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

Hyperledger Iroha community

Sep 12, 2020

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

ONE

IROHA V2.0

The following is a specification for Hyperledger Iroha v2.0.

1.1 1. Overview

Hyperledger Iroha v2 aims to be an even more simple, highly performant distributed ledger platform than Iroha v1. V2 carries on the tradition of putting on emphasis on having a library of pre-defined smart contracts in the core, so that developers do not have to write their own code to perform many tasks related to digital identity and asset management.

1.1.1 1.1. Relationship to Hyperledger Fabric, Hyperledger Sawtooth, Hyperledger Besu, and Others

It is our vision that in the future Hyperledger will consist less of disjointed projects and more of coherent libraries of components that can be selected and installed in order to run a Hyperledger network. Towards this end, it is the goal of Iroha to eventually provide encapsulated components that other projects (particularly in Hyperledger) can use.

1.1.2 1.2. Mobile and web libraries

Having a solid distributed ledger system is not useful if there are no applications that can easily utilize it. To ease use, we created and opened sourced software development kits for iOS, Android, and JavaScript. Using these libraries, cryptographic public/private key pairs that are compatible with iroha can be created and common API functions can be conveniently called.

1.2 2. System architecture

1.2.1 2.1. P2P Network

Generally, $3f+1$ nodes are needed to tolerate f Byzantine nodes in the network (albeit some consensus algorithms may have higher node requirements). The number of f that a system should be made to tolerate should be determined by the system maintainer, based on the requirements for expected use cases.

The following node types are considered:

- Client
- Validating peers (participate in consensus)

- Normal peer (receives and relays blocks, but does not participate in consensus; however, normal peers do validate all received data, with the mantra of *don't trust, verify*)

1.2.2 2.2. Membership service

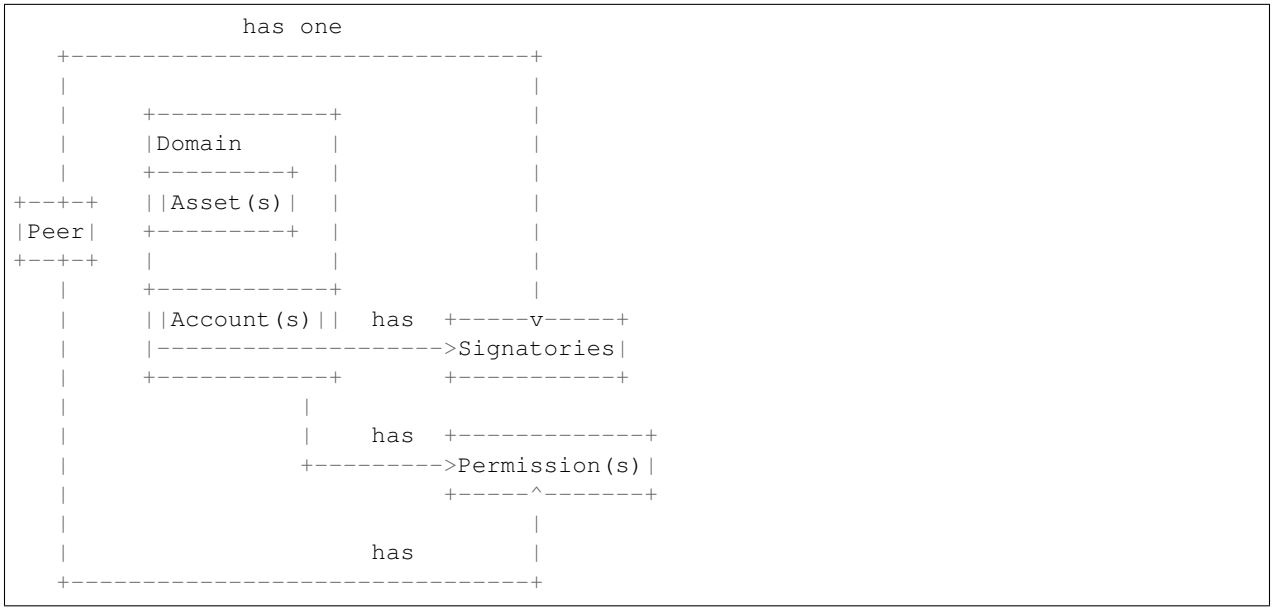
Membership is provided in a decentralized way, on ledger. By default $2f+1$ signatures are needed to confirm adding or removing nodes to or from the network. If nodes drop out and are unresponsive for "too long," then peers will automatically execute a remove peer smart contract to remove the peer from the consensus process.

1.2.3 2.3. Cryptography

We use Hyperledger Ursa.

