of pending transactions	Pending transactions	{tx1, tx2. . . }

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get pending transactions	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get pending transactions	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

7.2.8 Get Account Transactions

Purpose

In a case when a list of transactions per account is needed, *GetAccountTransactions* query can be formed.

Note: This query uses pagination for quicker and more convenient query responses.

Request Schema

```
message TxPaginationMeta {
    uint32 page_size = 1;
    oneof opt_first_tx_hash {
        string first_tx_hash = 2;
    }
}

message GetAccountTransactions {
    string account_id = 1;
    TxPaginationMeta pagination_meta = 2;
}
```

Request Structure

Field	Description	Constraint	Example
Account ID	account id to request transactions from	<account_name>@<domain_id>	makoto@soramitsu
Page size	size of the page to be returned by the query, if the response contains fewer transactions than a page size, then next tx hash will be empty in response	page_size > 0	5
First tx hash	hash of the first transaction in the page. If that field is not set — then the first transactions are returned	hash in hex format	bddd58404d1315e0eb27902c5d7c8eb0

Response Schema

```
message TransactionsPageResponse {
    repeated Transaction transactions = 1;
    uint32 all_transactions_size = 2;
    oneof next_page_tag {
        string next_tx_hash = 3;
    }
}
```

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get account transactions	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get account transactions	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories
4	Invalid pagination hash	Supplied hash does not appear in any of the user's transactions	Make sure hash is correct and try again
5	Invalid account id	User with such account id does not exist	Make sure account id is correct

Response Structure

Field	Description	Constraint	Example
Transactions	an array of transactions for given account	Committed transactions	{tx1, tx2...}
All transactions size	total number of transactions created by the given account		100
Next transaction hash	hash pointing to the next transaction after the last transaction in the page. Empty if a page contains the last transaction for the given account		bddd58404d1315e0eb27902c5d7c8eb0602c16238f005773df406bc1

7.2.9 Get Account Asset Transactions

Purpose

GetAccountAssetTransactions query returns all transactions associated with given account and asset.

Note: This query uses pagination for query responses.

Request Schema

```
message TxPaginationMeta {
    uint32 page_size = 1;
    oneof opt_first_tx_hash {
        string first_tx_hash = 2;
    }
}

message GetAccountAssetTransactions {
    string account_id = 1;
    string asset_id = 2;
```

```
    TxPaginationMeta pagination_meta = 3;
}
```

Request Structure

Field	Description	Constraint	Example
Account ID	account id to request transactions from	<account_name>@<domain_id>	makoto@soramitsu
Asset ID	asset id in order to filter transactions containing this asset	<asset_name>#<domain_id>	jpy#japan
Page size	size of the page to be returned by the query, if the response contains fewer transactions than a page size, then next tx hash will be empty in response	page_size > 0	5
First tx hash	hash of the first transaction in the page. If that field is not set — then the first transactions are returned	hash in hex format	bddd58404d1315e0eb27902c5d7c8eb0

Response Schema

```
message TransactionsPageResponse {
    repeated Transaction transactions = 1;
    uint32 all_transactions_size = 2;
    oneof next_page_tag {
        string next_tx_hash = 3;
    }
}
```

Response Structure

Field	Description	Constraint	Example
Transactions	an array of transactions for given account and asset	Committed transactions	{tx1, tx2...}
All transactions size	total number of transactions for given account and asset		100
Next transaction hash	hash pointing to the next transaction after the last transaction in the page. Empty if a page contains the last transaction for given account and asset	bddd58404d1315e0eb27902c5d7c8eb0602c16238f005773df406bc1	

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get account asset transactions	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get account asset transactions	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories
4	Invalid pagination hash	Supplied hash does not appear in any of the user's transactions	Make sure hash is correct and try again
5	Invalid account id	User with such account id does not exist	Make sure account id is correct
6	Invalid asset id	Asset with such asset id does not exist	Make sure asset id is correct

7.2.10 Get Account Assets

Purpose

To get the state of all assets in an account (a balance), *GetAccountAssets* query can be used.

Request Schema

```
message AssetPaginationMeta {
    uint32 page_size = 1;
    oneof opt_first_asset_id {
        string first_asset_id = 2;
    }
}

message GetAccountAssets {
    string account_id = 1;
    AssetPaginationMeta pagination_meta = 2;
}
```

Request Structure

Field	Description	Constraint	Example
Account ID	account id to request balance from	<account_name>@<domain_id>	makoto@soramitsu
AssetPaginationMeta.page_size	Requested page size. The number of assets in response will not exceed this value. If the response was truncated, the asset id immediately following the returned ones will be provided in next_asset_id.	0 < page_size < 32 bit unsigned int max (4294967296)	100
AssetPaginationMeta.first_asset_id	Requested page start. If the field is not set, then the first page is returned.	name#domain	my_asset#my_domain

Response Schema

```
message AccountAssetResponse {
    repeated AccountAsset account_assets = 1;
    uint32 total_number = 2;
    oneof opt_next_asset_id {
        string next_asset_id = 3;
    }
}

message AccountAsset {
    string asset_id = 1;
    string account_id = 2;
    string balance = 3;
}
```

Response Structure

Field	Description	Constraint	Example
Asset ID	identifier of asset used for checking the balance	<asset_name>#<domain_id>	jpy#japan
Account ID	account which has this balance	<account_name>@<domain_id>	makoto@soramitsu
Balance	balance of the asset	No less than 0	200.20
total_number	number of assets matching query without page limits	0 < total_number < 32 bit unsigned int max (4294967296)	100500
next_asset_id	the id of asset immediately following curent page	name#domain	my_asset#my_domain

Note: If page size is equal or greater than the number of assets matching other requested criteria, the next asset id will be unset in the response. Otherwise, it contains the value that clients should use for the first asset id if they want to fetch the next page.

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get account assets	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get account assets	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories
4	Invalid pagination metadata	Wrong page size or nonexistent first asset	Set a valid page size, and make sure that asset id is valid, or leave first asset id unspecified

7.2.11 Get Account Detail

Purpose

To get details of the account, *GetAccountDetail* query can be used. Account details are key-value pairs, splitted into writers categories. Writers are accounts, that added the corresponding account detail. Example of such structure is:

```
{
    "account@a_domain": {
        "age": 18,
        "hobbies": "crypto"
    },
    "account@b_domain": {
        "age": 20,
        "sports": "basketball"
    }
}
```

Here, one can see four account details - "age", "hobbies" and "sports" - added by two writers - "account@a_domain" and "account@b_domain". All of these details, obviously, are about the same account.

Request Schema

```
message AccountDetailRecordId {
  string writer = 1;
  string key = 2;
}

message AccountDetailPaginationMeta {
  uint32 page_size = 1;
  AccountDetailRecordId first_record_id = 2;
}

message GetAccountDetail {
  oneof opt_account_id {
    string account_id = 1;
  }
  oneof opt_key {
    string key = 2;
  }
  oneof opt_writer {
    string writer = 3;
  }
  AccountDetailPaginationMeta pagination_meta = 4;
}
```

Note: Pay attention, that all fields except pagination meta are optional. The reasons for that are described below.

Warning: Pagination metadata can be missing in the request for compatibility reasons, but this behaviour is deprecated and should be avoided.

Request Structure

Field	Description	Constraint	Example
Account ID	account id to get details from	<account_name>@<domain_id>	account@domain
Key	key, under which to get details	string	age
Writer	account id of writer	<account_name>@<domain_id>	account@domain
AccountDetailPaginationMeta.page_size	Requested page size. The number of records in response will not exceed this value. If the response was truncated, the record id immediately following the returned ones will be provided in next_record_id.	0 < page_size < 32 bit unsigned int max (4294967296)	100
AccountDetailPaginationMeta.first_record_id.writer	requested first record id writer	name#domain	my_asset#my_domain
AccountDetailPaginationMeta.first_record_id.key	requested first record id key	string	age

Note: When specifying first record id, it is enough to provide the attributes (writer, key) that are unset in the main query.

