Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Staking

Band Protocol Documentation

Introduction

Band Protocol is a cross-chain data oracle platform with the aspiration to build high-quality suites of web3 development products. We provides reliable, secure, and real-time data to smart contracts on various blockchain networks. It was designed to address the growing demand for accurate and timely data for blockchain-based applications.

This documentation provides a comprehensive guide to the BandChain protocol, including its architecture, functionalities, client libraries, and tools. It also covers the best practices and use cases for integrating BandChain into your blockchain applications.

Getting Started

Get familiar with BandChain and explore its main concepts.

Introduction to BandChain

High-level overview of the BandChain

Core Concepts

Learn about the core concepts of BandChain, including tokenomics, gas and fees, and more.

Products

Learn about the products that BandChain offers, including the Standard Dataset and ,VRF, and more.

Developer Guides

Learn how to use Band's integration tools to integrate Band's oracle data into your applications.

Node & Validators

Learn how to run a BandChain node and become a validator to earn rewards.

Development Stack

Check out the docs for the various parts of BandChain's core technical stack.

Cosmos SDK

The blockchain framework powering BandChain

WebAssembly

The standard underpinning BandChain's data oracle script

ReScript

The core language of our **CosmoScan** block explorer

Help & Support

Get in touch with Band Protocol devs and our community.

Devloper Discord

Chat with Band Protocol Developers on Discord.

Telegram Group

Join our Telegram group to get support from our team.

Found an Issue?

Help us improve this page by suggesting edits on GitHub.

Band Protocol Documentation

Introduction

High-Level Overview

The Need for Oracles

The BandChain Oracle

How BandChain Works?

Example Use Cases

Core Concepts

Products

Develop

Nodes & Validators

Staking

High-Level Overview

Band Protocol is a cross-chain data oracle aggregating and connecting real-world data and APIs to smart contracts.

The protocol is built on top of BandChain, a Cosmos-SDK-based blockchain designed to be compatible with most smart contract and blockchain development frameworks.

The network is designed to modularize and offload the heavy and resource-intensive tasks (i.e., fetching data from external sources aggregating them) from the smart contract platforms onto itself. This not only prevents such tasks from congesting or causing high transaction fees on the destination network, but the same data points can be packaged, used, and verified efficiently across multiple blockchains.

Its flexible design allows developers to query any range of data types, including both on-chain data (token balances, transaction data, etc.), real-world events (sports scores, flight status, weather, etc.), and any data that is available through the web or any other mediums (stocks/token prices, random numbers, etc.)

Since the launch of our GuanYu mainnet back in October 2020, we have seen an exponential increase in adoption in a diverse array of use cases. From applications in lending, money markets, gambling, asset, and tokenization, to both on-chain and real-world insurance.

With the **Phase 2 (Laozi) upgrade**, we aim to expand further the scope of what is possible with our oracle through multiple ways. Two, in particular, includes the option for data providers to receive payment directly on-chain from developers using their services on BandChain, and allowing for cross-chain oracle requests through the Inter-Blockchain Communication (IBC) standard.

The new feature also enables a new cohort of products and services that Band oracle can provide to developers. Some examples of these are a more decentralized price oracle, verifiable randomness, and facilitating cross-chain communication.

Band Protocol Documentation

Introduction

High-Level Overview

The Need for Oracles

The BandChain Oracle

How BandChain Works?

Example Use Cases

Core Concepts

Products

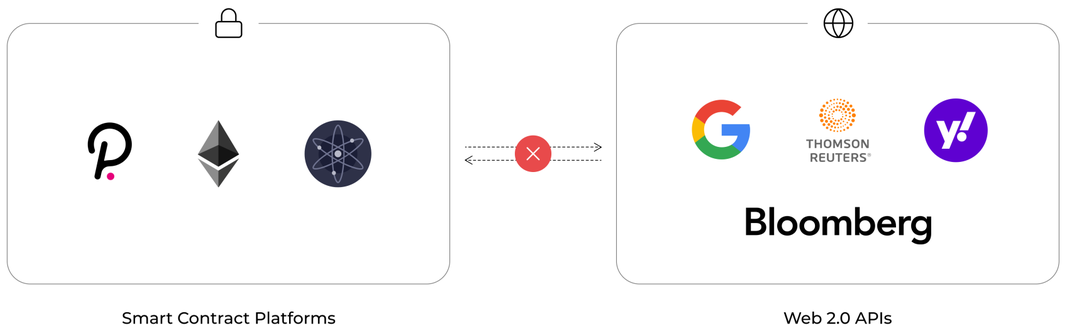
Develop

Nodes & Validators

Staking

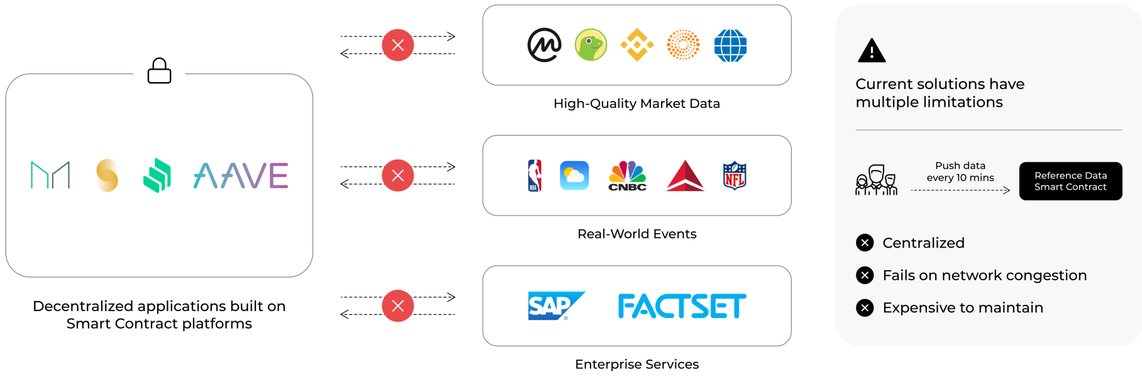
The Need for Oracles

Smart contracts are significant at immutable storage and verifiable transaction, but their use cases have previously been restricted due to their access to outside data. **Most blockchains are neither aware of anything going on in the real world**, nor can they access any data not native to the chain itself.



The data that they could not previously access includes any data available on the traditional web, as well as those accessible through APIs. When you start to consider just how many of the products and tools we use today rely on these data, the problem becomes apparent.

While there has been a multitude of efforts to solve this issue, most of the solutions have come to meet at least one of three main limitations.



1. **Centralization** Existing data solutions such as API feeds and some other oracle solutions are centralized by design. This not only goes against the ideology of decentralization and trustlessness but also represents a severe potential security flaw. Relying on a central authority to report data means that you are exposing yourself to the possibility of data manipulation and outages, both of which can have catastrophic implications on any services that depend on it, not to mention on the end users themselves.

2. **Network Congestion** Most of the existing oracle solution of them are constrained by network congestion. This is mostly the result of the solution being on the same blockchain as the application itself -- competing for block order. Thus, if the blockchain's network were to become full with pending transactions, the data request transaction themselves would also be delayed.

3. **High Cost** they are expensive. This comes from both the cost to research, develop, and deploy the solution, as well as the various costs associated with operating and maintaining it in the long run.

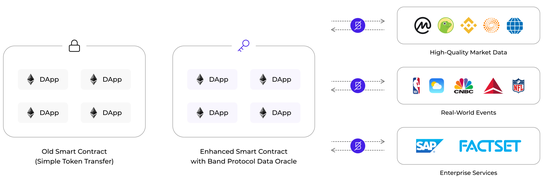
Band Protocol Documentation
Introduction
High-Level Overview
The Need for Oracles
The BandChain Oracle
How BandChain Works?
Example Use Cases
Core Concepts
Products
Develop
Nodes & Validators
Staking

The BandChain Oracle

The BandChain Oracle solution serves as an intermediary layer that operates between smart contract platforms or decentralized applications and various data providers.

The primary function of the oracles is as follows:

1. Responding to data requests from dApps
2. Querying data from the corresponding providers, and
3. Reporting results back to the application.



4.

BandChain Oracle stands out from other oracle solutions in four key areas:

1. **Decentralization**: Our solution is built on a decentralized network of validators, ensuring that data is secure and transparent.
2. **Flexibility**: BandChain Oracle supports a wide range of data sources and formats, making it easy to integrate with existing systems.
3. **Scalability**: Our solution is designed to handle high volumes of data requests, ensuring that your business can grow and scale without interruption.
4. **Cost**: Users can request data only when they need to and pay the associated fees on a per-request basis, making it significantly more cost-effective than updating the price of an entire set of assets when only the latest price of one is needed

Decentralization

BandChain Oracle is designed with maximum redundancy in its infrastructure to ensure decentralization at both the consensus and data source level.

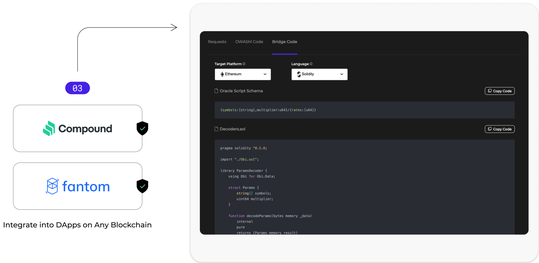
To achieve this, BandChain is operated by a globally distributed pool of validators whose actions can be easily monitored and verified by anyone. When a data request is made, validators are responsible for fetching the results.

To provide an additional layer of redundancy, the results are taken from multiple data sources. BandChain's delegated proof of stake design also ensures that validators have monetary incentives to properly and accurately report data, or risk losing capital and reputation.

Moreover, the entire data request flow is publicly available for viewing, verification, and auditing by anyone.

Flexibility

BandChain offers maximum customization and flexibility to its users through its data source and oracle scripts. These scripts allow users to query and compute their desired data feed with ease.



Data source scripts are custom scripts that allow users to query data from their desired sources. This data can then be fed into oracle scripts, which report back with the desired data feed.

Oracle scripts are pieces of code that define the logic of the data request. These scripts specify two things:

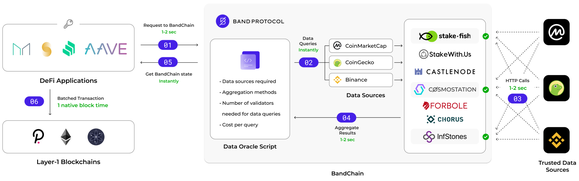
- the set of data sources that validators query data from
- the method to aggregate the result from those data sources into the final result.

These scripts can be programmed in multiple programming languages and function similarly to smart contracts.

Scalability

BandChain is designed specifically for oracle data requests and computations, which sets it apart from general-purpose blockchains.

This design results in several benefits, such as an average block time of just 3 seconds, compared to Ethereum's 10-15 seconds and Bitcoin's 10 minutes. As a result, data request transactions are received and resolved quickly.



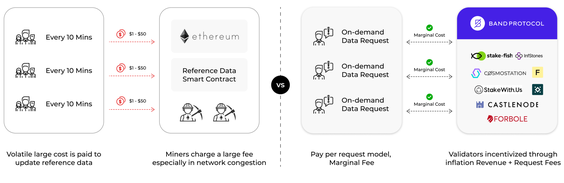
Furthermore, BandChain is optimized for heavy oracle computations, allowing it to offload these computations from the requester's chain.

This optimization, coupled with the design of BandChain, means that an average data request to BandChain's oracle can be expected to resolve in under six seconds. This allows BandChain Oracle to continuously enhance its throughput capacity with the same base-level infrastructure.

With its own chain, the oracle core logic and operations do not need to be duplicated onto a new chain or App for each integration, making integration with DApps streamlined and scalable.

Cost

In addition to its scalability, Band's oracle offers an economical approach to data requests. Users can request data only when they need to and pay the associated fees on a per-request basis, making it significantly more cost-effective than updating the price of an entire set of assets when only the latest price of one is needed.



Band Protocol Documentation
Introduction
High-Level Overview
The Need for Oracles
The BandChain Oracle
How BandChain Works?
Example Use Cases
Core Concepts
Products
Develop
Nodes & Validators
Staking

How BandChain Works?

BandChain is a high-performance public blockchain that allows anyone to make a request for APIs and services available on the traditional web. It is built on top of the Cosmos SDK, and utilizes Tendermint's Byzantine Fault Tolerance consensus algorithm to reach immediate finality. This finality is specifically reached upon getting confirmations from a sufficient number of block validators.

Goals

While the majority of existing smart contract platforms support trustless executions of arbitrary programs, they lack access to real-world data. This limitation hinders the potential of the applications that are developed on those platforms.

BandChain solves this important issue by connecting public blockchains to real-world, off-chain information. The project was created with the following design goals:

1. **Speed and Scalability**: The system must be able to serve a large quantity of data to multiple public blockchains with minimal latency, while maintaining a high throughput. The expected response time must be in the order of seconds.
2. **Cross-Chain Compatibility**: The system must be blockchain-agnostic and able to serve data to most publicly available blockchains. Verification of data authenticity on the target blockchains must be efficient and trustless by nature.
3. **Data Flexibility**: The system must be generic and able to support different methods of retrieving and aggregating data, including both permissionless, publicly available data as well as information guarded by centralized parties.

BandChain achieves the aforementioned goals with a blockchain specifically built for off-chain data curation. The blockchain supports generic data requests and on-chain aggregation with WebAssembly-powered oracle scripts. Oracle results on BandChain blockchain can be sent across to other blockchains via the Inter-Blockchain Communication protocol (IBC) or through customized one-way bridges with minimal latency.

Terminology

Data Sources

A data source is the most fundamental unit in BandChain's oracle system. It defines the procedure to retrieve raw data from a source and the fee associated with the data query.

On BandChain, a data source can be registered into the system by anyone. This is done through the registrant sending a `MsgCreateDataSource` message to the chain. In this message, they specify various parameters the data source they wish to register, including

- the **sender** who wish to create the data source
- the **owner** of the data source, if specified
- the **name** of the data source
- the per-query **fee** that someone looking to use that data source needs to pay
- the **executable** to be run by validators upon receiving a data request for this data source

Examples

The following two examples illustrate what a data source executable might look like. Both examples are written in bash.

Retrieve Cryptocurrency Price from CoinGecko

The data source requires that cURL and jq are installed on the executable runner's machine and expects one argument; the currency ticker symbol.

```
#!/bin/sh

# Cryptocurrency price endpoint: https://www.coingecko.com/api/documentations/v3
URL="https://api.coingecko.com/api/v3/simple/price?ids=$1&vs_currencies=usd"
KEY=".$1.usd"

# Performs data fetching and parses the result
curl -s -X GET $URL -H "Accept: application/json" | jq -r .[\"$1\"].usd"
```

Resolve Hostname to IP Addresses

Again, this script assumes that getent and awk are available on the host and the host is connected to the DNS network.

```
#!/bin/sh

getent hosts $1 | awk '{ print $1 }'
```

Oracle Scripts

When someone wants to request data from BandChain, however, it is not the data sources that they interact with. Instead, they do so by calling one of the available oracle scripts.

An oracle script is an executable program that encodes:

- the set of raw data requests to the data sources it needs
- the way to aggregate raw data reports into the final result

Oracle scripts are also Turing-complete and can be programmed in multiple languages. This composability and Turing-completeness makes oracle scripts very similar to smart contracts.

To create an oracle script, the creator must broadcast a `MsgCreateOracleScript` to BandChain. The contents of the message is similar to `MsgCreateDataSource`, and includes:

- the **sender** who wishes to create the oracle script
- the **owner** of the oracle script, if specified
- the **name** of the oracle script
- the OWasm compiled binary attached to this oracle script
- the **schema** detailing the inputs and outputs of this oracle script, as well as the corresponding types
- the **URL** for the source code of this oracle script

Similar to data sources, the sender who wishes to create the oracle script does not have to be the same as the owner of the oracle script specified in the message.

The execution flow of an oracle script can then be broken down into two phases.

In the first phase, the script outlines the data sources that are required for its execution. It then sends out a request to the chain's validators to retrieve the result from the required data sources. The content of this consists of the data sources' execution steps and the associated parameters.

The second phase then aggregates all of the data reports returned by the validators, with each report containing the values the validator received from the required data sources. The script then proceeds to combine those values into a single final result.

Note that the specifics of the aggregation process is entirely up to the design of the oracle script. BandChain does not enforce any regulations when it comes to the aggregation method used, and entirely leaves that design decision to the creator of the script or any subsequent editors.

Example

The pseudocode below shows an example of an oracle script that returns the current price of a cryptocurrency. The script begins by emitting requests to validators to query the price from three data sources (i.e. the `request` function calls to CoinGecko, CryptoCompare, CoinMarketCap inside `prepare`). Once a sufficient number of validators have reported the prices, the script then aggregates and averages out the reported values results into a single final result (the `aggregate` function).

In this particular oracle script, the aggregation process starts by summing all of the price values returned by the validators across all data sources, as well as the total number of reports returned. It then simply divides the summed price value with the number of data reports returned to arrive at the final average value.

```
# 1st Phase. Emits raw data requests that the oracle script needs.
def prepare(symbol):
    request(get_px_from_coin_gecko, symbol)
    request(get_px_from_crypto_compare, symbol)
    request(get_px_from_coin_market_cap, symbol)

# 2nd Phase. Aggregates raw data reports into the final result.
def aggregate(symbol, number_of_reporters):
    data_report_count = 0
    price_sum = 0.0
    for reporter_index in range(number_of_reporters):
        for data_source in (
            get_px_from_coin_gecko,
            get_px_from_crypto_compare,
            get_px_from_coin_market_cap,
        ):
            price_sum = receive(reporter_index, data_source, symbol)
            data_report_count += 1
    return price_sum / data_report_count
```

Raw Data Reports

Raw data reports are the results that BandChain's validators return when they have successfully responded to a data request and subsequently retrieved results from the required data sources. In these reports, the validators list out the result they got from each data source, using the data source's external ID as the reference key. The external ID is the identifier used to reference a data source within an oracle script, and each data source's external ID is unique within the context of that script.

Oracle Request Proof

When the final data request result is successfully stored onto BandChain, an oracle data proof is produced. This proof is a Merkle proof that shows the existence of the final result of the data request on BandChain. In addition to the actual result value of the request, the proof contains information on the request parameters (oracle script hash, the parameters, the time of execution, etc) as well as as well as those of the associated response (e.g. number of validators that responded to the request). This proof can then be used by smart contracts on other blockchain to verify the existence of the data as well as to decode and retrieve the result stored. Both of these can be done by interacting with our lite client.

Network Participants

BandChain's network consists of a number of network participants, each owning BAND tokens. In the Laozi mainnet, these participants can be broken down into three main groups; validators, delegators, and data providers.

Validators

> MORE INFO
> - How validator be choosing

Validators are responsible for performing two main functions on the network. First, they are responsible for proposing and committing new blocks to the blockchain. They participate in the block consensus protocol by broadcasting votes which contain cryptographic signatures signed by each validator's private key. This is similar to most other Cosmos-based delegated proof-of-stake blockchains.

Each validator will have a certain amount of BAND tokens bonded to them. The source of these tokens can either be their own holdings, or the tokens delegated to them by other token owners. In most cases, there will be a large number of parties with tokens staked to them. In that case, the top 100 validator candidates with the most token staked to them will become BandChain's validators.

The role the validators described above is similar to those of validators on many other Cosmos-based blockchains. In addition, most transactions supported by BandChain (asset transfer, staking, slashing, etc.) are also derived from Cosmos-SDK.

What makes BandChain unique, and the origin of the validators' second duty, is the chain's capability to natively support external data query. This role will be further explore in the Oracle Data Request Flow section.

Delegators

The second main group of participants are then the individual BAND token holders. On BandChain, BAND holders do not stake their tokens directly, but delegate holdings to a validator. This allows token holders who don't want to set up a validator node to participate in staking rewards.

Data Providers

Finally, the Laozi upgrade introduces a new third kind of participant in the BandChain network: data providers.

With the introduction of on-chain payments, API or data providers can now monetize their data and services directly on BandChain. This new flexibility benefits the network in multiple ways.

Data providers now have a new medium to collect revenue from. As the fees are collected per-query, the revenue that they stand to collect will scale alongside the adoption and usage of BandChain and our oracle as a whole.

The option for data providers to monetize their services directly on-chain will also bring official support for premium and paid data sources onto BandChain. This will allow any developer building on BandChain to access a much wider array of providers and data types they can choose from, enabling BandChain oracle infrastructure to power a much wider range of applications and services.

Oracle Data Request Flow

The flow of requesting data from BandChain can be broken down into four main steps:

1. Publishing the necessary data sources and oracle scripts to the network
2. Sending the oracle data request transaction
3. Fetching the necessary data
4. Aggregating and storing the request result onto BandChain

1. Publishing Data Sources and Oracle Scripts

Before any data requests can be made, two conditions must be met:

1. The oracle script that describes the data request must also have been published to Bandchain via `Msg`
2. The data sources related to the aforementioned oracle script must be published to BandChain

2. Oracle Data Request Initialization

Once the required data sources and oracle scripts are published, the user can initiate data request to Band's oracle by broadcasting `MsgRequestData`. The contents of the message includes the ID of the oracle script that the requester wants to invoke and other query and security parameters.

Once the data transaction is confirmed on BandChain, the requested oracle script will begin its execution. The script's execution process can be split into two phases.

3. Fetching the Data

First, the oracle script's preparation function will emit the set of raw data requests necessary to continue the script's execution. The chain's validators, who are chosen at random for security reasons, will then inspect the raw data requests and execute the associated data sources' procedures as instructed by the request. Specifically, each of the chosen validator will attempt to retrieve information from all of the data sources specified in the executed oracle script.

The validators that successfully retrieved data from all the sources will then submit a raw data report to BandChain, containing the results they got from each of the data sources, by broadcasting `MsgReportData`. Once a sufficient number of validators, specified in the data request's security parameters, have reported the their results, BandChain will begin executing the oracle script's second part of aggregating request result.

Note that for data from permissioned sources (e.g. under paywall), the data sources are expected to verify that payment has occurred on BandChain and supply data to requested validators accordingly. That way, BandChain allows API providers to monetize data with BandChain's on-chain payment settlement without needing to trust a middleman party.

4. Aggregating and Request Result Storage

This phase begins by aggregating all of the validators' reports (which contains the data each received from the data sources) into a final single result. This final result is then permanently stored in BandChain's application state. Once stored, the result becomes available on the chain's state tree and can be sent to other blockchain.

Band Protocol
Data sources required
Execution procedure
Number of validators needed for data queries
Cost per query
Oracle
Oracle data validators to fetch requested data
Oracle collects validator responses
stake.fish
StakeWithUs
CASTLENODE
CRYPTOSTATION
FIBROUS
CHAINUS
HYDROnes
Validators
Validators each fetch data from source
Source returns result
CoinMarketCap
CoinGecko
BINANCE
Data sources
Data request
Tx
Tx
Tx

Band Protocol Documentation

Introduction

High-Level Overview

The Need for Oracles

The BandChain Oracle

How BandChain Works?

Example Use Cases

Core Concepts

Products

Develop

Nodes & Validators

Staking

Example Use Cases

Decentralized Standard Price Reference

Since the initial launch of the Standrd Dataset back in October 2020, we have seen an exponential increase in adoption and usage of Band's price oracles across numerous chains and sectors.

As this trend continues to propagate, the need for stronger decentralization and robustness of our price feed becomes ever more crucial.

With the next major upgrade of our Standard Dataset, we will be adding a new mechanism that will allow anyone to send price update transactions to our oracle contract themselves, all secured and verified by our lite client verification architecture and a challenge mechanism.

Verifiable Random Number Generators

Aside from our Standard Dataset price oracle, the flexibility offered by Band's oracle design also allow it to support countless other use cases and data types.

One of these this is our upcoming Verifiable Random Function (VRF) functionality. This aims to provide a provable and verifiable source of randomness to smart contracts on any network supported by Band's oracle.

By providing a tamper-proof method of generating unpredictable yet verifiably random values, this upcoming addition to Band's oracle functionality will greatly extend the range of applications and use cases that Band can support. Examples of such areas are:

- NFTs
- gambling
- randomized selection

This feature is currently in active development and we will have much more to share on this soon.

Cross Chain Communication

BandChain's infrastructure design decision to be cross-chain compatible and operate on independent blockchain prepares its role as a key player in the ever-growing cross-chain and multi-chain narratives.

We believe that oracles such as Band will not only be useful wihtin the context of a single destination chain, but also will a core infrastructure in relaying information and faciliating communication between two independent chain.

This can be in the form of verifying token transfer transactions when bridging assets across chain, relaying transaction between chain, or any other arbitrary number of data or actions that needed to be tranasferred across networks.

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

BAND

BAND is BandChain's native staking token. Staked holders can govern and decide the future of the protocol.

BandChain Tokenomics

Inflation

BandChain applies an inflationary model to the BAND token, which incentivizes network participation by token holders. The aim is to encourage token holders to stake their coins on the network, rather than focusing solely on trading or leaving their coins idle.

The specific inflation parameters currently mirror those of the Cosmos network. The annual inflation rate ranges from 7% to 20% and is adjusted to target having 66% of the total supply of BAND tokens staked.

To illustrate how inflation incentivizes staking, consider a network participant with a certain amount of holdings. With inflation, if the participant chooses not to use their coins to participate in the network's activities, they will find that the percentage of their holdings with respect to the total supply decreases over time. However, if they decide to stake their coins, they will receive a share of coins proportional to the inflation, meaning their total token holding ratio will remain relatively unchanged.

Validators and Stakers

Similar to other Cosmos-based blockchains, one of the responsibilities of BandChain validators is to provision new blocks and process transactions. By performing these tasks, they earn BAND tokens as a reward. In the case of block provisioning, the reward comes from newly minted tokens on that block. Conversely, the reward for processing transactions comes from the fees that each validator chooses to set. Note that a percentage of the total block reward is diverted to the community fund pool.

Those who do not wish to become validators can still earn a portion of the validator rewards by becoming delegators. This is done by staking their holdings on the network's validators. By doing so, they will share the revenue earned by those validators.

The amount of reward each validator receives is based on the total amount of tokens staked to them. Before this revenue is distributed to their delegators, a validator can apply a commission. In other words, delegators pay a commission to their validators on the revenue they earn.

However, while delegators share the revenue of their validators, they also share the risks. Therefore, it is imperative for potential delegators to understand those risks, and that being a delegator is not a passive task.

Some actions that a delegator should perform are:

- **Perform due diligence on the validators you wish to stake on before committing:** If a validator you staked on misbehaves, a portion of the validator's stake, including those of their delegators, are slashed. Therefore, it is advisable for delegators to carefully consider their staking choices.
- **Actively monitor the validators you've committed to:** Delegators should ensure that the validators they delegate to behave correctly, meaning that they have good uptime, do not double sign or get compromised, and participate in governance.
- **Participate in network governance:** Delegators are expected to participate in network governance activities. A delegator's voting power is proportional to the size of their bonded stake. If a delegator does not cast their vote, they will inherit the vote of the validators they staked on. If they do vote, they instead override the vote of those validators. Delegators, therefore, act as an important counterbalance to their validators.

Community Fund Pool

Two percent of the total block rewards are diverted to the community fund pool. The funds are intended to promote the long-term sustainability of the ecosystem. These funds can be distributed in accordance with the decisions made by the governance system.

Slashing

If a validator misbehaves, their delegated stake will be partially slashed.

There are three reasons why a validator may be slashed:

- Excessive downtime
- Double signing, and
- Unresponsiveness.

The first two are derived from the Cosmos SDK, while the third is specific to BandChain.

Excessive Downtime

One of the validators' primary functions is to propose and subsequently commit new blocks onto the chain. Thus, if a validator has not participated in more than `[MIN_SIGNED_PER_WINDOW]` of the last `[SIGNED_BLOCK_WINDOW]` block proposals and commits, we will consider that they are not performing their function properly. Consequently, we will slash the total value staked to them by `[SLASH_FRACTION_DOWNTIME]`.

Double Signing

During a double signing, the block proposer in the consensus round attempts to create two conflicting blocks and broadcast them to the network. If there are any other validators who stand to profit from this double block-proposal, they will vote for both blocks.

If the votes pass, nodes on the network will see two different blocks at the same height, each with different contents and hashes. From this point on, the network has **"forked"**. The consequence of this is that there will now be two versions of the **"truth"**.

To prevent such attempts at double signing, Cosmos, and by extension BandChain, is implemented so that any validator that double-signs is slashed, with the slashed amount being `[SLASH_FRACTION_DOUBlESIGNING]` of all tokens bonded to them.

Unresponsiveness

Finally, we also slash validators if they consistently fail to respond to data requests. If a validator fails to respond to `[CONSECUTIVE_UNRESPONSIVE_REQUESTS]` consecutive requests, they will be slashed an amount equal to `[SLASH_FRACTION_UNRESPONSIVENESS]`.

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Gas and Fees

PRE-REQUISITE READINGS

- Cosmos SDK Gas

Introduction to Gas and Network Fees

In the Cosmos SDK, gas is a unit that is used to track the consumption of resources during process execution. It is typically consumed during read/write operations, or whenever a computationally expensive operation is performed.

The purpose of gas is twofold:

1. To prevent blocks from consuming excessive resources, thus ensuring that the block will be finalized
2. To prevent abuse from a malicious actor on the user side

Gas consumed during message execution is typically priced, resulting in a fee

```
fee = gas * gasPrice
```

Fees generally have to be paid by the sender of the message.

Meanwhile, each block validator can subjectively establish the minimum gas fee that must be reached for them to process the transaction and choose whatever transactions it wants to include in the block that it is proposing, as long as the total gas limit is not exceeded.

Gas Estimation

This section we will compare gas usage on each message type to compare how many fee that user need to pay to do these actions.

Message	Estimated fee
MsgSend	~70k
MsgDelegate	~120k
MsgWithdrawReward	~100k
MsgTransfer(IBC)	~85k
MsgUpdateClient + MsgReceivePacket (Transfer packet)	~350k
MsgRequestData*	>500k upto 5m
MsgUpdateClient + MsgReceivePacket (Oracle request packet)*	>560k upto 5m
MsgCreateDataSource / MsgEditDataSource**	30k - 100k
MsgCreateOracleScript / MsgEditOracleScript**	500k - 2m

(*) The gas is used on MsgRequestData or process oracle request packet based on complexity of oracle script and how many validator need to query data on this request.

(**) Based on size of data source and oracle script mostly data source should be smaller than oracle script.

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Accounts and Wallets

RECOMMENDED READINGS

- Cosmos SDK Accounts

Band Accounts

BandChain is a Cosmos SDK-based blockchain, which means it is compatible with the Cosmos SDK account system. However, BandChain has its own custom `Account` type to implement in Hierarchical Deterministic (HD) wallets.

The BandChain coin type HD path is `m/44'/494'/0'/0/0`. This path is used to derive BandChain addresses from a BIP44 standard HD wallet.

The first part of the path, `m/44'`, specifies the derivation path is based on BIP44 standard. The second part of the path, `494'`, indicates the BandChain is the coin type for this address. The third part of the path, `0'`, specifies that this is a hardened derivation path, which means that the child keys can only be derived by someone who has the parent key. The fourth part of the path, `0`, specifies the account index which is used to generate multiple accounts from the same master key. The last path, `0`, specifies the address index, which is used to generate multiple addresses for a single account.

Addresses and Public Keys

BandChain supports Bech32 (`band...`). The Bech32 format is the default format for Cosmos-SDK queries and transactions through CLI and REST clients.

For example, a BandChain address might look like this: `band18p27yl962l8283ct7srr5l3g7ydazj07dqrwph`. This address can be used to receive and send BandChain tokens and interact with smart contracts and dApps on the BandChain network.

`Addresses` and `PubKeys` are both public information that identifies actors in the application. Each account is identified using `Address` which is a sequence of bytes derived from a public key.

In the BandChain, we define 3 types of addresses that specify a context where an account is used:

Types	Address bech32 Prefix	Pubkey bech32 Prefix	Address byte length	Pubkey byte length
Accounts	`band`	`bandpub`	20	33
Validators	`bandvaloper`	`bandvaloperpub`	20	33
Consensus Nodes	`bandvalcons`	`bandvalconspub`	20	33

Mnemonic and Address Generation

To create an account you can either create a private key, a keystore file (a file that contains your encrypted private key), or a mnemonic phrase (a string of words that can access multiple private keys).

Mnemonic phrases also known as hierarchical deterministic key generation (HD keys). This allows the user to create accounts on multiple blockchains without having to manage multiple secrets. We can think of the derived accounts as child accounts of the root account created using the original mnemonic seed phrase.

A mnemonic phrase typically consists of 12, 24, or sometimes 16 words that are randomly generated from a pre-determined word list. Each word in the list corresponds to a unique sequence of numbers, which are used to generate the private key for your wallet. By writing down and securely storing your mnemonic phrase, you can always recover your BandChain account in case of loss or damage to your device.

```
## Example of mnemonic phrases used in BandChain

hedgehog pact leave raccoon empty various item profit uncover dune someone ball chat repair acquire mid
```

PROTECT YOUR SEED PHRASE

If the mnemonic phrase leaks, accounts cannot be derived without the initial password. If someone knows the mnemonic phrase and the derivation path, they will have access to your account. To protect your seed phrase, consider the following tips.

- Never share your mnemonic phrase with anyone.
- Always write down your seed phrase with a pen and paper.
- Store the paper with your seed phrase on it somewhere safe.
- Never give your seed phrase to anyone, not even support staff.

At BandChain, we offer a simple method for generating a mnemonic phrase to create a BandChain account using either PyBand or BandChain.js.

```
import { Wallet } from "@bandprotocol/bandchain.js";
const { PrivateKey } = Wallet

const [mnemonic, privateKey] = PrivateKey.generate("m/44'/494'/0'/0/0")
const address = PrivateKey.fromMnemonic(mnemonic).toPubkey().toAccBech32() // band1ycw2277nurr5zymw7exq
```

Querying an Account

You can query an account address using the Band CLI or REST clients:

```
# Query account address using bandd
bandd keys show <key-name> -a
```

Query account address using REST client

```
# GET /cosmos/auth/v1beta1/accounts/{address}
curl -X GET "https://laozi1.bandchain.org/api/cosmos/auth/v1beta1/accounts/band1ycs4g7xu8wmf7n4vwwtfsvh
```

Wallets

As mentioned earlier, building on the CosmosSDK allows us to enable our users to interact with BandChain using Cosmos native wallets. The most popular Cosmos and IBC-enabled wallets are supported on BandChain. These include:

1. Cosmostation
2. Ledger

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Protocol Messages

PRE-REQUISITE READINGS
- Cosmos SDK Messages

Native Cosmos SDK Messages

Stemming from its Cosmos SDK foundation, BandChain supports all types of messages that are native to the SDK.

BandChain Specific Messages

Apart from the messages that stems from the Cosmos SDK, BandChain also supports a number of messages native to its data oracle system. These messages' specification is presented below.

MsgCreateDataSource

Deploys and registers a new data source to BandChain. Once registered, the data source is assigned a unique `int64` identifier which can be used to refer to it forever.

Parameters

Parameter	Type	Description
Sender	sdk.AccAddress	The address of the message's sender. Note that the sender does not need to be the same as the owner
Owner	sdk.AccAddress	The address of the entity who will be responsible for maintaining the data source
Name	string	The human-readable string name for this data source
Description	string	The description of this data source
Executable	[]byte	The content of executable to be run by block upon receiving a data request for this data source. The executable can be in any format, as long as it is accepted by the general public.
Treasury	sdk.AccAddress	Treasury is the account address who receive data source fee from requester.
Fee	sdk.Coins	Fee is the data source fee per AskCount that data provider will receive from requester.

MsgEditDataSource

Edits an existing data source given the unique `int64` identifier (i.e. `dataSourceID`). The sender must be the owner of the data source for the transaction to succeed.

Parameters

Parameter	Type	Description
DataSourceID	int64	The unique identifier number assigned to the data source when it was first registered
Sender	sdk.AccAddress	The address of the message's sender. Note that the sender does not need to be the same as the owner
Owner	sdk.AccAddress	The address of the entity who will be responsible for maintaining the data source
Name	string	The human-readable string name for this data source
Description	string	The description of this data source
Executable	[]byte	The content of executable to be run by block validators upon receiving a data request for this data source. The executable can be in any format, as long as it is accepted by the general public.
Treasury	sdk.AccAddress	Treasury is the account address who receive data source fee from requester.
Fee	sdk.Coins	Fee is the data source fee per AskCount that data provider will receive from requester.

MsgCreateOracleScript

Deploys a new oracle script to BandChain's network. Once registered, the script is assigned a unique `int64` identifier which can be used to refer to it forever.

Parameters

Parameter	Type	Description
Sender	sdk.AccAddress	The address of the message's sender. Note that the sender does not need to be the same as the owner
Owner	sdk.AccAddress	The address of the entity who will be responsible for maintaining the oracle script
Name	string	The human-readable string name for this oracle script
Description	string	The description of this oracle script
Code	[]byte	The Owasm-compiled binary attached to this oracle script
Schema	string	The schema detailing the inputs and outputs of this oracle script, as well as the corresponding types
Source Code URL	string	The URL for the source code of this oracle script

MsgEditOracleScript

Edits an existing oracle script given the unique `int64` identifier (i.e. `oracleScriptID`). The sender must be the owner of the oracle script for the transaction to succeed.

Parameters

Parameter	Type	Description
OracleScriptID	int64	The unique identifier number assigned to the oracle script when it was first registered on Bandchain
Sender	sdk.AccAddress	The address of the message's sender. Note that the sender does not need to be the same as the owner
Owner	sdk.AccAddress	The address of the entity who will be responsible for maintaining the oracle script
Name	string	The human-readable string name for this oracle script
Description	string	The description of this oracle script
Code	[]byte	The Owasm-compiled binary attached to this oracle script
Schema	string	The schema detailing the inputs and outputs of this oracle script, as well as the corresponding types
Source Code URL	string	The URL for the source code of this oracle script

MsgRequestData

Requests a new data based on an existing oracle script. A data request will be assigned a unique identifier once the transaction is confirmed. After sufficient validators report the raw data points. The results of the data requests will be written and stored permanently on BandChain for future uses.

Parameters

Parameter	Type	Description
OracleScriptID	int64	The unique identifier number assigned to the oracle script when it was first registered on Bandchain
Sender	sdk.AccAddress	The address of the message's sender.
Calldata	string	The data passed over to the oracle script for the script to use during its execution
AskCount	int64	The number of validators that are requested to respond to this request
MinCount	int64	The minimum number of validators necessary for the request to proceed to the execution phase
ClientID	string	the unique identifier of this oracle request, as specified by the client. This same unique ID will be sent back to the requester with the oracle response.

MsgReportData

Reports raw data points for the given data request. Each data point corresponds to a data source query issued during the data request script's execution of `prepare` function.

Parameters

Parameter	Type	Description
RequestID	int64	The unique identifier number of the particular request
Validator	sdk.ValAddress	The reporting validator's actual validator address
Reporter	sdk.AccAddress	The address the reporting validator uses to sign when submitting the report
Data	[]struct{ externalDataId: int64, data: []byte }'	The array of raw data points. Each item corresponds to a data source query.

MsgAddReporter

Registers an address to the list of addresses available to a validator when signing and submitting a report.

Parameters

Parameter	Type	Description
ValidatorAddress	sdk.ValAddress	The address of the validator wishing to add the reporter address to their list
ReporterAddress	sdk.AccAddress	The address to add to the validator's available addresses

MsgRemoveReporter

Remove a previously registered address from the list of addresses available to a validator when signing and submitting a report.