1.2.4 2.4. Data Model

Iroha uses a simple data model made up of domains, peers, accounts, assets, signatories, and permissions, as shown in the figure below:



1.2.5 2.5. Smart Contracts

Iroha provides a library of smart contracts called **I**roha **S**pecial **I**nstructions (ISI). To execute logic on the ledger, these smart contracts can be invoked via either transactions or registered event listeners.

These include the following smart contracts to support asset management use cases:

- swapAssets
- invoiceAsset

Primitives:

- RegisterDomain(Name, *permissions*)
- RegisterAsset(Name, PrecisionDecimals, *quantity*)

- Add(WhatAsset, WhatAmount, WhereToAdd)
- Sub(WhatAsset, WhatAmount, WhereToSub)
- CreateAccount(Name, *permissions*)
- Batch(*)
  - Batch executes a list of instructions atomically and in order. Batches must always conserve assets (meaning that you cannot have more of an asset after executing a batch). If for for reason you want to mint an asset inside a batch, use *BatchUnconserved*.
- BatchUnconserved(*)
  - Works the same as Batch, above, but allows minting of assets (meaning that more of an asset can exist after executing a batch, if permissions exist for minting).
- GroupAssets(Alias, WhatAsset1, WhatAmount1, . . . , WhatAssetN, WhatAmountN)
  - Groups a set of assets and amounts into one atomic group that can be transfered together.
- DisbandGroup(Alias)
- StoreBlob(bytes)
  - Can be used to store arbitrary data in the blockchain.
- AddValidatingPeer(peerIP, peerPublicKey)
  - Adds a peer to the consensus process.
- RemoveValidatingPeer(peerIP, peerPublicKey)
  - Removes a peer from the consensus process.

Compound Instructions

Transfer = Batch(Sub(WhatAsset, WhatAmount, Account1) & Add(WhatAsset, WhatAmount, Account2)) NOTE: Should probably inline Transfer as a primitive because it is going to be one of the most common instructions that is executed.

SwapAssets = Batch(Sub(WhatAsset1, WhatAmount1, Account1) & Sub(WhatAsset2, WhatAmount2, Account2) & Add(WhatAsset2, WhatAmount2, Account1) & Add(WhatAsset1, WhatAmount1, Account2))

AddXYKExchangePair(asset1, asset2) = CreateAccount(Asset1Asset2Pair, Permissions(Permissions.Ownership.PoolShares))

The following are still WIP:

AddExchLimitBid = Batch(CreateEventTrigger(Event1, *condition*=bidPrice<=Exists(askPrice), *action*= SwapAssets(askAsset, bidAsset)) & CreateAccount(Account2, Permissions(Permissions.OwnedByEvent(Event1))) & Sub(Asset, Amount, Account1) & Add(Asset, Amount, Pool))

- AddOraclizedPair(mintableAsset, mintableAsset, oracle)
- XYKTrade(Asset)
- AddXYKLiquidity(asset, asset)
- ChainedTrade(inputAsset, outputAsset) # Does paythingfinding to get output; oraclized inputs/outputs are special
- addLimitAsk
- removeLimitBid
- removeLimitAsk

- DepositBTC
- DepositERC20
- DepositKusama
- DepositCBDC(cb_id)
- serveHttps(port)

Additionally, the following two transaction types take as input (i.e., "wrap") one of the above transaction types:

- Multisignature
- Interledger (i.e., cross-chain)
- Conditional Multisignature

Conditional Multisignature:

```
[condition,
   [signatory_set, ...],
    ...
]
```

where `signatory_set` is an m-of-n signatory set. If multiple `signatory_sets` exist for a given condition, they can be either `OR` or `AND` unioned.

As an example illustrating why conditional multisig is useful, consider a situation where a bank wants to allow either 2 tellers or 1 manager to sign off on any transfer transaction over $500 and under $1000. In this case, the condition will be: `Condition.asset("usd@nbc").qty(500).comparison(">").qty(1000).comparison("<")` and the `signatory_sets` for the tellers and manager will be `OR` unioned, so that either the m-of-n signatues from the tellers or the single signature from the manager will be acceptable for transaction signing.

1.2.6 2.5.1 Event Listeners

Triggers can be either based on time or on a confirmed transaction (event). This is very powerful and enables Turing complete computation for Iroha's smart contracts.

- Trigger at timestamp
- Trigger at blockchain
- Trigger at *condition*

1.2.7 2.6. Transactions

Transactions are used to either invoke a smart contract or store an event listener.

2.6.1 Transaction Data Structure

The data structure for transactions follows the interpreter pattern and is very simple:

(TODO)

2.6.2 Transaction Cache (Priority Queue)

(Tabu search for optimization of settlements in the priority queue)(TODO)

1.2.8 2.8. Data storage

Data in Hyperledger Iroha v2 are stored in two places: a block store (disk) and a a world-state-view (in memory).