Response Schema

```
message AccountDetailResponse {
  string detail = 1;
  uint64 total_number = 2;
  AccountDetailRecordId next_record_id = 3;
}
```

Response Structure

Field	Description	Constraint	Example
Detail	key-value pairs with account details	JSON	see below
total_number	number of records matching query without page limits	0 < total_number < 32 bit unsigned int max (4294967296)	100
next_record_id.writer	the writer account of the record immediately following curent page	<account_name>@<domain_id>	pushkin@lyceum.tsar
next_record_id.key	the key of the record immediately following curent page	string	cold and sun

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get account detail	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get account detail	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories
4	Invalid pagination metadata	Wrong page size or nonexistent first record	Set valid page size, and make sure that the first record id is valid, or leave the first record id unspecified

Usage Examples

Again, let's consider the example of details from the beginning and see how different variants of *GetAccountDetail* queries will change the resulting response.

```
{
    "account@a_domain": {
        "age": 18,
        "hobbies": "crypto"
    },
    "account@b_domain": {
        "age": 20,
        "sports": "basketball"
    }
}
```

account_id is not set

If account_id is not set - other fields can be empty or not - it will automatically be substituted with query creator's account, which will lead to one of the next cases.

only account_id is set

In this case, all details about that account are going to be returned, leading to the following response:

```
{
    "account@a_domain": {
        "age": 18,
        "hobbies": "crypto"
    },
    "account@b_domain": {
        "age": 20,
        "sports": "basketball"
    }
}
```

account_id and key are set

Here, details added by all writers under the key are going to be returned. For example, if we asked for the key "age", that's the response we would get:

```
{
    "account@a_domain": {
```

```
        "age": 18
    },
    "account@b_domain": {
        "age": 20
    }
}
```

account_id and writer are set

Now, the response will contain all details about this account, added by one specific writer. For example, if we asked for writer "account@b_domain", we would get:

```
{
    "account@b_domain": {
        "age": 20,
        "sports": "basketball"
    }
}
```

account_id, key and writer are set

Finally, if all three field are set, result will contain details, added the specific writer and under the specific key, for example, if we asked for key "age" and writer "account@a_domain", we would get:

```
{
    "account@a_domain": {
        "age": 18
    }
}
```

7.2.12 Get Asset Info

Purpose

In order to get information on the given asset (as for now - its precision), user can send *GetAssetInfo* query.

Request Schema

```
message GetAssetInfo {
    string asset_id = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Asset ID	asset id to know related information	<asset_name>#<domain_id>	jpy#japan

Response Schema

```
message Asset {
    string asset_id = 1;
    string domain_id = 2;
    uint32 precision = 3;
}
```

Note: Please note that due to a known issue you would not get any exception if you pass invalid precision value. Valid range is: 0 <= precision <= 255

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get asset info	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get asset info	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

Response Structure

Field	Description	Constraint	Example
Asset ID	identifier of asset used for checking the balance	<asset_name>#<domain_id>	jpy#japan
Domain ID	domain related to this asset	RFC1035[1], RFC1123[2]	japan
Precision	number of digits after comma	0 <= precision <= 255	2

7.2.13 Get Roles

Purpose

To get existing roles in the system, a user can send *GetRoles* query to Iroha network.

Request Schema

```
message GetRoles {
}
```

Response Schema

```
message RolesResponse {
    repeated string roles = 1;
}
```

Response Structure

Field	Description	Constraint	Example
Roles	array of created roles in the network	set of roles in the system	{MoneyCreator, User, Admin, ... }

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get roles	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get roles	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

7.2.14 Get Role Permissions

Purpose

To get available permissions per role in the system, a user can send *GetRolePermissions* query to Iroha network.

Request Schema

```
message GetRolePermissions {
    string role_id = 1;
}
```

Request Structure

Field	Description	Constraint	Example
Role ID	role to get permissions for	existing role in the system	MoneyCreator

Response Schema

```
message RolePermissionsResponse {
    repeated string permissions = 1;
}
```

Response Structure

Field	Description	Constraint	Example
Permissions	array of permissions related to the role	string of permissions related to the role	{can_add_asset_qty, ...}

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get role permissions	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get role permissions	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

7.2.15 Get Peers

Purpose

A query that returns a list of peers in Iroha network.

Request Schema

```
message GetPeers {
}
```

Response Schema

```
message Peer {
  string address = 1;
  string peer_key = 2; // hex string
}

message PeersResponse {
  repeated Peer peers = 1;
}
```

Response Structure

A list of peers with their addresses and public keys is returned.

Field	Description	Constraint	Example
Peers	array of peers from the network	non-empty list of peers	{Peer{"peer.domain.com", "292a8714694095edce6be799398ed5c ...}

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get peers	Internal error happened	Try again or contact developers
2	No such permissions	Query creator does not have enough permissions to get peers	Append a role with can_get_blocks or can_get_peers permission
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

Warning: Currently Get Peers query uses "can_get_blocks" permission for compatibility purposes. Later that will be changed to "can_get_peers" with the next major Iroha release.

7.2.16 Fetch Commits

Purpose

To get new blocks as soon as they are committed, a user can invoke *FetchCommits* RPC call to Iroha network.

Request Schema

No request arguments are needed

Response Schema