Parameters

Parameter	Type	Description
ValidatorAddress	sdk.ValAddress	The address of the validator wishing to add the reporter address to their list
ReporterAddress	sdk.AccAddress	The address to remove from the validator's available addresses

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Protobuf Documentation

oracle/v1/oracle.proto

ActiveValidator

ActiveValidator is information of currently active validator

Field	Type	Label	Description
address	string		Address is a validator address
power	uint64		Power is an amount of token that the validator is holding

DataSource

DataSource is the data structure for storing data sources in the storage.

Field	Type	Label	Description
owner	string		Owner is an address of the account who own the data source
name	string		Name is data source name used for display
description	string		Description is data source description used for display
filename	string		Filename is string of file name used as reference for locating data source file stored in bandchain nodes
treasury	string		Treasury is the account address who receive data source fee from requester.
fee	cosmos.base.v1beta1.Coin	repeated	Fee is the data source fee per ask_count that data provider will receive from requester.

IBCChannel

IBCChannel is information of IBC protocol to allow communicating with other chain

Field	Type	Label	Description
port_id	string		PortID is port ID used for sending response packet when request is resolved.
channel_id	string		ChannelID is channel ID used for sending response packet when request is resolved.

OracleRequestPacketAcknowledgement

OracleRequestPacketAcknowledgement encodes an oracle request acknowledgement send back to requester chain.

Field	Type	Label	Description
request_id	int64		RequestID is BandChain's unique identifier for this oracle request.

OracleRequestPacketData

OracleRequestPacketData encodes an oracle request sent from other blockchains to BandChain.

Field	Type	Label	Description
client_id	string		ClientID is the unique identifier of this oracle request, as specified by the client. This same unique ID will be sent back to the requester with the oracle response.
oracle_script_id	int64		OracleScriptID is the unique identifier of the oracle script to be executed.
calldata	bytes		Calldata is the OBI-encoded calldata bytes available for oracle executor to read.
ask_count	uint64		AskCount is the number of validators that are requested to respond to this oracle request. Higher value means more security, at a higher gas cost.
min_count	uint64		MinCount is the minimum number of validators necessary for the request to proceed to the execution phase. Higher value means more security, at the cost of liveness.
fee_limit	cosmos.base.v1beta1.Coin	repeated	FeeLimit is the maximum tokens that will be paid to all data source providers.
request_key	string		RequestKey is the key from request chain to match data source fee payer on Bandchain
prepare_gas	uint64		PrepareGas is amount of gas to pay to prepare raw requests
execute_gas	uint64		ExecuteGas is amount of gas to reserve for executing

OracleResponsePacketData

OracleResponsePacketData encodes an oracle response from BandChain to the requester.

Field	Type	Label	Description
client_id	string		ClientID is the unique identifier matched with that of the oracle request packet.
request_id	int64		RequestID is BandChain's unique identifier for this oracle request.
ans_count	uint64		AnsCount is the number of validators among to the asked validators that actually responded to this oracle request prior to this oracle request being resolved.
request_time	int64		RequestTime is the UNIX epoch time at which the request was sent to BandChain.
resolve_time	int64		ResolveTime is the UNIX epoch time at which the request was resolved to the final result.
resolve_status	ResolveStatus		ResolveStatus is the status of this oracle request, which can be OK, FAILURE, or EXPIRED.
result	bytes		Result is the final aggregated value encoded in OBI format. Only available if status if OK.

OracleScript

OracleScript is the data structure for storing oracle scripts in the storage.

Field	Type	Label	Description
owner	string		Owner is an address of the account who own the oracle script
name	string		Name is oracle script name used for display
description	string		Description is oracle script description used for display
filename	string		Filename is string of file name used as reference for locating compiled oracle script WASM file stored in bandchain nodes
schema	string		Schema is the schema of the oracle script input/output which is formatted in OBI format e.g. "{symbol:string,multiplier:u64}/{px:u64}"
source_code_url	string		SourceCodeURL is the URL of oracle script's source code. It is recommendded to store source code on IPFS and get its URL to preserve decentralization.

Params

Params is the data structure that keeps the parameters of the oracle module.

Field	Type	Label	Description
max_raw_request_count	uint64		MaxRawRequestCount is the maximum number of data source raw requests a request can make.
max_ask_count	uint64		MaxAskCount is the maximum number of validators a request can target.
expiration_block_count	uint64		ExpirationBlockCount is the number of blocks a request stays valid before it gets expired due to insufficient reports.
base_owasm_gas	uint64		BaseOwasmGas is the base amount of Cosmos-SDK gas charged for owasm execution.
per_validator_request_gas	uint64		PerValidatorRequestGas is the amount of Cosmos-SDK gas charged per requested validator.
sampling_try_count	uint64		SamplingTryCount the number of validator sampling tries to pick the highest voting power subset of validators to perform an oracle task.
oracle_reward_percentage	uint64		OracleRewardPercentage is the percentage of block rewards allocated to active oracle validators.
inactive_penalty_duration	uint64		InactivePenaltyDuration is the duration period where a validator cannot activate back after missing an oracle report.
ibc_request_enabled	bool		IBCRequestEnabled is a flag indicating whether sending oracle request via IBC is allowed

PendingResolveList

PendingResolveList is a list of requests that are waiting to be resolved

Field	Type	Label	Description
request_ids	int64	repeated	RequestIDs is a list of request IDs that are waiting to be resolved

PriceResult

PriceResult is a result from standard price reference

Field	Type	Label	Description
symbol	string		Symbol is unit of data indicating what the data is. It is price currencies for this case.
multiplier	uint64		Multiplier is a number used for left-shifting value to eliminate decimal digits
px	uint64		Px is the actual data, which is rate number multiplied by the multiplier.
request_id	int64		RequestID is oracle request ID that contains this price
resolve_time	int64		ResolveTime is epoch timestamp indicating the time when the request had been resolved

RawReport

RawRequest is the data structure for storing raw reporter in the storage.

Field	Type	Label	Description
external_id	int64		ExternalID is an ID of the raw request
exit_code	uint32		ExitCode is status code provided by validators to specify error, if any. Exit code is usually filled by the exit code returned from execution of specified data source script. With code 0 means there is no error.
data	bytes		Data is raw result provided by validators. It is usually filled by the result from execution of specified data source script.

RawRequest

RawRequest is the data structure for storing raw requests in the storage.

Field	Type	Label	Description
external_id	int64		ExternalID is an ID of the raw request
data_source_id	int64		DataSourceID is an ID of data source script that relates to the raw request
calldata	bytes		Calldata is the data used as argument params for executing data source script

Report

Report is the data structure for storing reports in the storage.

Field	Type	Label	Description
validator	string		Validator is a validator address who submit the report
in_before_resolve	bool		InBeforeResolve indicates whether the report is submitted before the request resolved
raw_reports	RawReport	repeated	RawReports is list of raw reports provided by the validator. Each raw report has different external ID

ReportersPerValidator

ReportersPerValidator is list of reporters that is associated with a validator

Field	Type	Label	Description
validator	string		Validator a validator address
reporters	string	repeated	Reporters is a list of reporter account addresses associated with the validator

Request

Request is the data structure for storing requests in the storage.

Field	Type	Label	Description
oracle_script_id	int64		OracleScriptID is ID of an oracle script
calldata	bytes		Calldata is the data used as argument params for the oracle script
requested_validators	string	repeated	RequestedValidators is a list of validator addresses that are assigned for fulfilling the request
min_count	uint64		MinCount is minimum number of validators required for fulfilling the request
request_height	int64		RequestHeight is block height that the request has been created
request_time	uint64		RequestTime is timestamp of the chain's block which contains the request
client_id	string		ClientID is arbitrary id provided by requester. It is used by client-side for referencing the request
raw_requests	RawRequest	repeated	RawRequests is a list of raw requests specified by execution of oracle script
ibc_channel	IBCChannel		IBCChannel is an IBC channel info of the other chain, which contains a channel and a port to allow bandchain connect to that chain. This field allows other chain be able to request data from bandchain via IBC.
execute_gas	uint64		ExecuteGas is amount of gas to reserve for executing

RequestVerification

RequestVerification is a message that is constructed and signed by a reporter to be used as a part of verification of oracle request.

Field	Type	Label	Description
chain_id	string		ChainID is the ID of targeted chain
validator	string		Validator is an validator address
request_id	int64		RequestID is the targeted request ID
external_id	int64		ExternalID is the oracle's external ID of data source

Result

Result encodes a result of request and store in chain

Field	Type	Label	Description
client_id	string		ClientID is the unique identifier of this oracle request, as specified by the client. This same unique ID will be sent back to the requester with the oracle response.
oracle_script_id	int64		OracleScriptID is the unique identifier of the oracle script to be executed.
calldata	bytes		Calldata is the calldata bytes available for oracle executor to read.
ask_count	uint64		AskCount is the number of validators that are requested to respond to this oracle request. Higher value means more security, at a higher gas cost.
min_count	uint64		MinCount is the minimum number of validators necessary for the request to proceed to the execution phase. Higher value means more security, at the cost of liveness.
request_id	int64		RequestID is BandChain's unique identifier for this oracle request.
ans_count	uint64		AnsCount is the number of validators among to the asked validators that actually responded to this oracle request prior to this oracle request being resolved.
request_time	int64		RequestTime is the UNIX epoch time at which the request was sent to BandChain.
resolve_time	int64		ResolveTime is the UNIX epoch time at which the request was resolved to the final result.
resolve_status	ResolveStatus		ResolveStatus is the status of this oracle request, which can be OK, FAILURE, or EXPIRED.
result	bytes		Result is the final aggregated value only available if status if OK.

ValidatorStatus

ValidatorStatus maintains whether a validator is an active oracle provider.

Field	Type	Label	Description
is_active	bool		IsActive is a boolean indicating active status of validator. The validator will be deactivated when they are unable to send reports to fulfill oracle request before the request expired.
since	google.protobuf.Timestamp		Since is a block timestamp when validator has been activated/deactivated

ResolveStatus

ResolveStatus encodes the status of an oracle request.

Name	Number	Description
RESOLVE_STATUS_OPEN_UNSPECIFIED	0	Open - the request is not yet resolved.
RESOLVE_STATUS_SUCCESS	1	Success - the request has been resolved successfully with no errors.
RESOLVE_STATUS_FAILURE	2	Failure - an error occured during the request's resolve call.
RESOLVE_STATUS_EXPIRED	3	Expired - the request does not get enough reports from validator within the timeframe.

Top

oracle/v1/query.proto

QueryActiveValidatorsRequest

QueryActiveValidatorsRequest is request type for the Query/ActiveValidators RPC method.

QueryActiveValidatorsResponse

QueryActiveValidatorsResponse is response type for the Query/ActiveValidators RPC method.

Field	Type	Label	Description
validators	ActiveValidator	repeated	Validators is a list of active validators

QueryCountsRequest

QueryCountsRequest is request type for the Query/Count RPC method.

QueryCountsResponse

QueryCountsResponse is response type for the Query/Count RPC method.

Field	Type	Label	Description
data_source_count	int64		DataSourceCount is total number of data sources available on the chain
oracle_script_count	int64		OracleScriptCount is total number of oracle scripts available on the chain
request_count	int64		RequestCount is total number of requests submitted to the chain

QueryDataRequest

QueryDataRequest is request type for the Query/Data RPC method.

Field	Type	Label	Description
data_hash	string		DataHash is SHA256 hash of the file's content, which can be data source or oracle script

QueryDataResponse

QueryDataResponse is response type for the Query/Data RPC method.

Field	Type	Label	Description
data	bytes		Data is file's content, which can be data source or oracle script

QueryDataSourceRequest

QueryDataSourceRequest is request type for the Query/DataSource RPC method.

Field	Type	Label	Description
data_source_id	int64		DataSourceID is ID of a data source script

QueryDataSourceResponse

QueryDataSourceResponse is response type for the Query/DataSource RPC method.

Field	Type	Label	Description
data_source	DataSource		DataSource is summary information of a data source

QueryOracleScriptRequest

QueryOracleScriptRequest is request type for the Query/OracleScript RPC method.

Field	Type	Label	Description
oracle_script_id	int64		OracleScriptID is ID of an oracle script

QueryOracleScriptResponse

QueryOracleScriptResponse is response type for the Query/OracleScript RPC method.

Field	Type	Label	Description
oracle_script	OracleScript		OracleScript is summary information of an oracle script

QueryParamsRequest

QueryParamsRequest is request type for the Query/Params RPC method.

QueryParamsResponse

QueryParamsResponse is response type for the Query/Params RPC method.

Field	Type	Label	Description
params	Params		pagination defines an optional pagination for the request.

QueryPendingRequestsRequest

QueryPendingRequestRequest is request type for the Query/PendingRequests RPC method.

Field	Type	Label	Description
validator_address	string		ValidatorAddress is address of a validator

QueryPendingRequestsResponse

QueryPendingRequestResponse is response type for the Query/PendingRequests RPC method.

Field	Type	Label	Description
request_ids	int64	repeated	RequestIDs is a list of pending request IDs assigned to the given validator

QueryReportersRequest

QueryReportersRequest is request type for the Query/Reporters RPC method.

Field	Type	Label	Description
validator_address	string		ValidatorAddress is a validator address

QueryReportersResponse

QueryReportersResponse is response type for the Query/Reporters RPC method.

Field	Type	Label	Description
reporter	string	repeated	Reporter is a list of account addresses of reporters

QueryRequestPoolRequest

QueryRequestPoolRequest is request type for the Query/RequestPool RPC method.

Field	Type	Label	Description
request_key	string		RequestKey is a user-generated key for each request pool
port_id	string		PortID is the corresponding port to the request pool
channel_id	string		ChannelID is the corresponding channel to the request pool

QueryRequestPoolResponse

QueryRequestPoolResponse is response type for the Query/RequestPool RPC method.

Field	Type	Label	Description
request_pool_address	string		RequestPoolAddress is an address of a request pool
balance	cosmos.base.v1beta1.Coin	repeated	

QueryRequestPriceRequest

QueryRequestPriceRequest is request type for the Query/RequestPrice RPC method.

Field	Type	Label	Description
symbols	string	repeated	Symbol is unit of data indicating what the data is
ask_count	int64		AskCount is number of validators allowed for fulfilling the request
min_count	int64		MinCount is number of validators required for fulfilling the request

QueryRequestPriceResponse

QueryRequestPriceResponse is response type for the Query/RequestPrice RPC method.

Field	Type	Label	Description
price_results	PriceResult	repeated	PriceResult is a list of price results for given symbols

QueryRequestRequest

QueryRequestRequest is request type for the Query/Request RPC method.

Field	Type	Label	Description
request_id	int64		RequestID is ID of an oracle request

QueryRequestResponse

QueryRequestResponse is response type for the Query/Request RPC method.

Field	Type	Label	Description
request	Request		Request is an oracle request
reports	Report	repeated	Reports is list of result data as raw reports that are fulfilled by assigned validators
result	Result		Result is a final form of result data

QueryRequestSearchRequest

QueryRequestSearchRequest is request type for the Query/RequestSearch RPC method.

Field	Type	Label	Description
oracle_script_id	int64		OracleScriptID is ID of an oracle script
calldata	string		Calldata is OBI-encoded data in hex format as argument params for the oracle script
ask_count	uint64		AskCount is number of validators allowed for fulfilling the request
min_count	uint64		MinCount is number of validators required for fulfilling the request

QueryRequestSearchResponse

QueryRequestSearchResponse is response type for the Query/RequestSearch RPC method.

Field	Type	Label	Description
request	QueryRequestResponse		Request is details of an oracle request

QueryRequestVerificationRequest

QueryRequestVerificationRequest is request type for the Query/RequestVerification RPC

Field	Type	Label	Description
chain_id	string		ChainID is the chain ID to identify which chain ID is used for the verification
validator	string		Validator is a validator address
request_id	int64		RequestID is oracle request ID
external_id	int64		ExternalID is an oracle's external ID
reporter	string		Reporter is an bech32-encoded public key of the reporter authorized by the validator
signature	bytes		Signature is a signature signed by the reporter using reporter's private key

QueryRequestVerificationResponse

QueryRequestVerificationResponse is response type for the Query/RequestVerification RPC

Field	Type	Label	Description
chain_id	string		ChainID is the targeted chain ID
validator	string		Validator is the targeted validator address
request_id	int64		RequestID is the ID of targeted request
external_id	int64		ExternalID is the ID of targeted oracle's external data source
data_source_id	int64		DataSourceID is the ID of a data source that relates to the targeted external ID

QueryValidatorRequest

QueryValidatorRequest is request type for the Query/Validator RPC method.

Field	Type	Label	Description
validator_address	string		ValidatorAddress is address of a validator

QueryValidatorResponse

QueryValidatorResponse is response type for the Query/Validator RPC method.

Field	Type	Label	Description
status	ValidatorStatus		Status is status of a validator e.g. active/inactive

Query

Query defines the gRPC querier service.

Method Name	Request Type	Response Type	Description
Counts	QueryCountsRequest	QueryCountsResponse	Counts queries the number of existing data sources, oracle scripts, and requests.
Data	QueryDataRequest	QueryDataResponse	Data queries content of the data source or oracle script for given SHA256 file hash.
DataSource	QueryDataSourceRequest	QueryDataSourceResponse	DataSource queries data source summary info for given data source id.
OracleScript	QueryOracleScriptRequest	QueryOracleScriptResponse	OracleScript queries oracle script summary info for given oracle script id.
Request	QueryRequestRequest	QueryRequestResponse	Request queries request info for given request id.
PendingRequests	QueryPendingRequestsRequest	QueryPendingRequestsResponse	PendingRequests queries list of pending request IDs assigned to given validator.
Validator	QueryValidatorRequest	QueryValidatorResponse	Validator queries properties of given validator address.
Reporters	QueryReportersRequest	QueryReportersResponse	Reporters queries all reporters associated with given validator address.
ActiveValidators	QueryActiveValidatorsRequest	QueryActiveValidatorsResponse	ActiveValidators queries all active oracle validators.
Params	QueryParamsRequest	QueryParamsResponse	Params queries parameters used for runnning bandchain network.
RequestSearch	QueryRequestSearchRequest	QueryRequestSearchResponse	RequestSearch queries the latest request that match search criteria.
RequestPrice	QueryRequestPriceRequest	QueryRequestPriceResponse	RequestPrice queries the latest price on standard price reference oracle script.
RequestVerification	QueryRequestVerificationRequest	QueryRequestVerificationResponse	RequestVerification verifies a request to make sure that all information that will be used to report the data is valid
RequestPool	QueryRequestPoolRequest	QueryRequestPoolResponse	RequestPool queries the request pool information corresponding to the given port, channel, and request key.

Top

oracle/v1/tx.proto

MsgActivate

MsgEditOracleScript is a message for activating a validator to become an oracle provider. However, the activation can be revoked once the validator is unable to provide data to fulfill requests

Field	Type	Label	Description
validator	string		Validator is the validator address who sign this message and request to be activated.

MsgActivateResponse

MsgActivateResponse is response data for MsgActivate message

MsgAddReporter

MsgAddReporter is a message for adding a new reporter for a validator.

Field	Type	Label	Description
validator	string		Validator is the validator address who requested to add a new reporter. Note that this is the signer.
reporter	string		Reporter is the account address to be added as a reporter to the validator.

MsgAddReporterResponse

MsgAddReporterResponse is response data for MsgAddReporter message

MsgCreateDataSource

MsgCreateDataSource is a message for creating a new data source.

Field	Type	Label	Description
name	string		Name is the name of this data source used for display (optional).
description	string		Description is the description of this data source used for display (optional).
executable	bytes		Executable is the content of executable script or binary file to be run by validators upon execution.
fee	cosmos.base.v1beta1.Coin	repeated	Fee is the data source fee per ask_count that data provider will receive from requester.
treasury	string		Treasury is the account address who receive data source fee from requester.
owner	string		Owner is the account address who is allowed to make further changes to the data source.
sender	string		Sender is the signer of this message.

MsgCreateDataSourceResponse

MsgCreateDataSourceResponse is response data for MsgCreateDataSource message

MsgCreateOracleScript

MsgCreateOracleScript is a message for creating an oracle script.

Field	Type	Label	Description
name	string		Name is the name of this oracle script used for display (optional).
description	string		Description is the description of this oracle script used for display (optional).
schema	string		Schema is the OBI schema of this oracle script (optional).
source_code_url	string		SourceCodeURL is the absolute URI to the script's source code (optional).
code	bytes		Code is the oracle WebAssembly binary code. Can be raw of gzip compressed.
owner	string		Owner is the address who is allowed to make further changes to the oracle script.
sender	string		Sender is the signer of this message.

MsgCreateOracleScriptResponse

MsgCreateOracleScriptResponse is response data for MsgCreateOracleScript message

MsgEditDataSource

MsgEditDataSource is a message for editing an existing data source.

Field	Type	Label	Description
data_source_id	int64		DataSourceID is the unique identifier of the data source to be edited.
name	string		Name is the name of this data source used for display (optional).
description	string		Description is the description of this data source used for display (optional).
executable	bytes		Executable is the executable script or binary to be run by validators upon execution.
fee	cosmos.base.v1beta1.Coin	repeated	Fee is the data source fee per ask_count that data provider will receive from requester.
treasury	string		Treasury is the address who receive data source fee from requester.
owner	string		Owner is the address who is allowed to make further changes to the data source.
sender	string		Sender is the signer of this message. Must be the current data source's owner.

MsgEditDataSourceResponse

MsgEditDataSourceResponse is response data for MsgEditDataSource message

MsgEditOracleScript

MsgEditOracleScript is a message for editing an existing oracle script.

Field	Type	Label	Description
oracle_script_id	int64		OracleScriptID is the unique identifier of the oracle script to be edited.
name	string		Name is the name of this oracle script used for display (optional).
description	string		Description is the description of this oracle script used for display (optional).
schema	string		Schema is the OBI schema of this oracle script (optional).
source_code_url	string		SourceCodeURL is the absolute URI to the script's source code (optional).
code	bytes		Code is the oracle WebAssembly binary code. Can be raw of gzip compressed.
owner	string		Owner is an account address who is allowed to make further changes to the oracle script.
sender	string		Sender is an account address who sign this message. Must be the current oracle script's owner.

MsgEditOracleScriptResponse

MsgEditOracleScriptResponse is response data for MsgEditOracleScript message

MsgRemoveReporter

MsgAddReporter is a message for removing an existing reporter from a validator.

Field	Type	Label	Description
validator	string		Validator is the validator address who requested to remove an existing reporter. Note that this is the signer.
reporter	string		Reporter is the account address to be removed from being the validator's reporter.

MsgRemoveReporterResponse

MsgRemoveReporterResponse is response data for MsgRemoveReporter message

MsgReportData

MsgReportData is a message for reporting to a data request by a validator.

Field	Type	Label	Description
request_id	int64		RequestID is the identifier of the request to be reported to.
raw_reports	RawReport	repeated	RawReports is the list of report information provided by data sources identified by external ID
validator	string		Validator is the address of the validator that owns this report.
reporter	string		Reporter is the message signer who submits this report transaction for the validator.

MsgReportDataResponse

MsgReportDataResponse is response data for MsgReportData message

MsgRequestData

MsgRequestData is a message for sending a data oracle request.

Field	Type	Label	Description
oracle_script_id	int64		OracleScriptID is the identifier of the oracle script to be called.
calldata	bytes		Calldata is the OBI-encoded call parameters for the oracle script.
ask_count	uint64		AskCount is the number of validators to perform the oracle task.
min_count	uint64		MinCount is the minimum number of validators sufficient to resolve the oracle tasks.
client_id	string		ClientID is the client-provided unique identifier to track the request.
fee_limit	cosmos.base.v1beta1.Coin	repeated	FeeLimit is the maximum tokens that will be paid to all data source providers.
prepare_gas	uint64		PrepareGas is amount of gas to pay to prepare raw requests
execute_gas	uint64		ExecuteGas is amount of gas to reserve for executing
sender	string		Sender is an account address of message sender.

MsgRequestDataResponse

MsgRequestDataResponse is response data for MsgRequestData message

Msg

Msg defines the oracle Msg service.

Method Name	Request Type	Response Type	Description
RequestData	MsgRequestData	MsgRequestDataResponse	RequestData defines a method for submitting a new request.
ReportData	MsgReportData	MsgReportDataResponse	ReportData defines a method for reporting a data to resolve the request.
CreateDataSource	MsgCreateDataSource	MsgCreateDataSourceResponse	CreateDataSource defines a method for creating a new data source.
EditDataSource	MsgEditDataSource	MsgEditDataSourceResponse	EditDataSource defines a method for editing an existing data source.
CreateOracleScript	MsgCreateOracleScript	MsgCreateOracleScriptResponse	CreateOracleScript defines a method for creating a new oracle script.
EditOracleScript	MsgEditOracleScript	MsgEditOracleScriptResponse	EditOracleScript defines a method for editing an existing oracle script.
Activate	MsgActivate	MsgActivateResponse	Activate defines a method for applying to be an oracle validator.
AddReporter	MsgAddReporter	MsgAddReporterResponse	AddReporter defines a method for adding a new reporter for a validator.
RemoveReporter	MsgRemoveReporter	MsgRemoveReporterResponse	RemoveReporter defines a method for removing an reporter from a validator

Top

oracle/v1/genesis.proto

GenesisState

GenesisState defines the oracle module's genesis state.

Field	Type	Label	Description
params	Params		Params defines all the paramaters of the module.
data_sources	DataSource	repeated	DataSources are data sources to be installed during genesis phase
oracle_scripts	OracleScript	repeated	OracleScripts are list of oracle scripts to be installed during genesis phase
reporters	ReportersPerValidator	repeated	Reporters are mapping between reporter's account address (key) and validator's validator address (value)

Scalar Value Types

.proto Type	Notes	C++	Java	Python	Go	C#	PHP	Ruby
double		double	double	float	float64	double	float	Float
float		float	float	float	float32	float	float	Float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	int32	int	integer	Bignum or Fixnum (as required)
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long	int64	long	integer/string	Bignum
uint32	Uses variable-length encoding.	uint32	int	int/long	uint32	uint	integer	Bignum or Fixnum (as required)
uint64	Uses variable-length encoding.	uint64	long	int/long	uint64	ulong	integer/string	Bignum or Fixnum (as required)
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	int32	int	integer	Bignum or Fixnum (as required)
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long	int64	long	integer/string	Bignum
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^28.	uint32	int	int	uint32	uint	integer	Bignum or Fixnum (as required)
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^56.	uint64	long	int/long	uint64	ulong	integer/string	Bignum
sfixed32	Always four bytes.	int32	int	int	int32	int	integer	Bignum or Fixnum (as required)
sfixed64	Always eight bytes.	int64	long	int/long	int64	long	integer/string	Bignum
bool		bool	boolean	boolean	bool	bool	boolean	TrueClass/FalseClass
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode	string	string	string	String (UTF-8)
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str	[]byte	ByteString	string	String (ASCII-8BIT)

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Cosmos IBC Integration

> **NOTE**
> This section presented an overview of how IBC can be used to make an oracle data request on BandChain.
>
> For more information on IBC itself, its architecture, and other related topics, please see Cosmos' Interchain Standards documentation.

IBC Overview

In addition to our own lite client protocol, we also allow interaction with our data oracle through Cosmos' Inter-Blockchain-Communication, or IBC, protocol. This protocol allows other IBC-compatible blockchains to request data from BandChain.

BandChain-Specific IBC Data Packet

OracleRequestPacketData

`OracleRequestPacketData` is a data packet sent by a blockchain to BandChain's oracle to request data. It contains the following parameters:

Parameter	Type	Description
ClientID	string	The unique identifier of this oracle request, as specified by the client. This same unique ID will be sent back to the requester with the oracle response
OracleScriptID	int64	The unique identifier number assigned to the oracle script when it was first registered on Bandchain
Calldata	string	The data that was passed over to the oracle script to use during its execution eg. list of requested symbols and multiplier
AskCount	uint64	The number of validators that are requested to respond to this request
MinCount	uint64	The minimum number of validators necessary for the request to proceed to the execution phase
FeeLimit	sdk.Coins	The maximum tokens that will be paid to all data source providers
PrepareGas	uint64	The amount of gas to pay to prepare raw requests
ExecuteGas	uint64	The amount of gas reserved for executing the oracle script during execution phase

OracleResponsePacketData

Subsequently, this is the packet that will be relayed from BandChain back to the requester's chain. It contains information on the response parameters as well as the requested data itself.

Parameter	Type	Description
ClientID	string	The unique identifier of this oracle request, as specified by the client. This matches the ID stated in the corresponding `OracleRequestPacketData`
RequestID	int64	The unique identifier number of the particular request
AnsCount	uint64	The number of validators that answers the request, retrieved the data, and submitted a report
RequestTime	int64	The timestamp of when the request was made
ResolveTime	int64	The timestamp of when the last validator submitted the report and the request is resolved
ResolveStatus	int32	The resolve status of the request. See here for the full list of possible values
Result	[]byte	The aggregated value of the results returned by the validators

Requesting Data Through IBC

To make a request to BandChain's oracle using IBC, the module on another IBC-compatible blockchain must first initialize a communication tunnel with the oracle module on BandChain. Once the connection has been established, a pair of channel identifiers is generated -- one for the counterparty chain and one for BandChain.

The channel identifier is an important piece for the counterparty IBC module to route outgoing oracle request packets to the targeted oracle module on BandChain.

Similarly, BandChain's oracle module uses the channel identifier when sending back the oracle response. This means these channel identifiers have to be unique within each module, and the right channel identifier needs to be specified when making an oracle request from the counterparty chain.

IBC Workflow

Once a relayer has been set up, the module on another IBC-compatible blockchain looking to make the request must generate an `OracleRequestPacketData` data packet to be relayed. Using their chain's IBC module, they must then relay the message through to BandChain's own IBC module, which will proceed to further send it to the chain's `oracle` module. Once the request packet is successfully received, the subsequent flow is the almost the same as how BandChain handles a native `MsgRequestData` message type with a few additional steps. To summarize, the data request flow consists of the following steps:

- First, requesters create a request from their chain which are then relayed to BandChain.
- Once the request is submitted to BandChain, the oracle module fetches the corresponding oracle script and starts the oracle script's preparation phase returning information of all related data sources.
- Then BandChain performs various checks such as preparation phase smaller than the provided prepare gas (This is actually done in the oracle's script preparation phase) and the total fee for the request does not exceed the provided fee limit (More details can be found in the next section).
- (IBC Process Only) Then an acknowledgement is sent back to the requester's chain which either contains the error from the checks or the request identifier created by BandChain.
- If there is no error, the request is then broadcasted. Each validator selected for the particular request will then proceeed to retrieve data from each of the data source
- If a validator's retrieval is successful, they will submit back a report to BandChain containing the result they received from each of the data source.
- If the number of validators that managed to successfully submit the report exceeds the `minCount` specified in the `OracleRequestPacketData`, BandChain then computes and stores an aggregate final value.
- (IBC Process Only) The final result is also directly relayed back to the requesting chain and module in the form of a `OracleResponsePacketData` data packet.

As a slight aside, a data request to BandChain generally takes roughly 5 seconds from submitting the initial request until the requester received back the requested result. This is because BandChain's blocktime is set at approximately 3 seconds

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Decentralized Validator Sampling

Motivation

All BandChain validators have an additional responsibility other than voting and proposing new blocks. That is submitting reports for incoming oracle requests. However, asking all validators to retrieve results and waiting for their reports for every request is time inefficient, since this would become a single queue for requests bottlenecking the whole system of BandChain. Since there are many validators on BandChain, we can process multiple requests simultaneously and mitigate the bottleneck effect by distributing each oracle request to a random subset of the validators.

The randomized selection is based on the voting power of validators. The underlying reason is the same as that of Proof-of-Stake concept. Validators with higher voting powers are those that have staked more BAND token. Due to the higher commitment to risks, they are incentivized to be good actors and to deliver the correct oracle results within the expected time interval.

Those with lower voting power have less risk exposure and thus are generally less incentivized to be a good actor as they do not have much to lose. Nevertheless, we still want every validator to have a chance in participating in the oracle retrieving result task. Otherwise, the platform would be centralized around few validators with high voting powers, undermining the very goal of any blockchain -- decentralization.

Therefore, when determining which algorithm to use in selecting the validators to respond to an oracle data request, there are two main characteristics that we have incorporated:

1. A validator with a higher voting power should have a higher chance of being chosen than one with a lower voting power
2. Every validator should still have a chance to be selected

Validators

From the pool of all validators, we have to select a specific set of validators from that pool. To do this, we make use of a random number generator.

Random Number and Seed Generation

As with most random number generator, our number generation proces require the use of a seed. In this specific case, our seed comprises of:

- A list of `blockHashes`
- The `requestID` of the request the validators are being chosen for
- `chainID` of BandChain

In the case of the list of `blockHashes`, we will use the blockHashes of the previous n blocks (currently we use 32). We then take `$\frac{32}{n}$` bytes from each of the hashes and concatenate them. The purpose of this is to prevent any potentially malicious validators from influencing the entire seed in the turn they have to propose the block. Even if they attempt to intentionally construct certain `blockHashes` in their proposed block, they can only control $\frac{n}{32}$ of the seed. Finally, we concatenate the result from `blockHashes` with `requestID` and `chainID` to create the seed.

Manipulation Resistant Seed Generation

After we have used our concatenated seed to generate a random number, we then use that value to select the validator for that round.

To do so, we again assume that the validators in the selection space is sorted in descending order of voting power (as shown below from validator with 100 voting power, to 85, then to 70 and so on.)

Then, we imagine that we have a cumulative scale running across that list, with the values being the validator's voting power. For example:

where $k > l > m$

${1, k}$ $\hspace{90pt}$ is assigned to the first validator in line with k voting power
${k+1, k+l}$ $\hspace{57pt}$ to the second in line with l power
${(k+l)+1, (k+l)+m}$ $\hspace{8pt}$to the third in line with m power
$. . .$
and so forth.

With that, the specific range in which our random number falls in along that cumulative scale determine which validator is ultimately chosen for that round. A visual representation of this method is shown below.

Setup

Selection using a Random Number

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Lite Client Protocol

In addition to the native IBC connections, we also provide a lite client for anyone who requested data from our oracle to verify the validity of the result they received. An instance of this client exists on each of the blockchains to which Band has integrated.

When someone submits a verification request to our lite client, they must also send in the encoded result they got from BandChain oracle. This is because the result is not just the data they requested, but also contains information on the request itself as well as the associated response. The lite client use these information to prove that the data the user requested exists on BandChain, thus verifying the oracle result's validity.

The lite client's verification process checks for 3 conditions:

1. that the proof received in the request can be used to construct a valid block header
2. that using the constructed block header, it can recover a valid set of validator addresses who signed on the block
3. that those validators have sufficient total voting power relative to the system total

The diagram below illustrates the above steps.

Lite Client Request Flow

Constructing the Block Header from the Oracle Data Proof

The proof that BandChain's oracle returns is a packet called `result` which encodes the information from both the oracle request sent to BandChain and the oracle response aggregated from reports submitted by validators assigned to this request. Some of the information includes

- The identifier of the oracle script requested,
- The number of validators that are requested to respond to this request,
- The number of validators that actually responded to the request,
- The timestamp of when the request was sent and when it was resolved to a final result
- The actual final result itself if the request was successful.

For a full detailed breakdown of the packet's contents, please see our GitHub repository.

The task of the lite client is then to use this packet to eventually arrive at the block header. The steps that make up this process are as follows:

- Use the proof sent to construct the oracle store's root hash
- Combine the oracle store hash with the hashes of the other stores in our application to compute the appHash
- Finally, use the appHash, in combination with other block information hashes, to compute the blockHash

Constructing the Oracle Store's Root Hash

Oracle Store Tree Contents

BandChain's oracle system resides in an `oracle` Cosmos module, also known as the `oracle` store. Each of these stores can then be represented as an iAVL tree, where the bottom or leaves of these store trees contains the byte representation of the data in that module. In our case of the `oracle` store root, the only piece of data that we will be looking at is the `result` packet.

As mentioned previously, the `result` packet contains information related to both the request that the user made to our oracle, such as the oracle script identifier and the number of requested validators, and the corresponding oracle response composed from validators' reports, such as the actual number of validators that responded to the request as well as the actual result value itself.

This `result` packet is what the lite client essentially returns to the requester upon successful validation. By also returning contextual information on the request and response, in addition to the actual result value itself, we aim to give as much information as possible for the user to use in their application or for any further validation they might want to perform.

Constructing the oracle store leaf node

Using the `result`, we can create an intermediary hash value called the `dataHash` by encoding then hashing the `result`. If we then append this `dataHash` by other information such as the version (i.e., the latest block height that the data node was updated), and the request ID of the request, we arrive at the leaf node of the oracle store tree, also known as the `resultHash`.

After we have the leaf node, we then need to use that node to gradually climb up the tree to reach the store's root node. To help us do so, we use an additional piece of information in the proof; the `merklePaths`.

Computing the oracle store root hash

Shown in the figure above, the left node of the `result` we want to verify is represented by the black node. The `merklePaths` we mentioned is a Merkle proof that shows how the `dataHash` leaf node we just computed is part of the larger oracle tree. The proof's content is the list of "Merkle paths" from the leaf to the root of the tree. Each merkle path contains the sibling hash for each step represented by the gray nodes. With a node hash and a corresponding merkle path, we can get the parent hash of the node hash and the sibling hash in the merkle path. Hence, we can the use these Merkle paths to compute the parent hash of our `dataHash`. If we then repeat this process, we can gradually climb up the store tree, finally getting the oracle store root hash we want. In the above figure, all parent hashes computed along the tree climbing process starting from the black node are represented by blue nodes.

Computing the appHash and the blockHash

After we have the oracle store root hash, we can begin to iteratively combine it with the hashes of the other stores in our application to compute the `appHash`. Since all the stores are stored in a form of tree, we can use the same climbing process to reach the root of this tree or the `appHash`. Note that the starting point now is the oracle store hash which is the only black node as depicted in the figure above.

We can then use that `appHash` to finally compute the `blockHash` using the very same method.

Recovering Signer Addresses

After we have constructed a `blockHash`, we can move on to prove its validity by attempting to use it to recover the addresses of the validators who signed on this block using Ethereum's ecrecover opcode. To ensure that the addresses we extracted are valid, we verify that each address we extract is unique.

As we recover each signer, we also add each extracted validator's voting power to a counting tally, which we will use in the next step.

Checking Total Voting Power

Once we have extracted all of the validators and ensure that the extracted order is correct, we proceed to check if the tallied voting power is sufficient. Specifically, we check that the tallied value is at least two-thirds of the system's total voting power. This threshold check is to ensure that we reach consensus.

If the tallied voting power exceeds the two-third threshold, we have successfully proven that the proof is valid. Our lite client can then decode the result and return it to the requester to either use or further validate themselves.

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

On-chain Payment

Overview

When oracle scripts make inquiries from premium data sources, parties that request data from these oracle scripts must pay additional fees that are proportional to the fees charged by the premium data sources. These data sources collect fees every time a validator requests oracle data. Thus, the final fee charged for a single oracle request is the sum of all data source fees corresponding to the requested oracle script, multiplied by the number of requested validators.

Since each data source sets its own fees, which can vary, users must specify a `feeLimit` for each request, to prevent any unexpectedly high fees when requesting oracle data. The fee limit defines the amount of tokens allowed to be paid for the total data source fees corresponding to the request. If the fee limit is less than the total fees, then the request becomes invalid, and the associated transaction fails.

The party responsible for paying the fees depends on the origin of the request, whether it is a direct on-chain request or an IBC request from a counterparty chain. The fees are paid directly to treasury accounts owned by each data source.

Direct Request

Direct Request Payment Flow

The following is an example of the on-chain payment flow when requesting oracle data directly on BandChain.