To reach the performance targets, Iroha v2 does not use a database to store data, but instead implements a custom storage solution, called **Kura**, that is specially designed for storing and validating blockchain data. One of the goals of Kura is that even without multiple-peer consensus (e.g., when running a blockchain as a single node), transactional data can still be processed using tamper-evident cryptographic proofs.

When Kura is initialized, data are read from the on-disk block store using either of two methods (depending on the config settings): `fastInit` or `strictInit`. `fastInit` reads all transactions in all blocks in order and recreates all the in-memory hashmaps, but without doing any validation. `strictInit` validates that all transactions and blocks have correct signatures and that all transactions follow the business rules (e.g., no accounts should have a negative balance).

Additionally, an auditor thread can be spawned that will go through the block store strictly and compare the result with the world-state-view, in case the server suspects that something has been compromised and wants to check.

Kura takes as input blocks, which comprise multiple transactions. Kura is meant to take only blocks as input that have passed stateless and stateful validation, and have been finalized by consensus. For finalized blocks, Kura simply commits the block to the block storage on the disk and updates atomically the in-memory hashmaps that make up the key-value store that is the world-state-view. To optimize networking syncing, which works on 100 block chunks, chunks of 100 blocks each are stored in files in the block store.

Kura also helps out with stateful validation, by providing functions that retrieve a copy of values affected in the world-state-view by the transactions in a block, returning the values as a copy. This then allows the stateful validation component to apply the transactions to update the world-state-view and confirm that no transactions in the block violate business rule invariants (e.g., no account shall have a negative balance of an asset after a transaction).

Kura uses the in-memory hashmap structures to also store information, such as the latest input/output transactions for an account and for an asset, in order to simplify the query API and allow real-time querying of Iroha v2 directly, without requiring end-user applications to rely on middleware. To confirm that transactions are indeed correct, Merkle proofs are also stored with and readily available from Kura.

Merkle tree structure

todo

1.2.9 2.9. Consensus

Byzantine fault tolerant systems are engineered to tolerate f numbers of Byzantine faulty nodes in a network. Iroha introduces a Byzantine Fault Tolerant consensus algorithm called Sumeragi. It is heavily inspired by the B-Chain algorithm:

Duan, S., Meling, H., Peisert, S., & Zhang, H. (2014). *Bchain: Byzantine replication with high throughput and embedded reconfiguration*. In International Conference on Principles of Distributed Systems (pp. 91-106). Springer.

As in B-Chain, we consider the concept of a global order over validating peers and sets **A** and **B** of peers, where **A** consists of the first $2f+1$ peers and **B** consists of the remainder. As $2f+1$ signatures are needed to confirm a transaction, under the normal case only $2f+1$ peers are involved in transaction validation; the remaining peers only join the validation when faults are exhibited in peers in set **A**. The $2f+1$th peer is called the *proxy tail*.

Sumeragi is a Byzantine fault tolerant consensus algorithm for permissioned, peer-to-peer networks that try to reach consensus about some set of data.

The Basics

- no ordering service; instead, the leader of each round just uses the transactions they have at hand to create a block and the leader changes each round to prevent long-term censorship
- $3f+1$ validators that are split into two groups, a and b, of $2f+1$ and f validators each
- $2f+1$ validators must sign off on a block in order for it to be committed
- the first node in set a is called the *leader* (sumeragi) and the $2f+1$th node in set a is called the *proxy tail*
- the basic idea is that up to f validators can fail and the system should run, so if there are f Byzantine faulty nodes, you want them to be in the b set as much as possible
- empty blocks are not produced, so to prevent an evil leader from censoring transactions and claiming there are no transactions to create blocks with, everytime a node sends a transaction to the leader, the leader has to submit a signed receipt of receiving it; then, if the leader does not create a block in an orderly amount of time (the *block time*), the submitting peer can use this as proof to convince non-faulty nodes to do a view change and elect a new leader
- after a node signs off on a block and forwards it to the *proxy tail*, they expect a commit message within a reasonable amount of time (the *commit time*); if there is no commit message in time, the node tries to convince the network to do a view change, which creates a new leader and proxy tail by shifting down the $2f+1$ nodes used by 1 to the right
- once a commit message is received from the proxy tail, all nodes commit the block locally; if a node complains that they never received the commit message, then a peer that has the block will provide that peer with the committed block (note: there is no danger of a leader creating a new block while the network is waiting for a commit message because the next round cannot continue nor can a new leader be elected until after the current round is committed or leader election takes place)
- every time there is a problem, such as a block not being committed in time, both the leader and the proxy tail are changed; this is becaues we want to just move on and not worry about assigning blame, which would come with considerable overhead
- $2f+1$ signatures are needed to commit, $f+1$ are needed to change the leader and proxy tail

The Details

Network Topology

A network of nodes is assumed, where each node knows the identity of all other nodes on the network. These nodes are called *validators*. We also assume that there are $3f+1$ validators on the network, where f is the number of simultaneous Byzantine faulty nodes that the network can contain and still properly function (albeit, performance will degrade in the presence of a Byzantine faulty node, but this is okay because Hyperledger Iroha is designed to operate in a permissioned environment).