```
message BlockQueryResponse {
  oneof response {
    BlockResponse block_response = 1;
    BlockErrorResponse block_error_response = 2;
  }
}
```

Please note that it returns a stream of *BlockQueryResponse*.

Response Structure

Field	Description	Constraint	Example
Block	Iroha block	only committed blocks	{ 'block_v1': ....}

Possible Stateful Validation Errors

Code	Error Name	Description	How to solve
1	Could not get block streaming	Internal error happened	Try again or contact developers
2	No such permissions	Query's creator does not have any of the permissions to get blocks	Grant the necessary permission: individual, global or domain one
3	Invalid signatures	Signatures of this query did not pass validation	Add more signatures and make sure query's signatures are a subset of account's signatories

Example

You can check an example how to use this query here: https://github.com/x3medima17/twitter

Commands here are parts of transaction - a state-changing set of actions in the system. When a transaction passes validation and consensus stages, it is written in a block and saved in immutable block store (blockchain).

Transactions consist of commands, performing an action over an entity in the system. The entity might be an account, asset, etc.

CHAPTER 8

Maintenance

Hardware requirements, deployment process in details, aspects related to security, configuration files — all of the listed is explained in this separate section, helpful for DevOps engineers or those who are digging deeper in the system capabilities.

8.1 Permissions

Hyperledger Iroha uses a role-based access control system to limit actions of its users. This system greatly helps to implement use cases involving user groups having different access levels — ranging from the weak users, who can't even receive asset transfer to the super-users. The beauty of our permission system is that you don't have to have a super-user in your Iroha setup or use all the possible permissions: you can create segregated and lightweight roles.

Maintenance of the system involves setting up roles and permissions, that are included in the roles. This might be done at the initial step of system deployment — in genesis block, or later when Iroha network is up and running, roles can be changed (if there is a role that can do that :)

This section will help you to understand permissions and give you an idea of how to create roles including certain permissions. Each permission is provided with an example written in Python that demonstrates the way of transaction or query creation, which require specific permission. Every example uses *commons.py* module, which listing is available at *Supplementary Sources* section.

8.2 List of Permissions

Permission Name	Category	Type
can_create_account	Account	Command
can_set_detail	Account	Command
can_set_my_account_detail `grantable`	Account	Command
can_create_asset	Asset	Command
can_receive	Asset	Command

Continued on next page

Table 8.1 – continued from previous page

Permission Name	Category	Type
can_transfer	Asset	Command
can_transfer_my_assets `grantable`	Asset	Command
can_add_asset_qty	Asset Quantity	Command
can_subtract_asset_qty	Asset Quantity	Command
can_add_domain_asset_qty	Asset Quantity	Command
can_subtract_domain_asset_qty	Asset Quantity	Command
can_create_domain	Domain	Command
can_grant_can_add_my_signatory	Grant	Command
can_grant_can_remove_my_signatory	Grant	Command
can_grant_can_set_my_account_detail	Grant	Command
can_grant_can_set_my_quorum	Grant	Command
can_grant_can_transfer_my_assets	Grant	Command
can_add_peer	Peer	Command
can_remove_peer	Peer	Command
can_append_role	Role	Command
can_create_role	Role	Command
can_detach_role	Role	Command
can_add_my_signatory `grantable`	Signatory	Command
can_add_signatory	Signatory	Command
can_remove_my_signatory `grantable`	Signatory	Command
can_remove_signatory	Signatory	Command
can_set_my_quorum `grantable`	Signatory	Command
can_set_quorum	Signatory	Command
can_get_all_acc_detail	Account	Query
can_get_all_accounts	Account	Query
can_get_domain_acc_detail	Account	Query
can_get_domain_accounts	Account	Query
can_get_my_acc_detail	Account	Query
can_get_my_account	Account	Query
can_get_all_acc_ast	Account Asset	Query
can_get_domain_acc_ast	Account Asset	Query
can_get_my_acc_ast	Account Asset	Query
can_get_all_acc_ast_txs	Account Asset Transaction	Query
can_get_domain_acc_ast_txs	Account Asset Transaction	Query
can_get_my_acc_ast_txs	Account Asset Transaction	Query
can_get_all_acc_txs	Account Transaction	Query
can_get_domain_acc_txs	Account Transaction	Query
can_get_my_acc_txs	Account Transaction	Query
can_read_assets	Asset	Query
can_get_blocks	Block Stream	Query
can_get_roles	Role	Query
can_get_all_signatories	Signatory	Query
can_get_domain_signatories	Signatory	Query
can_get_my_signatories	Signatory	Query
can_get_all_txs	Transaction	Query
can_get_my_txs	Transaction	Query
can_get_peers	Peer	Query

8.2.1 Command-related permissions

Account

can_create_account

Allows creating new accounts.

Related API method: Create Account

Example

Admin creates domain "test" that contains only can_create_account permission and Alice account in that domain. Alice can create Bob account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_create_account]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def create_account_tx():
    tx = iroha.transaction([
        iroha.command('CreateAccount', account_name='bob', domain_id='test', public_
↪key=bob['key'])
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_set_detail

Allows setting account detail.

The permission allows setting details to other accounts. Another way to set detail without can_set_detail permission is to grant *can_set_my_account_detail* permission to someone. In order to grant, transaction creator should have *can_grant_can_set_my_account_detail* permission.

Note: Transaction creator can always set detail for own account even without that permission.

Related API method: Set Account Detail

Example

Admin creates domain "test" that contains only can_set_detail permission and Alice account in that domain. Alice can set detail for Admin account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_set_detail]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def set_account_detail_tx():
    tx = iroha.transaction([
        iroha.command('SetAccountDetail', account_id=admin['id'], key='fav_color',␣
↪value='red')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_set_my_account_detail

Hint: This is a grantable permission.

Permission that allows a specified account to set details for the another specified account.

Note: To grant the permission an account should already have a role with *can_grant_can_set_my_account_detail* permission.

Related API method: Set Account Detail

Example

Admin creates domain “test” that contains only can_grant_can_set_my_account_detail permission and two accounts for Alice and Bob in that domain. Alice grants to Bob can_set_my_account_detail permission. Bob can set detail for Alice account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_grant_can_set_my_account_detail]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key']))
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_permission_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_set_my_account_detail)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def set_detail_tx():
    tx = iroha.transaction([
        iroha.command('SetAccountDetail', account_id=alice['id'], key='fav_year',␣
↪value='2019')
    ], creator_account=bob['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, bob['key'])
    return tx
```

Asset

can_create_asset

Allows creating new assets.

Related API method: Create Asset

Example

Admin creates domain “test” that contains only can_create_asset permission and Alice account in that domain. Alice can create new assets.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_create_asset]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def create_asset_tx():
    tx = iroha.transaction([
        iroha.command('CreateAsset', asset_name='coin', domain_id='test', precision=2)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_receive

Allows account receive assets.

Related API method: Transfer Asset

Example

Admin creates domain “test” that contains can_receive and can_transfer permissions and two accounts for Alice and Bob. Admin creates “coin” asset, adds some quantity of it and transfers the asset to Alice. Alice can transfer assets to Bob (Alice has can_transfer permission and Bob has can_receive permission).

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_transfer, primitive_pb2.can_receive]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
```

```
    genesis_commands.extend([
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])),
        iroha.command('CreateAsset', asset_name='coin', domain_id='test',␣
↪precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='90.00'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
                      dest_account_id=alice['id'],
                      asset_id='coin#test',
                      description='init top up',
                      amount='90.00')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def transfer_asset_tx():
    tx = iroha.transaction([
        iroha.command('TransferAsset',
                      src_account_id=alice['id'],
                      dest_account_id=bob['id'],
                      asset_id='coin#test',
                      description='transfer to Bob',
                      amount='60.00')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_transfer

Allows sending assets from an account of transaction creator.

You can transfer an asset from one domain to another, even if the other domain does not have an asset with the same name.

Note: Destination account should have *can_receive* permission.

Related API method: Transfer Asset

```
#
# Copyright Soramitsu Co., Ltd. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
#

import can_receive

# Please see example for can_receive permission.
# By design can_receive and can_transfer permissions
# can be tested only together.
```

can_transfer_my_assets

Hint: This is a grantable permission.

Permission that allows a specified account to transfer assets of another specified account.

See the example (to be done) for the usage details.

Related API method: Transfer Asset

Example

Admin creates domain "test" that contains can_grant_can_transfer_my_assets, can_receive, can_transfer permissions and two accounts for Alice and Bob in that domain. Admin issues some amount of "coin" asset and transfers it to Alice. Alice grants to Bob can_transfer_my_assets permission. Bob can transfer Alice's assets to any account that has can_receive permission, for example, to Admin.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [
        primitive_pb2.can_grant_can_transfer_my_assets,
        primitive_pb2.can_receive,
        primitive_pb2.can_transfer
    ]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.extend([
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])),
        iroha.command('CreateAsset', asset_name='coin', domain_id='test',␣
↪precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='100.00'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
                      dest_account_id=alice['id'],
                      asset_id='coin#test',
                      description='init top up',
                      amount='90.00')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx
```

```

@commons.hex
def grant_permission_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_transfer_my_assets)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def transfer_asset_tx():
    tx = iroha.transaction([
        iroha.command('TransferAsset',
                      src_account_id=alice['id'],
                      dest_account_id=admin['id'],
                      asset_id='coin#test',
                      description='transfer from Alice to Admin by Bob',
                      amount='60.00')
    ], creator_account=bob['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, bob['key'])
    return tx
```

Asset Quantity

can_add_asset_qty

Allows issuing assets.

The corresponding command can be executed only for an account of transaction creator and only if that account has a role with the permission.

Related API method: Add Asset Quantity

Example

Admin creates domain “test” that contains only can_add_asset_qty permission and Alice account in that domain. Admin craetes “coin” asset. Alice can add to own account any amount of any asset (e.g. “coin” asset).

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_add_asset_qty]
```

```
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAsset', asset_name='coin', domain_id='test',␣
↪precision=2))
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def add_asset_tx():
    tx = iroha.transaction([
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='5000.99')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_subtract_asset_qty

Allows burning assets.

The corresponding command can be executed only for an account of transaction creator and only if that account has a role with the permission.

Related API method: Subtract Asset Quantity

Example

Admin creates domain "test" that contains only can_subtract_asset_qty permission and Alice account in that domain. Admin issues some amount of "coin" asset and transfers some amount of "coin" asset to Alice. Alice can burn any amount of "coin" assets.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_subtract_asset_qty]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.extend([
        iroha.command('CreateAsset', asset_name='coin', domain_id='test',␣
↪precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='1000.00'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
                      dest_account_id=alice['id'],
                      asset_id='coin#test',
```

```
                        description='init top up',
                        amount='999.99')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def subtract_asset_tx():
    tx = iroha.transaction([
        iroha.command('SubtractAssetQuantity', asset_id='coin#test', amount='999.99')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_add_domain_asset_qty

Allows issuing assets only in own domain.

The corresponding command can be executed only for an account of transaction creator and only if that account has a role with the permission and only for assets in creator's domain.

Related API method: Add Asset Quantity