A user begins by making an oracle request, providing a specific oracle script ID and a desired fee limit. Once BandChain receives the request, it fetches the oracle script and calls the preparation step of the oracle script.

onchain-direct

After the preparation step is complete, BandChain obtains all the data sources required for this oracle script. The provided fee limit is used to ensure that the total fee does not exceed the expected price. If the fee limit is insufficient, the transaction is reverted. Otherwise, the balance of the user account is deducted by the total fee, and the treasury accounts of the data sources receive funds according to their specified fee times the number of requested validators.

IBC request

To make an oracle request via IBC, a communication channel must be created between a counterparty chain and BandChain, as stated in IBC. The main challenge when requesting from another chain is that the requester is an address on the other chain that cannot pay the data source fees on BandChain. Therefore, the fees are instead paid by the account of a relayer that the counterparty chain uses to send packets.

The relayer account is an address account owned by an entity from another chain, as the chain has to manage the relayer. Note that the method by which users fund the relayer account will depend entirely on the protocol of the counterparty chains.

IBC Request Payment Flow

onchain-ibc

Once the relayer account is funded, the payment flow for IBC requests is very similar to that of the direct request. The additional layer is that the flow begins with a user on the counterparty chain. The request is then relayed from the counterparty chain to BandChain via a relayer through a specific port and channel.

After BandChain receives the request via a relayer, the payment flow is almost identical to the payment flow of a direct request, except that the account used for paying all the fees is now the relayer account.

Band Protocol Documentation
Introduction
Core Concepts
BAND Token
Gas and Fees
Accounts and Wallets
Protocol Messages
Protobuf Documentation
Cosmos IBC Integration
Decentralized Validator Sampling
Lite Client Protocol
On-chain Payment
Oracle WebAssembly (Owasm)
Products
Develop
Nodes & Validators
Staking

Oracle WebAssembly (Owasm)

Oracle WebAssembly, or Owasm for short, is Band Protocol's Domain Specific Language (DSL) on top of the Rust programing language for writing oracle scripts to be used in the BandChain ecosystem. The Owasm library consists of two primary modules: `owasm/oei` and `owasm/ext`.

- **Owasm/OEI** - The OEI modules defines a set of functions that are part of the Owasm Oracle Environment Interface. These functions are then accessible to an oracle script during its execution. The complete list of functions and implementation can be found here.
- **Owasm/Ext** - The Owasm extension module provides a convenient way to write oracle scripts that connect to various public APIs. Examples of these are functions to calculate the mean, median, and majority values from the validator's reported results, which can be used during the aggregation phase of an oracle script. The full list of functions and its implementation can be found here.

Usage

To illustrate an example usage of the Owasm library, we will be using the example below. The code is based off an oracle script for retrieving the price of a stock.

```rust
use obi::{OBIDeserialize, OBISchema, OBISerialize};
use owasm::{execute_entry_point, ext, oei, prepare_entry_point};

#[derive(OBIDeserialize, OBISchema)]
struct Input {
    symbol: String,
    multiplier: u64,
}

#[derive(OBISerialize, OBISchema)]
struct Output {
    px: u64,
}

#[no_mangle]
fn prepare_impl(input: Input) {
    // Asking data source #14 for asset price
    oei::ask_external_data(14, 1, &input.symbol.as_bytes());
}

#[no_mangle]
fn execute_impl(input: Input) -> Output {
    let avg: f64 = ext::load_average(1);
    Output { px: (avg * input.multiplier as f64) as u64 }
}

prepare_entry_point!(prepare_impl);
execute_entry_point!(execute_impl);
```

The script starts off by defining the input and output structs. In this case, the input comprises of the stock ticker/symbol (`string`) and the multiplier we want to multiply the stock's price by (`u64`). On the other hand, the output is simply the price of the stock multiplied by the multiplier, returned as a `u64` value.

Once the structs and types of both input and output have been determined, we move on to defining the preparation and execution phases of the oracle script, defined by `prepare_impl` and `execute_impl`, respectively.

In order to call these functions, we need to pass appriopriate input values and make the function calls. To do so, oracle script writer can use our macros defined in `macros.rs`, also shown below. The aim of these macros is to reduce the load of the script writer by handling the work of retrieving the calldata, deserializing it, and using it to construct the appropriate input struct for them.

```rust
#[macro_export]
macro_rules! prepare_entry_point {
    ($name:ident) => {
        #[no_mangle]
        pub fn prepare() {
            $name(OBIDeserialize::try_from_slice(&oei::get_calldata()).unwrap());
        }
    };
}

#[macro_export]
macro_rules! execute_entry_point {
    ($name:ident) => {
        #[no_mangle]
        pub fn execute() {
            oei::save_return_data(
                &$name(OBIDeserialize::try_from_slice(&oei::get_calldata()).unwrap())
                    .try_to_vec()
                    .unwrap(),
            );
        }
    };
}
```

The last two lines of the oracle script above shows the macros in action.

Preparation phase

The `prepare_impl` function takes the previously-defined input struct as an argument. The function then has only one main task; calling the `ask_external_data` function in `oei` module.

In case extra information from relating oracle request is needed for implementing some logic before calling `ask_external_data`, here are functions and their details from `oei` module which can be used in `prepare_impl`.

- `get_ask_count()` returns `i64`
  - Returns the number of validators to asked to report for this oracle request.
- `get_min_count()` returns `i64`
  - Returns the minimum number of data reports as specified by the oracle request.
- `get_calldata()` returns `Vec<u8>`
  - Returns the raw calldata as specified when the oracle request is submitted.
  - **Note:** This function is already called in macros when preparing the input.
- `ask_external_data(eid: i64, did: i64, calldata: &[u8])`
  - Takes in external id, data source id, and data in bytes.
  - Issues a new raw request to the host environment using the specified data source ID and calldata, and assigns it to the given external ID.

Execution phase

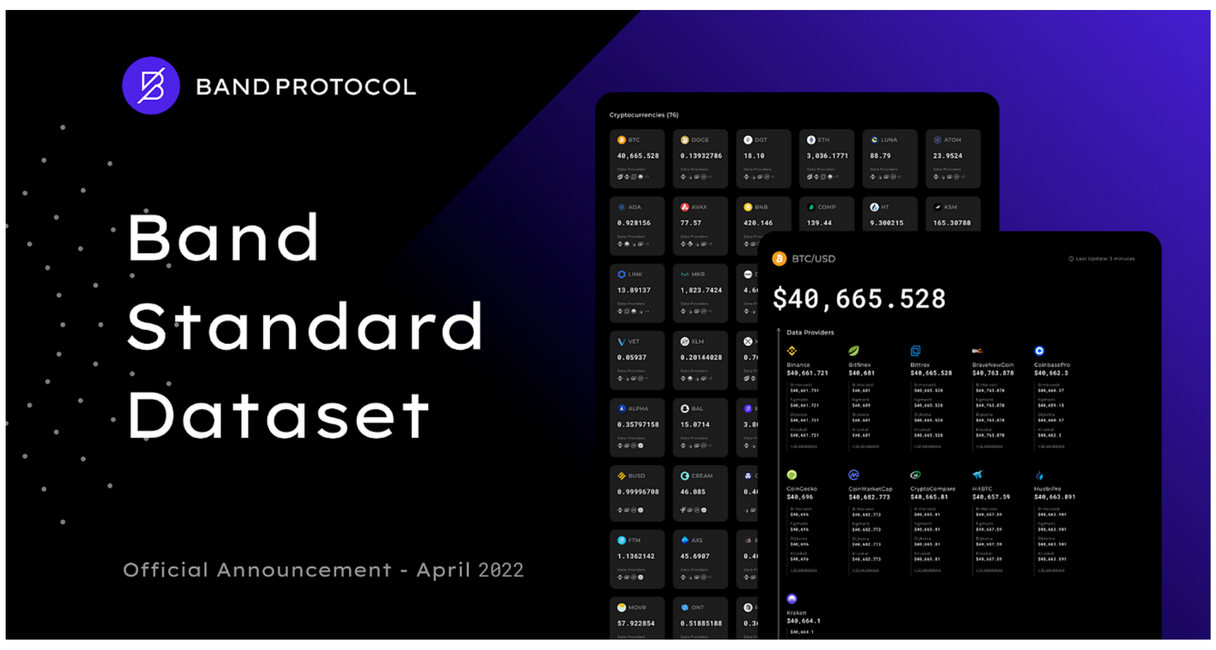
The `execute_impl` function takes in the input type as an argument as well, but also returns the output struct type. It then starts by computing the final value of the request through calling `load_average` function from the `ext` module and proceed to use the computed average to construct and return the appropriate output struct.

Below is the list of functions from `oei` that can be called in `execute_impl`.

- `get_ask_count()` returns `i64`
  - Returns the number of validators to asked to report for this oracle request.
- `get_min_count()` returns `i64`
  - Returns the minimum number of data reports as specified by the oracle request.
- `get_ans_count()` returns `i64`
  - Returns the number of validators that report data to this oracle request. Must only be called during execution phase.
- `get_calldata()` returns `Vec<u8>`
  - Returns the raw calldata as specified when the oracle request is submitted.
  - **Note:** This function is already called in macros when preparing the input.
- `save_return_data(data: &[u8])`
  - Saves the given data as the result of the oracle execution. Must only be called during execution phase and must be called exactly once.
- `get_external_data(eid: i64, vid: i64)` returns `Result<String, i64>`
  - Returns the data reported from the given validator index for the given external data ID. Result is OK if the validator reports data with zero return status, and Err otherwise.

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Introduction
Using Band Standard Dataset
Band VRF
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

Introduction



Overview

Band Standard Dataset, an exhaustive reference price dataset with frequent updates and comprehensive, customizable feeds. With Band Standard Dataset is will help to connect your smart contracts to the real-world data with expansive support for crypto assets, commodities and foreign exchange rates.

The Band Standard Dataset provides aggregated data from various data sources through a decentralized network of independent node operators. The Decentralized Data Model provides a detailed description of this process.

These Standard Dataset use cryptocurrencies price data aggregated from a number of premium data sources, including:

- CoinGecko
- CryptoCompare
- CoinMarketCap
- Binance
- Huobi Pro
- Coinbase Pro
- Bibox
- Bitfinex
- Bittrex
- HitBTC
- Kraken
- OKX

For the FX and commodity prices, we also aggregate data from premium sources:

- Fixer
- Open Exchange Rates
- XE
- AlphaVantage.

Data from these sources are retrieved by a subset of BandChain's 71+ reputable validators and medianized on two layers, the validator and data provider level, which is then aggregated on BandChain into a single final result to be stored on the blockchain state. The data retrieval, aggregation, and storage process is transparent, publicly viewable, auditable, and verifiable through our Cosmoscan, our official block explorer for BandChain.

Why Use Band Standard Dataset

Optimized for Efficiency and Cost-Effectiveness

Our Standard Dataset solutions are heavily optimized for gas usage, allowing developers to query multiple price pairs at once and retrieve accurate and reliable price updates for many feeds in the same block.

End-to-End Flexibility

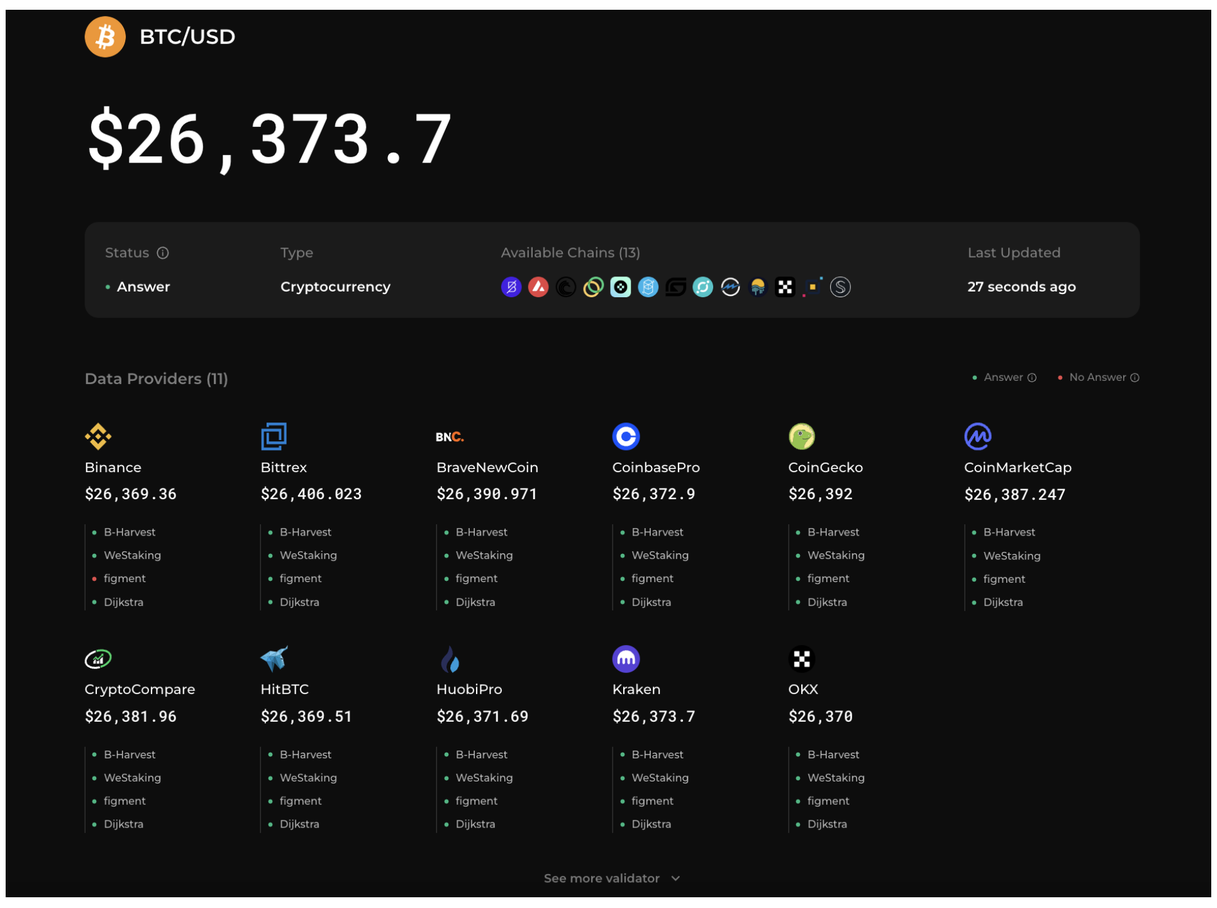
In addition, developers can query any price pair, not just the standard `<TOKEN>/USD`, `<TOKEN>/BTC`, and `<TOKEN>/ETH` pairs. Price feeds can be paired against any of the 100+ feeds supported on the Band Standard Dataset, including pairs such as `BTC/USD`, `BTC/ETH`, `ETH/CNY`, `EUR/BTC`, and `DOT/ATOM`. This flexibility, combined with the ability to query multiple price pairs simultaneously, significantly reduces the engineering resources required for our feed users and ensures that the resulting code is simpler and less error-prone.

Constantly Increasing Diversity

BandChain's infrastructure is designed with full customizability, allowing for streamlined and faster support for new tokens or price data. This greatly benefits developers and protocols looking to support new or less-known tokens.

Over 100+ Secure Price Feeds Available for Developers

The Standard Dataset currently offers over 100 asset price feeds, ranging from various cryptocurrency tokens to various foreign exchanges and commodities.



Symbol Listing Criteria

Symbols listed on the Band Standard Dataset are those that have been verified and passed several criteria including, but not limited to:

1. Listed on at least 4 sources which are either aggregators or exchanges that are listed above.
2. For each centralized exchange source, the pair must be a reputable stablecoin pairing and maintain an average daily trading volume of $1,000,000 within the last 90 days.
3. For tokens listed and traded on DEXes, they must have a minimum liquidity of $10Mn.

The first criteria guarantees the robustness of our feed in the event that one of the sources is down and/or reports incorrect prices. We can rest assured that Band Oracle would still be able to provide accurate price feeds in a timely manner. The second criteria serves to minimise the possibility of prices being easily manipulated.

As of today, the Band Standard Dataset is composed of 88 cryptocurrencies, 25 foreign exchange rates, and 2 commodities. The Band Standard Dataset is our first-step in entrusting the Band Standard with our partners, be it data users developers, data providers as well as our validators.

By listing your symbol on the Band Standard dataset, you gain instant credibility and recognition within the blockchain community. Imagine the opportunities that await as your project becomes discoverable by thousands of users searching for reliable, verified data sources.

🔥 Don't miss out on the chance to amplify your project's visibility and reach 🔥

👉 Join us today at Band Standard Dataset and take advantage of Band Protocol's Data Marketplace.

💪 Let's revolutionize the blockchain space together and unlock new possibilities for your symbol's success!

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Introduction
Using Band Standard Dataset
With Solidity
With CosmWasm
On Client Libraries
Through a REST API Endpoint
Band VRF
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

With Solidity

We have implemented a new `StdReference` interface contract to allow anyone to query data from our Standard Dataset.

To query prices from Band Protocol's oracle, a smart contract should reference Band's `StdReference` contract, specifically the `getReferenceData` and `getReferenceDatabulk` methods.

getReferenceData

`getReferenceData` takes two strings (the `base` and `quote` symbol) as the inputs, respectively. It then queries the `StdReference` contract for the latest rates for those two tokens, and returns a `ReferenceData` struct, shown below.

Input

- The base symbol as type `string`
- The quote symbol as type `string`

Output

The base quote pair result as type `ReferenceData`

```
struct ReferenceData {
    uint256 rate;
    uint256 lastUpdatedBase;
    uint256 lastUpdatedQuote;
}
```

The `ReferenceData` struct has the following elements:

- Rate: the exchange rate in terms of `base/quote`. The value returned is multiplied by `1e18`
- Last updated base: the last time when the base price was updated (since UNIX epoch)
- Last updated quote: the last time when the quoted price was updated (since UNIX epoch)

Example

For example, if we wanted to query the price of `BTC/USD`, the demo contract below shows how this can be done.

```
import interfaces/IStdReference.sol

contract Demo {
    IStdReference public ref;

    constructor(IStdReference _ref) public {
        ref = _ref;
    }

    function demo() external view returns (IStdReference.ReferenceData memory) {
        return ref.getReferenceData("BTC", "USD");
    }
}
```

The result from `Demo()` would yield:

```
ReferenceData(23131270000000000000000, 1659588229, 1659589497)
```

Where the results can be interpreted as:

```
{
    rate: 23131270000000000000000, // 23131.27 of BTC/USD
    lastUpdatedBase: 1659588229, // 2022-08-04 04:43:49
    lastUpdatedQuote: 1659589497 // 2022-08-04 05:04:57
}
```

getReferenceDatabulk

The second function, `getReferenceDataBulk`, takes information as data arrays. For example, if you pass in `['BTC','BTC','ETH']` as **base** and `['USD','ETH','EUR']` as **quote**, the `ReferenceDatareturned` array contains the information regarding the following pairs:

- `BTC/USD`
- `BTC/ETH`
- `ETH/EUR`

Input

- An array of base symbols as type `string[]`
- An array of quote symbol as type `string[]`

Output

- An array of the base quote pair results as type `ReferenceData[]`

Example Contract

The following smart contract code provides some simple examples of the `StdReference` contract and the `getReferenceData` function - these are not meant for production. The `IStdReference.sol` interface defines `ReferenceData` structure and the functions available to make the queries.

```
pragma solidity 0.6.11;
pragma experimental ABIEncoderV2;

interface IStdReference {
    /// A structure returned whenever someone requests for standard reference data.
    struct ReferenceData {
        uint256 rate; // base/quote exchange rate, multiplied by 1e18.
        uint256 lastUpdatedBase; // UNIX epoch of the last time when base price gets updated.
        uint256 lastUpdatedQuote; // UNIX epoch of the last time when quote price gets updated.
    }

    /// Returns the price data for the given base/quote pair. Revert if not available.
    function getReferenceData(string memory _base, string memory _quote)
        external
        view
        returns (ReferenceData memory);

    /// Similar to getReferenceData, but with multiple base/quote pairs at once.
    function getReferenceDataBulk(string[] memory _bases, string[] memory _quotes)
        external
        view
        returns (ReferenceData[] memory);
}

contract DemoOracle {
    IStdReference ref;

    uint256 public price;

    constructor(IStdReference _ref) public {
        ref = _ref;
    }

    function getPrice() external view returns (uint256){
        IStdReference.ReferenceData memory data = ref.getReferenceData("BTC","USD");
        return data.rate;
    }

    function getMultiPrices() external view returns (uint256[] memory){
        string[] memory baseSymbols = new string[](2);
        baseSymbols[0] = "BTC";
        baseSymbols[1] = "BTC";

        string[] memory quoteSymbols = new string[](2);
        quoteSymbols[0] = "USD";
        quoteSymbols[1] = "ETH";
        IStdReference.ReferenceData[] memory data = ref.getReferenceDataBulk(baseSymbols,quoteSymbols);

        uint256[] memory prices = new uint256[](2);
        prices[0] = data[0].rate;
        prices[1] = data[1].rate;

        return prices;
    }

    function savePrice(string memory base, string memory quote) external {
        IStdReference.ReferenceData memory data = ref.getReferenceData(base,quote);
        price = data.rate;
    }
}
```

Available Reference Data Contracts

You can access the `StdReference` data aggregator contract on the following supported networks

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Introduction
Using Band Standard Dataset
With Solidity
With CosmWasm
On Client Libraries
Through a REST API Endpoint
Band VRF
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

With CosmWasm

CosmWasm is an advanced smart contracting platform built for the Cosmos ecosystem. *Cosm* refers to "Cosmos" while *Wasm* refers to WebAssembly. CosmWasm uniquely enables developers to build multi-chain smart contracts, making use of the InterBlockchain Communication (IBC) Protocol.

CosmWasm contract can update and fetch the prices from Band Standard Dataset using Band's CosmWasm contract, deployed on their network.

The deployed contract addresses can be found in Supported Blockchains Section.

Build

Contract

To compile all contracts, run the following script in the repo root: `/scripts/build_artifacts.sh` or the command below: The optimized wasm code and its checksums can be found in the `/artifacts` directory

```
docker run --rm -v "$(pwd)":/code \
  --mount type=volume,source="$(basename "$(pwd)")_cache",target=/code/target \
  --mount type=volume,source=registry_cache,target=/usr/local/cargo/registry \
  cosmwasm/workspace-optimizer:0.12.7
```

Schema

To generate the JSON schema files for the contract call, queries and query responses, run the following script in the repo root: `/scripts/build_schemas.sh` or run `cargo schema` in the smart contract directory.

Usage

To query the prices from Band Protocol's StdReference contracts, the contract looking to use the price values should query Band Protocol's `std_reference` contract.

QueryMsg

The query messages used to retrieve price data for price data are as follows:

```
pub enum QueryMsg {
    GetReferenceData {
        // Symbol pair to query where:
        // symbol_pair := (base_symbol, quote_symbol)
        // e.g. BTC/USD ≡ ("BTC", "USD")
        symbol_pair: (String, String),
    },
    GetReferenceDataBulk {
        // Vector of Symbol pair to query
        // e.g. <BTC/USD ETH/USD, BAND/BTC> ≡ <("BTC", "USD"), ("ETH", "USD"), ("BAND", "BTC")>
        symbol_pairs: Vec<(String, String)>,
    },
}
```

ReferenceData

`ReferenceData` is the struct that is returned when querying with `GetReferenceData` or `GetReferenceDataBulk` where the bulk variant returns `Vec<ReferenceData>`

`ReferenceData` is defined as:

```
pub struct ReferenceData {
    // Pair rate e.g. rate of BTC/USD
    pub rate: Uint256,
    // Unix time of when the base asset was last updated. e.g. Last update time of BTC in Unix time
    pub last_updated_base: Uint64,
    // Unix time of when the quote asset was last updated. e.g. Last update time of USD in Unix time
    pub last_updated_quote: Uint64,
}
```

Examples

Single Query

For example, if we wanted to query the price of `BTC/USD`, the demo function below shows how this can be done.

```
fn demo(
    std_ref_addr: Addr,
    symbol_pair: (String, String),
) -> StdResult<ReferenceData> {
    deps.querier.query_wasm_smart(
        &std_ref_addr,
        &QueryMsg::GetReferenceData {
            symbol_pair,
        },
    )
}
```

Where the result from `demo(std_ref_addr, ("BTC", "USD"))` would yield:

```
ReferenceData(2313127000000000000000, 1659588229, 1659589497)
```

and the results can be interpreted as:

- BTC/USD
  - `rate = 23131.27 BTC/USD`
  - `lastUpdatedBase = 1659588229`
  - `lastUpdatedQuote = 1659589497`

Bulk Query

```
fn demo(
    std_ref_addr: Addr,
    symbol_pairs: Vec<String>,
) -> StdResult<ReferenceData> {
    deps.querier.query_wasm_smart(
        &std_ref_addr,
        &QueryMsg::GetReferenceDataBulk {
            symbol_pairs,
        },
    )
}
```

Where the result from `demo(std_ref_addr, [("BTC", "USD"), ("ETH", "BTC")])` would yield:

```
[
    ReferenceData(2313127000000000000000, 1659588229, 1659589497),
    ReferenceData(71601775432131482, 1659588229, 1659588229)
]
```

and the results can be interpreted as:

- BTC/USD
  - `rate = 23131.27 BTC/USD`
  - `lastUpdatedBase = 1659588229`
  - `lastUpdatedQuote = 1659589497`
- ETH/BTC
  - `rate = 0.07160177543213148 ETH/BTC`
  - `lastUpdatedBase = 1659588229`
  - `lastUpdatedQuote = 1659588229`

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Introduction
Using Band Standard Dataset
With Solidity
With CosmWasm
On Client Libraries
Through a REST API Endpoint
Band VRF
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

On Client Libraries

BandChain.js

Along with the Standard Dataset, we have also released a new version of our BandChain.js 🪄 JavaScript client library. This new version comes with a new `getReferenceData` function that allows developers to query data from the Dataset.

To get started, the library needs to be installed:

```
npm install @bandprotocol/bandchain.js
```

Then, the library can be imported and used as follows:

```
const { Client } = require('@bandprotocol/bandchain.js')

// BandChain's Proof-of-Authority REST endpoint
const endpoint = 'https://laozi-testnet6.bandchain.org/grpc-web'
const client = new Client(endpoint)

const minCount = 3
const askCount = 4

// This example demonstrates how to query price data from
// Band's standard dataset
async function exampleGetReferenceData() {
  const rate = await client.getReferenceData(
    ['BTC/USD', 'BTC/ETH', 'EUR/USD', 'EUR/ETH'],
    minCount,
    askCount
  )
  return rate
}

;(async () => {
  console.log(await exampleGetReferenceData())
})()
```

For a full tutorial and more usage on BandChain.js, please refer to the Getting Started section.

PyBand

```
from pyband.client import Client

def main():
    # Step 1
    grpc_url = "<GRPC>" # ex.laozi-testnet6.bandchain.org(without https://)
    c = Client(grpc_url)

    # Step 2
    print(c.get_reference_data(["BTC/USD", "ETH/USD"], 3, 4))

if __name__ == "__main__":
    main()
```

And running the code above should return a result that looks like this.

```
[
  ReferencePrice(
    (pair = "BTC/USD"),
    (rate = 34614.1),
    (updated_at = ReferencePriceUpdated((base = 1625655764), (quote = 1625715134)))
  ),
  ReferencePrice(
    (pair = "ETH/USD"),
    (rate = 2372.53),
    (updated_at = ReferencePriceUpdated((base = 1625655764), (quote = 1625715134)))
  )
]
```

For a full tutorial and more usage on PyBand, please refer to the Getting Started section.

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Introduction
Using Band Standard Dataset
With Solidity
With CosmWasm
On Client Libraries
Through a REST API Endpoint
Band VRF
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

Through a REST API Endpoint

The service allows users to easily query for recent price updates via a REST API.

Our REST Endpoints can be found in this Section.

Get Latest Price on Standard Price Reference

Method: `GET`

Path: `/oracle/v1/request_prices`

Parameters: `?ask_count={askCount}&min_count={minCount}&symbols={symbol-1}&symbols={symbol-2}&...&symbols={symbols-n}`

```
$ curl -X GET "${REST_API}/oracle/v1/request_prices?symbols=BTC&symbols=ETH&ask_count=4&min_count=3"
{
  "price_results": [
    {
      "symbol": "BTC",
      "multiplier": "100",
      "px": "4822795",
      "request_id": "235866",
      "resolve_time": "1631766659"
    },
    {
      "symbol": "ETH",
      "multiplier": "100",
      "px": "303321",
      "request_id": "245820",
      "resolve_time": "1632163802"
    }
  ]
}
```

For more usage example of our REST API, can be found in Band CLI and REST Endpoints Section.

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Introduction
Getting Started
VRF integration
Example Use Cases
VRF Supported Blockchains
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

Introduction

Band Verifiable Random Function (VRF)

The foundation of the randomness matters for security reasons. Modern distributed applications (dApps) often rely on "good" randomness, generated freshly and independently of the application's state. In addition, the random result should have integrity, which means it should be verifiable, tamper-proof and unpredictable. For example, consider an online lottery in which participants place their bids, a random result comes up, and winnings are awarded according to the bid placement. Similarly, consider a leadership election (often used in committee-based blockchain platforms) that proceeds in rounds to randomly elect a leader among a set of participants. For such applications, it is crucial to guarantee that the randomness is sampled uniformly and independently of the application's state, therefore making it hard to predict as well as ensuring that there are no malicious actors who can affect the outcome of such lottery winnings and leadership elections.

Verifiable Random Function (VRF)

A commonly used technique for applications that require "good" random values to be produced rely on cryptography to produce values that are pseudorandom, i.e., impossible to distinguish from uniformly randomized values. A verifiable random function (VRF) is a mathematical operation that takes some input and produces a pseudorandom output along with the proof of authenticity of the output generation process. Challengers can verify the proof to ensure the pseudorandom result is valid and does not come out of thin air.

In general, the core of the VRF system will have a set of secret keys used to generate verifiable results and a set of corresponding public keys used to verify the results produced. For example, given a secret key generated in private and a cryptographic function that maps a seed to an output value with its proof. The crucial property is that someone that does not have access to the secret key cannot distinguish in polynomial time the output from a value that is sampled uniformly at random from the range of all possible outcomes.

VRF security properties

- **Unpredictably**: This ensures that the computed outputs are distributed in a way that is, for all practical purposes, uniformly random. It is a fundamental property of a VRF, as it says that the VRF behaves like a random oracle. In practice, this implies that anyone not knowing the secret key has no way to predict the outcome that is better than "randomly guessing" even when knowing the seed. So, if the input seeds are chosen with sufficiently high entropy, it is practically impossible to predict the output.
- **Uniqueness**: This ensures that, after the VRF providers publish their secret key, they can only produce proof that will convince others of the correct VRF output value for every seed. In other words, for a given (secret key, seed), it is incredibly hard to find two different VRF values, both of which pass the verification. This property is crucial to protect against a cheating actor that tries to claim a specific output other than the correct one for their purposes.
- **Collision-Resistance**: This ensures that it is computationally hard to find two different inputs, "seed1" and "seed2", with the same secret key to obtain the identical output value —much like the classic property of cryptographic hash functions. The difference is that for VRFs, this holds even against an adversary that knows the secret key. Note that this offers a different type of protection than the unique property. For example, it protects against a party that tries to claim an output computed from one input seed as if it was computed from a different "seed2".

Band VRF

Our solution for verifiably (pseudo-)randomness is based on the BandChain blockchain. Our protocol uses a verifiable random function (VRF) to cryptographically secure that produced results have not been tampered with. We will present in detail how our protocol operates in this document.

Band VRF extends from the general form of the VRF system to serve randomness requests for dApps, which is based on the distributed BandChain oracle network. BandChain is a public blockchain that provides APIs for data and services stored "off-chain" on the traditional web or from third-party providers. It supports generic data requests from other public blockchains and performs on-chain aggregation via the BandChain oracle scripts. The aggregation process works like a smart contract on the EVM platform, which executes data on-chain to produce the oracle results. The oracle results are also stored on-chain. After that, the results are returned to the calling dApp on the main blockchain (e.g., Ethereum), accompanied by a proof of authenticity via customized one-way bridges (or via an Inter-Blockchain Communication protocol). To guarantee verifiably "good" randomness suitable for security-critical applications, we deploy the cryptographic primitive of verifiable random functions. At a high level, a VRF provides values that are indistinguishable from uniformly random ones and can be verified for their authenticity concerning a pre-published public key.

We chose the VRF of this paper, which has already been adopted in various other protocols. The construction is based on a well-studied cryptographic hardness assumption over prime-order elliptic curve groups. For our instantiation, we chose the widely-used Ed25519 curve that achieves very good performance and has a transparent choice of parameters, as well as the Elligator, for our hash-to-curve instantiation. Our implementation is fully compliant with the VRF draft standard. Moreover, we implemented all the necessary techniques to achieve full security. For completeness, we include the pseudo-code description of our VRF below.

Public parameters: G is a cyclic group of prime order q and generator g. Let H_1 be a hash function mapping arbitrary-length bitstrings onto $G-\{1_G\}$, H_2 a function that takes the bitstring representation of an element of G and shortens it to length 2λ (λ is the security parameter), and H_3 a hash function mapping arbitrary-length inputs to λ-bit integers.

KeyGen Return secret key $sk \in \{1, \ldots, q-1\}$ chosen uniformly at random, and public key $pk = g^{sk}$.

Eval On input seed a, compute output y and proof π, as follows:

1. $h \leftarrow H_1(a)$ and $\gamma \leftarrow h^{sk}$
2. Choose random nonce $k \in \{0, \ldots, q-1\}$
3. $c \leftarrow H_3(g, h, g^{sk}, h^{sk}, g^k, h^k)$.
4. $s \leftarrow (k - c \cdot sk) \bmod q$
5. Compute proof $\pi \leftarrow (\gamma, c, s)$, output value $y \leftarrow H_2(\gamma)$ and return (y, π)

Verify Check proof $\pi = (\gamma, c, s)$ corresponds to input x and output y, as follows:

1. $u \leftarrow (pk^c) \cdot g^s$ and $h \leftarrow H_1(a)$
2. Check that $\gamma \in G$
3. $v \leftarrow (\gamma)^c \cdot h^s$.
4. Output accept if $c = H_3(g, h, pk, \gamma, u, v)$, else output reject

Protocol Flow

At a high level, our protocol works as follows. First, the VRFProvider contract and the Bridge contract are deployed on the main-chain. The VRFProvider contract is in charge of receiving randomness requests from dApps and contains code that pre-processes the request in order to be ready for submission to the Band side-chain. It also works as the receiving end of the request's result. The Bridge contract, as the name denotes, works as the connecting "bridge" between the two chains in order to validate the latest state of the side-chain and verify that the received results for VRF requests are indeed the ones computed and stored on the BandChain.

A third-party dApp that wishes to request a random value submits its request to the VRFProvider contract, which then prepares the actual VRF input by expanding it into a VRF seed. This is picked up by incentivized actors and/or the Band foundation and is submitted as a VRF request to the BandChain. In particular, a VRF Oracle Script collects this request, maps it to the set of VRF Data Sources that is available to the chain, as well as a number of BandChain Validators. The VRF Oracle Script then randomly assigns it to a VRF Data Source, which in turn passes the request to a corresponding VRF Provider API. The assigned VRF Provider API evaluates the VRF based on the prescribed input using its VRF secret key, and broadcasts the result to the Band network. All chosen validators run the VRF verification algorithm using this provider's public key and, if the verification is successful, the result will be transmitted to the VRF Oracle Script. Finally, after collecting the necessary number of results from the validators, the oracle script accepts the majority as the final result which then becomes part of the BandChain state, which gets included in the next block's computation.

The final result is transmitted back to the main-chain's VRFProvider contract together with a Merkle tree proof for its inclusion on the BandChain's state. This proof is then checked with the Bridge contract. Upon successful checking, the result is returned to the original calling dApp.

Band VRF on EVM

From the protocol flow, we can categorize the EVM contracts into two parts: the Band-operated contracts and the consumer (third-party) contract. We provide detailed explanations regarding the consumer contract in the VRF Integration section. In this section, we will focus on the two Band-operated contracts: Bridge and VRFProvider.

Bridge

Conceptually, you can think of the Bridge contract as a generic logic that helps verify the availability of any data stored on the BandChain. In this case, the VRFProvider acts as a Bridge's user who wants to verify the data. In order to verify the data's availability, the Bridge contract contains a set of validators of the BandChain, which is used for signature verification when any external actors relay a BandChain's block. After the block relaying is successful (accumulated power of more than 2/3), the rest is the verification of the actual data (leaf) against the root hash. Finally, the result/leaf is extracted and returned to the caller if the data is successfully verified.

State

```
// The encoded chain's ID of Band.
// This value is only set at the deployment time
bytes public encodedChainID;


//********************************************** Set by the owner **********************************

struct ValidatorWithPower {
    address addr;
    uint256 power;
}

// Mapping from an address to its voting power.
EnumerableMap.AddressToUintMap private validatorPowers;

// The total voting power of active validators currently on duty.
uint256 public totalValidatorPower;

//********************************************** Set by the owner **********************************



//*********************************************       Public       ******************************

struct BlockDetail {
    bytes32 oracleState;
    uint64 timeSecond;
    uint32 timeNanoSecondFraction; // between 0 to 10^9
}

// Mapping from block height to the struct that contains block time and hash of "oracle" IAVL Merkle
mapping(uint256 => BlockDetail) public blockDetails;

//*********************************************       Public       ******************************
```

Functions

The two key functions of the Bridge contract are `updateValidatorPowers` and `relayAndVerify`.

The `updateValidatorPowers` function can only be called by the owner of the Bridge contract. The function is used for updating the Bridge contract's validator set in order for the contract's state to be consistent with the actual validator set on the BandChain.

The `relayAndVerify` function is a public function for anyone who wants to relay request data from the BandChain into the Bridge contract. As a result, the relayed and verified data can be used safely by those who wish to consume data from Band oracle.

VRFProvider

The VRFProvider contract is a contract that is used for tracking all VRF requests and their status. It is also the gateway for the users/consumers to interact with the protocol.

State

```
//********************************************** Set by the owner **********************************


// A reference to the Bridge contract
IBridge public bridge;

// An id of the VRF oracle script on Bandchain
uint64 public oracleScriptID;

// A minimum number of validators needed to resolve requests.
uint8 public minCount;

// A limited number of validators needed to resolve requests.
uint8 public askCount;

// A global counter for all tasks known to the VRFProvider
// As a new request is made, a new task will be created, and this number will keep increasing.
uint64 public taskNonce;

// An amount of minimum fee required to pay the VRF worker(resolver, relayer, etc.).
uint256 public minimumFee;

//********************************************** Set by the owner **********************************



//*********************************************       Public       ******************************

// Mapping that enforces the client to provide a unique seed for each request
mapping(address => mapping(string => bool)) public hasClientSeed;

// The struct of a task
struct Task {
    bool isResolved;
    uint64 time;
    address caller;
    uint256 taskFee;
    bytes32 seed;
    string clientSeed;
    bytes proof;
    bytes result;
}
// Mapping from nonce => task
// External entities can query a task via this mapping by specifying the task id.
// The task id is the taskNonce at the time the task was created.
mapping(uint64 => Task) public tasks;

//*********************************************       Public       ******************************
```

Functions

There are two main functions in the VRFProvider contract: `requestRandomData` and `relayProof`.

The `requestRandomData` function is a function that is called by VRF users/consumers. This function is used for registering a new task. It also performs a few checks, such as checking that the users/consumers are not using an already used seed, and that the fee meets the minimum requirement.