Because the identities of the nodes are known by all and can be proven through digital signatures, it makes sense to overlay a topology on top of the network of nodes in order to provide guarantees that can enable consensus to be reached faster.

For each round (e.g., block), the previous round's (block's) hash is used to determine an ordering over the set of nodes, such that there is a deterministic and canonical order for each block. In this ordering, the first $2f+1$ nodes are grouped into a set called set a. Under normal (non-faulty) conditions, consensus for a block is performed by set a. The remaining f nodes are grouped into a set called set b. Under normal conditions, set b acts as a passive set of validators to view and receive committed blocks, but otherwise they do not participate in consensus.

Data Flow: Normal Case

Assume the leader has at least one transaction. The leader creates a block either when the *block timer* goes off or its transaction cache is full. The leader then sends the block to each node in set a. Each peer in set a then validates and signs the block, and sends it to the proxy tail; after sending it to the proxy tail, each non-leader node in set a also sends the block to each node in set b, so they can act as observers on the block. When each node in set a sends the block to the proxy tail, they set a timer, the *commit timer*, within which time the node expects to get a commit message (or else it will suspect the proxy tail).

The proxy tail should at this point have received the block from at least one of the peers. From the time the first peer contacts the proxy tail with a block proposal, a timer is set, the *voting timer*. Before the *voting timer* is over, the proxy tail expects to get $2f$ votes from the other nodes in set a, to which it then will add its vote in order to get $2f+1$ votes to commit the block.

Handling Faulty Cases

Possible faulty cases related to the leader are:

- leader ignores all transactions and never creates a block
  - the solution to this is to have other nodes broadcast a transaction across the network and if someone sends a transaction to the leader and it gets ignored, then the leader can be suspected; the suspect message is sent around the network and a new leader is elected if $f+1$ nodes cannot get a reply from the leader for any transaction
- leader creates a block, but only sends it to a minority of peers, so that $2f+1$ votes cannot be obtained for consensus
  - the solution is to have a *commit timer* on each node where a new leader will be elected if a block is not agreed upon; the old leader is then moved to set b
- leader creates multiple blocks and sends them to different peers, causing the network to not reach consensus about a block
  - the solution is to have a *commit timer* on each node where a new leader will be elected if a block is not agreed upon; the old leader is then moved to set b

- the leader does not put *commit timer* block invalidation information, where applicable
  - the non-faulty nodes that see the block without block invalidation when required will not vote for a block

Possible faulty cases related to the proxy tail are:

- proxy tail received some votes, but does not receive enough votes for a block to commit
  - the *commit timer* on regular nodes or the *voting timer* on the proxy tail will go off and a new leader and proxy tail are elected
- proxy tail receives enough votes for a block, but lies and says that they didn't
  - the *commit timer* on nodes will go off and a new leader and proxy tail are elected
- proxy tail does not inform any other node about a block commit (block withholding attack)
  - the *commit timer* on nodes will go off and a new leader and proxy tail are elected; the signatures from at least $f+1$ nodes saying their *commit timer* goes off, invalidates a block hash forever; this invalidation is written in the next block created successfully, to prevent arbitrary rewriting of history in the future
- proxy tail does not inform set b about a block commit
  - through normal data synchronization (P2P gossip), set b will get up to date
- proxy tail selectively sends a committed block to some, but not other nodes
  - the *commit timer* on nodes will go off and a new leader and proxy tail are elected; the signatures from at least $f+1$ nodes saying their *commit timer* goes off, invalidates a block hash forever; this invalidation is written in the next block created successfully, to prevent arbitrary rewriting of history in the future

Possible faulty cases related to any node in set a are:

- a peer could delay signing on purpose so they slow down consensus, without witholding their signature
  - this is not very nice, but it is also hard to prove; the Hijiri reputation system can be used to lower the reputation of slow nodes anyway
- a peer may not sign off on a block
  - if the lack of a signature causes a block to not commit, a new node will be brought in from set b
- a peer may make a false claim that their *voting timer* went off
  - $f+1$ *voting timer* claims are required to make a block invalid and change the leader and proxy tail
- a peer may make a leader suspect claim
  - $f+1$ claims are needed to change a leader, so just one node is not enough; non-faulty nodes will not make false claims

1.2.10 2.10. Data synchronization and retrieval

When nodes gossip to each other, they include the latest known block hash. If a receiving node does not know about this block, they will then request data.