```
#
# Copyright Soramitsu Co., Ltd. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
#

import can_add_asset_qty

# Please see example for can_add_asset_qty permission.

# TODO igor-egorov 21.01.2019 IR-240
```

can_subtract_domain_asset_qty

Allows burning assets only in own domain.

The corresponding command can be executed only for an account of transaction creator and only if that account has a role with the permission and only for assets in creator's domain.

Related API method: Subtract Asset Quantity

```
#
# Copyright Soramitsu Co., Ltd. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
#
```

```

import can_subtract_asset_qty

# Please see example for can_subtract_asset_qty permission.

# TODO igor-egorov 21.01.2019 IR-240
```

Domain

can_create_domain

Allows creating new domains within the system.

Related API method: Create Domain

Example

Admin creates domain that contains only can_create_domain permission and Alice account in that domain. Alice can create new domains.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_create_domain]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def create_domain_tx():
    # 'test_role' was created in genesis transaction
    tx = iroha.transaction([
        iroha.command('CreateDomain', domain_id='another-domain', default_role='test_
↪role')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

Grant

can_grant_can_add_my_signatory

Allows role owners grant *can_add_my_signatory* permission.

Related API methods: Grant Permission, Revoke Permission

Example

Admin creates domain that contains only can_grant_can_add_my_signatory permission and two accounts for Alice and Bob in that domain. Alice can grant to Bob and revoke can_add_my_signatory permission.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_grant_can_add_my_signatory]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])))
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_add_my_signatory_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_add_my_signatory)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def revoke_can_add_my_signatory_tx():
    tx = iroha.transaction([
        iroha.command('RevokePermission', account_id=bob['id'], permission=primitive_
↪pb2.can_add_my_signatory)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_grant_can_remove_my_signatory

Allows role owners grant *can_remove_my_signatory* permission.

Related API methods: Grant Permission, Revoke Permission

Example

Admin creates domain that contains only can_grant_can_remove_my_signatory permission and two accounts for Alice and Bob in that domain. Alice can grant to Bob and revoke can_remove_my_signatory permission.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_grant_can_remove_my_signatory]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])))
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_remove_my_signatory_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_remove_my_signatory)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def revoke_can_remove_my_signatory_tx():
    tx = iroha.transaction([
        iroha.command('RevokePermission', account_id=bob['id'], permission=primitive_
↪pb2.can_remove_my_signatory)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_grant_can_set_my_account_detail

Allows role owners grant *can_set_my_account_detail* permission.

Related API methods: Grant Permission, Revoke Permission

Example

Admin creates domain that contains only can_grant_can_set_my_account_detail permission and two accounts for Alice and Bob in that domain. Alice can grant to Bob and revoke can_set_my_account_detail permission.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_grant_can_set_my_account_detail]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])))
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_set_my_account_detail_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_set_my_account_detail)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def revoke_can_set_my_account_detail_tx():
    tx = iroha.transaction([
        iroha.command('RevokePermission', account_id=bob['id'], permission=primitive_
↪pb2.can_set_my_account_detail)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_grant_can_set_my_quorum

Allows role owners grant *can_set_my_quorum* permission.

Related API methods: Grant Permission, Revoke Permission

Example

Admin creates domain that contains only can_grant_can_set_my_quorum permission and two accounts for Alice and Bob in that domain. Alice can grant to Bob and revoke can_set_my_quorum permission.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_grant_can_set_my_quorum]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key']))
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_set_my_quorum_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_set_my_quorum)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def revoke_can_set_my_quorum_tx():
    tx = iroha.transaction([
        iroha.command('RevokePermission', account_id=bob['id'], permission=primitive_
↪pb2.can_set_my_quorum)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_grant_can_transfer_my_assets

Allows role owners grant *can_transfer_my_assets* permission.

Related API methods: Grant Permission, Revoke Permission

Example

Admin creates domain that contains only can_grant_can_transfer_my_assets permission and two accounts for Alice and Bob in that domain. Alice can grant to Bob and revoke can_transfer_my_assets permission.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [
        primitive_pb2.can_grant_can_transfer_my_assets,
        primitive_pb2.can_receive,
        primitive_pb2.can_transfer
    ]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.extend([
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])),
        iroha.command('CreateAsset', asset_name='coin', domain_id='test',␣
↪precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='100.00'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
                      dest_account_id=alice['id'],
                      asset_id='coin#test',
                      description='init top up',
                      amount='90.00')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_transfer_my_assets_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_transfer_my_assets)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

```
39
40
41 @commons.hex
42 def revoke_can_transfer_my_assets_tx():
43     tx = iroha.transaction([
44         iroha.command('RevokePermission', account_id=bob['id'], permission=primitive_
   ↪pb2.can_transfer_my_assets)
45     ], creator_account=alice['id'])
46     irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
47     return tx
```

Peer

can_add_peer

Allows adding peers to the network.

A new peer will be a valid participant in the next consensus round after an agreement on transaction containing "addPeer" command.

Related API method: Add Peer

Example

Admin creates domain that contains only can_add_peer permission and Alice account in that domain. Alice can add new peers.

```
1  admin = commons.new_user('admin@test')
2  alice = commons.new_user('alice@test')
3  iroha = irohalib.Iroha(admin['id'])
4
5
6  @commons.hex
7  def genesis_tx():
8      test_permissions = [primitive_pb2.can_add_peer]
9      genesis_commands = commons.genesis_block(admin, alice, test_permissions)
10     tx = iroha.transaction(genesis_commands)
11     irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
12     return tx
13
14
15 @commons.hex
16 def add_peer_tx():
17     peer_key = irohalib.IrohaCrypto.private_key()
18     peer = primitive_pb2.Peer()
19     peer.address = '192.168.10.10:50541'
20     peer.peer_key = irohalib.IrohaCrypto.derive_public_key(peer_key)
21     tx = iroha.transaction([
22         iroha.command('AddPeer', peer=peer)
```

```
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_remove_peer

Allows removing peers from the network.

Removed peer will not participate in the next consensus round after an agreement on transaction containing "removePeer" command.

Related API method: Remove Peer

Example

Admin creates domain that contains only can_remove_peer permission and Alice account in that domain. Admin adds a second peer. Alice can remove existing peers.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])

peer_key = irohalib.IrohaCrypto.private_key()
peer = primitive_pb2.Peer()
peer.address = '192.168.10.10:50541'
peer.peer_key = irohalib.IrohaCrypto.derive_public_key(peer_key)


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_remove_peer]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(irohalib.Iroha.command('AddPeer', peer=peer))
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def remove_peer_tx():
    peer_key = irohalib.IrohaCrypto.private_key()
    tx = iroha.transaction([
        iroha.command('RemovePeer', public_key=peer.peer_key)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

Role

can_append_role

Allows appending roles to another account.

You can append only that role that has lesser or the same set of privileges as transaction creator.

Related API method: Append Role

Example

Admin creates domian that contains can_append_role and can_add_peer permissions and two accounts for Alice and Bob in that domain. Admin creates the second role that contains only can_add_peer permission. Alice can append role to Bob.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_append_role, primitive_pb2.can_add_peer]
    second_role_permissions = [primitive_pb2.can_add_peer]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.extend([
        iroha.command('CreateRole', role_name='second_role', permissions=second_role_
↪permissions),
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])),
        iroha.command('AppendRole', account_id=alice['id'], role_name='second_role')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def append_role_tx():
    # Note that you can append only that role that has
    # lesser or the same set of permissions as transaction creator.
    tx = iroha.transaction([
        iroha.command('AppendRole', account_id=bob['id'], role_name='second_role')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_create_role

Allows creating a new role within a system.

Possible set of permissions for a new role is limited to those permissions that transaction creator has.

Related API method: Create Role

Example

Admin creates domain that contains only can_create_role permission and Alice account in that domain. Alice can create new roles.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_create_role, primitive_pb2.can_create_
↪domain]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def create_role_tx():
    # You can pick only those permissions that
    # already belong to account of transaction creator.
    role_permissions = [primitive_pb2.can_create_domain]
    tx = iroha.transaction([
        iroha.command('CreateRole', role_name='newrole', permissions=role_permissions)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_detach_role

Allows revoking a role from a user.