The `relayProof` function is a function that is called by the VRF resolvers (worker bots, relayers, etc.). This function receives the Band's Merkle proof and then passes it to the Bridge contract for verification. After the verification has succeeded, the extracted result is sent back from the Bridge contract to the VRFProvider contract. The VRFProvider contract then verifies the parameters in the result's struct to check if they pass all conditions. After the result's parameters have been checked, the VRFProvider contract calls back to the consumer contract via the callback function on the consumer side with the final VRF result as one of the parameters.

Learn more

To learn more about the VRFProvider and Bridge, please visit this 👉 repository

Band VRF on BandChain

There are two main components on the BandChain which involve in the VRF random data delivery process: `oracle script` and `data source`.

The oracle script on the BandChain is similar to a smart contract on EVM chains or CosmWasm chains. It is the on-chain calculation part. VRF oracle script is an oracle script that has been designed specifically for the VRF protocol. It contains a list of VRF data sources to be chosen from during the preparation stage. After the preparation stage, the chosen validators run the VRF data sources to create the VRF random result, and then report the raw result to the BandChain's state. Finally, at the execution/aggregation stage, the oracle script is run automatically to summarize the raw results into a single result, which is then stored on the BandChain's state. When the final result is stored on the BandChain, anyone can retrieve its Merkle proof of availability from the BandChain. This proof is used together with the relay function of the Bridge contract on the client chain.

The data source is an off-chain execution script that is used to call external APIs. VRF data sources are linked to external VRF providers that provide the verifiable randomness service. After calling the provider, the data source verifies the returned result and proof against the predefined public key. The data source only returns the result if the verification is successful.

Learn more

- Band VRF Whitepaper
- Band oracle script docs
- Band data source docs

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Introduction
Getting Started
VRF integration
Example Use Cases
VRF Supported Blockchains
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

Getting Started with Band VRF

This guide serves as a quick reference on how to request random data from the Band VRF. For a detailed reference with examples, please refer to the VRF Integration section.

Step 1: Prepare a VRF consumer contract

1. Create a VRF consumer contract that can call the `requestRandomData` function on the `VRFProvider` contract.
2. Implement a callback function on the VRF consumer contract, which allows the `VRFProvider` contract to call back and execute some logic against the returned result.

Step 2: Choose a resolving method

There are currently 3 methods for relaying and resolving the VRF request:

- **Band's VRF worker solution** - We provide both standard and customized solutions for all clients. Please Email Us for more details.
- **Manually resolve on CosmoScan** - This is an ideal and low cost solution for one-off Band VRF requests. Please refer to this guide for how to resolve manually.
- **Implement your own resolver bot** - Anyone can implement their own version of resolver bot. An open-source version of Band's VRF worker bot is available at VRFWorkerV1 repository.

Step 3: Request a random value

You are now ready to request a random value from the Band VRF.

A summary of the Band VRF process is outlined below:

1. Simply call the request function on you VRF consumer contract that implements the `requestRandomData` function in Step 1, providing a `seed` and an optional `msg.value`.
2. Depending on the resolving method chosen in Step 2, the request is sent to the BandChain.
3. The VRF oracle script on the BandChain forwards the request to a randomly chosen data source, and then retrieves the returned result and the corresponding proof of authenticity.
4. Depending on the resolving method chosen in Step 2, the proof is relayed to the `Bridge` contract for verification on the client chain via the `VRFProvider` contract.
5. If the verification succeeds, the result (random value) is returned to the VRF consumer contract via the callback function mentioned in Step 1.

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Introduction
Getting Started
VRF integration
Example Use Cases
VRF Supported Blockchains
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

VRF integration

In this section, we will discuss how to integrate your smart contracts with the Band VRF. This section is separated into two sub-sections: **Requesting** and **Resolving**.

Typically, when building on-chain applications that rely on an unpredictable outcome, such as lottery apps or games, the system requires a reliable source of randomness, and that is when the Band VRF comes into play.

VRF Flow

As shown in the VRF workflow, a request and callback model is used to obtain new random data. We will summarize the steps shown in the figure above into 3 main steps to make it easier to understand; steps 1 and 3 are on the EVM side, and step 2 is on the Bandchain side.

1. The process begins with a transaction that contains a consumer's request for random data from Band's VRFProvider contract.
2. After the transaction in step 1 is confirmed, an off-chain entity will grab parameters from its log to make a request transaction on Bandchain, producing the VRF result.
3. The last step is grabbing the result in step 2 with the proof of availability on Bandchain for making a relay transaction on the EVM side to resolve the request in the first step.

Contracts integration

Requesting

Assume that you are building an on-chain application that uses the Band VRF as a reliable source of randomness. Your contract(s) should contain a reference to Band's VRFProvider contract to be able to request random values.

First, let's define an interface for the VRFProvider contract.

```
interface IVRFProvider {
    /// @dev The function for consumers who want random data.
    /// Consumers can simply make requests to get random data back later.
    /// @param seed Any string that used to initialize the randomness.
    function requestRandomData(string calldata seed) external payable;
}
```

Then, the consumer only needs to call `requestRandomData` with a string parameter called `seed`.

> **CAUTION**
> Please note that the seed is a generated string on the consumer side. Any two different consumers may be using the same seed. However, every consumer must use a unique seed for all their requests, as the VRFProvider has a mapping to check this condition.

```
contract VRFProvider {
    ...

    // Mapping that enforces the client to provide a unique seed for each request
    mapping(address => mapping(string => bool)) public hasClientSeed;

    ...
}
```

After including the `IVRFProvider`, the consumer can now make a request-call to the VRFProvider contract, as shown in the example implementation below.

```
contract MockVRFConsumer {
    IVRFProvider public provider;

    constructor(IVRFProvider _provider) {
        provider = _provider;
    }

    function requestRandomDataFromProvider(string calldata seed) external payable {
        provider.requestRandomData{value: msg.value}(seed);
    }
}
```

When calling `requestRandomData(seed)`, the consumer can specify `msg.value` to incentivize others to resolve the random data request. However, consumers can choose not to provide any incentive and resolve the request themselves.

After the consumer knows how to request from the `VRFProvider`, the consumer must also implement a callback function for the VRFProvider contract to call after the corresponding VRF result has been relayed.

The implementation below takes the previous step's code and adds the `consume` function.

```
contract MockVRFConsumer {
    IVRFProvider public provider;

    constructor(IVRFProvider _provider) {
        provider = _provider;
    }

    function requestRandomDataFromProvider(string calldata seed) external payable {
        provider.requestRandomData{value: msg.value}(seed);
    }

    // ==========================================================================

    string public latestSeed;
    uint64 public latestTime;
    bytes32 public latestResult;

    function consume(string calldata seed, uint64 time, bytes32 result) external override {
        require(msg.sender == address(provider), "Caller is not the provider");

        latestSeed = seed;
        latestTime = time;
        latestResult = result;
    }
}
```

As shown above, the `consume` function implements a logic that verifies whether the caller is the VRFProvider contract or not. This is to ensure that no one can call this function except the VRFProvider contract. With regards to the remaining logic in the example, the callback function only saves the callback data from the VRFProvider contract to its state.

Finally, we got our complete `MockVRFConsumer` contract that can make a request called the `VRFProvider` and then consume the VRF result once that request is resolved from the `VRFProvider` side.

Resolving

Anyone can get any existing requests in the VRFProvider contract by tracking the mapping called tasks. A task(VRF request) was designed to be resolved in a decentralized way. Therefore, there is no permission to resolve any task. The only thing that needs to resolve any unresolved tasks is the Merkle proof of the VRF result available on Bandchain.

The code below shows what a `task` looks like and the data structure(mapping) that helps track the tasks.

```
contract VRFProvider {

    ...

    struct Task {
        bool isResolved;
        uint64 time;
        address caller;
        uint256 taskFee;
        bytes32 seed;
        bytes32 result;
        string clientSeed;
    }

    // A global variable representing the number of all tasks in this contract
    uint64 public taskNonce;

    // Mapping from nonce => task
    mapping(uint64 => Task) public tasks;

    ...

    function relayProof(bytes calldata _proof, uint64 _taskNonce) external nonReentrant {
        ...
    }

    ...

}
```

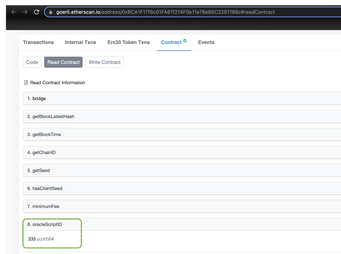
When a resolver(a self-implemented worker, a bot, a bounty hunter, etc.) finds an unresolved request, the resolver can resolve it by requesting the VRF randomness on the BandChain. After the VRF result is finalized on the BandChain, the resolver can retrieve the Merkle proof of availability of the result and then relay the proof via a `relayProof` function on the VRFProvider contract. The resolver also needs to specify the nonce of the task it wants to resolve.

The next step will demonstrate how to manually request and resolve the VRF randomness using the user interfaces of Cosmoscan(Band) and Etherscan(EVM).

Manually request and resolve

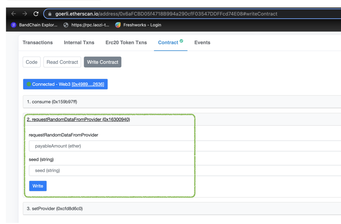
This section will demonstrate how to request random data from the `VRFProvider` and then resolve the request manually using Goerli and Laozi-Testnet6 UI.

Firstly, go to the VRFProvider contract on Goerli to view some of its global variables.



To request the VRF randomness on Bandchain, we need to know the `oracleScriptID`. The VRF oracle script is the ID 335 on the laozi-Testnet6.

Now, let's move to the MockVRFConsumer contract to begin the VRF flow started by calling a function `requestRandomDataFromProvider`.



The video below shows the flow of the `MockConsumer` that requests the VRF random value from the `VRFProvider`.

Recommended Oracle Script Request Settings

Parameter	Value
Prepare Gas	100000
Execute Gas	400000
Gas Limit	(Leave Blank)
Ask Count	4
Min Count	3

Implement your own resolver

An alternative to manually resolve the request is to use a resolver bot. Anyone can implement their own version of resolver bot to automate the resolving process. We provide an open-source version of Band's VRF worker bot, which is available at VRFWorkerV1 repository.

Example Use Cases

Band VRF provides deterministic pre-commitments for low entropy inputs, which must resist brute-force pre-image attacks. In addition, the VRF can be used as defense against offline enumeration attacks (such as dictionary attacks) on data stored in hash-based data structures. Therefore, it can be used as a secure source of on-chain randomness. The Band VRF use cases are outlined below.

Use cases

We separate the use cases into four categories based on the behaviors of the VRF users/consumers.

1. **One-time-use:** Consumers only request random data once in their product lifetime, typically when they initiate their contracts.
    - NFT minting - in the case where a single random seed is used to generate the entire collection.
2. **Batch-use:** Consumers request random data multiple times but with a countable number of requests for the entire life of their product.
    - NFT minting - in the case where every single ID will be minted one by one. Therefore, whenever the end-user tries to mint an NFT, the VRF request is created to resolve the minting. This process will continue until the entire collection is minted.
3. **Interval-use:** Consumers set their specific intervals to request random data from our VRF protocol. This process will continue indefinitely since some parts of their products rely on a trusted source of randomness.
    - Lottery dApps (predetermined start-end, calculatable start-end)
    - NFT minting with a specific minting interval
4. **Continuous-use:** Consumers use randomness as parts of their product with no specific interval. Therefore, they can request random data at any time based on the internal logic of their contracts/system.
    - Lottery dApps (no predetermined interval, unable to calculate future start-end)
    - On-chain games
    - NFT on-demand minting

Lottery Example (Continuous-use)

This on-chain lottery dApp is an example of a continuous-use VRF, and it satisfies the requirements below.

Requirements:

- Only the owner can set the minimum price and the round's duration.
- Only the owner can start a new round.
- The owner determined the seed of the `started` round.
- Anyone can buy lotteries during a `started` round.
- Anyone can request to resolve the current round if it has ended.
- Only the VRFProvider contract can resolve the `resolving` round.

```
pragma solidity ^0.8.17;

interface IVRFConsumer {
    /// @dev The function is called by the VRF provider in order to deliver results to the consumer.
    /// @param seed Any string that used to initialize the randomizer.
    /// @param time Timestamp where the random data was created.
    /// @param result A random bytes for given seed anfd time.
    function consume(
        string calldata seed,
        uint64 time,
        bytes32 result
    ) external;
}

interface IVRFProvider {
    /// @dev The function for consumers who want random data.
    /// Consumers can simply make requests to get random data back later.
    /// @param seed Any string that used to initialize the randomizer.
    function requestRandomData(string calldata seed) external payable;
}

contract SimpleLottery is IVRFConsumer {

    event Buy(address buyer, uint256 buyerIndex, uint256 roundNumber, uint256 buyPrice);
    event StartRound(uint256 roundNumber, string seed);
    event ResolvingRound(uint256 roundNumber, string seed);
    event RoundResolved(uint256 roundNumber, string seed, bytes32 result);

    address public owner;

    uint256 public minLotteryPrice;
    uint256 public roundDuration;
    uint256 public roundCount;
    bool public isResolvingCurrentRound;

    IVRFProvider public provider;

    struct Round {
        uint256 startBlock;
        uint256 endBlock;
        string seedOfRound;
        address[] buyers;
    }

    mapping(uint256 => Round) public rounds;

    constructor(IVRFProvider _provider, uint256 _minLotteryPrice, uint256 _roundDuration) {
        provider = _provider;
        minLotteryPrice = _minLotteryPrice;
        roundDuration = _roundDuration;
        owner = msg.sender;
    }

    function isCurrentRoundStart() public view returns(bool) {
        return rounds[roundCount].startBlock > 0;
    }

    function currentRoundBlocksRemaining() public view returns(uint256) {
        Round memory currentRound = rounds[roundCount];
        if (block.number > currentRound.endBlock) {
            return 0;
        }
        return currentRound.endBlock - block.number;
    }

    function setMinLotteryPrice(uint256 _minLotteryPrice) external {
        require(msg.sender == owner, "SimpleLottery: not the owner");
        require(!isCurrentRoundStart(), "SimpleLottery: this round is in progress");

        minLotteryPrice = _minLotteryPrice;
    }

    function setRoundDuration(uint256 _roundDuration) external {
        require(msg.sender == owner, "SimpleLottery: not the owner");
        require(!isCurrentRoundStart(), "SimpleLottery: this round is in progress");

        roundDuration = _roundDuration;
    }

    function startANewRound(string memory roundSeed) external {
        require(msg.sender == owner, "SimpleLottery: not the owner");
        require(!isCurrentRoundStart(), "SimpleLottery: this round is in progress");

        Round memory currentRound = rounds[roundCount];

        currentRound.seedOfRound = roundSeed;
        currentRound.startBlock = block.number;
        currentRound.endBlock = currentRound.startBlock + roundDuration;

        rounds[roundCount] = currentRound;
        emit StartRound(roundCount, roundSeed);
    }

    function buy() external payable {
        require(currentRoundBlocksRemaining() > 0, "SimpleLottery: this round is not in progress");
        require(msg.value >= minLotteryPrice, "SimpleLottery: given price is too low");

        uint256 currentBuyerIndex = rounds[roundCount].buyers.length;
        emit Buy(msg.sender, currentBuyerIndex, roundCount, msg.value);

        rounds[roundCount].buyers.push(msg.sender);
    }

    function resolveCurrentRound() external {
        require(isCurrentRoundStart(), "SimpleLottery: this round is not started yet");
        require(currentRoundBlocksRemaining() == 0, "SimpleLottery: this round has not ended yet");
        require(!isResolvingCurrentRound, "SimpleLottery: round is resolving");

        Round memory currentRound = rounds[roundCount];
        if (currentRound.buyers.length > 0) {
            isResolvingCurrentRound = true;

            provider.requestRandomData{value: 0}(currentRound.seedOfRound);
            emit ResolvingRound(roundCount, currentRound.seedOfRound);
        } else {
            emit RoundResolved(roundCount, currentRound.seedOfRound, bytes32(0));

            roundCount += 1;
            isResolvingCurrentRound = false;
        }
    }

    function consume(string calldata seed, uint64 time, bytes32 result) external override {
        require(msg.sender == address(provider), "Caller is not the provider");
        require(isResolvingCurrentRound, "SimpleLottery: round is not resolving");

        Round memory currentRound = rounds[roundCount];
        address winner = currentRound.buyers[uint256(result) % currentRound.buyers.length];

        emit RoundResolved(roundCount, seed, result);

        roundCount += 1;
        isResolvingCurrentRound = false;

        winner.call{value: address(this).balance}("");
    }
}
```

We have deployed the reference contracts to the Goerli testnet here.

Contract	Address
Bridge	0xD291A502e3ca4Bb13E09892e57d8Ff0271Bd198A
VRFProvider	0xF1F3554b6f46D8f172c89836FBeD1ea8551eabad
VRFLens	0x6e876b4Ed458af275Eb049a3f89BF0909618d154
SimpleLottery	0xCD3528283aA330003E50350134a48d1920BA70A0

NFT Minting Example (Batch-use)

This NFT is an example of a batch-use VRF, and it satisfies the requirements below.

Requirements:

- The max supply is set once at the time the contract is deployed.
- Anyone can call `mintWithVRF` to start minting an NFT for themself.
- An actual minting is done when the VRFprovider resolves the token id for the minter/receiver.

```
pragma solidity ^0.8.17;

import {ERC721Enumerable} from "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";

/**
 * @dev String operations.
 */
library Strings {
    bytes16 private constant _HEX_SYMBOLS = "0123456789abcdef";

    /**
     * @dev Converts a `uint256` to its ASCII `string` decimal representation.
     */
    function toString(uint256 value) internal pure returns (string memory) {
        // Inspired by OraclizeAPI's implementation - MIT licence
        // https://github.com/oraclize/ethereum-api/blob/b42146b063c7d6ee1358846c198246239e9360e8/oracl

        if (value == 0) {
            return "0";
        }
        uint256 temp = value;
        uint256 digits;
        while (temp != 0) {
            digits++;
            temp /= 10;
        }
        bytes memory buffer = new bytes(digits);
        while (value != 0) {
            digits -= 1;
            buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
            value /= 10;
        }
        return string(buffer);
    }
}

interface IVRFConsumer {
    /// @dev The function is called by the VRF provider in order to deliver results to the consumer.
    /// @param seed Any string that used to initialize the randomizer.
    /// @param time Timestamp where the random data was created.
    /// @param result A random bytes for given seed anfd time.
    function consume(
        string calldata seed,
        uint64 time,
        bytes32 result
    ) external;
}

interface IVRFProvider {
    /// @dev The function for consumers who want random data.
    /// Consumers can simply make requests to get random data back later.
    /// @param seed Any string that used to initialize the randomizer.
    function requestRandomData(string calldata seed) external payable;
}

contract ExampleNFT is ERC721Enumerable, IVRFConsumer {
   using Strings for uint256;

    IVRFProvider public immutable provider;
    uint256 public immutable maxSupply;

    uint256 public mintRequestCount = 0;
    uint256 public mintResolveCount = 0;

    mapping(uint256 => uint256) public tokenMintingLogs;
    mapping(string => address) public tokenSeedToMinter;

    constructor(IVRFProvider _provider, uint256 _maxSupply) ERC721("ExampleNFT", "ENFT") {
        provider = _provider;
        maxSupply = _maxSupply;
    }

    function mintWithVRF() external {
        require(mintRequestCount < maxSupply, "Reach max supply");
        string memory clientSeed = string(abi.encodePacked("ExampleNFT-", mintRequestCount.toString()))
        tokenSeedToMinter[clientSeed] = msg.sender;

        mintRequestCount++;

        provider.requestRandomData{value: 0}(clientSeed);
    }

    function consume(string calldata seed, uint64 time, bytes32 result) external override {
        require(msg.sender == address(provider), "Caller is not the provider");

        address _receiver = tokenSeedToMinter[seed];
        uint256 index = uint256(result) % (maxSupply - mintResolveCount);

        uint256 tokenID = tokenMintingLogs[index];
        if (tokenID == 0) {
            tokenID = index;
        }

        mintResolveCount++;
        tokenMintingLogs[index] = maxSupply - mintResolveCount;

        _safeMint(_receiver, tokenID);
    }
}
```

We have deployed the reference contracts to the Goerli testnet here.

Contract	Address
Bridge	0xD291A502e3ca4Bb13E09892e57d8Ff0271Bd198A
VRFProvider	0xF1F3554b6f46D8f172c89836FBeD1ea8551eabad
VRFLens	0x6e876b4Ed458af275Eb049a3f89BF0909618d154
NFTBatchMinting	0x0b590C537608d121F8e46c2b366f5d22EC942c0f

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Supported Blockchains

CAUTION

- We recommend experimenting with using our solution on a test network first before proceeding to mainnet.
- Please ONLY use the proxy contract as in the case that the base contract is updated, the proxy contract will always point to the correct base contract and show the most updated price feeds.

Currently, our `StdReferenceProxy` smart contracts are available on the following networks:

Testnets

Blockchain	`StdReferenceProxy` Contract Address	Explorer
Astar (Shibuya)	0x2Bf9a731f9A56C59DeB4DF1369286A3E69F5b418	link
Avalanche (Fuji)	0xD0b2234eB9431e850a814bCdcBCB18C1093F986B	link
Bitgert	0x8c064bCf7C0DA3B3b090BAbFE8f3323534D84d68	link
BitTorrent Chain	0x8c064bCf7C0DA3B3b090BAbFE8f3323534D84d68	link
Boba	0xe8e974b6b294AC1f5031AC54B968E8afFb7306Cb	link
Celo (Alfajores)	0x660cBc25F0cFD31F0Bdcaa43525f0bACC6DB2ABc	link
CLV	0xD0b2234eB9431e850a814bCdcBCB18C1093F986B	link
Cronos	0xD0b2234eB9431e850a814bCdcBCB18C1093F986B	link
Findora	0x8c064bCf7C0DA3B3b090BAbFE8f3323534D84d68	link
Godwoken	0x0c2362c9a0586dd7295549c65a4a5e3afe10a88a	link
Harmony	0xD0b2234eB9431e850a814bCdcBCB18C1093F986B	link
Icon (Berlin)	cxd2bac764a0277efb9a6861fa991be4e5a46f16a2	link
Icon (Lisbon)	cx734512ad03efdcedb69e0526415a7ce21340e0db	link
Meter	0xe1bCC505f2Bdd02C9480C924856f5080834A3897	link
Oasis (Emerald)	0x61704EFB8b8120c03C210cAC5f5193BF8c80852a	link
Oasis (Sapphire)	0x0c2362c9A0586Dd7295549C65a4A5e3aFE10a88A	link
OKC	0xb6256DCb23CEE06eDa2408E73945963606fdddd7	link
Osmosis	osmo1j8jnej8k5he7h27qzr4v999xgsd0q43n9ny8phtx9aj6w82ruwgslcp77p	link
PlatON	0xb6256DCb23CEE06eDa2408E73945963606fdddd7	link
Secret (Pulsar-2)	secret14swdnnllsfvtnvwmtvnvcj2zu0njsl9cdkk5xp	link

Mainnets

Blockchain	`StdReferenceProxy` Contract Address	Explorer
Astar (Astar)	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Astar (Shiden)	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Avalanche	0x75B01902D9297fD381bcF3B155a8cEAC78F5A35E	link
BitTorrent Chain	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
BNB	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Celo	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
CLV	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Cronos	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Ethereum	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Fantom	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Godwoken	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Icon (Java)	cxe647e0af68a4661566f5e9861ad4ac854de808a2	link
Meter	0x861C20f77f194EEa4f86e0d39069D789265A3A82	link
Moonriver	0x75B01902D9297fD381bcF3B155a8cEAC78F5A35E	link
Oasis (Emerald)	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
OKC	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Optimism	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
PlatON	0xDA7a001b254CD22e46d3eAB04d937489c93174C3	link
Secret 4	secret1ezamax2vrhjpy92fnujlpwfj2dpredaafss47k	link

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Pricefeed Module
Introduction
Getting started
Example Use Cases
cw-band
Develop
Nodes & Validators
Staking

Introduction

The Cosmos SDK is presently the most widely utilized framework for developing blockchain applications. A crucial necessity for such apps is an oracle. As decentralized applications (dApps) rely heavily on real-world data, ensuring the accuracy of this data holds significant importance.

It would be advantageous for your Cosmos SDK app to possess a built-in Oracle that obtains data from BandChain via IBC. By importing the pricefeed module implemented by Band Protocol.

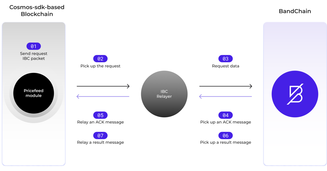
Proposal

The initial step for the pricefeed module is to obtain information about the symbols that require price data from BandChain on every `n` block. This is accomplished by submitting the `UpdateSymbolRequest` Proposal.

The Proposal submitted to update tasks for the pricefeed module consists of three components - the name of the symbol, the Oracle script ID required to obtain the price, and the block interval for requesting the data every `n` blocks.

Upon the proposal's approval, the pricefeed module will request price data from BandChain based on the `SymbolRequest` that was updated through the proposal.

Workflow



Request

At a high level, the workflow will be as follows. First, the pricefeed module creates an IBC packet to request data from BandChain. Then, relayers will pick up the IBC packet and relay it on BandChain.

After BandChain processes the request, it will send an acknowledgement message along with `request_id` back. And, when the result of the request is finalized, BandChain will send a new IBC packet that contains the final data back. Relayers will listen and pick up those packets and relay them to your Cosmos SDK app.

After this stage, the Cosmos SDK app can safely use the data obtained from BandChain in its application at every `n` blocks interval. If the Cosmos SDK app requires additional data, it can submit an update symbols request proposal at any time.

pricefeed

The pricefeed module obtains price data from BandChain through IBC and stores the most recent prices on your Cosmos SDK applications.

An example of the usage of this module is provided on the Oracle Consumer Chain.

Params

The pricefeed module stores its params in the state, it can be updated with governance. The information contained in these parameters is utilized to request data from BandChain.

proto/consumer/pricefeed/params.proto

```
message Params {
  option (gogoproto.goproto_stringer) = false;

  uint64 ask_count = 1;
  uint64 min_count = 2;
  uint64 min_ds_count = 3;
  uint64 prepare_gas_base = 4;
  uint64 prepare_gas_each = 5;
  uint64 execute_gas_base = 6;
  uint64 execute_gas_each = 7;
  string source_channel = 8;
  repeated cosmos.base.v1beta1.Coin fee_limit = 9 [
    (gogoproto.nullable) = false,
    (gogoproto.castrepeated) = "github.com/cosmos/Cosmos SDK/types.Coins"
  ];
}
```

Proposal

The pricefeed module includes the `UpdateSymbolRequestProposal` for updating symbols to request price data on BandChain based on `block_interval` configuration by submitting the proposal on your Cosmos SDK application.

proto/consumer/pricefeed/pricefeed.proto

```
message UpdateSymbolRequestProposal {
  option (gogoproto.goproto_getters) = false;

  string title = 1;
  string description = 2;

  repeated SymbolRequest symbol_requests = 3 [ (gogoproto.nullable) = false ];
}

message SymbolRequest {
  string symbol = 1;
  uint64 oracle_script_id = 2;
  uint64 block_interval = 3;
}
```

The example of submitting and voting on the proposal is demonstrated in the CLI section.

CLI

A user can query and interact with the pricefeed module using the CLI.

> Note: This example use `oracle-consumerd` as a command-line interface (CLI) from oracle consumer chain. Please replace it with your own cosmos app.

Query

The query commands allow users to query pricefeed state.

```
oracle-consumerd query pricefeed --help
```

Symbol Requests

The `symbol-requests` command enables users to retrieve information about all symbol requests that are saved in this Cosmos SDK application.

```
oracle-consumerd query pricefeed symbol-requests
```

Price

The `price` command allows users to query price data by the symbol.

```
oracle-consumerd query pricefeed price [symbol]
```

Example:

```
oracle-consumerd query pricefeed price BTC
```

Example Output:

```
price:
  price: "22702955000000"
  resolve_time: "1675935544"
  symbol: BTC
```

Proposal

The `tx gov submit-legacy-proposal` commands allow users to submit a proposal on your Cosmos SDK app.

```
oracle-consumerd tx gov submit-legacy-proposal -h
```

Source Channel param change proposal

To acquire BandChain data through the IBC, it is imperative to update the `source-channel` parameter by submitting a proposal for the change that reflects your source channel.

```
oracle-consumerd tx gov submit-legacy-proposal param-change [proposal-file]
```

Example:

1. create `param_change.json` file

```
{
  "title": "Param change for SourceChannel",
  "description": "Proposal for change SourceChannel param in pricefeed module",
  "changes": [
    {
      "subspace": "pricefeed",
      "key": "SourceChannel",
      "value": "channel-0"
    }
  ],
  "deposit": "10000000stake"
}
```

2. submit the proposal

```
oracle-consumerd tx gov submit-legacy-proposal param-change param_change.json --from alice
```

Update symbol request proposal

The `update-symbol-request` command allows users to update symbol requests to specify which symbols they desire to obtain price data from BandChain.

```
oracle-consumerd tx gov submit-legacy-proposal update-symbol-request [proposal-file]
```

Example:

1. create `update_symbol_requests.json` file

```
{
  "title": "Update Symbol requests",
  "description": "Update symbol that request price from BandChain",
  "symbol_requests": [
    {
      "symbol": "BTC",
      "oracle_script_id": "396",
      "block_interval": "40"
    },
    {
      "symbol": "ETH",
      "oracle_script_id": "396",
      "block_interval": "40"
    }
  ],
  "deposit": "10000000stake"
}
```

> Note: You can also delete symbol request by set `"block_interval": "0"` on this proposal.

2. submit the proposal

```
oracle-consumerd tx gov submit-legacy-proposal update-symbol-request update_symbol_requests.json --f
```

Another way to initiate source channel and symbol requests

To utilize the Ignite feature to replace the genesis state, insert the code shown below into the `config.yml` file, and restart the chain by using `ignite chain serve -r -v` command.

```
genesis:
  app_state:
    pricefeed:
      params:
        source_channel: 'channel-0'
      symbol_requests: [{ 'symbol': 'BAND', 'oracle_script_id': 396, 'block_interval': 40 }]
```

Learn more

- Cosmos SDK
  - building-modules
  - ibc-app-packets
- BandChain
  - oracle-script

Getting started

This guide serves as a brief guide on how to utilize the pricefeed module in your Cosmos SDK app.

Prerequisites

Be sure you have met the prerequisites before you follow this guide.

Operating systems

- Ubuntu 22.04

Go (for ignite and consumer chain)

- 1.19.5 or higher

Ignite CLI

- 0.26.1 or higher

Rust (for Hermes Relayer)

- 1.65.0 or higher

Installation

Installing build-essential package in Ubuntu

```
sudo apt update && sudo apt install build-essential
```

Creating a new blockchain

To create a new blockchain project with Ignite, you will need to run the following command:

```
ignite scaffold chain example
```

The ignite scaffold chain command will create a new blockchain in a new directory example.

Step 1: Replace the genesis state

Config proposal voting period by Ignite

To expedite the testing of the pricefeed module, modify the default voting period to 40 seconds using Ignite feature to replace the genesis state by incorporating this code in `config.yml`.

```
...
genesis:
  app_state:
    gov:
      voting_params:
        voting_period: "40s"
```

Initiate source channel and symbol requests by Ignite

To utilize the Ignite feature to replace the genesis state without open `update-symbol-requests` proposal, insert the code shown below into the `config.yml` file.

```
...
genesis:
  app_state:
    ...
    pricefeed:
      params:
        source_channel: "channel-0"
      symbol_requests: [{"symbol": "BAND", "oracle_script_id": 396, "block_interval": 40}]
```

Step 2: Import pricefeed module to your cosmos app

Edit Cosmos SDK and IBC-go version

To ensure compatibility with the pricefeed module, kindly update the Cosmos SDK version to `v0.46.12`.

```
require (
    ...
    github.com/cosmos/Cosmos SDK v0.46.12
)
```

Additionally, modify the ibc-go dependency in both the go.mod and app.go files, replacing the version `v6.1.0` from the repository github.com/cosmos/ibc-go/v4 with version `v5.2.0` from the repository github.com/cosmos/ibc-go/v5.

```
require (
    ...
    github.com/cosmos/ibc-go/v5 v5.2.0
)
```

Replace Tendermint with CometBFT

The pricefeed module now uses the version implemented by CometBFT. Therefore, to replace the Tendermint version, kindly add this line in `go.mod`.

```
replace (
    ...
    github.com/tendermint/tendermint => github.com/cometbft/cometbft v0.34.27
)
```

Then run `go mod tidy` to update all module packages.

Add pricefeed keeper in `app/app.go`

Add pricefeed proposal

```
import (
    ...
    pricefeedclient "github.com/bandprotocol/oracle-consumer/x/pricefeed/client"
)

func getGovProposalHandlers() []govclient.ProposalHandler {
    var govProposalHandlers []govclient.ProposalHandler

    govProposalHandlers = append(
        ...
        pricefeedclient.ProposalHandler,
    )

    return govProposalHandlers
}
```

Add pricefeed module basic

```
import (
    ...
    pricefeed "github.com/bandprotocol/oracle-consumer/x/pricefeed"
)

ModuleBasics = module.NewBasicManager(
    ...
    pricefeed.AppModuleBasic{},
)
```

Add pricefeed keeper type

```
import (
    ...
    pricefeedkeeper "github.com/bandprotocol/oracle-consumer/x/pricefeed/keeper"
)

type App struct {
    ...
    PricefeedKeeper       pricefeedkeeper.Keeper

    ...
    ScopedPricefeedKeeper capabilitykeeper.ScopedKeeper
}
```

Add pricefeed store key

```
import (
    ...
    pricefeedtypes "github.com/bandprotocol/oracle-consumer/x/pricefeed/types"
)

keys := sdk.NewKVStoreKeys(
    ...
    pricefeedtypes.StoreKey,
)
```

Create a new pricefeed keeper

```
scopedPricefeedKeeper := app.CapabilityKeeper.ScopeToModule(pricefeedtypes.ModuleName)
app.ScopedPricefeedKeeper = scopedPricefeedKeeper
app.PricefeedKeeper = pricefeedkeeper.NewKeeper(
    appCodec,
    keys[pricefeedtypes.StoreKey],
    keys[pricefeedtypes.MemStoreKey],
    app.IBCKeeper.ChannelKeeper,
    app.IBCKeeper.ChannelKeeper,
    &app.IBCKeeper.PortKeeper,
    scopedPricefeedKeeper,
)
```

Create a new pricefeed module

```
import (
    ...
    pricefeedmodule "github.com/bandprotocol/oracle-consumer/x/pricefeed"
)

pricefeedModule := pricefeedmodule.NewAppModule(appCodec, app.PricefeedKeeper)
pricefeedIBCModule := pricefeedmodule.NewIBCModule(app.PricefeedKeeper)
```

Add pricefeed module in IBC router

```
ibcRouter.
    AddRoute(...).
    AddRoute(pricefeedtypes.ModuleName, pricefeedIBCModule)
```

Add pricefeed module in governance Handler router

```
govRouter.
    AddRoute(...).
    AddRoute(pricefeedtypes.RouterKey, pricefeedmodule.NewUpdateSymbolRequestProposalHandler(app.Pricef
```

Add pricefeed module in the module manager

```
app.mm = module.NewManager(
    ...
    pricefeedModule,
)
```

Set pricefeed order in begin block, end block and init genesis

```
app.mm.SetOrderBeginBlockers(
    ...
    pricefeedtypes.ModuleName,
)

app.mm.SetOrderEndBlockers(
    ...
    pricefeedtypes.ModuleName,
)

app.mm.SetOrderInitGenesis(
    pricefeedtypes.ModuleName,
)
```

Set pricefeed order for deterministic simulations

```
app.sm = module.NewSimulationManager(
    ...
    pricefeedModule,
)
```

Add pricefeed subspace in params Keeper

```
func initParamsKeeper(...) paramskeeper.Keeper {
    paramsKeeper.Subspace(...)
    paramsKeeper.Subspace(pricefeedmoduletypes.ModuleName)
}
```

Once you have completed the addition of the pricefeed module in the app.go file, execute the command `go mod tidy` to import and update the necessary modules.

Now have completed importing the pricefeed module and can now execute the chain by running this command 🎉:

```
ignite chain serve -v
```

Step 3: Setup a relayer

The second step is to set up a relayer to listen and relay IBC packets between your chain and BandChain.

Here are the simple guides for setting up a relayer.

- Setup Hermes relayer *(Recommend)*
- Setup Go relayer

Step 4 (optional): Open proposal for change params and update symbol requests

Since you have already configured the symbol requests and source channel in the `config.yml` file during the step 1 , you may skip this particular step.

Step 4.1 Open source channel param change proposal and vote

The current default value for the source channel is `[not_set]`. If you wish to obtain BandChain data through IBC, you will need to open the proposal to change the source channel parameter to your own source channel. An example of how to open a parameter change proposal is provided below.

create param-change-proposal.json

```
{
  "title": "Param change for SourceChannel",
  "description": "Proposal for change SourceChannel param in pricefeed module",
  "changes": [
    {
      "subspace": "pricefeed",
      "key": "SourceChannel",
      "value": "channel-0"
    }
  ],
  "deposit": "10000000stake"
}
```

Submit proposal

```
exampled tx gov submit-legacy-proposal param-change param-change-proposal.json --from alice
```

Vote the proposal

```
exampled tx gov vote 1 yes --from alice
```

```
exampled tx gov vote 1 yes --from bob
```

Step 4.2: Open update symbol request proposal and vote

The purpose of this proposal is to request price data from BandChain at `block_interval` specified in the proposal. If the proposal is approved, the pricefeed module will retrieve the data and store the response on the consumer chain.

create update-symbol-requests-proposal.json

```
{
  "title": "Update Symbol requests",
  "description": "Update symbol that request price from BandChain",
  "symbol_requests": [
    {
      "symbol": "BTC",
      "oracle_script_id": "396",
      "block_interval": "40"
    },
    {
      "symbol": "ETH",
      "oracle_script_id": "396",
      "block_interval": "40"
    }
  ],
  "deposit": "10000000stake"
}
```

Note: You can also delete symbol request by set `"block_interval": "0"` on this proposal.

Submit proposal

```
exampled tx gov submit-legacy-proposal update-symbol-request update-symbol-requests-proposal.json --fro
```

Vote the proposal

```
exampled tx gov vote 2 yes --from alice
```

```
exampled tx gov vote 2 yes --from bob
```

Check proposal status

```
exampled query gov proposals
```

Query the latest price that got from BandChain

Once the proposal has been approved, the pricefeed module will query BTC and ETH from BandChain every 40 blocks on your chain, and you can view the latest price by executing this command:

```
exampled query pricefeed price [symbol]
```

Example Use Cases

By utilizing pricefeed module in your Cosmos SDK application, you can request price data from BandChain via IBC. This creates a chance to obtain a variety of real-world data, including cryptocurrency and stock prices. Please take a look at the following example of an oracle-consumer chain that requests data from BandChain to use in their module.

---

oracle-consumer

The oracle-consumer is an application of the Cosmos SDK that demonstrates the use of the pricefeed module implemented by BandProtocol. This module allows other Cosmos SDK applications to easily obtain data from BandChain through IBC.

Requirements: oracle-consumer is built on the Cosmos SDK using the following modules:

- `x/consumer`: Consume data from pricefeed module.
- `x/pricefeed`: Logic of requesting data from BandChain.