1.2.11 2.11. Data permissions

Data permissioning is crucial to many real use cases. For example, companies will not likely accept distributed ledger technology if it means that competing institutions will know the intricate details of transactions.

1.2.12 2.12. Hijiri: Peer reputation system

The hijiri reputation system is based on rounds. At each round, validating peers that are registered with the membership service perform the following tasks to establish trust (reliability) ratings for the peers:

- data throughput test
- version test
- computational test
- data consistency test

Which peers validate each other are based on the pairwise distance between hashes (e.g., `sort(abs(hash && 0x0000ffff - publicKey && 0x0000ffff))`). The hashes are computed based on the public keys of the peers that are concatenated with the round number and then SHA-3 hashed. Rounds occur whenever the Merkle root is less than TODO:XXX. Results are shared in a separate Merkle tree, maintained independently of the transactions (so the systems can run in parallel).

1.2.13 2.13. Channels

1.3 3.0 Queries

- getTotalQtyEverCreated(assetId)
- getTotalQtyEverDeleted(assetId)
- getAssetQty(assetId)
- getTransactions(accountId)

This is a paged query that returns up to the last 100 transactions.

- getTransactions(accountId, startingIndex)

This is a paged query that returns up to the last 100 transactions, starting at the starting index, and returning in descending order temporally.

- getTransactions(accountId, startingIndex, endingIndex)

1.4 4.0 Performance Goals

- 20,000 tps
- 2-3 s block time

1.5 Appendix

1.5.1 A.1. Developing for Hyperledger Iroha

todo

CHAPTER

TWO

GUIDES

2.1 How to Develop a New Iroha Module

When you need to add some functionality to Iroha, use this guide to develop a new Iroha Module.

2.1.1 Prerequisites

- Rust
- Text Editor or IDE

2.1.2 Steps

1. Create new Rust module inside Iroha crate

Inside `iroha/src/lib.rs` add a declaration of your new module. For example, for `bridge` module we add the following declaration,

```
#[cfg(feature = "bridge")]
pub mod bridge;
```

so for you module `x` you would add `pub mod x;`. You should also place your new module under the Cargo feature so other developers would be able to turn it on and off when needed.

Now, create a separate file for your module. For `bridge` module it will be `iroha/src/bridge.rs`. Likewise, for your module `x` you will need to create a new file `iroha/src/x.rs`.

2. Add documentation

Each module must provide description of its own functionality via Rust Docs.

For that, at the beginning of the module file you should place docs block for the enclosing item.

```
//! Here you can see a good description of the module `x` and its functionality.
```

All public entites of your module should be documented as well. But first, let's create them.

3. Write your logic

The development of a new Iroha Module has a goal - to bring new functionality to Iroha. So based on the goal and requirements, you have you will introduce new entities and place them inside newly created module.

Let's specify particular categories of such entities and look how they can be implemented according to Iroha best practices.

4. Add custom Iroha Special Instruction

If you need to have some module-related Iroha Special Instructions you should add `isi` submodule to the file of your newly created module, like that:

```
...
pub mod isi {
}
```

Inside this submodule you may declare new Iroha Special Instructions. To provide safety guarantees, Iroha Modules can create new Iroha Special Instructions composed of the Out of the Box Instructions.

Let's look at the example from the `bridge` Iroha Module:

```
...
pub fn register_bridge(&self, bridge_definition: BridgeDefinition) -> Instruction {
    let seed = crate::crypto::hash(bridge_definition.encode());
    let public_key = crate::crypto::generate_key_pair_from_seed(seed)
        .expect("Failed to generate key pair.")
        .0;
    let domain = Domain::new(bridge_definition.id.name.clone());
    let account = Account::new("bridge", &domain.name, public_key);
    Instruction::If(
        Box::new(Instruction::ExecuteQuery(IrohaQuery::GetAccount(
            GetAccount {
                account_id: bridge_definition.owner_account_id.clone(),
            },
        ))),
        Box::new(Instruction::Sequence(vec![
            Add {
                object: domain.clone(),
                destination_id: self.id.clone(),
            }
            .into(),
            Register {
                object: account.clone(),
                destination_id: domain.name,
            }
            .into(),
            Mint {
                object: (
                    "owner_id".to_string(),
                    bridge_definition.owner_account_id.to_string(),
                ),
                destination_id: AssetId {
                    definition_id: owner_asset_definition_id(),
                    account_id: account.id.clone(),
                },
            }
```