Note: Due to a known issue the permission allows to detach any role without limitations https://soramitsu.atlassian.net/browse/IR-1468

Related API method: Detach Role

Example

Admin creates domain that contains only can_detach_role permission and creates Alice account in that domain. Admin has two roles test_role and admin_role. Alice can detach test_role from Admin account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_detach_role]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def detach_role_tx():
    tx = iroha.transaction([
        iroha.command('DetachRole', account_id=admin['id'], role_name='test_role')
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

Signatory

can_add_my_signatory

Hint: This is a grantable permission.

Permission that allows a specified account to add an extra public key to the another specified account.

Related API method: Add Signatory

Example

Admin creates domain that contains only can_grant_can_add_my_signatory permission and two accounts for Alice and Bob in that domain. Alice can grant to Bob can_add_my_signatory permission. Bob can add an extra key to Alice account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])
```

```

@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_grant_can_add_my_signatory]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key'])))
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_add_my_signatory_tx():
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_add_my_signatory)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def add_signatory_tx():
    extra_key = irohalib.IrohaCrypto.private_key()
    tx = iroha.transaction([
        iroha.command('AddSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(extra_key))
    ], creator_account=bob['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, bob['key'])
    return tx
```

can_add_signatory

Allows linking additional public keys to account.

The corresponding command can be executed only for an account of transaction creator and only if that account has a role with the permission.

Related API method: Add Signatory

Example

Admin creates domain that contains only can_add_signatory permission and Alice account in that domain. Alice can add to own account additional keys.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
```

```
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_add_signatory]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def add_signatory_tx():
    extra_key = irohalib.IrohaCrypto.private_key()
    tx = iroha.transaction([
        iroha.command('AddSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(extra_key))
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_remove_my_signatory

Hint: This is a grantable permission.

Permission that allows a specified account remove public key from the another specified account.

See the example (to be done) for the usage details.

Related API method: Remove Signatory

Example

Admin creates domain that contains can_add_signatory and can_grant_can_remove_my_signatory permissions and two accounts for Alice and Bob. Alice grants can_remove_my_signatory permission to Bob and adds additional key to own account. Bob can remove one of Alice's keys.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [
        primitive_pb2.can_grant_can_remove_my_signatory,
```

```
        primitive_pb2.can_add_signatory
    ]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key']))
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_remove_my_signatory_tx():
    extra_key = irohalib.IrohaCrypto.private_key()
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_remove_my_signatory),
        iroha.command('AddSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(extra_key))
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def remove_signatory_tx():
    tx = iroha.transaction([
        iroha.command('RemoveSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(alice['key']))
    ], creator_account=bob['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, bob['key'])
    return tx
```

can_remove_signatory

Allows unlinking additional public keys from an account.

The corresponding command can be executed only for an account of transaction creator and only if that account has a role with the permission.

Related API method: Remove Signatory

Example

Admin creates domian that contains can_remove_signatory permission and Alice account in that domain. Admin adds an extra key to Alice account. Alice can remove one of the keys.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
```

```
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_remove_signatory]
    extra_key = irohalib.IrohaCrypto.private_key()
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('AddSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(extra_key))
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def remove_signatory_tx():
    tx = iroha.transaction([
        iroha.command('RemoveSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(alice['key']))
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

can_set_my_quorum

Hint: This is a grantable permission.

Permission that allows a specified account to set quorum for the another specified account.

Account should have greater or equal amount of keys than quorum.

Related API method: Set Account Quorum

Example

Admin creates domain that contains can_grant_can_set_my_quorum and can_add_signatory permissions and create two accounts for Alice and Bob in that domain. Alice grants to Bob can_set_my_qourum permission and adds an extra key to account. Bob can set quorum for Alice.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
bob = commons.new_user('bob@test')
iroha = irohalib.Iroha(admin['id'])


```

```
@commons.hex
def genesis_tx():
    test_permissions = [
        primitive_pb2.can_grant_can_set_my_quorum,
        primitive_pb2.can_add_signatory
    ]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAccount', account_name='bob', domain_id='test',
                      public_key=irohalib.IrohaCrypto.derive_public_key(bob['key']))
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def grant_can_set_my_quorum_tx():
    extra_key = irohalib.IrohaCrypto.private_key()
    tx = iroha.transaction([
        iroha.command('GrantPermission', account_id=bob['id'], permission=primitive_
↪pb2.can_set_my_quorum),
        iroha.command('AddSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(extra_key))
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def set_quorum_tx():
    tx = iroha.transaction([
        iroha.command('SetAccountQuorum', account_id=alice['id'], quorum=2)
    ], creator_account=bob['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, bob['key'])
    return tx
```

can_set_quorum

Allows setting quorum.

At least the same number (or more) of public keys should be already linked to an account.

Related API method: Set Account Quorum

Example

Admin creates domain that contains only can_set_quorum permission and creates Alice account in that domain. Admin adds an extra key for Alice account. Alice can set quorum equals two.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_set_quorum]
    extra_key = irohalib.IrohaCrypto.private_key()
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('AddSignatory', account_id=alice['id'],
                      public_key=irohalib.IrohaCrypto.derive_public_key(extra_key))
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def set_quorum_tx():
    # Quourum cannot be greater than amount of keys linked to an account
    tx = iroha.transaction([
        iroha.command('SetAccountQuorum', account_id=alice['id'], quorum=2)
    ], creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx
```

8.2.2 Query-related permissions

Account

can_get_all_acc_detail

Allows getting all the details set to any account within the system.

Related API method: Get Account Detail

Example

Admin creates Alice account in a diffrerent domain that has only can_get_all_acc_detail permission. Alice can access details set to Admin account.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
```

```
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_all_acc_detail]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_detail_query():
    query = iroha.query('GetAccountDetail', creator_account=alice['id'], account_
↪id=admin['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_all_accounts

Allows getting account information: quorum and all the details related to the account.

With this permission, query creator can get information about any account within a system.

All the details (set by the account owner or owners of other accounts) will be returned.

Related API method: Get Account

Example

Admin creates Alice account in a different domain that has only can_get_all_accounts permission. Alice can access account information of Admin.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_all_accounts]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_query():
    query = iroha.query('GetAccount', creator_account=alice['id'], account_id=admin[
↪'id'])
```

```
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_domain_acc_detail

Allows getting all the details set to any account within the same domain as a domain of query creator account.

Related API method: Get Account Detail

Example

Admin creates Alice account in the same domain that has only can_get_domain_acc_detail permission. Alice can get details set to Admin account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_domain_acc_detail]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_detail_query():
    query = iroha.query('GetAccountDetail', creator_account=alice['id'], account_
↪id=admin['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_domain_accounts

Allows getting account information: quorum and all the details related to the account.

With this permission, query creator can get information only about accounts from the same domain.

All the details (set by the account owner or owners of other accounts) will be returned.

Related API method: Get Account

Example

Admin creates Alice account in the same domain that has only can_get_domain_accounts. Alice can access account information of Admin.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_domain_accounts]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_query():
    query = iroha.query('GetAccount', creator_account=alice['id'], account_id=admin[
↪'id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_acc_detail

Allows getting all the details set to the account of query creator.

Related API method: Get Account Detail

Example

Admin creates Alice account in the domain that has only can_get_my_acc_detail permission. Alice can get details set to own account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_my_acc_detail]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx
```

```

@commons.hex
def account_detail_query():
    query = iroha.query('GetAccountDetail', creator_account=alice['id'], account_
↪id=alice['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_account

Allows getting account information: quorum and all the details related to the account.

With this permission, query creator can get information only about own account.

All the details (set by the account owner or owners of other accounts) will be returned.

Related API method: Get Account

Example

Admin creates Alice account in the domain that has only can_get_my_account permission. Alice can access own account information.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_my_account]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_query():
    query = iroha.query('GetAccount', creator_account=alice['id'], account_id=alice[
↪'id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Account Asset

can_get_all_acc_ast

Allows getting a balance of assets on any account within the system.