You can see the full implementation of the oracle-consumer chain here

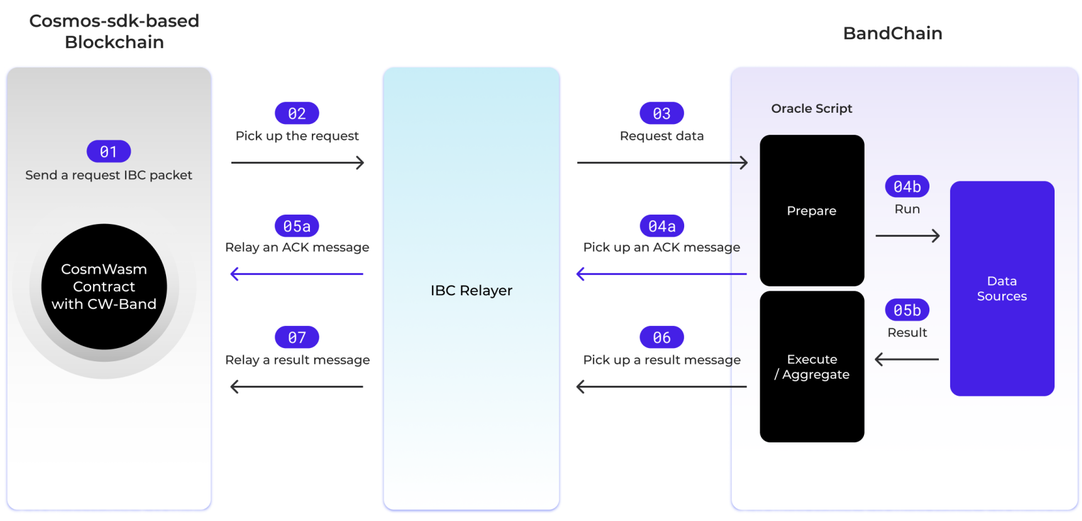
Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Pricefeed Module
cw-band
Introduction
Getting started
Example Use Cases
Develop
Nodes & Validators
Staking

Introduction

The correctness of the data is a crucial part of decentralized applications (dApps) that rely on data from the real world. Manipulated data can impact the security and the objective of the applications.

The `cw-band` introduces a standard for anyone looking to integrate data from Band's oracle into their CosmWasm smart contract on a Cosmos-SDK-based blockchain through Inter-Blockchain Communication (IBC). The standard itself consists of data types that require for requesting and receiving data.

Workflow



At a high level, the workflow will be as follows. First, the smart contract creates an IBC packet to request data from BandChain. Then, relayers will pick up the IBC packet and relay it on BandChain.

After BandChain processes the request, it will send an acknowledgement message along with `request_id` back. And, when the result of the request is finalized, BandChain will send a new IBC packet that contains the final data back. Relayers will listen and pick up those packets and relay them to the smart contract.

At this point, the smart contract can use the data from BandChain in its smart contract securely. If the smart contract wants to have new or more data, it can always send a new IBC packet and start the process again at any time.

cw-band

Our library (`cw-band`) provides data types and functions that are necessary for requesting and receiving data from BandChain. It will help developers to easily integrate their CosmWasm smart contract with BandChain.

Here are the data types and functions that the library provides for you.

Version

First, the library provides the version of the IBC channel of the Oracle module.

```
pub const IBC_APP_VERSION: &str = "bandchain-1";
```

Packet

The library also provides the types of packets that will be sent through IBC.

```
// Type of request packet data that will accept
#[cw_serde]
pub struct OracleRequestPacketData {
    pub client_id: String, // The client_id that you want to send. This will help you to identify the c
    pub oracle_script_id: Uint64, // The oracle_script_id that you want to request data from.
    pub calldata: Binary, // The data that you want to send to your oracle script.
    pub ask_count: Uint64, // Number of validators that you want to provide you data.
    pub min_count: Uint64, // Minimum number of validators that you will accept the final result.
    pub fee_limit: Vec<Coin>, // The fee limit of the data sources that you are willing to pay for the 
    pub prepare_gas: Uint64, // Prepare gas that you will use in preparation process of the oracle scri
    pub execute_gas: Uint64, // Execution gas that you will use in execution process of the oracle scri
}

// Type of response packet data from BandChain
#[cw_serde]
pub struct OracleResponsePacketData {
    pub client_id: String, // The client_id that you send in the OracleRequestPacketData.
    pub request_id: Uint64, // The id of the request of this response on BandChain.
    pub ans_count: Uint64, // The number of validators that helps provide this result.
    pub request_time: Uint64, // The time that this request is accepted on BandChain.
    pub resolve_time: Uint64, // The time that this request is resolved on BandChain.
    pub resolve_status: ResolveStatus, // The status of resolving the request.
    pub result: Binary, // The result of the request.
}

// Enum of possible resolve_status that BandChain will send
#[cw_serde]
pub enum ResolveStatus {
    #[serde(rename = "RESOLVE_STATUS_OPEN_UNSPECIFIED")]
    Open, // The request is not resolved yet.
    #[serde(rename = "RESOLVE_STATUS_SUCCESS")]
    Success, // The request is resolved successfully.
    #[serde(rename = "RESOLVE_STATUS_FAILURE")]
    Failure, // The request is failed to solve.
    #[serde(rename = "RESOLVE_STATUS_EXPIRED")]
    Expired, // The request is expired before solving.
}

// IBC Acknowledgement message (either result or error)
#[cw_serde]
pub enum AcknowledgementMsg {
    Result(Binary),
    Error(String),
}

// create a serialized success message
pub fn ack_success() -> Binary {
    let res = AcknowledgementMsg::Result(b"1".into());
    to_binary(&res).unwrap()
}

// create a serialized error message
pub fn ack_fail(err: String) -> Binary {
    let res = AcknowledgementMsg::Error(err);
    to_binary(&res).unwrap()
}

// Acknowledge message that BandChain will send after receives the request.
#[cw_serde]
pub struct BandAcknowledgement {
    pub request_id: Uint64, // The request_id of your request.
}
```

Oracle script - Price

This is the example of the Input and Output of the oracle script for getting the rates of symbols.

```
// Input of the oracle script.
#[derive(OBIEncode)]
pub struct Input {
    pub symbols: Vec<String>, // Symbol list that you want to request.
    pub minimum_sources: u8, // Minimum sources
}

// Output of the oracle script.
#[derive(OBIDecode)]
pub struct Output {
    pub responses: Vec<Response>, // Arrary of response of each each symbol.
}

// Rate of the symbol.
#[derive(OBIDecode)]
pub struct Response {
    pub symbol: String, // symbol.
    pub response_code: u8, // response_code.
    pub rate: u64, // the price of the symbol.
}
```

Learn more

- IBC docs
- IBC tutorials

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Pricefeed Module
cw-band
Introduction
Getting started
Example Use Cases
Develop
Nodes & Validators
Staking

Getting started

This guide serves as a quick reference on how to request and receive data from BandChain. For examples of CosmWasm contracts, please refer to the Example Use Cases section.

Step 1: Prepare an oracle script and data sources

There are two main components on the BandChain in the requesting process:

- Oracle script
- Data source

For requesting data from BandChain on the smart contract, you have to deploy both of them in the BandChain first so that your smart contract can specify the `oracle_script_id` when sending the request.

Step 2: Prepare a CosmWasm contract

To request data from BandChain's Oracle via a CosmWasm contract, your contract will need to implement the functions listed below:

```
ibc_channel_open()
ibc_channel_connect()
ibc_channel_close()
ibc_packet_ack()
ibc_packet_receive()
ibc_packet_timeout()
```

The behavior of the functions mentioned can be separated into 2 parts of the IBC protocol communication.

1. Channel
2. Packet

Channel

A channel between your contract and our oracle module on BandChain needs to be created to relay an IBC message via a relayer. To do that, 2 entry points need to be provided.

Note: As creating a channel can be started from either the contract or the oracle; A function that accepts messages regardless of the initiator side needs to be provided.

IBC Channel Open

`ibc_channel_open (OpenInit, OpenTry)`

```
#[entry_point]
pub fn ibc_channel_open(
    _deps: DepsMut,
    _env: Env,
    msg: IbcChannelOpenMsg,
) -> StdResult<IbcChannelOpenResponse> {

    // logic to verify channel order and the counterparty's version
    // ...

    Ok(Some(Ibc3ChannelOpenResponse {
        version: msg.channel().version.clone(),
    }))
}
```

This function should verify the order type of the channel and the counterparty's version and respond to its version to the counterparty's module.

IBC Channel Connect

`ibc_channel_connect (OpenAck, OpenConfirm)`

```
#[entry_point]
pub fn ibc_channel_connect(
    deps: DepsMut,
    _env: Env,
    msg: IbcChannelConnectMsg,
) -> Result<IbcBasicResponse, ContractError> {

    // logic to store channel_id for sending future IBC messages
    // ...

    Ok(IbcBasicResponse::default())
}
```

This function is called along with the channel detail so that the `channel_id` or other detail can be stored for any future IBC actions.

IBC Channel Close

```
#[entry_point]
pub fn ibc_channel_close(
    _deps: DepsMut,
    _env: Env,
    _msg: IbcChannelCloseMsg,
) -> StdResult<IbcBasicResponse> {

    // logic to handle when channel is closed.
    // ...

    Ok(IbcBasicResponse::default())
}
```

This function is called when the channel is somehow closed. You can add your logic to handle it in this function.

Packet

Three callback functions will need to be provided to accept the three outgoing messages from Oracle: `IbcPacketAckMsg`, `IbcPacketTimeoutMsg` and `IbcPacketReceiveMsg`.

IBCPacketAckMsg

```
#[entry_point]
pub fn ibc_packet_ack(
    _deps: DepsMut,
    _env: Env,
    _msg: IbcPacketAckMsg,
) -> StdResult<IbcBasicResponse> {

    // logic to process request_id
    // ...

    Ok(IbcBasicResponse::new().add_attribute("action", "ibc_packet_ack"))
}
```

The oracle will send an acknowledgement message with the corresponding request_id on BandChain if the request can be processed so that the sender's side can process the data as needed.

IBCPacketReceiveMsg

```
#[entry_point]
pub fn ibc_packet_receive(
    deps: DepsMut,
    _env: Env,
    msg: IbcPacketReceiveMsg,
) -> Result<IbcReceiveResponse, Never> {
    let packet = msg.packet;
    let resp: OracleResponsePacketData = from_slice(&packet.data)?;

    // logic to use/store result from BandChain
    // ...

    Ok(IbcReceiveResponse::new()
        .set_ack(ack_success())
        .add_attribute("action", "ibc_packet_received"))
}
```

After BandChain finishes your request, an OracleResponsePacketData packet will be sent to this function in your contract. The output of the oracle script that you requested will be contained in the result field.

IBCPacketTimeoutMsg

```
#[entry_point]
pub fn ibc_packet_timeout(
    _deps: DepsMut,
    _env: Env,
    _msg: IbcPacketTimeoutMsg,
) -> StdResult<IbcBasicResponse> {

    // logic to handle when the packet is timeout e.g. retry.
    // ...

    Ok(IbcBasicResponse::new().add_attribute("action", "ibc_packet_timeout"))
}
```

In the case where an acknowledgement message from the destination module hasn't been received, the relayers will call this function. Requests that are timeout can be handled within this function. e.g. retry, marking status.

Sending request

To send an IBC message to request data from BandChain, cosmwasm_std::IbcMsg::SendPacket needs to be used with the following three parameters:

- channel_id: The channel_id that you want to send a packet to
- data: The binary data of the packet that you want to send contained in an OracleRequestPacketData structure.
- timeout: The timeout timestamp.

Here is the example code for sending an IBC packet.

```
IbcMsg::SendPacket {
    channel_id: endpoint.channel_id,
    data: to_binary(&oracleRequestPacketData)?,
    // IBC timeout based on how long your contract will wait acknowkedgement until trigger timeout packe
    timeout: IbcTimeout::with_timestamp(env.block.time.plus_seconds(60)),
}
```

Step 3: Setup a relayer

The last step is to set up a relayer to listen and relay IBC packets between a client chain and BandChain.

Here are the simple guides for setting up a relayer.

- Hermes relayer
- Go relayer

Now, you are ready to request and receive data from BandChain.

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Pricefeed Module
cw-band
Introduction
Getting started
Example Use Cases
Develop
Nodes & Validators
Staking

Example Use Cases

As you can request any oracle script in BandChain from your CosmWasm contract. it opens the opportunity for getting any data from the real world such as crypto prices, stock prices, and weather data. Please see the examples of CosmWasm contracts that requests data from BandChain below.

Pricefeed

This price feed contract is an example of getting crypto prices from BandChain using the push model to allow other contracts can read the latest prices and execute actions in one transaction.

Requirements:

- Can trigger function to request new crypto prices from BandChain. (Execute `Request`)
- Able to query the price of a symbol in the USD unit. (Query `GetRate`)
- Able to query the price of a symbol in any unit. (Query `GetReferenceData`)
- Able to bulk query the price of symbols in any unit. (Query `GetReferenceDataBulk`)

You can see the full implementation of the contract here

Lottery

Lottery dApp is one of the use cases that can do with `cw-band` using the pull model to get the winning number when it reaches the end of the period to find the winner.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Oracle Binary Encoding (OBI)
Oracle WebAssembly (Owasm)
Remote Data Source Executor
Developer Tools
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Oracle Binary Encoding (OBI)

Oracle Binary Encoding (OBI) is a standard method for serializing and deserializing binary data in the BandChain ecosystem. Under the concept of Ethereum's Contract ABI Specification and Google's ProtoBuf, an OBI schema explains how a data object in any supported programming language can be encoded to and decoded from plain bytes.

OBI is designed with the following properties in mind:

- **Compactness**: OBI schema will be stored on-chain and passed around between blockchains. Thus, it is essential to keep the size of the schema specification tiniest.
- **Simplicity & Portability**: As a blockchain-agnostic protocol, OBI serialization and deserialization must be easy to implement in any environment. Consequently, complex platform-specific features are not supported.
- **Readability**: Lastly, OBI is intended to be used as a communication tool between oracle script creators and smart contract developers. It must be intuitive for readers to understand the OBI underlying objects from reading the schema.

Specification

An OBI schema is a non-self-describing binary serialization format of multiple objects. Some particular notes:

- An OBI schema consists of one or more individual schemas. In most cases, an OBI schema will consist of two individual schemas: the input type and the output type.
- bool is supported.
- 6 sizes (8-bit, 16-bit, 32-bit, 64-bit, 128-bit, and 256-bit) of signed and unsigned integers are supported. There are all serialized into big-endian bytes.
- Strings, bytes, vectors are serialized with their length as u32 first, followed by their contents.
- Structs are serialized field by field in the declaration order.

Backus–Naur Form (BNF) Grammar Specification

Below is the Backus–Naur form (BNF) grammar of an OBI schema.

```
<obi_schema>    ::= <obi_schema> | <obi_schema> "/" <indv_schema>
<indv_schema>   ::= <int_schema> | <uint_schema> | <string_schema> |
                    <bytes_schema> | <vector_schema> | <struct_schema>
<bool_schema>   ::= "bool"
<int_schema>    ::= "i8" | "i16" | "i32" | "i64" | "i128" | "i256"
<uint_schema>   ::= "u8" | "u16" | "u32" | "i64" | "u128" | "u256"
<string_schema> ::= "string"
<bytes_schema>  ::= "bytes"
<vector_schema> ::= "[" <indv_schema> "]"
<struct_schema> ::= "{" <struct_fields> "}"
<struct_fields> ::= <struct_member> | <struct_fields> "," <struct_member>
<struct_member> ::= <identifier> ":" <indv_schema>
```

Pseudocode Implementation

Below is an example pseudocode implementation of OBI schema declaration and the corresponding serializing function in a somewhat broken function language :P. The deserialization function is essentially the inverse of the serialization function.

```
(* An individual schema consists of 6 possible cases. *)
type indv_schema :=
| Bool(bool)
| Int(int)
| Uint(int)
| String
| Bytes
| Vector(indv_schema)
| Struct([(string, indv_schema)])

(* An OBI schema is essentially a list of individual schemas. *)
type obi_schema := [indv_schema]

(* Encode serializes the given object into bytes. *)
let encode (s : indv_schema) (o : object) :=
  match s with
  | Bool(sz) => be_unsigned_encode(o, sz)
  | Int(sz) => be_signed_encode(o, sz)
  | Uint(sz) => be_unsigned_encode(o, sz)
  | String => be_unsigned_encode(len(o), 32) ++ bytes_of_string(o)
  | Bytes => be_unsigned_encode(len(o), 32) ++ o
  | Vector(s) => be_unsigned_encode(len(o), 32) ++ concat (map o (encode s))
  | Struct(fs) => concat (map fs (fun (f, s) => encode s o[f]))
```

OBI Schema Examples

Below is an example OBI schema of an oracle script to fetch a cryptocurrency price, which is then multiplied by a specific multiplier. The OBI itself schema consists of two internal schemas, one for the inputs to the oracle script and the other for the output.

- The input consists with two fields: a string symbol and a u64 multiplier.
- The output consists with two fields: a u64 final price and a vector of struct each has string name and u64 timestamp.

```
# Compact OBI representation...
{symbol:string,multiplier:u64}/{price:u64,sources:[{name:string,time:u64}]}

# Prettified OBI representation...
{
  symbol: string,
  multiplier: u64
} / {
  price: u64,
  sources: [{ name: string, time: u64 }]
}
```

Example Object Serialization

```
{"symbol": "BTC", "multiplier": 1000000000}
0x000000034254430000000003b9aca00
  ^       ^     ^
  |       |     +- 64-bit be encode of 1000000000 is 0x000000003b9aca00
  |       +------- "BTC" data is encoded as 0x425443
  +--------------- 32-bit be encode of length 3 is 0x00000003 #BTC is a string with a length of 3


{
  "price": 9268300000000,
  "sources": [
    {"name": "CoinGecko", "time": 1590305341},
    {"name": "CryptoCompare", "time": 1590305362}
  ]
}
0x0000086df1baab0000000002000000009436f696e4765636b6f000000005eca223d0000000d43727970746f436f6d706172650
  ^               ^       ^       ^                 ^               ^       ^                         ^
  |               |       |       |                 |               |       |                         +
  |               |       |       |                 |               |       +- "CryptoCompare" data is e
  |               |       |       |                 |               +- 32-bit be encode of length 13 is
  |               |       |       |                 +- 64-bit be encode of 1590305341 is 0x000000005eca2
  |               |       |       +- "CoinGecko" data is encoded as 0x436f696e4765636b6f
  |               |       +- 32-bit be encode of length 9 is 0x00000009 #CoinGecko length is 9
  |               +------- 32-bit be encode of length 2 is 0x00000002 #sources has a value that is an a
  +--------------- 64-bit be encode of 9268300000000 is 0x0000086df1baab00
```

Reference Implementations

OBI serialization libraries are being actively developed in multiple programming languages. Head over to BandChain's OBI module to see all currently available implementations.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Oracle Binary Encoding (OBI)
Oracle WebAssembly (Owasm)
Remote Data Source Executor
Developer Tools
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Oracle WebAssembly (Owasm)

Oracle WebAssembly, or Owasm for short, is Band Protocol's Domain Specific Language (DSL) for writing oracle scripts to be used in the BandChain ecosystem.

Owasm Library Structure

The Owasm library is separated into `owasm-vm` and `owasm-kit` for different use cases.

Owasm-vm

The Owasm-vm library implements a wasmer virtual machine runtime and logic for the on-chain execution of oracle scripts.

Owasm-kit

Owasm-kit/OEI

The OEI modules defines a set of functions that are part of the Owasm Oracle Environment Interface. These functions are then accessible to an oracle script during its execution.

OEI Module Functions

get_ask_count()

GetAskCount returns the number of validators asked to work for this oracle query.

Returns:

- `ask_count: i64` - The requested validator count.

get_min_count()

GetMinCount returns the minimum number of validators to move to aggregation phase.

Returns:

- `min_count: i64` - The sufficient validator count.

get_ans_count()

GetAnsCount returns the number of validators that submit answers. Zero during preparation.

Returns:

- `ans_count: i64` - The received validator count.

get_prepare_time()

GetPrepareTime returns the time of *preparation* phase was run.

Returns:

- `timestamp: i64` - The block timestamp during the preparation phase.

get_execute_time()

GetExecuteTime returns the time of *aggregation* phase. Zero during preparation.

Returns:

- `timestamp: i64` - The block timestamp during the aggregation phase.

get_calldata()

GetCalldata returns the raw calldata as specified when the oracle request is submitted.

Returns:

- `calldata: Vec<u8>` - The calldata in bytes.

save_return_data(&[u8])

SaveReturnData saves the given data as the result of the oracle execution. Must only be called during execution phase and must be called exactly once.

Parameters:

- `data: &[u8]` - The return data in bytes.

ask_external_data(i64, i64, &[u8])

AskExternalData issues a new raw request to the host environment using the specified data source id and calldata, and assigns it to the given external id. Must only be called during preparation phase.

Parameters:

- `eid: i64` - The external id to be assigned for the raw data source request.
- `did: i64` - The data source id to be requested.
- `calldata: &[u8]` - The calldata in bytes to be used in a `did` data source execution

get_external_data(i64, i64)

GetExternalData returns the data reported from the given validator index for the given external data id. Result is OK if the validator reports data with zero return status, and Err otherwise. Must only be called during execution phase.

Parameters:

- `eid: i64` - The external id of the data source execution to be returned.
- `vid: i64` - The validator index for the given external id.

Returns:

- `result: Result<String, i64>` - The data reported in String format from the given validator index for the given external data id.

Owasm-kit/Ext

The Owasm Extension module provides a convenient way to write oracle scripts that connect to various public APIs. Examples of these are functions to calculate the mean, median, and majority values from the validator's reported results , which can be used during the aggregation phase of an oracle script.

load_input<T>(i64)

LoadInput returns an iterator of raw data points for the given external id, parsed into the parameterized type. Skip data points with nonzero status or cannot be parsed.

Parameters:

- `eid: i64` - External id.

Returns:

- `iterator: Iterator` - Iterator of raw data points.

load_average(<T>(i64)

LoadAverage returns the average value of the given external id, ignoring unsuccessful reports.

Parameters:

- `eid: i64` - External id.

Returns:

- `result: Option<T>` - Average value of valid data points.

load_median<T>(i64)

LoadMedian returns the median value of the given external ID, ignoring unsuccessful reports.

Parameters:

- `eid: i64` - External id.

Returns:

- `result: Option<T>` - Median value of valid data points.

load_majority(<T>(i64)

LoadMajority returns the majority value of the given external ID, ignoring unsuccessful reports.

Parameters:

- `eid: i64` - External id.

Returns:

- `result: Option<T>` - Median value of valid data points.

The full implementation can be found here.

Usage

To illustrate an example usage of the Owasm library, we will be using the example below. The code is based off an oracle script that retrieves the price of a stock.

```
use obi::{OBIDecode, OBIEncode, OBISchema};
use owasm_kit::{execute_entry_point, ext, oei, prepare_entry_point};

#[derive(OBIDecode, OBISchema)]
struct Input {
    symbol: String,
    multiplier: u64,
}

#[derive(OBIEncode, OBISchema)]
struct Output {
    px: u64,
}

#[no_mangle]
fn prepare_impl(input: Input) {
    oei::ask_external_data(19, 1, &input.symbol.as_bytes());
}

#[no_mangle]
fn execute_impl(input: Input) -> Output {
    let avg: f64 = ext::load_average(1);
    Output { px: (avg * input.multiplier as f64) as u64 }
}

prepare_entry_point!(prepare_impl);
execute_entry_point!(execute_impl);
```

The script starts off by defining the input and output structs. In this case, the input comprises of the stock ticker (`string`) and the multiplier we want to multiply the stock's price by (`u64`). On the other hand, the output is simply the price of the stock multiplied by the multiplier, returned as a `u64` value.

Once the input and output structs and types have been defined, we move on to define the preparation and execution phases the oracle script, defined by `prepare_impl` and `execute_impl`, respectively.

The `prepare_impl` function takes the previously-defined input struct as an argument. The function itself then only peforms one task; calling the `ask_external_data` function in the `oei` module. This call to `ask_external_data` is then caught and ultimately resolved by BandChain through `exec_env.go`.

The `execute_impl` function takes in the input type as an argument as well, but also returns the output struct type, as one might expect. It then starts by computing the final value of the request through calling `load_average` function from the `ext` module. It then proceeds to use the computed average to construct and return the appropriate output struct.

Once we have defined the functions for both stages of the oracle script's execution, we need to pass in the appropriate input values and actually make the function calls. To do so, oracle script writer can use our macros defined in `macros.rs`, also shown below. The aim of these macros is to reduce the load of the script writer by handling the work of retrieving the calldata, deserializing it, and using it to construct the appropriate input struct for them.

```
#[macro_export]
macro_rules! prepare_entry_point {
    ($name:ident) => {
        #[no_mangle]
        pub fn prepare() {
            $name(OBIDecode::try_from_slice(&oei::get_calldata()).unwrap());
        }
    };
}

#[macro_export]
macro_rules! execute_entry_point {
    ($name:ident) => {
        #[no_mangle]
        pub fn execute() {
            oei::save_return_data(
                &$name(OBIDecode::try_from_slice(&oei::get_calldata()).unwrap())
                    .try_to_vec()
                    .unwrap(),
            );
        }
    };
}
```

The last two lines of the oracle script above shows the macros in action.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Oracle Binary Encoding (OBI)
Oracle WebAssembly (Owasm)
Remote Data Source Executor
Developer Tools
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Remote Data Source Executor

The remote data source executor executes the various data sources stored on BandChain in order to retrieve data from the data providers. The executor itself is a `function-as-a-Service` hosted on a cloud service provider.

The goal of this design is to separate data source execution processes from processes related to the validator node itself. This separation of execution will be beneficial in cases where the data sources are not working correctly (infinite loop, bad code execution, for example). In those cases, such separation can prevent validator nodes from overloading or crashing due to the error.

Also, from a validator's perspective, all they have to do to execute a data source is to send an executable file content in JSON format (the data source script), along with the relevant calldata, to an endpoint, thus minimizing the work they have to do themselves.

Resource

Endpoint	HTTP Request Method	Description
`/execute`	`POST`	Executes the data source

Request

Request Headers

```
{
  "content-type": "application/json"
}
```

Request Body

Field	Type	Description
executable	string`	Base64-encoded data source that the executor will execute
calldata	string	Bash-encoded input parameters of data source
timeout	number	Execution time limit (in milliseconds)

Example

```
{
  "executable": "IyEvYmluL3NoXG5cbnN5bWJvbD0kMVxuXG4jIENyeXB0b2N1cnJlbmN5IHByaWNlIGVuZHBvaW50OiBodHRwcz
  "calldata": "bitcoin",
  "timeout": 1000
}
```

Response

Response Headers

```
{
  "content-type": "application/json",
  "access-control-allow-methods": "OPTIONS, POST"
}
```

Response Body

Field	Type	Description
returncode	number	Exit code (0-255)
stdout	string	Data source output
stderr	string	Data source error output
error	string	`""` or error message from executor

Response Codes

200 OK

Exit code	Description	Error Message
0 - 255	Executable errors from data source run successfully	N/A
111	Execution timeout	Execution time limit exceeded
112	Stdout exceed	Stdout exceeded max size
113	Stderr exceed	Stderr exceeded max size
126	Execution fail	Execution fail

400 Bad Request

Error	Error message
Missing executable value	`executable` field is missing from JSON request
Executable value exceed	`executable` value exceed max size
Missing calldata value	`calldata` field is missing from JSON request
Calldata value exceed	`calldata` exceeded max size
Missing timeout value	`timeout` field is missing from JSON request
Invalid timeout format	`timeout` type is invalid
Runtime value exceed	Runtime exceeded max size

Getting Started

Overview

PyBand is a library written in Python used for interacting with BandChain. The library provides classes, methods, and protobuf classes for the ease of sending transactions, querying data, OBI encoding, and wallet management.

The library is implemented based on gRPC-web protocol which sends HTTP/1.5 or HTTP/2 requests to a gRPC proxy server, before serving them as HTTP/2 to gRPC server.

This library is only implemented on Python.

System Requirements

- Recommended Python version: `3.8.x` or below
- MacOS, Windows (including WSL), and Linux are supported

Installation

This library is available on PyPI

```
pip install pyband
```

Basic Usages

Making an oracle request

This section describes the methods used to send a transaction containing an oracle request to BandChain

Step 1: Import `pyband` and create a parameter: `grpc_url` with the required `<GRPC>` endpoint which can be found here. Then the client instance needs to be initialized in order to allow for the methods in client module to be used.

```
from pyband.client import Client
def main():
    # Step 1
    grpc_url = "<GRPC>" # ex.laozi-testnet6.bandchain.org(without https://)
    c = Client(grpc_url)

if __name__ == "__main__":
    main()
```

Step 2: As the sender's address is required for sending a transaction, we will have to initialize the address first. To do this, start by importing `PrivateKey` from wallet module. In this example, we will get Mnemonic from environment variables.

```
from pyband.wallet import PrivateKey

MNEMONIC = os.getenv("MNEMONIC")
private_key = PrivateKey.from_mnemonic(MNEMONIC)
public_key = private_key.to_public_key()
sender_addr = public_key.to_address()
sender = sender_addr.to_acc_bech32()
```

After that, we will transform the private key to a public key, the public key to an address, and an address of type `Address` to an address of type `str`.

Step 3: Before constructing a transaction, additional information is needed.

As a transaction requires:

- messages
- sequence
- account_num
- chain_id
- fee
- gas
- memo

to be constructed, we will need to get those.

Messages

In this example, we will use `MsgRequestData` with the following parameters as our message.

- **oracle_script_id** `<int>`: The oracle script ID.
- **calldata** `<bytes>`: The calldata from a request (e.g., the hex string representing OBI-encoded value of `{"symbols": ["ETH"], "multiplier": 100}`).
- **ask_count** `<int>`: The number of validator required to process this transaction.
- **min_count** `<int>`: The minimum number of validator required to process this transaction.
- **client_id** `<str>`: Name of the client (can be any name or an empty string).
- **fee_limit** `<Coin>`: The fee limit.
- **prepare_gas** `<int>`: The amount of gas used in the preparation stage.
- **execute_gas** `<int>`: The amount of gas used in the execution stage.
- **sender** `<str>`: The sender's address.

```
from pyband.proto.oracle.v1.tx_pb2 import MsgRequestData

request_msg = MsgRequestData(
    oracle_script_id=37,
    calldata=bytes.fromhex("0000000200000003425443000000034554480000000000000064"),
    ask_count=4,
    min_count=3,
    client_id="BandProtocol",
    fee_limit=[Coin(amount="100", denom="uband")],
    prepare_gas=50000,
    execute_gas=200000,
    sender=sender,
)
```

Instead from using bytes for the calldata, oracle binary encoding (obi) can also be used.

```
from pyband.obi import PyObi

obi = PyObi("{symbols:[string],multiplier:u64}/{rates:[u64]}")
calldata = obi.encode({"symbols": ["ETH"], "multiplier": 100})
```

The message can be any message as listed in Oracle Modules or Cosmos Based Messages. However, please note that our message should be imported from the generated protobuf files.

Sequence and Account Number

Sequence and account number can be retrieved by calling `get_account` from the client module created in **step 1**.

```
account = c.get_account(sender)
account_num = account.account_number
sequence = account.sequence
```

Fee

Fee can be created by using `Coin` from the generated protobuf file.

```
from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin

fee = [Coin(amount="0", denom="uband")]
```

Step 4: Now we can construct a `Transaction` from the transaction module.

```
from pyband.transaction import Transaction

txn = (
    Transaction()
    .with_messages(request_msg)
    .with_sequence(sequence)
    .with_account_num(account_num)
    .with_chain_id(chain_id)
    .with_gas(2000000)
    .with_fee(fee)
    .with_memo("")
)
```

Step 5: Preparing the transaction before sending

Call `get_sign_doc` to get a signed transaction which we can use to get the signature from.

After that, we can get the raw transaction by calling `get_tx_data` and putting the signature and public key as the parameters.

```
sign_doc = txn.get_sign_doc(public_key)
# Need to serialize sign_doc of type cosmos_tx_type.SignDoc to string
signature = private_key.sign(sign_doc.SerializeToString())
tx_raw_bytes = txn.get_tx_data(signature, public_key)
```

Step 6: After getting the raw transaction, the transaction can now be sent.

While there are 3 modes for sending the transaction, Block mode will be used in this example. We can call `send_tx_block_mode` with the raw transaction as parameter.

```
import os

from pyband.client import Client
from pyband.transaction import Transaction
from pyband.wallet import PrivateKey

from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgRequestData
from google.protobuf.json_format import MessageToJson

def main():
    # Step 1
    grpc_url = "<GRPC>" # ex.laozi-testnet6.bandchain.org(without https://)
    c = Client(grpc_url)

    # Step 2
    MNEMONIC = os.getenv("MNEMONIC")
    private_key = PrivateKey.from_mnemonic(MNEMONIC)
    public_key = private_key.to_public_key()
    sender_addr = public_key.to_address()
    sender = sender_addr.to_acc_bech32()

    # Step 3
    request_msg = MsgRequestData(
        oracle_script_id=37,
        calldata=bytes.fromhex("0000000200000003425443000000034554480000000000000064"),
        ask_count=4,
        min_count=3,
        client_id="BandProtocol",
        fee_limit=[Coin(amount="100", denom="uband")],
        prepare_gas=50000,
        execute_gas=200000,
        sender=sender,
    )

    account = c.get_account(sender)
    account_num = account.account_number
    sequence = account.sequence

    fee = [Coin(amount="0", denom="uband")]
    chain_id = c.get_chain_id()

    # Step 4
    txn = (
        Transaction()
        .with_messages(request_msg)
        .with_sequence(sequence)
        .with_account_num(account_num)
        .with_chain_id(chain_id)
        .with_gas(2000000)
        .with_fee(fee)
        .with_memo("")
    )

    # Step 5
    sign_doc = txn.get_sign_doc(public_key)
    signature = private_key.sign(sign_doc.SerializeToString())
    tx_raw_bytes = txn.get_tx_data(signature, public_key)

    # Step 6
    tx_block = c.send_tx_block_mode(tx_raw_bytes)
    print(MessageToJson(tx_block))

if __name__ == "__main__":
    main()
```

And the result should look like this.


```
{
  "height": "603247",
  "txhash": "587FF6D48E5CB8A23715389FE3CAC10262777B395E4D0C554916127461F63446",
  "data": "0A090A0772657175657374",
  "rawLog": "[{\"events\":[{\"type\":\"message\",\"attributes\":[{\"key\":\"action\",\"value\":\"request
  "logs": [
    {
      "events": [
        {
          "type": "message",
          "attributes": [{ "key": "action", "value": "request" }]
        },
        {
          "type": "raw_request",
          "attributes": [
            { "key": "data_source_id", "value": "61" },
            {
              "key": "data_source_hash",
              "value": "07be7bd61667327aae10b7a13a542c7dfba31b8f4c52b0b60bf9c7b11b1a72ef"
            },
            { "key": "external_id", "value": "6" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "57" },
            {
              "key": "data_source_hash",
              "value": "61b369daa5c0918020a52165f6c7662d5b9c1eee915025cb3d2b9947a26e48c7"
            },
            { "key": "external_id", "value": "0" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "62" },
            {
              "key": "data_source_hash",
              "value": "107048da9dbf7960c79fb20e0585e080bb9be07d42a1ce09c5479bbada8d0289"
            },
            { "key": "external_id", "value": "3" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "5" },
            { "key": "calldata", "value": "huobipro BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "59" },
            {
              "key": "data_source_hash",
              "value": "5c011454981c473af3bf6ef93c76b36bfb6cc0ce5310a70a1ba569de3fc0c15d"
            },
            { "key": "external_id", "value": "2" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "4" },
            { "key": "calldata", "value": "binance BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "9" },
            { "key": "calldata", "value": "bittrex BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "7" },
            { "key": "calldata", "value": "kraken BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "8" },
            { "key": "calldata", "value": "bitfinex BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "58" },
            {
              "key": "data_source_hash",
              "value": "7e6759fade717a06fb643392bfde837bfc3437da2ded54feed706e6cd35de461"
            },
            { "key": "external_id", "value": "1" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" }
          ]
        },
        {
          "type": "request",
          "attributes": [
            { "key": "id", "value": "306633" },
            { "key": "client_id", "value": "BandProtocol" },
            { "key": "oracle_script_id", "value": "37" },
            {
              "key": "calldata",
              "value": "0000000200000003425443000000034554480000000000000064"
            },
            { "key": "ask_count", "value": "4" },
            { "key": "min_count", "value": "3" },
            { "key": "gas_used", "value": "111048" },
            { "key": "total_fees" },
            {
              "key": "validator",
              "value": "bandvaloper1zl5925n5u24njn9axpygz8lhjl5a8v4cpkzx5g"
            },
            {
              "key": "validator",
              "value": "bandvaloper17n5rmujk78nkgss7tjecg4nfzn6geg4cqtyg3u"
            },
            {
              "key": "validator",
              "value": "bandvaloper1p46uhvdk8vr829v747v85hst3mur2dzlhfemmz"
            },
            {
              "key": "validator",
              "value": "bandvaloper1ldtwjzsplhxzhrg3k5hhr8v0qterv05vpdxp9f"
            }
          ]
        }
      ]
    }
  ],
  "gasWanted": "2000000",
  "gasUsed": "566496"
}
```


Sending BAND token

The process of sending BAND token is similar to **making an oracle request** , except we will use `MsgSend` as our message.

The `MsgSend` contains the following parameters:

- **from_address** `<str>`: The sender address which as a string.
- **to_address** `<str>`: The receiver address which as a string.
- **amount** `<int>`: The amount of BAND in Coin that you want to send. In this case, we want to send 1 BAND or 1000000 UBAND

```
from pyband.proto.cosmos.bank.v1beta1.tx_pb2 import MsgSend

msg = MsgSend(
   from_address = sender,
   to_address = "band1jrhuqrymzt4mnvgw8cvy3s9zhx3jj0dq30qpte",
   amount = [Coin(amount="100", denom="uband")]
)
```

The final code should look as shown below.

```
import os

from pyband.client import Client
from pyband.transaction import Transaction
from pyband.wallet import PrivateKey

from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.cosmos.bank.v1beta1.tx_pb2 import MsgSend
from google.protobuf.json_format import MessageToJson

def main():
    # Step 1
    grpc_url = "<GRPC>" # ex.laozi-testnet6.bandchain.org(without https://)
    c = Client(grpc_url)

    # Step 2
    MNEMONIC = os.getenv("MNEMONIC")
    private_key = PrivateKey.from_mnemonic(MNEMONIC)
    public_key = private_key.to_public_key()
    sender_addr = public_key.to_address()
    sender = sender_addr.to_acc_bech32()

    # Step 3
    send_msg = MsgSend(
      from_address = sender,
      to_address = "band1jrhuqrymzt4mnvgw8cvy3s9zhx3jj0dq30qpte",
      amount = [Coin(amount="1000000", denom="uband")]
    )

    account = c.get_account(sender)
    account_num = account.account_number
    sequence = account.sequence

    fee = [Coin(amount="0", denom="uband")]
    chain_id = c.get_chain_id()

    # Step 4
    txn = (
        Transaction()
        .with_messages(send_msg)
        .with_sequence(sequence)
        .with_account_num(account_num)
        .with_chain_id(chain_id)
        .with_gas(2000000)
        .with_fee(fee)
        .with_memo("")
    )

    # Step 5
    sign_doc = txn.get_sign_doc(public_key)
    signature = private_key.sign(sign_doc.SerializeToString())
    tx_raw_bytes = txn.get_tx_data(signature, public_key)

    # Step 6
    tx_block = c.send_tx_block_mode(tx_raw_bytes)
    print(MessageToJson(tx_block))

if __name__ == "__main__":
    main()
```

And the result should look like this.


```
{
  "height": "603302",
  "txhash": "815F488B3F05F2CBDD57C433DBEAF01FBFB06F378716A8ECDF5888095D6F7F7C",
  "data": "0A060A0473656E64",
  "rawLog": "[{\"events\":[{\"type\":\"message\",\"attributes\":[{\"key\":\"action\",\"value\":\"send\"
  "logs": [
    {
      "events": [
        {
          "type": "message",
          "attributes": [
            { "key": "action", "value": "send" },
            {
              "key": "sender",
              "value": "band18p27yl962l8283ct7srr5l3g7ydazj07dqrwph"
            },
            { "key": "module", "value": "bank" }
          ]
        },
        {
          "type": "transfer",
          "attributes": [
            {
              "key": "recipient",
              "value": "band1jrhuqrymzt4mnvgw8cvy3s9zhx3jj0dq30qpte"
            },
            {
              "key": "sender",
              "value": "band18p27yl962l8283ct7srr5l3g7ydazj07dqrwph"
            },
            { "key": "amount", "value": "1000000uband" }
          ]
        }
      ]
    }
  ],
  "gasWanted": "2000000",
  "gasUsed": "49029"
}
```


Getting reference data

This section shows an example on how to query data from BandChain. This example queries the standard price reference based on the given symbol pairs, min count, and ask count.