(continues on next page)

(continued from previous page)

```
            .into(),
            Mint {
                object: (
                    "bridge_definition".to_string(),
                    format!("{:?}", bridge_definition.encode()),
                ),
                destination_id: AssetId {
                    definition_id: bridge_asset_definition_id(),
                    account_id: account.id,
                },
            }
            .into(),
        ])),
        None,
    )
}
...
```

And see what it does to register a new Bridge:

1. Check that Bridge's Owner's Account exists and terminate execution if not.
2. Add new Domain.
3. Register new Account.
4. Mint one Asset.
5. Mint another Asset.

We will not discuss Bridge-related terminology here – the thing we want to look at is how we can compose these steps into one new Iroha Special Instruction.

As you can see, we have `Instruction::If(...)` here - it's the utility Iroha Special Instruction. It takes three arguments - `condition`, `instruction_to_do_if_true`, `instruction_to_do_if_false_or_nothing`. By this instruction we've made the first step of our algorithm - run a check and terminated execution if there is no Owner's Account. Inside `condition` we placed `Instruction::ExecuteQuery(...)` which fails if Iroha Query fails.

If the first step succeeds, we should move forward and execute sequence of the following steps. For this purpose we also have a utility Iroha Special Instruction `Sequence` with a vector of Iroha Special Instructions executed one by one.

Inside this sequence we use domains-related Iroha Special Instructions `Add`, `Register`, and `Mint` twice.

2.1.3 Additional resources

- //TODO: add link to the pair programming session on `Bridge` module.

2.2 How to Monitor Iroha Peer

When you need to monitor state and work of Iroha peer, use this guide.

2.2.1 Prerequisites

- Iroha CLI

2.2.2 Steps

1. Run CLI Command to check Peer's Health

```
./iroha_client_cli maintenance health
```

2. Check Output

If output is `Health is Healthy` then peer is in a good state.

3. Run CLI Command to scrape Peer's Metrics

```
./iroha_client_cli maintenance metrics
```

4. Check Output

Output will contain information about peer's metrics:

```
Metrics { cpu: Cpu { load: Load { frequency: "Ok(CpuFrequency { current: 1204494750 s^
↪-1, min: Some(800000000 s^-1), max: Some(3700000000 s^-1) })", stats: "Ok(CpuStats
↪{ ctx_switches: 420120348, interrupts: 88638100 })", time: "Ok(CpuTime { user:␣
↪17592.36 s^1, system: 6387.2 s^1, idle: 66334.01 s^1 })" } }, disk: Disk { block_
↪storage_size: 0, block_storage_path: "./blocks" }, memory: Memory { memory:
↪"Ok(Memory { total: 7972520000, available: 1874804000, free: 599556000 })", swap:
↪"Ok(Swap { total: 16777212000, used: 5232588000, free: 11544624000 })" } }
```

2.2.3 Conclusion

CLI Client or custom solution using `iroha-client` library can easily check Iroha peers helth and metrics.

2.3 How to Write a Custom Iroha Special Instruction

When you need to add new instruction to Iroha, use this guide to write custom Iroha Special Instruction.

2.3.1 Prerequisites

- Rust
- Text Editor or IDE

2.3.2 Steps

1. Declare your intention

Iroha Special Instruction is a high level representation of changes to the World State View. They have more imperative instructions under the hood, while you may concentrate on the business logic. Let's take an example the following User's Story for `Bridge` module:

```
Feature: Bridge feature
  Scenario: Owner registers Bridge
    Given Iroha Peer is up
    And Iroha Bridge module enabled
    And Iroha has Domain with name company
    And Iroha has Account with name bridge_owner and domain company
    When bridge_owner Account from company domain registers Bridge with name polkadot
    Then Iroha has Domain with name polkadot
    And Iroha has Account with name bridge and domain polkadot
    And Iroha has Bridge Definition with name polkadot and kind iclaim and owner␣
↪bridge_owner
    And Iroha has Asset with definition bridge_asset in domain bridge and under␣
↪account bridge in domain polkadot
```

2. Extract the algorithm

As you can see - **Then** section contains expected output of our **When** instruction.

Let's look at it from another perspective and instead of **Then** use **Do**

```
Register polkadot domain
Register bridge account under polkadot domain
Register Bridge Definition polkadot with kind iclaim and bridge_owner owner
Mint Asset with definition bridge_asset in domain bridge under account bridge in␣
↪domain polkadot
```

This representation looks more like an algorithm and can be used to compose several *out-of-the-box instructions* into a new *custome Iroha special instruction*.