Query response will contain information about all the assets that ever been assigned to an account.

Related API method: Get Account Assets

Example

Admin creates Alice account in a different domain that has only can_get_all_acc_ast permission. Alice can access assets balance on Admin account.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_all_acc_ast]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_assets_query():
    query = iroha.query('GetAccountAssets', creator_account=alice['id'], account_
↪id=admin['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_domain_acc_ast

Allows getting a balance of specified asset on any account within the same domain as a domain of query creator account.

Query response will contain information about all the assets that ever been assigned to an account.

Related API method: Get Account Assets

Example

Admin creates Alice account in the same domain that has only can_get_domain_acc_ast permission. Alice can access assets balance on Admin account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_domain_acc_ast]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_assets_query():
    query = iroha.query('GetAccountAssets', account_id=admin['id'], creator_
↪account=alice['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_acc_ast

Allows getting a balance of specified asset on account of query creator.

Query response will contain information about all the assets that ever been assigned to an account.

Related API method: Get Account Assets

Example

Admin creates Alice account in a domain that has only can_get_my_acc_ast permission. Alice can access assets balance on own account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_my_acc_ast]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
```

```
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_assets_query():
    query = iroha.query('GetAccountAssets', creator_account=alice['id'], account_
↪id=alice['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Account Asset Transaction

can_get_all_acc_ast_txs

Allows getting transactions associated with a specified asset and any account within the system.

Note: Incoming asset transfers will also appear in the query response.

Related API method: Get Account Asset Transactions

Example

Admin creates Alice account in a different domain that has can_get_all_acc_ast_txs, can_receive and can_transfer permissions. Admin issues some amount of coins and transfers them to Alice. Alice can query all transactions related to coins and Admin account.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [
        primitive_pb2.can_get_all_acc_ast_txs,
        primitive_pb2.can_receive,
        primitive_pb2.can_transfer
    ]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    genesis_commands.extend([
        iroha.command('CreateAsset', asset_name='coin', domain_id='first',␣
↪precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#first', amount='300.00'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
```

```
                        dest_account_id=alice['id'],
                        asset_id='coin#first',
                        description='top up',
                        amount='200.00')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_asset_transactions_query():
    query = iroha.query('GetAccountAssetTransactions', creator_account=alice['id'], page_size=10,
                        account_id=admin['id'], asset_id='coin#first')
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_domain_acc_ast_txs

Allows getting transactions associated with a specified asset and an account from the same domain as query creator.

Note: Incoming asset transfers will also appear in the query response.

Related API method: Get Account Asset Transactions

Example

Admin creates Alice in the same domain that has only can_get_domain_acc_ast_txs permission. Admin issues some amount of coins and transfers them to Alice. Alice can query all transactions related to coins and Admin account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_domain_acc_ast_txs]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.extend([
        iroha.command('CreateAsset', asset_name='coin', domain_id='test', precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='500.69'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
                      dest_account_id=alice['id'],
```

```
                        asset_id='coin#test',
                        description='top up',
                        amount='10.00')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_asset_transactions_query():
    query = iroha.query('GetAccountAssetTransactions', account_id=admin['id'],
                        asset_id='coin#test', creator_account=alice['id'], page_
↪size=10)
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_acc_ast_txs

Allows getting transactions associated with the account of query creator and specified asset.

Note: Incoming asset transfers will also appear in the query response.

Related API method: Get Account Asset Transactions

Example

Admin creates Alice account in a domain that has only can_get_my_acc_ast_txs permission. Admin issues some amount of coins and transfers them to Alice. Alice can query all transactions related to coins and own account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_my_acc_ast_txs]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.extend([
        iroha.command('CreateAsset', asset_name='coin', domain_id='test',␣
↪precision=2),
        iroha.command('AddAssetQuantity', asset_id='coin#test', amount='500.69'),
        iroha.command('TransferAsset',
                      src_account_id=admin['id'],
                      dest_account_id=alice['id'],
                      asset_id='coin#test',
```

```
                        description='top up',
                        amount='10.00')
    ])
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_asset_transactions_query():
    query = iroha.query('GetAccountAssetTransactions', creator_account=alice['id'],␣
↪account_id=alice['id'],
                         asset_id='coin#test', page_size=10)
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Account Transaction

can_get_all_acc_txs

Allows getting all transactions issued by any account within the system.

Note: Incoming asset transfer inside a transaction would NOT lead to an appearance of the transaction in the command output.

Related API method: Get Account Transactions

Example

Admin creates Alice account in a different domain that has only can_get_all_acc_txs permiison. Alice can request all the transactions issues by Admin.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_all_acc_txs]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


```

```
@commons.hex
def account_transactions_query():
    query = iroha.query('GetAccountTransactions', creator_account=alice['id'],␣
↪account_id=admin['id'], page_size=10)
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_domain_acc_txs

Allows getting all transactions issued by any account from the same domain as query creator.

Note: Incoming asset transfer inside a transaction would NOT lead to an appearance of the transaction in the command output.

Related API method: Get Account Transactions

Example

Admin creates Alice account in the same domain that has only can_get_domain_acc_txs permission. Alice can request all the transactions issued by Admin.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_domain_acc_txs]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_transactions_query():
    query = iroha.query('GetAccountTransactions', creator_account=alice['id'],␣
↪account_id=admin['id'], page_size=10)
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_acc_txs

Allows getting all transactions issued by an account of query creator.

Note: Incoming asset transfer inside a transaction would NOT lead to an appearance of the transaction in the command output.

Related API method: Get Account Transactions

Example

Admin creates Alice account in a domain that has only can_get_my_acc_txs permission. Alice can get all transactions issued by own account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_my_acc_txs]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def account_transactions_query():
    query = iroha.query('GetAccountTransactions', creator_account=alice['id'],␣
↪account_id=alice['id'], page_size=10)
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Asset

can_read_assets

Allows getting information about asset precision.

Related API method: Get Asset Info

Example

Admin creates Alice account in a domain that has can_read_assets permissions. Alice can query information about any asset.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_read_assets]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    genesis_commands.append(
        iroha.command('CreateAsset', asset_name='coin', domain_id='test', precision=2)
    )
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def get_asset_query():
    query = iroha.query('GetAssetInfo', asset_id='coin#test', creator_account=alice[
↪'id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Block Stream

can_get_blocks

Allows subscription to the stream of accepted blocks.

Role

can_get_roles

Allows getting a list of roles within the system. Allows getting a list of permissions associated with a role.

Related API methods: Get Roles, Get Role Permissions

Example

Admin creates Alice account in a domain that has can_get_roles permission. Alice can query list of all existing roles. Alice can query list of permissions contained in any role.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_roles]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def get_system_roles_query():
    query = iroha.query('GetRoles', creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query


@commons.hex
def get_role_permissions_query():
    query = iroha.query('GetRolePermissions', creator_account=alice['id'], counter=2,␣
↪role_id='admin_role')
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Signatory

can_get_all_signatories

Allows getting a list of public keys linked to an account within the system.

Related API method: Get Signatories

Example

Admin creates Alice account in a different domain that has only can_get_all_signatories permission. Alice can query a list of public keys related to Admin account.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_all_signatories]
```

```
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def signatories_query():
    query = iroha.query('GetSignatories', creator_account=alice['id'], account_
↪id=admin['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_domain_signatories

Allows getting a list of public keys of any account within the same domain as the domain of query creator account.

Related API method: Get Signatories

Example

Admin creates Alice account in the same domain that has only can_get_domain_signatories permission. Alice can query a list of public keys related to Admin account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_domain_signatories]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def signatories_query():
    query = iroha.query('GetSignatories', creator_account=alice['id'], account_
↪id=admin['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_signatories

Allows getting a list of public keys of query creator account.

Related API method: Get Signatories

Example

Admin creates Alice account in a domain that has only can_get_my_signatories permission. Alice can query a list of public keys related to own account.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_my_signatories]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def signatories_query():
    query = iroha.query('GetSignatories', creator_account=alice['id'], account_
↪id=alice['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Transaction

can_get_all_txs

Allows getting any transaction by hash.