Step 1: Import `pyband` and create a parameter: `grpc_url` with the required `<GRPC>` endpoint which can be found here. Then the client instance needs to be initialized in order to allow for the methods in client module to be used.

```
from pyband.client import Client

def main():
    # Step 1
    grpc_url = "<GRPC>" # ex.laozi-testnet6.bandchain.org(without https://)

if __name__ == "__main__":
    main()
```

Step 2 After importing `Client`, the function `get_reference_data` can now be used to get the latest price.

The function contains the following parameters

- **pairs** `<List[str]>`: list of cryptocurrency pairs
- **min_count** `<int>`: integer of min count
- **ask_count** `<int>`: integer of ask count

```
from pyband.client import Client

def main():
    # Step 1
    grpc_url = "<GRPC>" # ex.laozi-testnet6.bandchain.org(without https://)
    c = Client(grpc_url)

    # Step 2
    print(c.get_reference_data(["BTC/USD", "ETH/USD"], 3, 4))

if __name__ == "__main__":
    main()
```

And running the code above should return a result that looks like this.

```
[
  ReferencePrice(
    (pair = "BTC/USD"),
    (rate = 34614.1),
    (updated_at = ReferencePriceUpdated((base = 1625655764), (quote = 1625715134)))
  ),
  ReferencePrice(
    (pair = "ETH/USD"),
    (rate = 2372.53),
    (updated_at = ReferencePriceUpdated((base = 1625655764), (quote = 1625715134)))
  )
]
```

Client Module

This module provides a function to get the data from Bandchain gRPC and send the data to Bandchain gRPC.

Note: Get the `<GRPC>` here

get_data_source(id)

This function returns data source details of the given ID.

Parameter

- **id** `<int>`: Data source ID

Return

- oracle_type.DataSource

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
id = 1

c = Client(grpc_url)
data_source = c.get_data_source(id)
print(MessageToJson(data_source))
```

Result

```
{
  "owner": "band1jfdmjkxs3hvddsf4ef2wmsmte3s5llqhxqgcfe",
  "name": "DS1",
  "description": "TBD",
  "filename": "32ee6262d4a615f2c3ca0589c1c1af79212f24823453cb3f4cfff85b8d338045",
  "treasury": "band1jfdmjkxs3hvddsf4ef2wmsmte3s5llqhxqgcfe"
}
```

get_oracle_script(id)

This function returns oracle script details of the given ID.

Parameter

- **id** `<int>`: Oracle Script ID

Return

- oracle_type.OracleScript

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
id = 1

c = Client(grpc_url)
oracle_script = c.get_oracle_script(id)
print(MessageToJson(oracle_script))
```

Result

```
{
  "owner": "band1jfdmjkxs3hvddsf4ef2wmsmte3s5llqhxqgcfe",
  "name": "OS1",
  "description": "TBD",
  "filename": "f86b37dbe62c3b8c86ae28523bf09e9963a6b2951dd1a5be79f29f66d8236abf",
  "schema": "{gas_option:string}/{gweix10:u64}"
}
```

get_request_by_id(id)

This function returns request details of the given ID.

Parameter

- **id** `<int>`: Request ID

Return

- oracle_query.QueryRequestResponse

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
id = 1

c = Client(grpc_url)
request = c.get_request_by_id(id)
print(MessageToJson(request))
```

Result

```
{
  "result": {
    "clientId": "from_bandd",
    "oracleScriptId": "37",
    "calldata": "AAAABgAAAANCVEMAAAADRVRIAAAAA01JUgAAAANBTkMAAAAERE9HRQAAAARMVU5BAAAAADuaygA=",
    "askCount": "1",
    "minCount": "1",
    "requestId": "1",
    "ansCount": "1",
    "requestTime": "1624374833",
    "resolveTime": "1624374844",
    "resolveStatus": "RESOLVE_STATUS_SUCCESS",
    "result": "AAAABgAAHBpu4YHAAAABqf4l8EAAAAABFPDhkAAAAACK0bhAAAAAAsYHsgAAAABLZl5AA=="
  }
}
```

get_reporters(validator)

This function returns a list of reporters associated with the given validator.

Parameter

- **validator** `<str>`: Validator address

Return

- `List<str>`

Example

```
from pyband import Client

grpc_url = "<GRPC>" # without https://
validator = "bandvaloper1p46uhvdk8vr829v747v85hst3mur2dzlhfemmz"

c = Client(grpc_url)
print(c.get_reporters(validator))
```

Result

```
[
  "band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f",
  "band1zgly2mgx7ykckfgm4dqgc58vrntdlgcemy62eq",
  "band1sfmc6995mk0d55zy2vy8dxu8s54y5e7yqquxkr",
  "band1jtjuucr5wea43up4d3d98e3xn737lry800j6tf",
  "band1jd95fjm3j43pqurc2k4suzmznhux85hsjrx0a8",
  "band1l5kxfkd7gvvtpd37gmjnpvd0suzwypgz5u9ysc"
]
```

get_latest_block

This function returns the latest block in the chain.

Return

- `tendermint_query.GetLatestBlockResponse`

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://

c = Client(grpc_url)
latest_block = c.get_latest_block()
print(MessageToJson(latest_block))
```

Result

```
{
  "blockId": {
    "hash": "FHpPvUlEF2WxXGGkrN9Lc4pI5oj/3Z5/U1UvLXsX/z0=",
    "partSetHeader": {
      "total": 1,
      "hash": "wgzw1SgKKd+uZwKSHWONJ6qNigdwOPhqW2nTq3AYKv0="
    }
  },
  "block": {
    "header": {
      "version": { "block": "11" },
      "chainId": "band-laozi-testnet2",
      "height": "603404",
      "time": "2021-07-12T08:05:22.386235207Z",
      "lastBlockId": {
        "hash": "XMfyuM/nZZcoWVCQG+SQ93zaYuVt47i/ISlgi3KJqJw=",
        "partSetHeader": {
          "total": 1,
          "hash": "u4XfI/858RuZExpK0D1mtQzC90R7fbhBAyIboqRQcoI="
        }
      },
      "lastCommitHash": "caU5MlsAHEpy29PzMDQJ0OdIGkCfIyttQLQoTrVNpfA=",
      "dataHash": "TyORhRUl3QS/stvNbHsld2uU47aOMhbHfbWbmzCgt9s=",
      "validatorsHash": "e0hw8Ieib1SLF87P75KUsV/Zh4UZDZocxtN13v+temM=",
      "nextValidatorsHash": "e0hw8Ieib1SLF87P75KUsV/Zh4UZDZocxtN13v+temM=",
      "consensusHash": "ek5k0qm1ziK3XpVuICUnTcA7aEbM13JRUqa8DQcn4z4=",
      "appHash": "VPpwYxRcdOtU5OclkZ+W1zuVCgo+P5TOkqwWXYY/Stw=",
      "lastResultsHash": "RTYNyG6gfZr3/J5OPnmv49+qR5wQmUUFypdaoZDT188=",
      "evidenceHash": "47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=",
      "proposerAddress": "Mi9CIpvJaLKLAUv/2dg6nGeAZGw="
    },
    "data": {
      "txs": [
        "CuYCCrYCChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESmQIIot0SEg8IDBoLMC4wMDA4NzM5NAoSDQgKGgkwLjAwMDg3MA
        "CqMECvMDChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESlgMIn90SEl8IAhABGlk0MjkgQ2xpZW50IEVycm9yOiBUb28gTW
        "CvACCsACChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESowIIoN0SEl8IAhABGlk0MjkgQ2xpZW50IEVycm9yOiBUb28gTW
        "CvACCsACChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESowIIoN0SEl8IAhABGlk0MjkgQ2xpZW50IEVycm9yOiBUb28gTW
        "CtABCs0BChkvb3JhY2xlLnYxLk1zZ1JlcXVlc3REYXRhEq8BCCUSWAAAAAoAAAAEVVNEVAAAAANCU1YAAAADWE1SAAAABF
        "CqcBCqQBChkvb3JhY2xlLnYxLk1zZ1JlcXVlc3REYXRhEoYBCCUSOAAAAAYAAAADQlRDAAAAA0VUSAAAAANSVIAAAADUS
        "CoABCn4KGS9vcmFjbGUudjEuTXNnUmVxdWVzdERhdGESYQgrEhwAAAACAAAABEJUQ0IAAAAEQkVUSAAAAAA7msoAGAYgA
        "CpoCCpcCChkvb3JhY2xlLnYxLk1zZ1JlcXVlc3REYXRhEvkBCCwSggEAAAAVAAAABEFBUEwAAAAFR09PR0wAAAAEVFNMQQ
        "CsYBCsMBChkvb3JhY2xlLnYxLk1zZ1JlcXVlc3REYXRhEqUBCCUSTgAAAAkAAAADTVlCAAAABESQWFMAAAADT1NUAAAAA1
        "CpEECvADChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGES0wMIpN0SEkIIAho+MC45OTg2MjksMTQwLjgsMjEzLjc2LDEuMC
        "CpAECu8DChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGES0gMIpd0SEhUIBhoRMzQyODEuMDcsMjE0NC40NQoSFhoUMzQyOD
        "Co8ECuwDChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGES0QMIpd0SEhUIBhoRMzQyODEuMDcsMjE0NC40NQoSJggBGiIzND
        "CvEDCtADChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESswMIn90SEj0IAho5OS45LDAuODE4ODY4NDY1LDEzLjA5LDQzMS
        "CoYDCsoCChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESrQII0N0SEhcIoDgaEjkwMi4zMjM1MjgzMzMxNDU2ChIXCKM4Gh
        "CvkCCskCChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESrAIIo90SEhcIozgaEjkwMi4zMjM1MjgzMzM1NDU2ChIXCKA4Gh
        "CsQCCpQCChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGES9wEIp90SEiIIARofMzQyNjcuMzM5OTk5OTk5OTk5OTk1LDIxND
        "Co0ECuwDChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESzwMIod0SEhcIoDgaEjkwMi4zMjM1MjgzMzMxNDU2ChIXCKM4Gh
        "CogFCpQCChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESwEIp90SEiEaHzM0MjY3LjMzOTk5OTk5OTk5OTk1LDIxNDMuOTk
        "CvEDCrUDChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGESmAMIpt0SEpQBCAEajwExNDUuMDM4MjE1LDI1NDUuOTk0LDY1Nz
        "CpEECvADChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGES0wMIpN0SEkIIAho+MC45OTg2MjksMTQwLjgsMjEzLjc2LDEuMC
        "CskEC08ECjChgvb3JhY2xlLnYxLk1zZ1JlcG9ydERhdGES8AMIpN0SEjIQARowMSwxNDAuNywyMTMuNjIsMSwwLjU0NiwxMz
      ]
    },
    "evidence": {},
    "lastCommit": {
      "height": "603403",
      "blockId": {
        "hash": "XMfyuM/nZZcoWVCQG+SQ93zaYuVt47i/ISlgi3KJqJw=",
        "partSetHeader": {
          "total": 1,
          "hash": "u4XfI/858RuZExpK0D1mtQzC90R7fbhBAyIboqRQcoI="
        }
      },
      "signatures": [
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "Zdyy3QL8E8XZYhDrA8fAUD4m2Jc=",
          "timestamp": "2021-07-12T08:05:22.304030411Z",
          "signature": "ISRrl36Kv2m6vOv7d2OpoOQtR6fmeodsg/rxGKXfch5IutvFw4/p8/LMDg3zUT64raHz8YW2aXid88B
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "xLnySmLJL6Qq4ebq+oPMs+KEerU=",
          "timestamp": "2021-07-12T08:05:22.435650954Z",
          "signature": "fxFAi9FQEe98Dm7cv8psNwWdQg1B/h6RaEIrfKuPwBhfiN80WE36+ioNKK7BIvfOx8QuBBuNQ+ic6mM
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "D3OpjoewGqrIf2g+qADd6sKpM24=",
          "timestamp": "2021-07-12T08:05:22.395247997Z",
          "signature": "Ulku3yfZDtUxXdidUwlClrKAgns0O2z4suYZSef+rdkPxVxmCZayA4wwMUWVNnTfEdkLzsSkl6TVBSf
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "KLbUBDhTpSjBO2cco4cmyhJiR7w=",
          "timestamp": "2021-07-12T08:05:22.454990956Z",
          "signature": "pf/usYgYQrfmcIqul9YxUbu8Ruajit0L2HXU0xz8ayg1SBZqfJcu3I7tO77iAw76S5PMZHBvFO+cig3
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "biAZveTotPyzzCPfYXiWCcKDKr8=",
          "timestamp": "2021-07-12T08:05:22.302760757Z",
          "signature": "dSf7bobDcdvvJdKKIK9CbIGGiASyBDhis1fMEivpXmozjvAAMRcBtxoekLMzNfbu7ZhXZklFH7DzoVs
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "AFmBCh4x3+WV/oQfZTcQ4a97ms0=",
          "timestamp": "2021-07-12T08:05:22.303799803Z",
          "signature": "Nrwb1ztSW27fEqzfgJ6G4lUqecILKRGowezklOmX4l4OZNpJfihHuh7AHoPhj4SLkeNBud1vlW96zmr
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "GXmy0WdxkkwltyFvLJbIa9v5Kt8=",
          "timestamp": "2021-07-12T08:05:22.409681572Z",
          "signature": "HF5CZDNMKih/fe35Cwpqa5EIw2s1oTT87iVMRL5Fo51keUOB+Ly21SguFb6SeXusJ3W6KF+4YjtMM7O
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "HW6680aDSSb7WNZU6GKkL1PYgiA=",
          "timestamp": "2021-07-12T08:05:22.394570566Z",
          "signature": "t1brNDe7aCu3Q13ePI8Lsp6JwI8nTjKBvnRBOtdvbcUcSIy4WbadIbnXou8bXdoKnXeEZCqf7zsULZt
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "Mi9CIpvJaLKLAUv/2dg6nGeAZGw=",
          "timestamp": "2021-07-12T08:05:22.304367649Z",
          "signature": "w9nyw1GsE4EVViKnLuTHGHdcCUBA7EoetN6ztVHdqs4moh82yrhLxtJjoRgQy54G8SBZg7BS8dJuwpK
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "TO8fbCX9iAPgPb7NP60N5c+1XCw=",
          "timestamp": "2021-07-12T08:05:22.478276190Z",
          "signature": "UJ5YrJDIR55lEm9iwTtaodaKz38aTJC9ba4NoULsh0cZKWRMdXSsEo62gQd4XWZesRU9VbpwETOkr7a
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "UX8oPBDTFGmRZsWIhGKc2EUE86Q=",
          "timestamp": "2021-07-12T08:05:22.412614197Z",
          "signature": "KfxPKErTsEtpAKtSRE9XlfekK0XSUEdka5B+/i7OXGt0P2d7T7rztBKqYSASuPaAvCicTgEvWIhyNaC
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "aAyok6Qr/EGfdJrBDiG9Yycw4VA=",
          "timestamp": "2021-07-12T08:05:22.502941178Z",
          "signature": "Z9S74/SM6J5Hvqce4i7KlrXZwh6kG39pU5jUs8NVlFZvye+s4lnqR0vxQafLi8apyk2UdNW4KWLDUXA
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "adhSwssyskWcBBWKzyItym8rKBU=",
          "timestamp": "2021-07-12T08:05:22.330196188Z",
          "signature": "qah8/op1dvTB8CaPW/TPy89nJkUw7NEn7ZkgkQ/r6eA8bwT/+QJz2qHlUNxIZ/7KE6LsqDiy+Ss4xsZ
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "tWg//Wbq1Rj5b0nPLi1yZE794FI=",
          "timestamp": "2021-07-12T08:05:22.386235207Z",
          "signature": "SrQW8WDsO0USkDZmRYNDzCXLj/y4AEWCLJ3JzU12LOIFBs+uIRZ5dHDx2jsBytYOmDdXsh1udKiRrkB
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "056lMQcoc6aX+fZODSBBYOPz5DY=",
          "timestamp": "2021-07-12T08:05:22.402139725Z",
          "signature": "RF65UL0Hyd/0Z7aAPXqpOhcc0TaqZ614BN7IXOMNjfANXycZYgiNZvbppumWwsgI9AeAv/+6xPKXAG/
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "2MqI5T0aGLK0h3fNWAcQEKLplio=",
          "timestamp": "2021-07-12T08:05:22.416599751Z",
          "signature": "stsHa0JHU1ipj8ahGdNNxm/FrYGyAGYh16ry8tsC0kx7ymBhj3uvmu9CQpBMdb7bTQNOuOTLRKLaln+
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "299xvCm2ZUwTV/Tcty4zcOBT1AU=",
          "timestamp": "2021-07-12T08:05:22.356250712Z",
          "signature": "VX+TAitw0TeG25vdAyRTTqFD9ZQCMLyCFDfhzA8MmNQDAXLDF79I4sQGkQFnuAOYvAANKZtGdXdx+1n
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "+nva6pxhpfHNdpnVtUg9kpTft8s=",
          "timestamp": "2021-07-12T08:05:22.400389951Z",
          "signature": "haUPT+DIwnXFU+zxwZukakxLU/Vzz3tpwel1MA9tgN4YNLsK1DOKJjjs4U5Zj4qvYXLH5tvGUXz2XhN
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "lZEVu5kiC4YXL2FxwaAMRv282MQ=",
          "timestamp": "2021-07-12T08:05:22.393823192Z",
          "signature": "QuhuFlfGNzHzoAijw/PrFRUR7TmaaphyI/ybpFTpGBJxNGcmUZ+YtNJtzJwRhUqkhV/i5Ngv3Xx/uQX
        },
        {
          "blockIdFlag": "BLOCK_ID_FLAG_COMMIT",
          "validatorAddress": "a0hzz+h4moaQrE64yXSjzRWM+p4=",
          "timestamp": "2021-07-12T08:05:22.389074186Z",
          "signature": "8XggPj2IR9uWseeZKtejNwrkBUWcQDdThz4CcGH+X1l2RuFlZq/xIP2v0j2tw0Et3aFzOyMdMHl9N1/
        }
      ]
    }
  }
}
```

get_account(address)

This function returns the account details of the specified address.

Parameter

- **address** `<str>`

Return

- Optional[auth_type.BaseAccount]

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
address = "band1ee656yzw6y9swqayu9v0kgu5pua2kgjq3hd6g3"

c = Client(grpc_url)
addr = c.get_account(address)
print(MessageToJson(addr))
```

Result

```
{
  "address": "band1ee656yzw6y9swqayu9v0kgu5pua2kgjq3hd6g3",
  "pubKey": {
    "@type": "/cosmos.crypto.secp256k1.PubKey",
    "key": "AsBzzfeupPh2IM9xJ7SnhtIl7kVGX2QoY3Ro2DRKsmIF"
  },
  "accountNumber": "171",
  "sequence": "10"
}
```

get_request_id_by_tx_hash(tx_hash)

This function returns request ID of the given transaction hash.

Parameter

- **tx_hash** `<bytes>`: Transaction hash

Return

- `List[int]`

Exception

Type	Description
NotFoundError	Request Id is not found

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
tx_hash = "DCC09AD0087DFB30AD552DAFA6C52FE9676F157B24812FF4B9994B97CAC914AC"

c = Client(grpc_url)
print(c.get_request_id_by_tx_hash(tx_hash))
```

Result

```
[37625, 37626, 37627, 37628, 37629]
```

get_chain_id

This function returns a chain ID.

Return

- `<str>`

Example

```
from pyband import Client

grpc_url = "<GRPC>" # without https://

c = Client(grpc_url)
print(c.get_chain_id())
```

Result

```
band-laozi-testnet2
```

get_reference_data(pairs, min_count, ask_count)

This function returns the rates of the given cryptocurrency pairs.

Parameter

- **pairs** `List<str>` List of cryptocurrency pairs.
- **min_count** `<int>`: Minimum number of validators necessary for the request to proceed to the execution phase.
- **ask_count** `<int>`: Number of validators that are requested to response to the corresponding request.

Return

- List[ReferencePrice]

Exception

Type	Description
EmptyMsgError	Pairs are required

Example

```
from pyband import Client

grpc_url = "<GRPC>" # without https://
client = Client(grpc_url)

min_count = 3
ask_count = 4

pairs = ["BTC/USD", "ETH/USD"]

print(client.get_reference_data(pairs, min_count, ask_count))
```

Result

```
[
  ReferencePrice(
    (pair = "BTC/USD"),
    (rate = 33373.93),
    (updated_at = ReferencePriceUpdated(
      (base = 1625715297),
      (quote = 1625715749)
    ))
  ),
  ReferencePrice(
    (pair = "ETH/USD"),
    (rate = 2261.97),
    (updated_at = ReferencePriceUpdated(
      (base = 1625715297),
      (quote = 1625715749)
    ))
  )
]
```

get_latest_request(oid, calldata, min_count, ask_count)

This function returns the latest request.

Parameter

- **oid** `<int>`: Oracle script ID
- **calldata** `<bytes>`: Calldata of a request.
- **min_count** `<int>`: Minimum number of validators necessary for the request to proceed to the execution phase.
- **ask_count** `<int>`: Number of validators that are requested to response to the corresponding request.

Return

- oracle_query.QueryRequestSearchResponse

Example

```
from pyband import Client
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
c = Client(grpc_url)

oid = 43
calldata = "00000002000000044254430000000445544800000003b9aca00"
min_count = 3
ask_count = 4

latest_req = c.get_latest_request(oid, calldata, min_count, ask_count)
print(MessageToJson(latest_req))
```

Result

```
{
  "request": {
    "request": {
      "oracleScriptId": "43",
      "calldata": "AAAAAgAAAARCVENCAAAABEJFVEgAAAAAO5rKAA==",
      "requestedValidators": [
        "bandvaloper1p46uhvdk8vr829v747v85hst3mur2dzlhfemmz",
        "bandvaloper17n5rmujk78nkgss7tjecg4nfzn6geg4cqtyg3u",
        "bandvaloper1e9sa38742tzhmandc4gkqve9zy8zc0yremaa3j",
        "bandvaloper1zl5925n5u24njn9axpygz8lhjl5a8v4cpkzx5g",
        "bandvaloper1ldtwjzsplhxzhrg3k5hhr8v0qterv05vpdxp9f",
        "bandvaloper19eu9g3gka6rxlevkjlvjq7s6c498tejnwxjwxx"
      ],
      "minCount": "3",
      "requestHeight": "603449",
      "requestTime": "1626077260",
      "clientId": "linear",
      "rawRequests": [
        {
          "dataSourceId": "74",
          "calldata": "aHR0cHM6Ly91cy1ycGMuYmFuZGNoYWluLm9yZy9vcmFjbGUvcmVxdWVzdF9wcmljZXMgQlRDIEVUSA=='
        },
        {
          "externalId": "1",
          "dataSourceId": "74",
          "calldata": "aHR0cHM6Ly9ldS1ycGMuYmFuZGNoYWluLm9yZy9vcmFjbGUvcmVxdWVzdF9wcmljZXMgQlRDIEVUSA=='
        },
        {
          "externalId": "2",
          "dataSourceId": "74",
          "calldata": "aHR0cHM6Ly9hc2lhLXJwYy5iYW5kY2hhaW4ub3JnL29yYWNsZS9yZXF1ZXN0X3ByaWNlcyBCVEMgRVRI'
        },
        {
          "externalId": "3",
          "dataSourceId": "74",
          "calldata": "aHR0cHM6Ly9hdXMtcnBjLmJhbmRjaGFpbi5vcmcvb3JhY2xlL3JlcXVlc3RfcHJpY2VzIEJUQyBFVEg='
        }
      ],
      "executeGas": "1000000"
    },
    "reports": [
      {
        "validator": "bandvaloper1p46uhvdk8vr829v747v85hst3mur2dzlhfemmz",
        "inBeforeResolve": true,
        "rawReports": [
          { "externalId": "1", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "2", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "3", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" }
        ]
      },
      {
        "validator": "bandvaloper1zl5925n5u24njn9axpygz8lhjl5a8v4cpkzx5g",
        "inBeforeResolve": true,
        "rawReports": [
          { "externalId": "2", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "3", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "1", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" }
        ]
      },
      {
        "validator": "bandvaloper19eu9g3gka6rxlevkjlvjq7s6c498tejnwxjwxx",
        "inBeforeResolve": true,
        "rawReports": [
          { "externalId": "3", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "2", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "1", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" }
        ]
      },
      {
        "validator": "bandvaloper1e9sa38742tzhmandc4gkqve9zy8zc0yremaa3j",
        "inBeforeResolve": true,
        "rawReports": [
          { "externalId": "3", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "2", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "1", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" }
        ]
      },
      {
        "validator": "bandvaloper17n5rmujk78nkgss7tjecg4nfzn6geg4cqtyg3u",
        "inBeforeResolve": true,
        "rawReports": [
          { "externalId": "2", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "3", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "1", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" }
        ]
      },
      {
        "validator": "bandvaloper1ldtwjzsplhxzhrg3k5hhr8v0qterv05vpdxp9f",
        "inBeforeResolve": true,
        "rawReports": [
          { "externalId": "3", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "1", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "externalId": "2", "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" },
          { "data": "MzQyNzkuOTYgMjE0NC43MTk5OTk5OTk5OTkK" }
        ]
      }
    ],
    "result": {
      "clientId": "linear",
      "oracleScriptId": "43",
      "calldata": "AAAAAgAAAARCVENCAAAABEJFVEgAAAAAO5rKAA==",
      "askCount": "6",
      "minCount": "3",
      "requestId": "306920",
      "ansCount": "6",
      "requestTime": "1626077260",
      "resolveTime": "1626077266",
      "resolveStatus": "RESOLVE_STATUS_SUCCESS",
      "result": "AAAAAgAAHy1s1rYAAAAB81tGE/4="
    }
  }
}
```

send_tx_sync_mode(tx_bytes)

This function sends a transaction in sync mode, that is, send and wait until a transaction has passed CheckTx phase.

Parameter

- tx_bytes `<bytes>`: Transaction raw bytes that is already signed.

Return

- `abci_type.TxResponse`

Example

```
import os

from pyband.client import Client
from pyband.transaction import Transaction
from pyband.wallet import PrivateKey

from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgRequestData
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
c = Client(grpc_url)

MNEMONIC = os.getenv("MNEMONIC")
private_key = PrivateKey.from_mnemonic(MNEMONIC)
public_key = private_key.to_public_key()
sender_addr = public_key.to_address()
sender = sender_addr.to_acc_bech32()

request_msg = MsgRequestData(
    oracle_script_id=37,
    calldata=bytes.fromhex("0000000200000003425443000000034554480000000000000064"),
    ask_count=4,
    min_count=3,
    client_id="BandProtocol",
    fee_limit=[Coin(amount="100", denom="uband")],
    prepare_gas=50000,
    execute_gas=200000,
    sender=sender,
)

account = c.get_account(sender)
account_num = account.account_number
sequence = account.sequence

fee = [Coin(amount="0", denom="uband")]
chain_id = c.get_chain_id()

txn = (
    Transaction()
    .with_messages(request_msg)
    .with_sequence(sequence)
    .with_account_num(account_num)
    .with_chain_id(chain_id)
    .with_gas(2000000)
    .with_fee(fee)
    .with_memo("")
)

sign_doc = txn.get_sign_doc(public_key)
signature = private_key.sign(sign_doc.SerializeToString())
tx_raw_bytes = txn.get_tx_data(signature, public_key)

tx_sync = c.send_tx_sync_mode(tx_raw_bytes)
print(MessageToJson(tx_sync))
```

Result

```
{
  "txhash": "FEE8A58F7A68326A50B974C13721B55A6ABA6A1761A2D466A9940FF393F02C9E",
  "rawLog": "[]"
}
```

send_tx_async_mode(tx_bytes)

This function sends a transaction in async mode, that is, send and return immediately without waiting for the transaction process.

Parameter

- tx_bytes `<bytes>`: Transaction raw bytes that is already signed.

Return

- `abci_type.TxResponse`

Example

```
import os

from pyband.client import Client
from pyband.transaction import Transaction
from pyband.wallet import PrivateKey

from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgRequestData
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
c = Client(grpc_url)

MNEMONIC = os.getenv("MNEMONIC")
private_key = PrivateKey.from_mnemonic(MNEMONIC)
public_key = private_key.to_public_key()
sender_addr = public_key.to_address()
sender = sender_addr.to_acc_bech32()

request_msg = MsgRequestData(
    oracle_script_id=37,
    calldata=bytes.fromhex("0000000200000003425443000000034554480000000000000064"),
    ask_count=4,
    min_count=3,
    client_id="BandProtocol",
    fee_limit=[Coin(amount="100", denom="uband")],
    prepare_gas=50000,
    execute_gas=200000,
    sender=sender,
)
account = c.get_account(sender)
account_num = account.account_number
sequence = account.sequence

fee = [Coin(amount="0", denom="uband")]
chain_id = c.get_chain_id()

txn = (
    Transaction()
    .with_messages(request_msg)
    .with_sequence(sequence)
    .with_account_num(account_num)
    .with_chain_id(chain_id)
    .with_gas(2000000)
    .with_fee(fee)
    .with_memo("")
)

sign_doc = txn.get_sign_doc(public_key)
signature = private_key.sign(sign_doc.SerializeToString())
tx_raw_bytes = txn.get_tx_data(signature, public_key)

tx_async = c.send_tx_async_mode(tx_raw_bytes)
print(MessageToJson(tx_async))
```

Result

```
{ "txhash": "C685F799E4D870353364155602C14520416FC274293DFC9EFC3575357F9A8893" }
```

send_tx_block_mode(tx_bytes)

This function sends a transaction in block mode, that is, send and wait until the transaction has been committed to a block.

Parameter

- tx_bytes `<bytes>`: Transaction raw bytes that is already signed.

Return

- `abci_type.TxResponse`

Example

```
import os

from pyband.client import Client
from pyband.transaction import Transaction
from pyband.wallet import PrivateKey

from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgRequestData
from google.protobuf.json_format import MessageToJson

grpc_url = "<GRPC>" # without https://
c = Client(grpc_url)

MNEMONIC = os.getenv("MNEMONIC")
private_key = PrivateKey.from_mnemonic(MNEMONIC)
public_key = private_key.to_public_key()
sender_addr = public_key.to_address()
sender = sender_addr.to_acc_bech32()

request_msg = MsgRequestData(
    oracle_script_id=37,
    calldata=bytes.fromhex("0000000200000003425443000000034554480000000000000064"),
    ask_count=4,
    min_count=3,
    client_id="BandProtocol",
    fee_limit=[Coin(amount="100", denom="uband")],
    prepare_gas=50000,
    execute_gas=200000,
    sender=sender,
)

account = c.get_account(sender)
account_num = account.account_number
sequence = account.sequence

fee = [Coin(amount="0", denom="uband")]
chain_id = c.get_chain_id()

txn = (
    Transaction()
    .with_messages(request_msg)
    .with_sequence(sequence)
    .with_account_num(account_num)
    .with_chain_id(chain_id)
    .with_gas(2000000)
    .with_fee(fee)
    .with_memo("")
)

sign_doc = txn.get_sign_doc(public_key)
signature = private_key.sign(sign_doc.SerializeToString())
tx_raw_bytes = txn.get_tx_data(signature, public_key)

tx_block = c.send_tx_block_mode(tx_raw_bytes)
print(MessageToJson(tx_block))
```

Result

```
{
  "height": "603561",
  "txhash": "A50970334A74461CF045D962EEA1230B18AAAC2CEE2E96C3C348672100D46A93",
  "data": "0A090A0772657175657374",
  "rawLog": "[{\"events\":[{\"type\":\"message\",\"attributes\":[{\"key\":\"action\",\"value\":\"request
  "logs": [
    {
      "events": [
        {
          "type": "message",
          "attributes": [{ "key": "action", "value": "request" }]
        },
        {
          "type": "raw_request",
          "attributes": [
            { "key": "data_source_id", "value": "61" },
            {
              "key": "data_source_hash",
              "value": "07be7bd61667327aae10b7a13a542c7dfba31b8f4c52b0b60bf9c7b11b1a72ef"
            },
            { "key": "external_id", "value": "6" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "57" },
            {
              "key": "data_source_hash",
              "value": "61b369daa5c0918020a52165f6c7662d5b9c1eee915025cb3d2b9947a26e48c7"
            },
            { "key": "external_id", "value": "0" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "62" },
            {
              "key": "data_source_hash",
              "value": "107048da9dbf7960c79fb20e0585e080bb9be07d42a1ce09c5479bbada8d0289"
            },
            { "key": "external_id", "value": "3" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "5" },
            { "key": "calldata", "value": "huobipro BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "59" },
            {
              "key": "data_source_hash",
              "value": "5c011454981c473af3bf6ef93c76b36bfb6cc0ce5310a70a1ba569de3fc0c15d"
            },
            { "key": "external_id", "value": "2" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "4" },
            { "key": "calldata", "value": "binance BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "9" },
            { "key": "calldata", "value": "bittrex BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "7" },
            { "key": "calldata", "value": "kraken BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "60" },
            {
              "key": "data_source_hash",
              "value": "2e588de76a58338125022bc42b460072300aebbcc4acaf55f91755c1c1799bac"
            },
            { "key": "external_id", "value": "8" },
            { "key": "calldata", "value": "bitfinex BTC ETH" },
            { "key": "fee" },
            { "key": "data_source_id", "value": "58" },
            {
              "key": "data_source_hash",
              "value": "7e6759fade717a06fb643392bfde837bfc3437da2ded54feed706e6cd35de461"
            },
            { "key": "external_id", "value": "1" },
            { "key": "calldata", "value": "BTC ETH" },
            { "key": "fee" }
          ]
        },
        {
          "type": "request",
          "attributes": [
            { "key": "id", "value": "307081" },
            { "key": "client_id", "value": "BandProtocol" },
            { "key": "oracle_script_id", "value": "37" },
            {
              "key": "calldata",
              "value": "0000000200000003425443000000034554480000000000000064"
            },
            { "key": "ask_count", "value": "4" },
            { "key": "min_count", "value": "3" },
            { "key": "gas_used", "value": "111048" },
            { "key": "total_fees" },
            {
              "key": "validator",
              "value": "bandvaloper1l2hchtyawk9tk43zzjrzr2lcd0zyxngcjdsshe"
            },
            {
              "key": "validator",
              "value": "bandvaloper17n5rmujk78nkgss7tjecg4nfzn6geg4cqtyg3u"
            },
            {
              "key": "validator",
              "value": "bandvaloper1e9sa38742tzhmandc4gkqve9zy8zc0yremaa3j"
            },
            {
              "key": "validator",
              "value": "bandvaloper1lm2puy995yt8dh53cnazk3ge3m27t7cay4ndaq"
            }
          ]
        }
      ]
    }
  ],
  "gasWanted": "2000000",
  "gasUsed": "566496"
}
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
Getting Started
Client Module
Data Module
OBI Module
Transaction Module
Wallet Module
BandChain.js
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Data Module

This module is used to construct the data in term of dataclass to make sure that all the inputs and outputs follow this data schema.

Although there are Protobuf classes, some types do not exists. Therefore, additional types of data will be declared here.

> **NOTE**
> Note that `base` and `quote` is the first and the second price symbols respectively e.g. BTC/USD price rate has a base of BTC and a quote of USD.

The dataclasses declaration are as follows:

```
from dataclasses import dataclass


@dataclass
class ReferencePriceUpdated(object):
    base: int
    quote: int


@dataclass
class ReferencePrice(object):
    pair: str
    rate: float
    updated_at: ReferencePriceUpdated
```

ReferencePriceUpdated

Attribute	Type	Description
base	integer	Base resolve time
quote	integer	Quote resolve time

ReferencePrice

Attribute	Type	Description
pair	string	The token pair e.g. "BTC/USDT"
rate	float	Price rate
updated_at	ReferencePriceUpdated	Price updated time

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
Getting Started
Client Module
Data Module
OBI Module
Transaction Module
Wallet Module
BandChain.js
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

OBI Module

Oracle Binary Encoding (OBI) is the standard way to serialized and deserialize binary data in the BandChain ecosystem. This module provides the functionality to serialize data. More details

Constructor

- **schema** `<str>`: Input and output schema.

Example

```
from pyband import PyObi

obi = PyObi("{symbol: string,px: u64,in: {a: u8,b: u8}, tb:bool} / string")
```

---

encode_input(value)

This function encodes the input value by using input schema.

Parameter

- **value** `<Any>`: Value to be encoded.

Return

- `<bytes>`: An encoded value

Example

```
from pyband.obi import PyObi

obi = PyObi("{symbol: string,px: u64,in: {a: u8,b: u8}, tb:bool} / string")
test_input = {"symbol": "BTC", "px": 9000, "in": {"a": 1, "b": 2}, "tb": False}

print(obi.encode_input(test_input).hex())
```

Result

```
00000003425443000000000002328010200
```

---

encode_output(value)

This function encodes the output value by using output schema.

Parameter

- **value** `<Any>`: Value to be encoded.

Return

- `<bytes>`: An encoded value

Example

```
from pyband.obi import PyObi

obi = PyObi("{symbol: string,px: u64,in: {a: u8,b: u8}, tb:bool} / string")
test_output = "test"
print(obi.encode_output(test_output).hex())
```

Result

```
0000000474657374
```

---

decode_input(value)

This function decode the input value by using input schema

Parameter

- **value** `<bytes>`: Value to be decoded.

Return

- `<Any>`: A decoded value

Example

```
from pyband.obi import PyObi

obi =PyObi("{symbol: string,px: u64,in: {a: u8,b: u8}, tb:bool} / string")
print(obi.decode_input(bytearray.fromhex("00000003425443000000000002328010200")))
```

Result

```
{"symbol": "BTC", "px": 9000, "in": {"a": 1, "b": 2}, "tb": False}
```

---

decode_output(value)

This function decode the output value by using output schema

Parameter

- **value** `<bytes>`: Value to be decoded.

Return

- `<Any>`: A decoded value

Example

```
from pyband.obi import PyObi

obi = PyObi("{symbol: string,px: u64,in: {a: u8,b: u8}, tb:bool} / string")
print(obi.decode_output(bytearray.fromhex("0000000474657374")))
```

Result

```
test
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
Getting Started
Client Module
Data Module
OBI Module
Transaction Module
Wallet Module
BandChain.js
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Transaction Module

This module provides the preparatory component that is required for sending a transaction.

with_message(msgs)

This function add one or more messages to `<Transaction>`

The message can be any message as listed in Oracle Modules or Cosmos Based Messages. Please note that our message should be imported from the generated protobuf files.