3. Write your own instruction

Now let's write some code:

```
    /// Constructor of Iroha Special Instruction for bridge registration.
    pub fn register_bridge(
        peer_id: <Peer as Identifiable>::Id,
        bridge_definition: &BridgeDefinition,
    ) -> Instruction {
        let domain = Domain::new(bridge_definition.id.name.clone());
        let account = Account::new(BRIDGE_ACCOUNT_NAME, &domain.name);
        Instruction::If(
            Box::new(Instruction::ExecuteQuery(IrohaQuery::GetAccount(
                GetAccount {
                    account_id: bridge_definition.owner_account_id.clone(),
                },
            ))),
            Box::new(Instruction::Sequence(vec![
                Add {
                    object: domain.clone(),
                    destination_id: peer_id,
                }
                .into(),
                Register {
                    object: account.clone(),
                    destination_id: domain.name,
                }
                .into(),
                Mint {
                    object: (
                        BRIDGE_ASSET_BRIDGE_DEFINITION_PARAMETER_KEY.to_string(),
                        bridge_definition.encode(),
                    ),
                    destination_id: AssetId {
                        definition_id: bridge_asset_definition_id(),
                        account_id: account.id,
                    },
                }
                .into(),
                Mint {
                    object: (
                        bridge_definition.id.name.clone(),
                        bridge_definition.encode(),
                    ),
                    destination_id: AssetId {
                        definition_id: bridges_asset_definition_id(),
                        account_id: bridge_definition.owner_account_id.clone(),
                    },
                }
                .into(),
                // TODO: add incoming transfer event listener
            ])),
            Some(Box::new(Instruction::Fail(
                "Account not found.".to_string(),
            ))),
        )
    }
```

Using a sequence of Iroha Special Instructions we compose existing functionality into a new one.

2.3.3 Additional resources

//TODO add additional references

CHAPTER

THREE

REFERENCES

3.1 Glossary

//TODO: add links to docs.rs

Definitions of all Iroha-related entities can be found here.

3.1.1 DEX

A decentralized exchange (DEX) is a marketplace for cryptocurrencies or blockchain investments that is totally open sourced. Nobody is in control at a DEX, instead buyers and sell deal with each other on a one-on-one basis via peer-peer (P25) trading applications. source

In Iroha DEX represented as a *module* with a set of Iroha Special Instructions and Queries.

Order

Is a proposal to transfer (to or from) some *assets* inside Iroha implemented via *trigger*.

3.1.2 Iroha Query

Iroha read-only request to the *World State View*.

3.1.3 Iroha Special Instruction

Iroha provides a library of smart contracts called Iroha Special Instructions (ISI). To execute some logic on the ledger, these smart contracts can be invoked via either transactions or registered event listeners.

Out of the Box Iroha Special Instruction

Iroha provides several basic Instructions for utility purposes or domain-related functionality out-of-the-box:

Utility Iroha Special Instruction

This set contains logical instructions like `If`, I/O related like `Notify` and compositions like `Sequence`. They are mostly used by *custom Instructions*.

Domain-related Iroha Special Instruction

This set contains domain-related instructions (asset/account/domain/peer) and provides the opportunity to make changes to the *World State View* safely.

Custom Iroha Special Instruction

These Instructions provided by *Iroha Modules*, clients or 3rd parties. They can only be build on top of *the Out of the Box Instructions*.

3.1.4 World State View

In-memory representation of the current blockchain state.

3.1.5 Trigger

Triggers are Iroha Special Instructions registered on *peer*. Their execution depends on some conditions, for example on the blockchain height, time or *query* result.

3.2 Iroha Special Instructions

Because Iroha Special Instructions is a very important topic in Iroha functionality we have made this reference with detailed description of every aspect related to **ISI**.

3.2.1 TL;DR

In Iroha 2.0 we have Iroha Special Instructions (inspired by commands from Iroha 1 and smart contracts in other systems). They are provided out-of-the-box for all possible actions and contain implementations with Permissions check inside. But we have several Instructions for composition like 'If', 'Sequence', etc. Therefore Iroha modules and other applications are able to compose their custom complex commands on top of out-of-the-box Iroha Special Instructions with the help of these composition oriented instructions. Permissions are implemented in form of assets with possibility to check action and domain this action is applied to.

CHAPTER

FOUR

TUTORIALS

4.1 Minting your first Asset

When you use Iroha in your application you will definitely face the need to mint an asset. Use this tutorial to master Iroha Special Instructions in general and `MintAsset` in particular.