Related API method: Get Transactions

Example

Admin issues several transactions and creates Alice account in a different domain that has only can_get_all_txs permission. Alice (knowing transactions hashes) can query transactions issued by Admin Account.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])

admin_tx1_hash = None
admin_tx2_hash = None


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_all_txs]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def admin_action_1_tx():
    global admin_tx1_hash
    tx = iroha.transaction([
        iroha.command('CreateAsset', asset_name='coin', domain_id='second',␣
↪precision=2)
    ])
    admin_tx1_hash = irohalib.IrohaCrypto.hash(tx)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def admin_action_2_tx():
    global admin_tx2_hash
    tx = iroha.transaction([
        iroha.command('SetAccountDetail', account_id=admin['id'], key='hyperledger',␣
↪value='iroha')
    ])
    admin_tx2_hash = irohalib.IrohaCrypto.hash(tx)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def transactions_query():
    hashes = [
        binascii.hexlify(admin_tx1_hash),
        binascii.hexlify(admin_tx2_hash)
    ]
    query = iroha.query('GetTransactions', tx_hashes=hashes, creator_account=alice['id
↪'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

can_get_my_txs

Allows getting transaction (that was issued by query creator) by hash.

Related API method: Get Transactions

Example

Admin creates Alice account in a different domain. Alice (knowing transactions hashes) issues several transactions. Alice can query own transactions.

```
admin = commons.new_user('admin@first')
alice = commons.new_user('alice@second')
iroha = irohalib.Iroha(admin['id'])

alice_tx1_hash = None
alice_tx2_hash = None


@commons.hex
def genesis_tx():
    test_permissions = [
        primitive_pb2.can_get_my_txs,
        primitive_pb2.can_add_asset_qty,
        primitive_pb2.can_create_asset
    ]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions,␣
↪multidomain=True)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def alice_action_1_tx():
    global alice_tx1_hash
    tx = iroha.transaction([
        iroha.command('CreateAsset', asset_name='coin', domain_id='first',␣
↪precision=2)
    ], creator_account=alice['id'])
    alice_tx1_hash = irohalib.IrohaCrypto.hash(tx)
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
def alice_action_2_tx():
    global alice_tx2_hash
    tx = iroha.transaction([
        iroha.command('AddAssetQuantity', asset_id='coin#first', amount='600.30')
    ], creator_account=alice['id'])
    alice_tx2_hash = irohalib.IrohaCrypto.hash(tx)
    irohalib.IrohaCrypto.sign_transaction(tx, alice['key'])
    return tx


@commons.hex
```

```
def transactions_query():
    hashes = [
        binascii.hexlify(alice_tx1_hash),
        binascii.hexlify(alice_tx2_hash)
    ]
    query = iroha.query('GetTransactions', creator_account=alice['id'], tx_
↪hashes=hashes)
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

Peer

can_get_peers

Allows to request the list of peers in the Iroha network.

Related API method: Get Peers

Example

Admin creates Alice account in any domain that has can_get_peers. Alice can now request the list of peers in the system.

```
admin = commons.new_user('admin@test')
alice = commons.new_user('alice@test')
iroha = irohalib.Iroha(admin['id'])


@commons.hex
def genesis_tx():
    test_permissions = [primitive_pb2.can_get_peers]
    genesis_commands = commons.genesis_block(admin, alice, test_permissions)
    tx = iroha.transaction(genesis_commands)
    irohalib.IrohaCrypto.sign_transaction(tx, admin['key'])
    return tx


@commons.hex
def get_system_peers_query():
    query = iroha.query('GetPeers', creator_account=alice['id'])
    irohalib.IrohaCrypto.sign_query(query, alice['key'])
    return query
```

8.2.3 Supplementary Sources

Listing 8.1: commons.py

```
#
# Copyright Soramitsu Co., Ltd. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
#

import ed25519
import irohalib
import primitive_pb2
import binascii
from time import time

command = irohalib.Iroha.command


def now():
    return int(time() * 1000)


def all_permissions():
    return [
        primitive_pb2.can_append_role,
        primitive_pb2.can_create_role,
        primitive_pb2.can_detach_role,
        primitive_pb2.can_add_asset_qty,
        primitive_pb2.can_subtract_asset_qty,
        primitive_pb2.can_add_peer,
        primitive_pb2.can_add_signatory,
        primitive_pb2.can_remove_signatory,
        primitive_pb2.can_set_quorum,
        primitive_pb2.can_create_account,
        primitive_pb2.can_set_detail,
        primitive_pb2.can_create_asset,
        primitive_pb2.can_transfer,
        primitive_pb2.can_receive,
        primitive_pb2.can_create_domain,
        primitive_pb2.can_read_assets,
        primitive_pb2.can_get_roles,
        primitive_pb2.can_get_my_account,
        primitive_pb2.can_get_all_accounts,
        primitive_pb2.can_get_domain_accounts,
        primitive_pb2.can_get_my_signatories,
        primitive_pb2.can_get_all_signatories,
        primitive_pb2.can_get_domain_signatories,
        primitive_pb2.can_get_my_acc_ast,
        primitive_pb2.can_get_all_acc_ast,
        primitive_pb2.can_get_domain_acc_ast,
        primitive_pb2.can_get_my_acc_detail,
        primitive_pb2.can_get_all_acc_detail,
        primitive_pb2.can_get_domain_acc_detail,
        primitive_pb2.can_get_my_acc_txs,
        primitive_pb2.can_get_all_acc_txs,
        primitive_pb2.can_get_domain_acc_txs,
        primitive_pb2.can_get_my_acc_ast_txs,
        primitive_pb2.can_get_all_acc_ast_txs,
        primitive_pb2.can_get_domain_acc_ast_txs,
        primitive_pb2.can_get_my_txs,
```

```
        primitive_pb2.can_get_all_txs,
        primitive_pb2.can_get_blocks,
        primitive_pb2.can_grant_can_set_my_quorum,
        primitive_pb2.can_grant_can_add_my_signatory,
        primitive_pb2.can_grant_can_remove_my_signatory,
        primitive_pb2.can_grant_can_transfer_my_assets,
        primitive_pb2.can_grant_can_set_my_account_detail
    ]


def genesis_block(admin, alice, test_permissions, multidomain=False):
    """
    Compose a set of common for all tests' genesis block transactions
    :param admin: dict of id and private key of admin
    :param alice: dict of id and private key of alice
    :param test_permissions: permissions for users in test domain
    :param multidomain: admin and alice accounts will be created in
    different domains and the first domain users will have admin right
    by default if True
    :return: a list of irohalib.Iroha.command's
    """
    peer = primitive_pb2.Peer()
    peer.address = '0.0.0.0:50541'
    # ed25519.publickey_unsafe takes and returns a bytes object, while we have hex␣
↪strings
    peer.peer_key = binascii.hexlify(ed25519.publickey_unsafe(binascii.
↪unhexlify(admin['key'])))
    commands = [
        command('AddPeer', peer=peer),
        command('CreateRole', role_name='admin_role', permissions=all_permissions()),
        command('CreateRole', role_name='test_role', permissions=test_permissions)]
    if multidomain:
        commands.append(command('CreateDomain', domain_id='first', default_role=
↪'admin_role'))
    commands.extend([
        command('CreateDomain',
                domain_id='second' if multidomain else 'test',
                default_role='test_role'),
        command('CreateAccount',
                account_name='admin',
                domain_id='first' if multidomain else 'test',
                public_key=irohalib.IrohaCrypto.derive_public_key(admin['key'])),
        command('CreateAccount',
                account_name='alice',
                domain_id='second' if multidomain else 'test',
                public_key=irohalib.IrohaCrypto.derive_public_key(alice['key']))
    ])
    if not multidomain:
        commands.append(command('AppendRole', account_id=admin['id'], role_name=
↪'admin_role'))
    return commands


def new_user(user_id):
    private_key = irohalib.IrohaCrypto.private_key()
    if user_id.lower().startswith('admin'):
        print('K{}'.format(private_key.decode('utf-8')))
    return {
```

```
        'id': user_id,
        'key': private_key
    }


def hex(generator):
    """
    Decorator for transactions' and queries generators.

    Allows preserving the type of binaries for Binary Testing Framework.
    """
    prefix = 'T' if generator.__name__.lower().endswith('tx') else 'Q'
    print('{}{}'.format(prefix, binascii.hexlify(generator().SerializeToString()).
↪decode('utf-8')))
```

8.3 Restarting Iroha node with existing WSV

Previously, in cases when you had to update a node or it shut down for some reason, there was only one option of re-reading all of the blocks to recreate consistent world state view (aka WSV). To start up a node quicker, it is now possible to reuse an existing WSV database after a quick check. For that, `hash` of the top block and the `height` of the blockstorage are included in the WSV.

Warning: It is up to Administrators of the node to make sure the WSV is not edited manually – only by Iroha or the *migration script*. Manual editing or editing of the migration script not following a trustworthy guideline can lead to inconsistent network. Only do so at your own risk (we warned you).

Although it can be a great idea for some of the cases, but please consider that there are certain specifics of reusing WSV, compared to restoring it from blockstorage:

Trust point

Reusing WSV: we need to rely on both blockstorage and WSV.

Restore WSV from block storage: we trust only the genesis block.

Integrity

Reusing WSV: blockstorage and WSV must match each other! Iroha will not check for that.

Restore WSV from block storage: Iroha will check every block, while restoring WSV. Any error in blockstorage will be found (except genesis block, of course). WSV is guaranteed to match the blockstorage.

Time

Reusing WSV: Iroha is almost immediately ready to operate in the network.

Restore WSV from block storage: the larger blockstorage - the longer it takes to restore it and begin operation.

Note: If the local ledger that shut down has more blocks than it should and the correct WSV is among them - it is ok, Iroha will take the WSV of the correct block. If blocks are less than should be – the option of reusing WSV will not

work for you. Please, restore it from blocks.

8.3.1 Enabling WSV Reuse

If you want to reuse WSV state, start Iroha with *–reuse_state* flag. Given this flag, Iroha will not reset or overwrite the state database if it fails to start for whatever reason.

8.3.2 State Database Schema version

When reusing existing WSV, Iroha performs a schema version compatibility check. It will not start or somehow alter the database, if its schema is not compatible with the Iroha in use.

If your schema was created by Iroha of version v1.1.1 or lower, most likely it does not include the version information. In this case you need to add it manually. You are encouraged to use our script for this purpose, it is located here. To forcefully (i.e. without any *migration process*) set your schema version, launch the script with *–force_schema_version* flag and pass the version of Iroha binary that was used to create your schema.

Warning: Before forcefully writing the schema version numbers, double check the version of irohad that created the schema. No checks are performed when you force schema numbers, hence it is easy to break the state database in the future (during the next migration).

8.3.3 Changing Iroha version. Migration.

In case you want to change Iroha version while keeping the WSV, you are encouraged to perform a migration. Although it might be unnecessary (Iroha will refuse to start if the schema is incompatible), as a general rule, we improve the schema with each version and migration might be a good idea for a better performance. You are encouraged to perform a database backup before migration using standard PostgreSQL guidelines for that.

To perform migration, please use our script. It will load the schema information from the database and match it with migration steps (by default, migration scenarios are defined in `migration_data` directory in the same folder as the script). Then it will find all migration paths that will transition your database to the desired version and ask you to choose one.

CHAPTER 9

Contribution

First off, thanks for taking the time to contribute!

The following is a short set of guidelines for contributing to Iroha.

9.1 Table Of Contents

9.1.1 How Can I Contribute?

- *Reporting Bugs*
- *Reporting Vulnerabilities*
- *Suggesting Improvements*
- *Asking Questions*
- *Your First Code Contribution*
- *Pull Requests*

9.1.2 Styleguides

- *Git Style Guide*
- *C++ Style Guide*
- *Documentation Styleguide*

Contact Developers

How Can I Contribute?

Reporting Bugs

Bug is an error, design flaw, failure or fault in Iroha that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.

Bugs are tracked as JIRA issues in Hyperledger Jira.

To submit a bug, create new issue and include these details:

Field	What to enter
Project	Iroha (IR)
Issue Type	Bug
Summary	Essence of the problem
Description	What the issue is about; if you have any logs, please provide them
Priority	You can use Medium though if you see the issue as a high priority, please choose that
Environment	Your OS, device's specs, Virtual Environment if you use one, version of Iroha etc.

Reporting Vulnerabilities

While we try to be proactive in preventing security problems, we do not assume they'll never come up.

It is standard practice to responsibly and privately disclose to the vendor (Hyperledger organization) a security problem before publicizing, so a fix can be prepared, and damage from the vulnerability minimized.

Before the First Major Release (1.0) all vulnerabilities are considered to be bugs, so feel free to submit them as described above. After the First Major Release please utilize a bug bounty program here in order to submit vulnerabilities and get your reward.

In any case — feel free to reach to any of existing maintainers in Rocket.Chat private messages or in an e-mail (check CONTRIBUTORS.md file) if you want to discuss whether your discovery is a vulnerability or a bug.

Suggesting Improvements

An *improvement* is a code or idea, which makes **existing** code or design faster, more stable, portable, secure or better in any other way.

Improvements are tracked as JIRA improvements. To submit new improvement, create new issue and include these details:

Field	What to enter
Project	Iroha (IR)
Issue Type	Improvement
Summary	Essence of the idea
Description	What the idea is about; if you have any code suggestions, you are welcome to add them here
Priority	You can use Medium
Assign	You can assign the task to yourself if you are planning on working on it

Asking Questions

A *question* is any discussion that is typically neigher a bug, nor feature request or improvement. If you have a question like "How do I do X?" - this paragraph is for you.

Please post your question in *your favourite messenger* so members of the community could help you. You can also help others!

Your First Code Contribution

Read our *C++ Style Guide* and start with beginner-friendly issues with JIRA label *‘‘[good-first-issue]‘* <https://jira.hyperledger.org/issues/?jql=project%20%3D%20IR%20and%20labels%20%3D%20good-first-issue%20ORDER%20BY%20updated%20DESC>‘_ <- click here. Indicate somehow that you are working on this task: get in touch with maintainers team, community or simply assign this issue to yourself.

Pull Requests

- Fill in the required template
- End all files with a newline
- **Write tests** for new code. Test coverage for new code must be at least 70%
- Every pull request should be reviewed and **get at least two approvals from maintainers team**. Check who is a current maintainer in MAINTAINERS.md file
- When you’ve finished work make sure that you’ve got all passing CI checks — after that **squash and merge** your pull request
- Follow the *C++ Style Guide*
- Follow the *Git Style Guide*
- **Document new code** based on the *Documentation Styleguide*
- When working with **PRs from forks** check this manual

Styleguides

Git Style Guide

- **Sign-off every commit** with DCO: `Signed-off-by: $NAME <$EMAIL>`. You can do it automatically using `git commit -s`
- **Use present tense** (“Add feature”, not “Added feature”).
- **Use imperative mood** (“Deploy docker to...” not “Deploys docker to...”).
- Write meaningful commit message.
- Limit the first line of commit message to 50 characters or less
- First line of commit message must contain summary of work done, second line must contain empty line, third and other lines can contain list of commit changes

C++ Style Guide

- Use clang-format settings file. There are guides available on the internet (e.g. Kratos wiki)
- Follow CppCoreGuidelines and Cpp Best Practices.
- Avoid platform-dependent code.
- Use C++14.
- Use camelCase for class names and methods, use snake_case for variables.

Documentation Styleguide

- Use Doxygen.
- Document all public API: methods, functions, members, templates, classes...

Places where community is active

Our community members are active at:

Service	Link
RocketChat	https://chat.hyperledger.org/channel/iroha
StackOverflow	https://stackoverflow.com/questions/tagged/hyperledger-iroha
Mailing List	hyperledger-iroha@lists.hyperledger.org
Gitter	https://gitter.im/hyperledger-iroha/Lobby
Telegram	https://t.me/hl_iroha
YouTube	https://www.youtube.com/channel/UCYlK9OrZo9hvNYFuf0vrwww

---

Thank you for reading the document!