4.1.1 Prerequisites

- Rust
- Text Editor or IDE
- Existing Rust project //TODO: create some sandbox project

4.1.2 Steps

1. Add dependency on Iroha crate

Inside `path_to_your_project/Cargo.toml` add a new dependency:

```
iroha = "2.0.0"
```

2. Find a place where you need to mint an asset

Usually any business operation will have a need to mint an asset:

- Uploading of digital documents
- Withdrawl and initial supply of crypto currencies
- Etc.

Look at your project and find a good place to put it, which will encapsulate Iroha related logic inside and may be easily modified if needed in future.

3. Use Iroha CLI Client to prepare Iroha Peer

Iroha Special Instructions executed on behalf of an authority - Account.

If you already has an account to store assets on - feel free to skip these step. If not - you will need to receive Account's Key Pair with permissions to Register an Account.

TL;DR - after configuration of Iroha CLI run this command:

```
./iroha_client_cli account register --domain="my_domain" --name="my_account" --key="
↪{account_public_key}"
```

4. Construct Iroha Special Instruction

Now we can write some real code, let's imagine that you need to mint 200 amount of crypto currency "xor":

```
let mint_asset = isi::Mint {
    object: 200,
    destination_id: AssetId {
        definition_id: "xor#soramitsu",
        account_id: "my_account@my_domain",
    },
};
```

And let's see what it consist of:

- `isi` module contains basic Iroha special instructions functionality
- `Mint` structure is a declaration - "Iroha - Mint this object to the following destination"
- `object` in our case is an amount of "xor" to mint. In other cases it can be bytes of digital document or other data.
- `destination_id` in our case is an asset's identification which consist of a asset's definition identification and account's identification cross product.

This functionality also available via Iroha CLI Client but we did it via Rust code on purpose.

5. Submit Iroha Special Instruction

```
iroha_client
  .submit(mint_asset.into())
  .await
  .expect("Failed to mint an asset.");
```

`iroha_client` provides functionality to submit iroha special instructions and it will be automatically "pack" them into transaction signed on behalf of the client. As a result of this operation you will receive transaction acceptance status - if transaction was accepted by the peer (signature was valid and payload is legal set of iroha special instructions) then it will be `Result::Ok(())`, if some problems arrive - it will be `Result::Err(String)` with textual error message.

4.1.3 Conclusion

It is necessary to be aware of Iroha domain model and set of out-of-the-box Iroha Special Instructions and Queries to develop custom projects based on it. But once you understand this model, it provides very simple and clean approach to declaratively define desired steps and send them for execution.

4.1.4 Further reading

CHAPTER

FIVE

CONTRIBUTING GUIDE

First off, thanks for taking the time to contribute!

The following is a short set of guidelines for contributing to Iroha.

5.1 How Can I Contribute?

5.1.1 TL;DR

- Find Jira
- Write Tests
- `cargo test && cargo bench && cargo fmt -all && cargo clippy`
- `git pull -r origin iroha2-dev && git commit -s`

5.1.2 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:

5.1.3 Reporting Vulnerabilities

While we try to be proactive in preventing security problems, we do not assume they will 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 of questions feel free to reach to any of existing maintainers in Rocket.Chat private messages.

5.1.4 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:

5.1.5 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!

5.1.6 Your First Code Contribution

Read our *Rust Style Guide* and start with beginner-friendly issues with JIRA label good-first-issue. Indicate somehow that you are working on this task: get in touch with maintainers team, community or simply assign this issue to yourself.

5.1.7 Pull Requests

- Fill in the required template
- **Write tests** for new code. Test coverage for new code must be at least 70% and to check coverage use `cargo tarpaulin -v`
- 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 **rebase and merge** your pull request
- Follow the *Rust Style Guide*
- Follow the *Git Style Guide*
- **Document new code** based on the *Documentation Styleguide*
- When working with **PRs from forks** check this manual

5.1.8 Tests and Benchmarks

- To run tests execute `cargo test` command
- To run benchmarks execute `cargo bench` command, if you want to debug output in benchmark, execute `RUSTFLAGS="--cfg debug_assertions" cargo bench` command

5.2 Styleguides

5.2.1 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
- Use Git Rebase Workflow

5.2.2 Rust Style Guide

- Use `cargo fmt --all`
- Do not place code inside `mod.rs` files
- Use domain-first modules structure. For example `domain::isi::*`. Such a way complex uses will be easier to incude in dependent modules.
- Do not use whitespaces or empty lines inside function bodies.
- Put public methods first in your impl blocks.
- Put inner modules after `self` module content, but before `tests` module.
- Prefer to return `Result` instead of panic.
- Use `expect` with explicit error message instead of `unwrap`.

5.2.3 Documentation Styleguide

- Use `Rust Docs`

5.3 Places where community is active

Our community members are active at:

Thank you for reading the document!