Parameter

- **msgs** `<google-protobuf.message.Message>`: Messages to be included into the transaction.

Return

- `<Transaction>`

---

with_sender(client, sender)

This function set `account_num` and `sequence` from `<Client>` with the address from `sender`. `<Transaction>` must have at least 1 message added before calling `with_sender()`

Parameter

- **client** `<Client>`: Client used to set `account_num` and `sequence` by calling `get_address()`.
- **sender** `<str>`: Address of the sender.

Return

- `<Transaction>`

Exception

Type	Description
EmptyMsgError	Message is empty, please use with_messages at least 1 message
NotFoundError	Account doesn't exist

---

with_account_num(account_num)

This function sets the account number in `<Transaction>`.

Parameter

- **account_num** `<int>`

Return

- `<Transaction>`

---

with_sequence(sequence)

This function sets the sequence number in `<Transaction>`.

Parameter

- **sequence** `<int>`

Return

- `<Transaction>`

---

with_chain_id(chain_id)

This function sets the chain ID in `<Transaction>`.

Parameter

- **chain_id** `<str>`

Return

- `<Transaction>`

---

with_fee(fee)

This function sets the fee by using the given fee and gas limit `<Transaction>`.

Parameter

- **fee** `<List[Coin]>`

Return

- `<Transaction>`

---

with_gas(gas)

This function sets the gas limit in `<Transaction>`.

Parameter

- **gas** `<int>`

Return

- `<Transaction>`

---

with_memo(memo)

This function sets the memo in `<Transaction>`.

Parameter

- **memo** `<str>`: Maximum length of memo is 256.

Return

- `<Transaction>`

Exception

Type	Description
ValueTooLargeError	Memo is too large

---

get_sign_doc(public_key)

This function returns a sign data from `<Transaction>`.

Parameter

- **public_key** `<PublicKey>`, default = None: Public key.

Return

- `<cosmos_tx_type.SignDoc>`

Exception

Type	Description
EmptyMsgError	message is empty
UndefinedError	account_num should be defined
UndefinedError	sequence should be defined
UndefinedError	chain_id should be defined

---

get_tx_data(signature, public_key)

This function returns a transaction that need to be sent.

Parameter

- **signature** `<bytes>`: Signature from sign from get_sign_doc
- **public_key** `<PublicKey>`, default = None: Public key

Return

- `<bytes>`

Example use case

Note: Get the `<GRPC>` here

```
import os

from pyband.client import Client
from pyband.transaction import Transaction
from pyband.wallet import PrivateKey

from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgRequestData

grpc_url = "<GRPC>"
c = Client(grpc_url)

MNEMONIC = os.getenv("MNEMONIC")
private_key = PrivateKey.from_mnemonic(MNEMONIC)
public_key = private_key.to_public_key()
sender_addr = public_key.to_address()
sender = sender_addr.to_acc_bech32()

request_msg = MsgRequestData(
    oracle_script_id=37,
    calldata=bytes.fromhex("0000000200000003425443000000034554480000000000000064"),
    ask_count=4,
    min_count=3,
    client_id="BandProtocol",
    fee_limit=[Coin(amount="100", denom="uband")],
    prepare_gas=50000,
    execute_gas=200000,
    sender=sender,
)

account = c.get_account(sender)
account_num = account.account_number
sequence = account.sequence

fee = [Coin(amount="0", denom="uband")]
chain_id = c.get_chain_id()

txn = (
    Transaction()
    .with_messages(request_msg)
    .with_sequence(sequence)
    .with_account_num(account_num)
    .with_chain_id(chain_id)
    .with_gas(2000000)
    .with_fee(fee)
    .with_memo("")
)

sign_doc = txn.get_sign_doc(public_key)
signature = private_key.sign(sign_doc.SerializeToString())
tx_raw_bytes = txn.get_tx_data(signature, public_key)

print(tx_raw_bytes.hex())
```

Result

```
0a93010a90010a192f6f7261636c652e76312e4d73675265717565737444461746112730825121a0000000200000003425443000
```

Wallet Module

This module provides the functionality to control the account.

Private Key

Class for wrapping `SigningKey` that is used for signature creation and public key derivation.

generate(path)

This function generates new private key with random mnemonic phrase.

> INFO
> If path is not given, default to Band's HD prefix 494 and all other indexes being zeroes.

Parameter

- **path** <str> optional: HD path that follows the BIP32 standard.

Return

- Tuple[str, <PrivateKey>]: Tuple of mnemonic phrase and `PrivateKey` instance.

Example

```
from pyband.wallet import PrivateKey

mnemonic, priv = PrivateKey.generate(path="m/44'/494'/0'/0/3")
```

from_mnemonic(word, path)

This function creates a `PrivateKey` instance from a given mnemonic phrase and a HD derivation path.

> INFO
> If path is not given, default to Band's HD prefix 494 and all other indexes being zeroes.

Parameter

- **words** <str>: Mnemonic phrase for recovers a private key.
- **path** <str> (optional): HD path that follows the BIP32 standard.

Return

- <PrivateKey>: `PrivateKey` object.

Example

```
from pyband.wallet import PrivateKey

priv = PrivateKey.from_mnemonic("test mnemonic")
```

from_hex(priv)

This function creates a `PrivateKey` instance from a given hex that represents a signing key.

Parameter

- **priv** <str>: Hex representation of signing key.

Return

- <PrivateKey>:`PrivateKey` object.

Example

```
from pyband.wallet import PrivateKey

priv = PrivateKey.from_hex("2442b724db7189468f16accc0fc505f0609817eb129e13702e696d8b84609ea9")
```

to_hex

This function returns a hex representation of a signing key.

Return

- <str>: A hex representation of signing key.

Example

```
from pyband.wallet import PrivateKey

priv = PrivateKey.from_mnemonic("test mnemonic")
print(priv.to_hex())
```

Result

```
2cb2e2d3582cebf0664d9cda0b89c5d478ae12fac19a6f4ed9c10a7406a19615
```

to_public_key

This function returns a `PublicKey` that is associated with this private key.

Return

- <PublicKey>: A `PublicKey` that can be used to verify the signatures made with this `PrivateKey`.

Example

```
from pyband.wallet import PrivateKey

priv = PrivateKey.from_mnemonic("test mnemonic")
print(priv.to_public_key().to_hex())
```

Result

```
02b2b0d35cb1c6d3923813c64e46a82d29e12d03288f438b9d3cf232d9a22bcb83
```

sign(msg)

This function returns a signature of the associated private key. The message is signed by using the edcsa sign_deterministic function.

Parameter

- **msg** <bytes>: Message that will be hashed and signed.

Return

- <bytes>: A signature of this private key over the given message.

Example

```
from pyband.wallet import PrivateKey

priv = PrivateKey.from_mnemonic("test mnemonic")
print(priv.sign(b"test message").hex())
```

Result

```
4bbc9a7ea54b47b11c67a4074e8d9bca068cb64c788f67342c4033b1b6f0553e1bc63cdf9bc2fb6e89c1e965c1e0f0712a51c25
```

Public Key

Class for wrapping `VerifyKey` that is used for signature verification.

from_acc_bech32(bech)

This function creates a `PublicKey` instance from bech32-encoded with account public key prefix `bandpub`.

Parameter

- **bech** <str>: A bech32-encoded with account public key prefix.

Return

- `PublicKey`: A `PublicKey` instance

Exception

Type	Description
ValueError	Invalid bech32 prefix

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_acc_bech32("bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05
```

from_val_bech32(bech)

This function creates a `PublicKey` instance from bech32-encoded with validator public key prefix `bandvaloperpub`.

Parameter

- **bech** <str>: A bech32-encoded with validator public key prefix.

Return

- `PublicKey`: A `PublicKey` instance

Exception

Type	Description
ValueError	Invalid bech32 prefix

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_val_bech32("bandvaloperpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlua
```

from_cons_bech32(bech)

This function creates a `PublicKey` instance from a bech32-encoded with validator consensus public key prefix `bandvalconspub`.

Parameter

- **bech** <str>: A bech32-encoded with validator consensus public key prefix.

Return

- `PublicKey`: A `PublicKey` instance

Exception

Type	Description
ValueError	Invalid bech32 prefix

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_cons_bech32("bandvalconspub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlua
```

to_hex

This function returns a hex representation of the verified key.

Return

- <str>: A hex representationof the verified key.

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_acc_bech32("bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05
print(public_key.to_hex())
```

Result

```
0351e98e1be097250f2ff4188c0aace0a716e69a992cd77f9dfe436b3e8b34280d
```

to_public_key_proto()

This function returns a public key in type protobuf.

Return

- <PubKeyProto>: A public key of type protobuf (SECP256k1).

to_acc_bech32

This function returns bech32-encoded with account public key prefix `bandpub`.

Return

- <str>: A bech32-encoded with account public key prefix.

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_acc_bech32("bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05
print(public_key.to_acc_bech32())
```

Result

```
bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v
```

to_val_bech32

This function returns a bech32-encoded with validator public key prefix `bandvaloperpub`.

Return

- <str>: A bech32-encoded with validator public key prefix.

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_acc_bech32("bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05
print(public_key.to_val_bech32())
```

Result

```
bandvaloperpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q69gsm29
```

to_cons_bech32

This function returns a bech32-encoded with validator consensus public key prefix `bandvalconspub`.

Return

- <str>: A bech32-encoded with validator consensus public key prefix.

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_acc_bech32("bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05
print(public_key.to_cons_bech32())
```

Result

```
bandvalconspub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6r8ytws
```

to_address

This function returns an `Address` instance from this public key.

Return

- <Address>: An `Address` instance

Example

```
from pyband.wallet import PublicKey

public_key = PublicKey.from_acc_bech32("bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05
print(public_key.to_address().to_hex())
```

Result

```
8bb66ae5bb7e5ce1035557cf1c77430b987683d2
```

verify(msg, sig)

This function is used to verify a signature made from the given message.

Parameter

- **msg** <bytes>: A data signed by the signature, will be hashed using sha256 function.
- **sig** <bytes: An encoded signature

Return

- <bool>: True, if the verification success.

Exception

Type	Description
BadSignatureError	If the signature is invalid or malformed

Example

```
from pyband.wallet import PrivateKey

priv = PrivateKey.from_mnemonic("test mnemonic")
public_key = priv.to_public_key()
msg = b"test message"
sig = priv.sign(msg)
print(public_key.verify(msg, sig))
```

Result

```
True
```

Address

Class for wrapping `Address`.

from_acc_bech32(bech)

This function creates an `Address` instance from the given bech32-encoded with account prefix `band`.

Parameter

- **bech**<str>: A bech32-encoded with account prefix.

Return

- <Address>: An `Address` instance

Exception

Type	Description
ValueError	Invalid bech32 prefix

Example

```
from pyband.wallet import Address

address = Address.from_acc_bech32("band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c")
```

from_val_bech32(bech)

This function creates an Address instance from the given bech32-encoded with validator prefix `bandvaloper`.

Parameter

- **bech** <str>: A bech32-encoded with validator prefix.

Return

- <Address>: An `Address` instance

Exception

Type	Description
ValueError	Invalid bech32 prefix

Example

```
from pyband.wallet import Address

address = Address.from_val_bech32("bandvaloper13eznuehmqzd3r84fkxu8wklxl22r2qfm8f05zn")
```

from_cons_bech32(bech)

This function creates an Address instance from the given bech32-encoded with validator consensus prefix `bandvalcons`.

Parameter

- **bech** <str>: A bech32-encoded with validator consensus prefix.

Return

- <Address>: An `Address` instance

Exception

Type	Description
ValueError	Invalid bech32 prefix

Example

```
from pyband.wallet import Address

address = Address.from_cons_bech32("bandvalcons13eznuehmqzd3r84fkxu8wklxl22r2qfmn6ugwj")
```

to_hex

This function returns a hex representation of `Address`.

Return

- <str>: A hex representation of `Address`.

Example

```
from pyband.wallet import Address

address = Address.from_acc_bech32("band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c")
print(address.to_hex())
```

Result

```
8e453e66fb009b119ea9b1b8775be6fa9435013b
```

to_acc_bech32

This function returns a bech32-encoded with account prefix band `band`.

Return

- <str>: A bech32-encoded with account prefix.

Example

```
from pyband.wallet import Address

address = Address.from_acc_bech32("band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c")
print(address.to_acc_bech32())
```

Result

```
band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c
```

to_val_bech32

This function returns a bech32-encoded with validator prefix `bandvaloper`.

Return

- <str>: A bech32-encoded with account prefix.

Example

```
from pyband.wallet import Address

address = Address.from_acc_bech32("band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c")
print(address.to_val_bech32())
```

```
bandvaloper13eznuehmqzd3r84fkxu8wklxl22r2qfm8f05zn
```

to_cons_bech32

This function returns a bech32-encoded with validator consensus prefix `bandvalcons`.

Return

- <str>: A bech32-encoded with account prefix.

Example

```
from pyband.wallet import Address

address = Address.from_acc_bech32("band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c")
print(address.to_cons_bech32())
```

Result

```
bandvalcons13eznuehmqzd3r84fkxu8wklxl22r2qfmn6ugwj
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
BandChain.js
Getting Started
Installation
Common Usage Example
Client Module
Data Module
Message Module
OBI Module
Transaction Module
Wallet Module
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Getting Started

BandChain.js is a library written in TypeScript used for interacting with BandChain. The library provides classes and methods to conveniently send transactions, query data, OBI encoding, and wallet management.

The library is implemented based on gRPC-web protocol which sends HTTP/1.5 or HTTP/2 requests to a gRPC proxy server, before serving them as HTTP/2 to gRPC server.

The library support both Node.js and browsers.

Previous
« Wallet Module

Next
Installation »

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
BandChain.js
Getting Started
Installation
Common Usage Example
Client Module
Data Module
Message Module
OBI Module
Transaction Module
Wallet Module
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Installation

System Requirements

- Node.js 12.22.0 or later
- MacOS, Windows (including WSL), and Linux are supported

Browser Support

The default build targets browsers that support both native ESM via script tags and native ESM dynamic import.

Installation

NPM

```
npm install --save @bandprotocol/bandchain.js
```

Yarn

```
yarn add @bandprotocol/bandchain.js
```

Previous
« Getting Started

Next
Common Usage Example »

Common Usage Example

Make an oracle request

This section describes methods to send a transaction of oracle request to BandChain

Step 1: Import `Client` from `@bandprotocol/bandchain.js` and creates a new instance of `grpcUrl` as a parameter and you can get `<GRPC_WEB>` endpoint from here. Then initialize the client instance. Every method in client module can now be used.

```
import { Client } from '@bandprotocol/bandchain.js'

const grpcUrl = '<GRPC_WEB>' // ex.https://laozi-testnet6.bandchain.org/grpc-web
const client = new Client(grpcUrl)
```

Step 2: Specify an account wallet for sending the transaction. This can be done by import `PrivateKey` from `Wallet` module and input a 24-words mnemonic string as a seed. For this example, we will use following mnemonic for demonstration.

```
subject economy equal whisper turn boil guard giraffe stick retreat wealth card only buddy joy leave ge
```

Here is the example on how to get a private key as an account.

```
import { Wallet } from '@bandprotocol/bandchain.js'
const { PrivateKey } = Wallet

// Step 2.1
const privkey = PrivateKey.fromMnemonic(
  'subject economy equal whisper turn boil guard giraffe stick retreat wealth card only buddy joy leave
)
```

Then, we will use the private key to generate public key and a BAND address, as shown below

```
const pubkey = privkey.toPubkey()
const sender = pubkey.toAddress().toAccBech32()
```

Step 3: As we have both an Client instance and an account wallet, we can now construct a transaction by importing `Transaction` and `MsgRequestData`.

As the transaction object requires following attributes,

- a list of messages
- account number
- sequence number
- chain ID
- fee

with following optional fields

- memo (default is empty string)

We will firstly construct a `MsgRequestData` to be included in a list of messages of the transaction. The message requires 9 fields as shown in the example below.

Within `MsgRequestData`

```
import { Obi, Message, Coin } from '@bandprotocol/bandchain.js'

const obi = new Obi('{symbols:[string],multiplier:u64}/{rates:[u64]}')
const calldata = obi.encodeInput({ symbols: ['ETH'], multiplier: 100 })

const oracleScriptId = 37
const askCount = 4
const minCount = 3
const clientId = 'from_bandchain.js'

let feeLimit = new Coin()
feeLimit.setDenom('uband')
feeLimit.setAmount('100000')

const prepareGas = 100000
const executeGas = 200000

const requestMessage = new Message.MsgRequestData(
  oracleScriptId,
  calldata,
  askCount,
  minCount,
  clientId,
  sender,
  [feeLimit],
  prepareGas,
  executeGas
)
```

After constructed `MsgRequestData`, we can get other required fields by following methods to constructs a transaction

- Account and Sequence number are automatically gathered from Transaction's `withSender` method.
- Chain ID can be gathered from Client's `getChainId` method.

```
import { Coin, Fee, Transaction } from '@bandprotocol/bandchain.js'

let feeCoin = new Coin()
feeCoin.setDenom('uband')
feeCoin.setAmount('50000')

const fee = new Fee()
fee.setAmountList([feeCoin])
fee.setGasLimit(1000000)

const chainId = await client.getChainId()
const txn = new Transaction()
txn.withMessages(requestMessage)
await txn.withSender(client, sender)
txn.withChainId(chainId)
txn.withFee(fee)
txn.withMemo('')
```

Step 4: Sign and send the transaction

Now, we had an instance of constructed transaction. In order to sign the transaction, `getSignDoc` method in `Transaction` instance can be used to get serialzed data of the transaction to be used for signing. Then, use `PrivateKey`'s `sign` to sign transaction. Finally, use `getTxData` to include signature and public key to the transaction to get a complete signed transaction.

```
const signDoc = txn.getSignDoc(pubkey)
const signature = privateKey.sign(signDoc)
```

Step 5: Send the signed transaction to Bandchain be using following method of choices

- `sendTxBlockMode` Send the transaction and wait until committed
- `sendTxSyncMode` Send the transaction and wait until `CheckTx` phase is done
- `sendTxAsyncMode` Send the transaction and immediately returned

For our example, we will use `sendTxBlockMode` to send the transaction.

The final code should now look like the code below.

```
import { Client, Wallet, Obi, Message, Coin, Transaction, Fee } from '@bandprotocol/bandchain.js'

const grpcUrl = '<GRPC_WEB>' // ex.https://laozi-testnet6.bandchain.org/grpc-web
const client = new Client(grpcUrl)

async function makeRequest() {
  // Step 1: Import a private key for signing transaction
  const { PrivateKey } = Wallet
  const mnemonic = 'test'
  const privateKey = PrivateKey.fromMnemonic(mnemonic)
  const pubkey = privateKey.toPubkey()
  const sender = pubkey.toAddress().toAccBech32()

  // Step 2.1: Prepare oracle request's properties
  const obi = new Obi('{symbols:[string],multiplier:u64}/{rates:[u64]}')
  const calldata = obi.encodeInput({ symbols: ['ETH'], multiplier: 100 })

  const oracleScriptId = 37
  const askCount = 4
  const minCount = 3
  const clientId = 'from_bandchain.js'

  let feeLimit = new Coin()
  feeLimit.setDenom('uband')
  feeLimit.setAmount('100000')

  const prepareGas = 100000
  const executeGas = 200000

  // Step 2.2: Create an oracle request message
  const requestMessage = new Message.MsgRequestData(
    oracleScriptId,
    calldata,
    askCount,
    minCount,
    clientId,
    sender,
    [feeLimit],
    prepareGas,
    executeGas
  )

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('50000')

  // Step 3.1: Construct a transaction
  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(1000000)

  const chainId = await client.getChainId()
  const txn = new Transaction()
  txn.withMessages(requestMessage)
  await txn.withSender(client, sender)
  txn.withChainId(chainId)
  txn.withFee(fee)
  txn.withMemo('')

  // Step 3.2: Sign the transaction using the private key
  const signDoc = txn.getSignDoc(pubkey)
  const signature = privateKey.sign(signDoc)

  const txRawBytes = txn.getTxData(signature, pubkey)

  // Step 4: Broadcast the transaction
  const sendTx = await client.sendTxBlockMode(txRawBytes)
  console.log(sendTx)
}

;(async () => {
  await makeRequest()
})()
```

After, we run the script above, the result should look like this. The following log contains logs, which have events related to sent request.

```
{
  "height":438884,
  "txhash":"313DBAD237E3E672B432D7F9A422EF953EA42E1029F3562C9EE2AEFB70E7D5A3",
  "codespace":"",
  "code":0,
  "data":"0A090A0772657175657374",
  "rawLog":"[{\"events\":[{\"type\":\"message\",\"attributes\":[{\"key\":\"action\",\"value\":\"request\
  "logsList":[{
    "msgIndex":0,
    "log":"",
    "eventsList": [
      {"type":"message","attributesList":[{"key":"action","value":"request"}]},
      {"type":"raw_request","attributesList":[{"key":"data_source_id","value":"61"},
      {"key":"data_source_hash","value":"07be7bd61667327aae10b7a13a542c7dfba31b8f4c52b0b60bf9c7b11b1a72
      {"key":"external_id","value":"6"},
      {"key":"calldata","value":"BTC ETH"},
      {"key":"fee","value":""},
      {"key":"data_source_id","value":"57"},
      {"key":"data_source_hash","value":"61b369daa5c0918020a52165f6c7662d5b9c1eee915025cb3d2b9947a26e48
      {"key":"external_id","value":"0"},
      {"key":"calldata","value":"BTC ETH"},
      {"key":"fee","value":""},
      {"key":"data_source_id","value":"62"},
      {"key":"data_source_hash","value":"107048da9dbf7960c79fb20e0585e080bb9be07d42a1ce09c5479bbada8d02
      {"key":"external_id","value":"3"},
      {"key":"calldata","value":"BTC ETH"},
      {"key":"fee","value":""},
      {"key":"data_source_id","value":"60"},
      ...,
      {"key":"calldata","value":"BTC ETH"},
      {"key":"fee","value":""}]},
      {"type":"request","attributesList":[{"key":"id","value":"74473"},
      {"key":"client_id","value":"from_bandchain.js"},
      {"key":"oracle_script_id","value":"37"},
      {"key":"calldata","value":"0000000200000003425443000000034554480000000000000064"},
      {"key":"ask_count","value":"4"},
      {"key":"min_count","value":"3"},
      {"key":"gas_used","value":"111048"},
      {"key":"total_fees","value":""},
      {"key":"validator","value":"bandvaloper1p46uhvdk8vr829v747v85hst3mur2dzlhfemmz"},
      {"key":"validator","value":"bandvaloper1v0u0tsptnkcdrju4qlj0hswqhnqcn47d20prfy"},
      {"key":"validator","value":"bandvaloper17n5rmujk78nkgss7tjecg4nfzn6geg4cqtyg3u"},
      {"key":"validator","value":"bandvaloper19eu9g3gka6rxlevkjlvjq7s6c498tejnwxjwxx"}
    ]}]}],
  "info":"",
  "gasWanted":1500000,
  "gasUsed":660549,
  "timestamp":""
}
```

Send BAND token

Sending BAND token has code pattern similar to the previous section, except that different type of message is used.

The message used for this section is `MsgSend` which can be used as shown below

```
const receiver = 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f'
const sendAmount = new Coin()
sendAmount.setDenom('uband')
sendAmount.setAmount('10')
const msg = new MsgSend(sender, receiver, [sendAmount])
```

Therefore, final result is as shown follow

```
import { Client, Wallet, Transaction, Message, Coin, Fee } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const client = new Client('<GRPC_WEB>') // ex.https://laozi-testnet6.bandchain.org/grpc-web

// Step 2.1 import private key based on given mnemonic string
const privkey = PrivateKey.fromMnemonic(
  'subject economy equal whisper turn boil guard giraffe stick retreat wealth card only buddy joy leave
)
// Step 2.2 prepare public key and its address
const pubkey = privkey.toPubkey()
const sender = pubkey.toAddress().toAccBech32()

const sendCoin = async () => {
  // Step 3.1 constructs MsgSend message
  const { MsgSend } = Message

  // Here we use different message type, which is MsgSend
  const receiver = 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f'
  const sendAmount = new Coin()
  sendAmount.setDenom('uband')
  sendAmount.setAmount('10')
  const msg = new MsgSend(sender, receiver, [sendAmount])
  // Step 3.2 constructs a transaction
  const account = await client.getAccount(sender)
  const chainId = 'band-laozi-testnet6'

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('1000')

  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(1000000)
  const tx = new Transaction()
    .withMessages(msg)
    .withAccountNum(account.accountNumber)
    .withSequence(account.sequence)
    .withChainId(chainId)
    .withFee(fee)

  // Step 4 sign the transaction
  const txSignData = tx.getSignDoc(pubkey)
  const signature = privkey.sign(txSignData)
  const signedTx = tx.getTxData(signature, pubkey)

  // Step 5 send the transaction
  const response = await client.sendTxBlockMode(signedTx)
  console.log(response)
}

;(async () => {
  await sendCoin()
})()
```

The response should be similar to as shown below

```
{
  "height": 443301,
  "txhash": "026760F78665C03DD4A8786304E01848A4747F0B62F7DB4B31F93C792B2D3D52",
  "codespace": "",
  "code": 0,
  "data": "0A060A0473656E64",
  "rawLog": "[{\"events\":[{\"type\":\"message\",\"attributes\":[{\"key\":\"action\",\"value\":\"send\"
  "logsList": [
    {
      "msgIndex": 0,
      "log": "",
      "eventsList": [
        {
          "type": "message",
          "attributesList": [
            { "key": "action", "value": "send" },
            {
              "key": "sender",
              "value": "band168ukdplr7nrljaleef8ehpyvfhe4n78hz0shsy"
            },
            { "key": "module", "value": "bank" }
          ]
        },
        {
          "type": "transfer",
          "attributesList": [
            {
              "key": "recipient",
              "value": "band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f"
            },
            {
              "key": "sender",
              "value": "band168ukdplr7nrljaleef8ehpyvfhe4n78hz0shsy"
            },
            { "key": "amount", "value": "10uband" }
          ]
        }
      ]
    }
  ],
  "info": "",
  "gasWanted": 1500000,
  "gasUsed": 49013,
  "timestamp": ""
}
```

Get reference data

This section shows an example on how to query data from BandChain. This example query standard price references based on given symbol pairs, min count, and ask count.

Step 1: Import `bandchain.js` and put `grpc_url_web` as a parameter and you can get `<GRPC_WEB>` endpoint from here. Then initialize the client instance. Every method in client module can now be used.

```
import { Client } from '@bandprotocol/bandchain.js'
// Step 1
const grpcUrl = '<GRPC_WEB>' // ex.https://laozi-testnet6.bandchain.org/grpc-web
const client = new Client(grpcUrl)
```

Step 2: After we import the `Client` already, then we call the `Client`'s `getReferenceData` function to get the latest price

There are 3 parameters

- minCount: Integer of min count
- askCount: Integer of ask count
- pairs: The list of cryprocurrency pairs

The final code should look like the code below.

```
import { Client } from '@bandprotocol/bandchain.js'
// Step 1
const grpcUrl = '<GRPC_WEB>' // ex.https://laozi-testnet6.bandchain.org/grpc-web
const client = new Client(grpcUrl)

// Step 2
const minCount = 3
const askCount = 4
const pairs = ['BTC/USD', 'ETH/USD']

;(async () => {
  console.log(JSON.stringify(await client.getReferenceData(pairs, minCount, askCount)))
})()
```

And the result should look like this.

```
[
  {
    "pair": "BTC/USD",
    "rate": 34078.0954,
    "updatedAt": {
      "base": 1625579610,
      "quote": 1625579627
    },
    "requestId": {
      "base": 79538,
      "quote": 0
    }
  },
  {
    "pair": "ETH/BTC",
    "rate": 0.06759872648278929,
    "updatedAt": {
      "base": 1625579610,
      "quote": 1625579610
    },
    "requestId": {
      "base": 79538,
      "quote": 79538
    }
  }
]
```

Send BAND token via IBC Transfer

With BandChain built based on the Cosmos-SDK, we also allow interaction with our data oracle through **Cosmos Inter-Blockchain-Communication protocol,** `IBC`, which connects other compatible blockchains to request data from BandChain.

To send BAND tokens through IBC Protocol, we will use [`MsgTransfer`] as a method to represents a message to send coins from one account to another between ICS20 enabled chains. See ICS spec here.

Step 1: First, you need to create a `MsgTransfer` instance from the Message module. The following code shows you how to create the instance.

```
import { Message } from '@bandprotocol/bandchain.js'

const { MsgTransfer } = Message
```

Step 2: Now we can construct the `MsgTransfer` method, this method requires 5 fields to interact with:

Field	Type	Description
sourcePort	string	The port on which the packet will be sent
sourceChannel	string	The channel by which the packet will be sent
sender	string	The sender address
receiver	string	The recipient address on the destination chain
token	Coin	The tokens to be transferred
timeoutTimestamp	number	Timeout timestamp (in nanoseconds) relative to the current block timestamp.

Your code should look like this.

```
const receiver = 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f'
const sourcePort = 'transfer'
const sourceChannel = 'channel-25'
const sendAmount = new Coin()
sendAmount.setDenom('uband')
sendAmount.setAmount('10')
const timeoutTimestamp = moment().unix() + 600 * 1e9 // timeout in 10 mins

const msg = new MsgTransfer(
  sourcePort,
  sourceChannel,
  sendAmount,
  sender,
  receiver,
  timeout_timestamp
)
```

Your final code should look something like this:

```
import { Client, Wallet, Transaction, Message, Coin, Fee } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const client = new Client('<GRPC_WEB>') // ex.https://laozi-testnet6.bandchain.org/grpc-web
const privkey = PrivateKey.fromMnemonic(
  'subject economy equal whisper turn boil guard giraffe stick retreat wealth card only buddy joy leave
)
const pubkey = privkey.toPubkey()
const sender = pubkey.toAddress().toAccBech32()

const sendCoinIbc = async () => {
  // Step 1 constructs MsgTransfer message
  const { MsgTransfer } = Message

  const receiver = 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f'
  const sourcePort = 'transfer'
  const sourceChannel = 'channel-25'
  const sendAmount = new Coin()
  sendAmount.setDenom('uband')
  sendAmount.setAmount('10')
  const timeoutTimestamp = moment().unix() + 600 * 1e9 // timeout in 10 mins

  const msg = new MsgTransfer(
    sourcePort,
    sourceChannel,
    sendAmount,
    sender,
    receiver,
    timeout_timestamp
  )

  // Step 2 constructs a transaction
  const account = await client.getAccount(sender)
  const chainId = 'band-laozi-testnet6'

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('1000')

  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(1000000)
  const tx = new Transaction()
    .withMessages(msg)
    .withAccountNum(account.accountNumber)
    .withSequence(account.sequence)
    .withChainId(chainId)
    .withFee(fee)

  // Step 3 sign the transaction
  const txSignData = tx.getSignDoc(pubkey)
  const signature = privkey.sign(txSignData)
  const signedTx = tx.getTxData(signature, pubkey)

  // Step 4 send the transaction
  const response = await client.sendTxBlockMode(signedTx)
  console.log(response)
}

;(async () => {
  await sendCoinIbc()
})()
```

Connect to your app with Ledger

This tutorial guides you through how to set up and use the Bandchain.js library with your Ledger device to access your Ledger Cosmos (BAND) account(s).

Supported Browsers

- Chrome
- Edge (89.0 and later)
- Opera (76.0 and later)

we strongly recommend using Chrome.

To connect your app you will need to install:

1. Ledger Live
2. The Cosmos Nano App
3. At least one account for Band

Accessing your Ledger

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Ledger } = Wallet

const connectLedger = async () => {
  const ledger = await Ledger.connectLedgerWeb()
  console.log(ledger)
}

;(async () => {
  await connectLedger()
})()
```

Sending Band Token using Ledger

To send BAND token using Ledger device, the final code should look something like this:

```
import { Wallet, Client, Transaction, Message, Coin, Fee } from '@bandprotocol/bandchain.js'

const sendCoinWithLedger = async () => {
  const ledger = await Ledger.connectLedgerWeb()

  const { MsgSend } = Message
  const receiver = 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f'

  const sendAmount = new Coin()
  sendAmount.setDenom('uband')
  sendAmount.setAmount('10')

  const msg = new MsgSend(sender, receiver, [sendAmount])

  const account = await client.getAccount(sender)
  const chainId = 'band-laozi-testnet6'

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('1000')

  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(1000000)

  const tx = new Transaction().withMessages(msg).withChainId(chainId).withFee(fee).withMemo('')

  await tx.withSender(client, bech32_address)

  // Sign a message with Ledger device
  const signature = await ledger.sign(tx)
  const signedTx = tx.getTxData(signature, pubKey, 127)

  // Create a transaction
  const response = await client.sendTxBlockMode(signedTx)
  console.log(response)
}

;(async () => {
  await sendCoinWithLedger()
})()
```

Getting Testnet BAND from Faucet

```
async function getFaucet() {
  const body = {
    address: 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f',
    amount: '10',
  }

  let options = {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json;charset=utf-8',
    },
    body: JSON.stringify(body),
  }

  // See https://docs.bandchain.org/develop/api-endpoints#laozi-testnet-5
  let response = await fetch(`${BAND_FAUCET_ENDPOINT}`, options)

  console.log(response)
}

getFaucet()
// {"txHash": "07EA6C439A72DE3A3FEBD6FC952EBEF54B802CC0A9C00C9A1265561AFE9169A7"}
```

And these are examples of Bandchain.js usages, for more information, feel free to dive into specifications in each module.

Client Module

This Client Metrics API provides data about the Blockchain and Cardano Blockchain Transaction. It is paired with Cardano Rosetta API and Cardano Blockfrost API to provide a full picture.

Base URL: `https://cardano-mainnet.blockfrost.io/api/v0/metrics/`

gHTC Errors

When there are gHTC errors, the gHTC error code is in the `X-Error` header and will be appended on all responses. The error message is in the `X-Error-Info` header.

Code	Name	Description
`1000`	`Unspecified`	gHTC Error
`1001`	`Not Implemented`	gHTC Error
`1002`	`Unauthorized`	gHTC Error
`1003`	`Forbidden`	gHTC Error
`1004`	`Not Acceptable`	gHTC Error

getChainID

Get the Blockchain Chain ID

Method:

- `GET` `/chain/{chain-id}` - chain-id

Example:

```
import { client, BlockfrostClientConfig, }
const config = new BlockfrostClientConfig({
  project_id: "mainnet",
});
const result = client.getChainID(config);
```

Result:

```
"cardano-mainnet"
```

getLatestBlock

Get the latest block data.

Method:

- `GET` `/blocks/latest` - latest block

Example:

```
import { client, BlockfrostClientConfig, }
const config = new BlockfrostClientConfig({
  project_id: "mainnet",
});
const result = client.getLatestBlock(config);
```

Result:

```
{
  "time": 1588388092,
  "height": 4388093,
  "hash": "4ea1f6fb2cb753ddfcf2d0c7d0f0e3ba6d7d1ac2f7f533d90d78b0f3ac4a5ca7",
  "slot": 4492800,
  "epoch": 208,
  "epoch_slot": 412800,
  "slot_leader": "pool18qx2fzzf5qujqwa5rqs6n6zqfw6kaj34ccz8wf63d8sx5qqfzqr",
  "size": 3072,
  "tx_count": 2,
  "output": "128314491794",
  "fees": "592661",
  "block_vrf": "vrf_vk1wf2k6lhujezqcfe03fs4mfs5qzngzxq7cqmz72dcpqd3suqqgcqsjw8ep7",
  "previous_block": "4ea1f6fb2cb753ddfcf2d0c7d0f0e3ba6d7d1ac2f7f533d90d78b0f3ac4a5ca6",
  "next_block": null,
  "slot_qty": 432000
}
```

getLatestBlockTxs

Get the latest block txs.

Method:

- `GET` `/blocks/latest/txs` - latest block txs

Example:

```
import { client, BlockfrostClientConfig, }
const config = new BlockfrostClientConfig({
  project_id: "mainnet",
});
const result = client.getLatestBlockTxs(config);
```

Result:

```
[
  "d1bc86475b8a5ff5e0f9f9f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0",
  "d2cd97586b9a6gg6f1g0g0g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1",
  "d3de08697c0b7hh7g2h1h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2",
  "d4ef19708d1c8ii8h3i2i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3",
  "d5fg20819e2d9jj9i4j3j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4",
  "d6gh31920f3e0kk0j5k4k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5",
  "d7hi42031g4f1ll1k6l5l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6",
  "d8ij53142h5g2mm2l7m6m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7",
  "d9jk64253i6h3nn3m8n7n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8",
  "daklm75364j7i4oo4n9o8o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9",
  "dblmn86475k8j5pp5o0p9p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p",
  "dcmno97586l9k6qq6p1q0q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q",
  "ddnop08697m0l7rr7q2r1r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r",
  "deopq19708n1m8ss8r3s2s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s",
  "dfpqr20819o2n9tt9s4t3t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4",
  "dgqrs31920p3o0uu0t5u4u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u",
  "dhrst42031q4p1vv1u6v5v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v",
  "dist53142r5q2ww2v7w6w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w",
  "djtu64253s6r3xx3w8x7x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x",
  "dkuv75364t7s4yy4x9y8y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9",
  "dlvw86475u8t5zz5y0z9z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0",
  "dmwx97586v9u6aa6z1a0a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a",
  "dnxy08697w0v7bb7a2b1b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b",
  "doyz19708x1w8cc8b3c2c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c",
  "dpza20819y2x9dd9c4d3d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d",
  "dqab31920z3y0ee0d5e4e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e",
  "drbc42031a4z1ff1e6f5f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f",
  "dscd53142b5a2gg2f7g6g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g",
  "dtde64253c6b3hh3g8h7h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h",
  "dufe75364d7c4ii4h9i8i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9",
  "dvgf86475e8d5jj5i0j9j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j",
  "dwhg97586f9e6kk6j1k0k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k",
  "dxih08697g0f7ll7k2l1l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l",
  "dyji19708h1g8mm8l3m2m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m",
  "dzkj20819i2h9nn9m4n3n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n",
  "ealk31920j3i0oo0n5o4o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o",
  "fbml42031k4j1pp1o6p5p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p",
  "gcnm53142l5k2qq2p7q6q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q",
  "hdon64253m6l3rr3q8r7r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r",
  "ipo75364n7m4ss4r9s8s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s",
  "jqp86475o8n5tt5s0t9t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0",
  "krq97586p9o6uu6t1u0u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u",
  "lsr08697q0p7vv7u2v1v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v",
  "mts19708r1q8ww8v3w2w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w",
  "nut20819s2r9xx9w4x3x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x",
  "ovu31920t3s0yy0x5y4y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y",
  "pwv42031u4t1zz1y6z5z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z",
  "qxw53142v5u2aa2z7a6a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a",
  "ryx64253w6v3bb3a8b7b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b",
  "szy75364x7w4cc4b9c8c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c",
  "taz86475y8x5dd5c0d9d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d",
  "ubz97586z9y6ee6d1e0e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e",
  "vca08697a0z7ff7e2f1f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f",
  "wdb19708b1a8gg8f3g2g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g",
  "xec20819c2b9hh9g4h3h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h",
  "yfd31920d3c0ii0h5i4i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i",
  "zge42031e4d1jj1i6j5j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j",
  "ahf53142f5e2kk2j7k6k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k",
  "big64253g6f3ll3k8l7l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8",
  "cjh75364h7g4mm4l9m8m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m",
  "dki86475i8h5nn5m0n9n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n",
  "elj97586j9i6oo6n1o0o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1",
  "fmk08697k0j7pp7o2p1p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p",
  "gnl19708l1k8qq8p3q2q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q",
  "hom20819m2l9rr9q4r3r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r",
  "ipn31920n3m0ss0r5s4s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s",
  "jqo42031o4n1tt1s6t5t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t",
  "krp53142p5o2uu2t7u6u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u",
  "lsq64253q6p3vv3u8v7v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v",
  "mtr75364r7q4ww4v9w8w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w",
  "nus86475s8r5xx5w0x9x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x",
  "ovt97586t9s6yy6x1y0y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y",
  "pwu08697u0t7zz7y2z1z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z",
  "qxv19708v1u8aa8z3a2a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a",
  "ryw20819w2v9bb9a4b3b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b",
  "szx31920x3w0cc0b5c4c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c",
  "tay42031y4x1dd1c6d5d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d",
  "ubz53142z5y2ee2d7e6e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e",
  "vca64253a6z3ff3e8f7f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f",
  "wdb75364b7a4gg4f9g8g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g",
  "xec86475c8b5hh5g0h9h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h",
  "yfd97586d9c6ii6h1i0i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i",
  "zge08697e0d7jj7i2j1j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j",
  "ahf19708f1e8kk8j3k2k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k",
  "big20819g2f9ll9k4l3l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l",
  "cjh31920h3g0mm0l5m4m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m",
  "dki42031i4h1nn1m6n5n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n",
  "elj53142j5i2oo2n7o6o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o",
  "fmk64253k6j3pp3o8p7p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p",
  "gnl75364l7k4qq4p9q8q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q",
  "hom86475m8l5rr5q0r9r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r",
  "ipn97586n9m6ss6r1s0s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s",
  "jqo08697o0n7tt7s2t1t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t",
  "krp19708p1o8uu8t3u2u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u",
  "lsq20819q2p9vv9u4v3v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v",
  "mtr31920r3q0ww0v5w4w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w",
  "nus42031s4r1xx1w6x5x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x",
  "ovt53142t5s2yy2x7y6y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y",
  "pwu64253u6t3zz3y8z7z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z",
  "qxv75364v7u4aa4z9a8a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a",
  "ryw86475w8v5bb5a0b9b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b",
  "szx97586x9w6cc6b1c0c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c",
  "tay08697y0x7dd7c2d1d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d",
  "ubz19708z1y8ee8d3e2e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e",
  "vca20819a2z9ff9e4f3f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f4f",
  "wdb31920b3a0gg0f5g4g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g5g",
  "xec42031c4b1hh1g6h5h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h6h",
  "yfd53142d5c2ii2h7i6i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i7i",
  "zge64253e6d3jj3i8j7j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j8j",
  "ahf75364f7e4kk4j9k8k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k9k",
  "big86475g8f5ll5k0l9l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l0l",
  "cjh97586h9g6mm6l1m0m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m1m",
  "dki08697i0h7nn7m2n1n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n2n",
  "elj19708j1i8oo8n3o2o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o3o",
  "fmk20819k2j9pp9o4p3p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p4p",
  "gnl31920l3k0qq0p5q4q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q5q",
  "hom42031m4l1rr1q6r5r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r6r",
  "ipn53142n5m2ss2r7s6s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s7s",
  "jqo64253o6n3tt3s8t7t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t8t",
  "krp75364p7o4uu4t9u8u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u9u",
  "lsq86475q8p5vv5u0v9v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v",
  "mtr97586r9q6ww6v1w0w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w",
  "nus08697s0r7xx7w2x1x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x",
  "ovt19708t1s8yy8x3y2y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y3y",
  "pwu20819u2t9zz9y4z3z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z4z",
  "qxv31920v3u0aa0z5a4a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a",
  "ryw42031w4v1bb1a6b5b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b6b",
  "szx53142x5w2cc2b7c6c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c",
  "tay64253y6x3dd3c8d7d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d8d",
  "ubz75364z7y4ee4d9e8e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e9e",
  "vca86475a8z5ff5e0f9f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f",
  "wdb97586b9a6gg6f1g0g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g1g",
  "xec08697c0b7hh7g2h1h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h2h",
  "yfd19708d1c8ii8h3i2i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i3i",
  "zge20819e2d9jj9i4j3j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j4j",
  "ahf31920f3e0kk0j5k4k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k5k",
  "big42031g4f1ll1k6l5l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l6l",
  "cjh53142h5g2mm2l7m6m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m7m",
  "dki64253i6h3nn3m8n7n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n8n",
  "elj75364j7i4oo4n9o8o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9o9",
  "fmk86475k8j5pp5o0p9p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p0p",
  "gnl97586l9k6qq6p1q0q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q1q",
  "hom08697m0l7rr7q2r1r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r2r",
  "ipn19708n1m8ss8r3s2s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s3s",
  "jqo20819o2n9tt9s4t3t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t4t",
  "krp31920p3o0uu0t5u4u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u5u",
  "lsq42031q4p1vv1u6v5v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v6v",
  "mtr53142r5q2ww2v7w6w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w7w",
  "nus64253s6r3xx3w8x7x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x8x",
  "ovt75364t7s4yy4x9y8y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9y9",
  "pwu86475u8t5zz5y0z9z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0",
  "qxv97586v9u6aa6z1a0a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a",
  "ryw08697w0v7bb7a2b1b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b",
  "szx19708x1w8cc8b3c2c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c",
  "tay20819y2x9dd9c4d3d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d",
  "ubz31920z3y0ee0d5e4e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e5e",
  "vca42031a4z1ff1e6f5f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f",
  "wdb53142b5a2gg2f7g6g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g7g",
  "xec64253c6b3hh3g8h7h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h8h",
  "yfd75364d7c4ii4h9i8i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i9i",
  "zge86475e8d5jj5i0j9j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j0j",
  "ahf97586f9e6kk6j1k0k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k",
  "big08697g0f7ll7k2l1l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l2l",
  "cjh19708h1g8mm8l3m2m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m3m",
  "dki20819i2h9nn9m4n3n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n4n",
  "elj31920j3i0oo0n5o4o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5o",
  "fmk42031k4j1pp1o6p5p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p6p",
  "gnl53142l5k2qq2p7q6q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q7q",
  "hom64253m6l3rr3q8r7r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r8r",
  "ipn75364n7m4ss4r9s8s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s9s",
  "jqo86475o8n5tt5s0t9t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0",
  "krp97586p9o6uu6t1u0u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u1u",
  "lsq08697q0p7vv7u2v1v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v2v",
  "mtr19708r1q8ww8v3w2w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w",
  "nus20819s2r9xx9w4x3x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x4x",
  "ovt31920t3s0yy0x5y4y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y5y",
  "pwu42031u4t1zz1y6z5z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z6z",
  "qxv53142v5u2aa2z7a6a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a7a",
  "ryw64253w6v3bb3a8b7b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b8b",
  "szx75364x7w4cc4b9c8c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c9c",
  "tay86475y8x5dd5c0d9d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d",
  "ubz97586z9y6ee6d1e0e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e",
  "vca08697a0z7ff7e2f1f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f",
  "wdb19708b1a8gg8f3g2g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g",
  "xec20819c2b9hh9g4h3h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4h",
  "yfd31920d3c0ii0h5i4i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i5i",
  "zge42031e4d1jj1i6j5j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j6j",
  "ahf53142f5e2kk2j7k6k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k7k",
  "big64253g6f3ll3k8l7l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l8l",
  "cjh75364h7g4mm4l9m8m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m9m",
  "dki86475i8h5nn5m0n9n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n",
  "elj97586j9i6oo6n1o0o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1o1",
  "fmk08697k0j7pp7o2p1p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p",
  "gnl19708l1k8qq8p3q2q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q3q",
  "hom20819m2l9rr9q4r3r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r",
  "ipn31920n3m0ss0r5s4s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s5s",
  "jqo42031o4n1tt1s6t5t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t6t",
  "krp53142p5o2uu2t7u6u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u7u",
  "lsq64253q6p3vv3u8v7v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v8v",
  "mtr75364r7q4ww4v9w8w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w9w",
  "nus86475s8r5xx5w0x9x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x",
  "ovt97586t9s6yy6x1y0y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y",
  "pwu08697u0t7zz7y2z1z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z2z",
  "qxv19708v1u8aa8z3a2a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a",
  "ryw20819w2v9bb9a4b3b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b4b",
  "szx31920x3w0cc0b5c4c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c",
  "tay42031y4x1dd1c6d5d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d6d",
  "ubz53142z5y2ee2d7e6e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e7e",
  "vca64253a6z3ff3e8f7f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f8f",
  "wdb75364b7a4gg4f9g8g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g9g",
  "xec86475c8b5hh5g0h9h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h0h",
  "yfd97586d9c6ii6h1i0i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i1i",
  "zge08697e0d7jj7i2j1j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j2j",
  "ahf19708f1e8kk8j3k2k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k3k",
  "big20819g2f9ll9k4l3l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l",
  "cjh31920h3g0mm0l5m4m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m",
  "dki42031i4h1nn1m6n5n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n",
  "elj53142j5i2oo2n7o6o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o7o",
  "fmk64253k6j3pp3o8p7p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p8p",
  "gnl75364l7k4qq4p9q8q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q9q",
  "hom86475m8l5rr5q0r9r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r0r",
  "ipn97586n9m6ss6r1s0s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s1s",
  "jqo08697o0n7tt7s2t1t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t",
  "krp19708p1o8uu8t3u2u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u3u",
  "lsq20819q2p9vv9u4v3v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v",
  "mtr31920r3q0ww0v5w4w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w5w",
  "nus42031s4r1xx1w6x5x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x6x",
  "ovt53142t5s2yy2x7y6y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y7y",
  "pwu64253u6t3zz3y8z7z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8z8
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
BandChain.js
Getting Started
Installation
Common Usage Example
Client Module
Data Module
Message Module
OBI Module
Transaction Module
Wallet Module
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Data Module

This module is to construct the data in term of dataclass to make sure that all the input and output follow this data schema.

Although there are Protobuf classes, some types are not exists. Therefore, additional types of data will be declared here.

> NOTE
> Note that `base` and `quote` is the first and the second price symbols respectively e.g. BTC/USD price rate has a base of BTC and a quote of USD.

ReferenceDataUpdated

Attribute	Type	Description
base	integer	Base's resolve time
quote	integer	Quote's resolve time

ReferenceDataRequestID

Attribute	Type	Description
base	integer	Base's request ID
quote	integer	Quote's request ID

ReferenceData

Attribute	Type	Description
pair	string	The token pair e.g. "BTC/USD"
rate	float	Price rate
updatedAt	ReferenceDataUpdated	Price update time
requestId	ReferenceDataRequestID	OracleRequest ID

Syntax

```
(method) Client.getReferenceData(pairs: string[], minCount: number, askCount: number): Promise<Data.Refe
```

Example

```
import { Client, Data } from "@bandprotocol/bandchain.js"

async function getReferenceData() {
  const grpcEndpoint = ${GRPC_ENDPOINT}
  const client = new Client(grpcEndpoint)
  const data = await client.getReferenceData(["BTC/USD", "ETH/BTC"], 3, 4)
  console.log(typeof data) // Promise<Data.ReferenceData[]>
}
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
BandChain.js
Getting Started
Installation
Common Usage Example
Client Module
Data Module
Message Module
OBI Module
Transaction Module
Wallet Module
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Message Module

Messages specified by BandChain. All messages presented in this module are extended from Protobuf's messages, which are generated from these proto files. For specification can be found here.

Here are methods that are extended from original protobuf classes.

---

toAny

Returns an Google Protobuf's `Any` instance that are used to construct transactions.

Return

- `Any` - a Google Protobuf's `Any` instance containing serialized messaged and type URL

Example

```
import { Message, Coin } from '@bandprotocol/bandchain.js'
const { MsgRequestData } = Message

const sender = 'band17n5rmujk78nkgss7tjecg4nfzn6geg4cvaqt5h'
const oracleScriptId = 37
const calldata = Buffer.from('0000000200000003425443000000034554480000000000000064', 'hex')
const askCount = 4
const minCount = 3
const clientId = 'from_bandchain.js'
const msg = new MsgRequestData(oracleScriptId, calldata, askCount, minCount, clientId, sender)

const any = msg.toAny()
console.log(any.getTypeUrl())
console.log(any.getValue_asB64())
```

Result

```
/oracle.v1.MsgRequestData
CCUSGgAAAAIAAAADQlRDAAAAA0VUSAAAAAAAAABkGAQgAyoRZnJvbV9iYW5kY2hhaW4uanM40IYDQOCnEkorYmFuZDE3bjVybXVqazc4
```

validate

Validates the message in a basic manner to ensure that there are no invarient values stored in the message instance. If message's validation fails, it will throw the exception.

Example

```
import { Message, Coin } from '@bandprotocol/bandchain.js'
const { MsgSend } = Message

const sender = 'band17n5rmujk78nkgss7tjecg4nfzn6geg4cvaqt5h'
const receiver = 'band1p46uhvdk8vr829v747v85hst3mur2dzlmlac7f'
const sendAmount = new Coin()
sendAmount.setDenom('uband')
sendAmount.setAmount('10')
const msg = new MsgSend(sender, receiver, [sendAmount])

msg.validate()
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
PyBand
BandChain.js
Getting Started
Installation
Common Usage Example
Client Module
Data Module
Message Module
OBI Module
Transaction Module
Wallet Module
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

OBI Module

Oracle Binary Encoding (OBI) is the standard way to serialized and deserialize binary data in the BandChain ecosystem. This module provides the functionality to serialize data. More details.

Bandchain.js provide a class named `Obi` to help encode/decode binary data using OBI encoding. Here is the usage of the class.

Constructor

- **schema** `string` - A string of OBI schema, including input and output schemas.

Example

```
import { Obi } from '@bandprotocol/bandchain.js'

const obi = new Obi('{symbol:string, px: u64, w: {a: u8, b: u8}, tb: [string]} / string')
```

---

encodeInput(value)

Encode the value based on given OBI input schema

Parameter

- value `any` - A value to be encoded. can be any type of data.

Return

- `Buffer` - An encoded value

Example

```
import { Obi } from '@bandprotocol/bandchain.js'

const obi = new Obi('{symbol:string, px: u64, w: {a: u8, b: u8}, tb: [string]} / string')
const testInput = {
  symbol: 'BTC',
  px: 9000,
  w: { a: 1, b: 2 },
  tb: ['a', 'b'],
}
console.log(obi.encodeInput(testInput).toString('hex'))
```

Result

```
00000003425443000000000000232801020000000200000001610000000162
```

---

encodeOutput(value)

Encode the value based on OBI output schema

Parameter

- value `any` - The value to be encoded

Return

- `Buffer` - An encoded value

Example

```
import { Obi } from '@bandprotocol/bandchain.js'

const obi = new Obi('{symbol:string, px: u64, w: {a: u8, b: u8}, tb: [string]} / string')
const testOutput = 'test'
console.log(obi.encodeOutput(testOutput).toString('hex'))
```

Result

```
0000000474657374
```

---

decodeInput(buff)

Decode the value based on given OBI input schema

Parameter

- value `Buffer` - The value to be decoded

Return

- `any` - A decoded value

Exceptions

Type	Description
DecodeError	Not all data is consumed after decoding output

Example

```
import { Obi } from '@bandprotocol/bandchain.js'

const obi = new Obi('{symbol:string, px: u64, w: {a: u8, b: u8}, tb: [string]} / string')
console.log(
  obi.decodeInput(
    Buffer.from('00000003425443000000000000232801020000000200000001610000000162', 'hex')
  )
)
```

Result

```
{ "symbol": "BTC", "px": 9000n, "w": { "a": 1n, "b": 2n }, "tb": ["a", "b"] }
```

---

decodeOutput(buff)

Decode the output value by using output schema

Parameter

- value `Buffer` - The value to be decoded

Return

- `any` - A decoded value

Exceptions

Type	Description
DecodeError	Not all data is consumed after decoding output

Example

```
import { Obi } from '@bandprotocol/bandchain.js'

const obi = new Obi('{symbol:string, px: u64, w: {a: u8, b: u8}, tb: [string]} / string')
console.log(obi.decodeOutput(Buffer.from('0000000474657374', 'hex')))
```

Result

```
test
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Pyband
BandChain.js
Getting Started
Installation
Common Usage Example
Client Module
Data Module
Message Module
OEE Module
Transaction Module
Wallet Module
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Transaction Module

This module provides functionalities to send transactions on BandChain which requires `<Msg>` to be included.

Exceptions

Type	Description
ValueError	Invalid value
NoIntegerError	The value is not an integer
EmptyMsgError	No messages given to the transaction
UndefinedError	The value should not be undefined

withMessages(msgs)

Add one or multiple messages as a list of Google Protobufs `Any` to `Transaction`. There are predefined message classes that can be used to convert to `Any` instance using `toAny()` method, but for the other type of message can be converted to `Any` instance using `Any.pack()` as shown in the code below.

```
import { MsgCreateDataSource } from "@bandprotocol/bandchain.js/proto/oracle/v1/tx.pb"
import { Any } from "google-protobuf/google/protobuf/any.pb"
import { Transaction } from "@bandprotocol/bandchain.js"

const msg = new MsgCreateDataSource()
msg.setName('dsName')
msg.setDescription('dsDescription')
// msg.set...() for every fields

const anyMsg = new Any()
const typeUrl = 'oracle.v1.MsgCreateDataSource'
anyMsg.pack(msg.serializeBinary(), typeUrl, '/')

const tx = new Transaction()
tx.withMessages(anyMsg)
```

Parameter

- *msgs `Any` Messages converted to `Any` to be included in the transaction

Return

- `Transaction` - This transaction instance for function chaining

withSender(client, sender)

Set account number and sequence number to `Transaction` from querying the account via Client's `getAccount`.

Parameter

- **client** `Client` - An instance of Client module
- **sender** `string` - A bech32-encoded with account prefix

Return

- `Promise<Transaction>` - This transaction instance for function chaining

Exceptions

Type	Description
EmptyMsgError	Message is empty, please use withMessages at least 1 message
NotFoundError	Account doesn't exist.

withAccountNum(accountNum)

Set account number to `Transaction`.

Parameter

- **accountNum** `number` - An integer of account number, which can be gathered from querying the account via Client's `getAccount`.

Return

- `Transaction` - This transaction instance for function chaining

Exceptions

Type	Description
NoIntegerError	accountNum is not an integer

withSequence(sequence)

Set sequence number to `Transaction`.

Parameter

- **sequence** `Number` - An integer of account's sequence number, which can be gathered from querying the account via Client's `getAccount`.

Return

- `Transaction` - This transaction instance for function chaining

Exceptions

Type	Description
NoIntegerError	sequence is not an integer

withChainId(chainId)

Set chain id to `Transaction`.

Parameter

- **chainId** `string` - a string of chain ID, which can be gathered from Client's `getChainId`.

Return

- `Transaction` - This transaction instance for function chaining

withFee(fee)

Set fee to `Transaction`.

Parameter

- **fee** - Set fee limit spent for gas price of the transaction.

Return

- `Transaction` - This transaction instance for function chaining

Exceptions

Type	Description
NoIntegerError	fee is not an integer

withMemo(memo)

Set memo to `Transaction`.

Parameter

- **memo** `string` - an arbitrary string to remember the transaction. Memo length is limited to 256.

Exceptions

Type	Description
ValueTooLargeError	memo is too large.

Return

- `Transaction` - This transaction instance for function chaining

getSignDoc()

Get serialized data of transaction's content to be signed from `Transaction` by using `SIGNMODE_DIRECT`. See more about signing mode.

Return

- `UintArray` - A byte array of serialized transaction content data, ready for signing.

Exceptions

Type	Description
EmptyMsgError	message is empty.
UndefinedError	accountNum should be defined
UndefinedError	sequence should be defined
UndefinedError	chainID should be defined

getSignMessage()

Get serialized data of transaction's content to be signed from `Transaction` by using `SIGN_MODE_LEGACY_AMINO_JSON`. See more about signing mode. **When using the Ledger to sign message, you need to use this method.**

Return

- `UintArray` - A byte array of serialized transaction content data, ready for signing.

getTxData(signature, publicKey, signMode)

Get transaction data from `Transaction`.

Parameter

- **signature** `UintArray` - signature for the transaction
- **pubkey** `PublicKey` - an instance of public key to be included in the transaction
- **signMode** `SignMode` - a signing mode with its own security guarantees representation. (Default: `SIGN_MODE_DIRECT`)

Return

- `UintArray` - Serialized data of signed transaction, ready to be broadcasted to BandChain.

Exceptions

Type	Description
UndefinedError	accountNum should be defined
UndefinedError	sequence should be defined

Example

```
import { Client, Wallet, Transaction, Message, Coin, Fee } from "@bandprotocol/bandchain.js"

const { privateKey } = Wallet
const client = new Client("https://laozi-testnet6.bandchain.org/grpc-web")

// Step 2.1 import private key based on given mnemonic string
const privkey = PrivateKey.fromMnemonic(
  'subject economy equal whisper turn boil guard giraffe stick retreat wealth card only buddy joy leave
)
// Step 2.2 prepare public key and its address
const pubkey = privkey.toPubKey()
const sender = pubkey.toAddress().toAccBech32()

const sendCoin = async () => {
  // Step 3.1 construct MsgSend message
  const { MsgSend } = Message

  // here we use different message type, which is MsgSend
  const receiver = 'band1p4u5vdk8vr82fv74v85hs13mur2d2lm1ac7f'
  const sendAmount = new Coin()
  sendAmount.setDenom('uband')
  sendAmount.setAmount('10')
  const msg = new MsgSend(sender, receiver, [sendAmount])
  // Step 3.2 construct a transaction
  const account = await client.getAccount(sender)
  const chainId = 'band-laozi-testnet6'

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('1000')

  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(1000000)
  const tx = new Transaction()
    .withMessages(msg)
    .withAccountNum(account.accountNumber)
    .withSequence(account.sequence)
    .withChainId(chainId)
    .withFee(fee)

  // Step 4 sign the transaction
  const txSignDoc = tx.getSignDoc(pubkey)
  const signature = privkey.sign(txSignDoc)
  const signedTx = tx.getTxData(signature, pubkey)

  // Step 5 send the transaction
  const response = await client.sendTxBlockMode(signedTx)
  console.log(JSON.stringify(response))
}

;(async () => {
  await sendCoin()
})()
```

Result


```
{
  "height": 144075,
  "txhash": "F9BC1B12211EC1C94E3BC05FC8B8410EC5B0A2540CC7F5A395AF61E05965C648",
  "codespace": "",
  "code": 0,
  "data": "0A10A01CF9336F73609F732E62616E6432E7363162657461312E407367595664",
  "rawLog": "[{\"events\":[{\"type\":\"coin_received\",\"attributes\":[{\"key\":\"receiver\",\"value\":\"
  "logs":[{
    "msgIndex": 0,
    "log": "",
    "eventsList": [
      {
        "type": "coin_received",
        "attributesList": [
          {
            "key": "receiver",
            "value": "band1p4u5vdk8vr82fv74v85hs13mur2d2lm1ac7f"
          },
          {"key": "amount", "value": "10uband" }
        ]
      },
      {
        "type": "coin_spent",
        "attributesList": [
          {
            "key": "spender",
            "value": "band1dpz2fy19621282b3ctfvrfs13q7ydxzj87drwh"
          },
          {"key": "amount", "value": "10uband" }
        ]
      },
      {
        "type": "message",
        "attributesList": [
          {"key": "action", "value": "/cosmos.bank.v1beta1.MsgSend" },
          {
            "key": "sender",
            "value": "band1dpz2fy19621282b3ctfvrfs13q7ydxzj87drwh"
          },
          {"key": "module", "value": "bank" }
        ]
      },
      {
        "type": "transfer",
        "attributesList": [
          {
            "key": "recipient",
            "value": "band1p4u5vdk8vr82fv74v85hs13mur2d2lm1ac7f"
          },
          {
            "key": "sender",
            "value": "band1dpz2fy19621282b3ctfvrfs13q7ydxzj87drwh"
          },
          {"key": "amount", "value": "10uband" }
        ]
      }
    ]
  }]
],
"info": "",
"gasWanted": 200000,
"gasUsed": 66987,
"timestamp": ""
}
```

Wallet Module

This module provides functionalities to manage BandChain account. It can be used to create a private key from mnemonic, sign and verify signature, and account address manipulation.

PrivateKey

A class for wrapping signing key, which is a byte array of private key, used for signature creation and public key derivation.

generate(path)

A static method for generating new private key with random mnemonic phrase.

> INFO
> If path is not given, default to Band's HD prefix 494 and all other indexes being zeroes.

Parameter

- path `string` - The HD path that follows the BIP32 standard (optional)

Return

- `[string, PrivateKey]` - A tuple of mnemonic phrase and `[PrivateKey]` instance

Exception

Type	Description
CreateError	Cannot create private key

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const [mnemonic, priv] = PrivateKey.generate("m/44'/494'/0'/0/3")
```

fromMnemonic(word, path)

A static method for creating a `[PrivateKey]` instance from a given mnemonic phrase and a HD derivation path.

> INFO
> If path is not given, default to Band's HD prefix 494 and all other indexes being zeroes.

Parameter

- **words** `string` - The mnemonic phrase for recover private key
- **path** `string` - The HD path that follows the BIP32 standard (optional)

Return

- `[PrivateKey]` - Initialized PrivateKey object

Exception

Type	Description
CreateError	Cannot create private key

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const priv = PrivateKey.fromMnemonic('test mnemonic')
```

fromHex(priv)

Create a `[PrivateKey]` instance from a given a hex representation of signing key.

Parameter

- **priv** `string` - A hex representation of signing key.

Return

- `[PrivateKey]` - Initialized PrivateKey object

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const priv = PrivateKey.fromHex('2442b724db7189468f16accc0fc505f0609817eb129e13702e696d8b84609ea9')
```

toHex

Return a hex representation of signing key.

Return

- `string` - A hex representation of signing key.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const priv = PrivateKey.fromMnemonic('test mnemonic')
console.log(priv.toHex())
```

Result

```
2cb2e2d3582cebf0664d9cda0b89c5d478ae12fac19a6f4ed9c10a7406a19615
```

toPubkey

Create a `[PublicKey]` instance which is associated with given private key.

Return

- `[PublicKey]` - A PublicKey that can be used to verify the signatures made with this PrivateKey.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const priv = PrivateKey.fromMnemonic('test mnemonic')
console.log(priv.toPubkey().toHex())
```

Result

```
02b2b0d35cb1c6d3923813c64e46a82d29e12d03288f438b9d32cf232d9a22bcb83
```

sign(msg)

Sign and the given bytes array.

Parameter

- **msg** `Buffer` - The message that will be hashed and signed.

Return

- `Buffer` - A signature of this private key over the given message

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const priv = PrivateKey.fromMnemonic('test mnemonic')
console.log(priv.sign(Buffer.from('test message')).toString('hex'))
```

Result

```
4bbc9a7ea54b47b11c67a4074e8d9bca068cb64c788f67342c4033b1b6f0553e1bc63cdf9bc2fb6e89c1e965c1e0f0712a51c25
```

PublicKey

Class for wraping verify Key, which is a byte array of public key, used for signature verification, and generate bech32-encoded address.

fromAccBech32(bech)

Creates a `[PublicKey]` instance from a bech32-encoded public key with account-pubkey prefix.

Parameter

- **bech** `string` - A bech32-encoded with account public key prefix.

Return

- `[PublicKey]` - A PublicKey instance

Exception

Type	Description
ValueError	Invalid bech32 prefix
DecodeError	Cannot decode bech32

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
```

fromValBech32(bech)

Creates a `[PublicKey]` instance from a bech32-encoded public key with validator-pubkey prefix

Parameter

- **bech** `string` - A bech32-encoded with validator public key prefix

Return

- `[PublicKey]` - A PublicKey instance

Exception

Type	Description
ValueError	Invalid bech32 prefix
DecodeError	Cannot decode bech32

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromValBech32(
  'bandvaloperpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q69gsm29'
)
```

fromConsBech32(bech)

Creates a `[PublicKey]` instance from a bech32-encoded public key with validator-consensus-pubkey prefix

Parameter

- **bech** `string` A bech32-encoded with validator consensus public key prefix

Return

- `[PublicKey]` - A PublicKey instance

Exception

Type	Description
ValueError	Invalid bech32 prefix
DecodeError	Cannot decode bech32

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromConsBech32(
  'bandvalconspub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6r8ytws'
)
```

toHex

Returns a hex representation of verifying key.

Return

- `string` - A hex representation of verifying key.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
console.log(publickey.toHex())
```

Result

```
0351e98e1be097250f2ff4188c0aace0a716e69a992cd77f9dfe436b3e8b34280d
```

toPubkeyProto

Returns an instance of Cosmos SDK Protobuf's `[PubKey]`, which can be used for constructing transactions.

Return

- `[PubKey]` - An instance of Cosmos SDK Protobuf's `PubKey`

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
// Converting to base64 is for displaying only
console.log(publickey.toPubkeyProto().getKey_asB64())
```

Result

```
A1HpjhvglyUPL/QYjAqs4KcW5pqZLNd/nf5Daz6LNCgN
```

toAccBech32

Returns bech32-encoded public key with account-pubkey prefix

Return

- `string` - A bech32-encoded with account public key prefix.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
console.log(publickey.toAccBech32())
```

Result

```
bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v
```

toValBech32

Returns bech32-encoded public key with validator-pubkey prefix

Return

- `string` - A bech32-encoded with validator public key prefix.

Exceptions

Type	Description
UnsuccessfulCallError	Unsuccessful bech32.toWords call

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
console.log(publickey.toValBech32())
```

Result

```
bandvaloperpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q69gsm29
```

toConsBech32

Returns bech32-encoded public key with validator-consensus-pubkey prefix

Return

- `string` - A bech32-encoded with validator consensus public key prefix.

Exceptions

Type	Description
UnsuccessfulCallError	Unsuccessful bech32.toWords call

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
console.log(publickey.toConsBech32())
```

Result

```
bandvalconspub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6r8ytws
```

toAddress

Returns an `[Address]` instance from given public key

Return

- `[Address]` - An Address instance.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PublicKey } = Wallet
const publickey = PublicKey.fromAccBech32(
  'bandpub1addwnpepqdg7nrsmuztj2re07svgcz4vuzn3de56nykdwlualepkk05txs5q6mw8s9v'
)
console.log(publickey.toAddress().toHex())
```

Result

```
8e453e66fb009b119ea9b1b8775be6fa9435013b
```

verify(msg, sig)

Verify a signature made over provided data.

Parameter

- **msg** `Buffer` A data signed by the `signature`, will be hashed using sha256 function
- **sig** `Buffer` A encoding of the signature

Return

- `boolean` True if the verification was successful

Exception

Type	Description
BadSignatureError	if the signature is invalid or malformed

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { PrivateKey } = Wallet
const priv = PrivateKey.fromMnemonic('test mnemonic')
const pubkey = priv.toPubkey()
const msg = Buffer.from('test message')
const sig = priv.sign(msg)
console.log(pubkey.verify(msg, sig))
```

Result

```
true
```

Address

Class for wraping Address. Adding method to encode/decode to Bech32 format.

fromAccBech32(bech)

Create an `[Address]` instance from a bech32-encoded with account prefix.

Parameter

- **bech** `string` - A bech32-encoded with account prefix.

Return

- `[Address]` - A Address instance

Exception

Type	Description
ValueError	Invalid bech32 prefix
DecodeError	Cannot decode bech32

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromAccBech32('band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c')
```

fromValBech32(bech)

Create an `[Address]` instance from a bech32-encoded with validator prefix

Parameter

- **bech** `string` - A bech32-encoded with validator prefix

Return

- `[Address]` - A Address instance

Exception

Type	Description
ValueError	Invalid bech32 prefix
DecodeError	Cannot decode bech32

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromValBech32('bandvaloper13eznuehmqzd3r84fkxu8wklxl22r2qfm8f05zn')
```

fromConsBech32(bech)

Create an `[Address]` instance from a bech32-encoded with validator consensus prefix

Parameter

- **bech** `string` - A bech32-encoded with validator consensus prefix

Return

- `[Address]` - A Address instance

Exception

Type	Description
ValueError	Invalid bech32 prefix
DecodeError	Cannot decode bech32

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromConsBech32('bandvalcons13eznuehmqzd3r84fkxu8wklxl22r2qfmn6ugwj')
```

toHex

Return a hex representation of `Address`.

Return

- `string` - A hex representation of Address.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromAccBech32('band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c')
console.log(address.toHex())
```

Result

```
8e453e66fb009b119ea9b1b8775be6fa9435013b
```

toAccBech32

Return bech32-encoded with account prefix

Return

- `string` - A bech32-encoded with account prefix.

Exceptions

Type	Description
UnsuccessfulCallError	Unsuccessful bech32.toWords call

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromAccBech32('band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c')
console.log(address.toAccBech32())
```

Result

```
band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c
```

toValBech32

Return bech32-encoded with validator prefix

Return

- `string` A bech32-encoded with validator prefix.

Exceptions

Type	Description
UnsuccessfulCallError	Unsuccessful bech32.toWords call

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromAccBech32('band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c')
console.log(address.toValBech32())
```

Result

```
bandvaloper13eznuehmqzd3r84fkxu8wklxl22r2qfm8f05zn
```

toConsBech32

Return bech32-encoded with validator consensus prefix

Return

- `string` - A bech32-encoded with validator consensus prefix.

Exceptions

Type	Description
UnsuccessfulCallError	Unsuccessful bech32.toWords call

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Address } = Wallet
const address = Address.fromAccBech32('band13eznuehmqzd3r84fkxu8wklxl22r2qfmtlth8c')
console.log(address.toConsBech32())
```

Result

```
bandvalcons13eznuehmqzd3r84fkxu8wklxl22r2qfmn6ugwj
```

Ledger

Ledger is a fundamental class that allows you to interact and perform actions with your Ledger devices; signing, and obtaining the account address.

connectLedgerNode(hdPath)

A static method used to connect Ledger with Node HID implementation.

Parameter

- hdPath `<string>` - The HD path that follows the BIP32 standard (optional). Defaults to `m/44'/118'/0'/0/0` if not provided.

Return

- `<Ledger>`: `Ledger` object.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Ledger } = Wallet

const connectLedger = async () => {
  const ledger = await Ledger.connectLedgerNode("m/44'/118'/0'/0/0")
  console.log(ledger)
}

;(async () => {
  await connectLedger()
})()
```

connectLedgerWeb(hdPath)

A static method used to connect Ledger with WebUSB/WebHID protocol implementation.

Parameter

- hdPath `<string>` - The HD path that follows the BIP32 standard (optional). Defaults to `m/44'/118'/0'/0/0` if not provided.

Return

- `<Ledger>`: `Ledger` object.

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Ledger } = Wallet

const connectLedger = async () => {
  const ledger = await Ledger.connectLedgerWeb("m/44'/118'/0'/0/0")
  console.log(ledger)
}

;(async () => {
  await connectLedger()
})()
```

getPubKeyAndBech32Address

This function returns public key and bech32 address with band prefix of the associated Ledger.

Return

An object containing public key and bech32 address with the following keys

- bech32_address `<string>`
- pubKey `<PublicKey>`

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Ledger } = Wallet

;(async () => {
  const ledger = await Ledger.connectLedgerWeb("m/44'/118'/0'/0/0")
  const addressObject = await ledger.getPubKeyAndBech32Address()
  console.log(addressObject)
})()
```

Return

```
{
    bech32_address: "band1pxzqj53rg87e2n0swh8h7a6m2umjlja6uwffp9",
    pubKey: PublicKey {verifyKey: Uint8Array(33)}
}
```

sign(transaction)

This function returns the signature of the associated Ledger. The message is signed by using the Nano App `signTransaction` function.

Parameter

- transaction `<Transaction>`: Sign message will be extracted from Transaction class and signed.

Return

- `<Buffer>`: A signature of the connected ledger over the given Transaction message.

Example in Bandchain.js Basic Usage

disconnect

As its namesake dictates, this function will disconnect the Ledger device.

Example

```
import { Wallet } from '@bandprotocol/bandchain.js'

const { Ledger } = Wallet

;(async () => {
  const ledger = await Ledger.connectLedgerWeb("m/44'/118'/0'/0/0")
  await ledger.disconnect()
})()
```

Introduction

A data source is the most fundamental unit in BandChain's oracle system. At the simplest level, it is an executable that describes the procedure to retrieve some type of data.

A data source is executed off-chain in order to reduce on-chain workloads as some tasks cannot be performed on the chain due to the heavy computation require and/or the network latency associated with running the data source which can decrease performance of the chain.

The sources a data source can use can either be a traditional API or any other method that returns the desired result.

Permissionless APIs are openly available to anyone who might want to audit the data source, any subsequent oracle scripts that depend on it, or the actual application that requests the oracle script itself. This openness helps build the trustworthiness of each of those components, which is ultimately what we are looking to do.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Data Source
Introduction
Creating a Data Source
Deployment
Oracle Script
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Creating a Data Source

In this section, we will take a look at how to create a data source.

Writing the data source

In order for the data source to run, always make sure to add a shebang line containing `#!/usr/bin/env python3` and to print the function output. A simple `Hello World!` example can be seen below.

```
#!/usr/bin/env python3

import sys


def main():
    return "Hello World!"


if __name__ == "__main__":
    try:
        print(main())
    except Exception as e:
        print(str(e), file=sys.stderr)
        sys.exit(1)
```

More Examples

However, in order to better understand the structure and implementation of a data source, let's look at two more examples below: Gold Price and Token Total Supply.

Gold Price

First, let's look at a data source that queries GoldPrice.org for the current price of gold. The script itself is written in Python and although this specific data source does not expect any arguments, a data source can have any number of arguments required.

```
#!/usr/bin/env python3

import requests
import sys


URL = "https://data-asg.goldprice.org/dbXRates/USD"
HEADERS = {'User-Agent': 'curl/7.64.1'}

def main():
    try:
        pxs = requests.get(URL, headers=HEADERS).json()
        return pxs['items'][0]['xauPrice']
    except Exception as e:
        raise e

if __name__ == "__main__":
    try:
        print('{0:.2f}'.format(main()))
    except Exception as e:
        print(str(e), file=sys.stderr)
        sys.exit(1)
```

Token Total Supply

As mentioned above, a data source can take any number of arguments. The example below, gives an example of a data source which requires two arguments. The data source shown below queries the given network's RPC endpoint for the total supply of a given token address.

```
#!/usr/bin/env python3

import requests
import sys


def main(rpc, to):
    headers = {
        "Content-Type": "application/json",
    }
    data = (
        """{ "jsonrpc": "2.0", "method": "eth_call", "params": [ { "to": "%s", "data": "0x18160ddd" }, '
        % (to)
    )
    response = requests.post(
        rpc,
        headers=headers,
        data=data,
    )
    return int(response.json()["result"], 16)


if __name__ == "__main__":
    try:
        print(main(sys.argv[1], sys.argv[2]))
    except Exception as e:
        print(str(e), file=sys.stderr)
        sys.exit(1)
```

More examples

For more data source examples, please refer to the ones available on our Mainnet to get an idea of the different types of data source used on BandChain:

- Latest crypto prices from CoinGecko
- Latest stock prices from Finage
- Latest forex prices from Alpha Vantage

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Data Source
Oracle Script
Introduction
Creating an Oracle Script
Deployment
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Deployment

On BandChain, an oracle script can be registered into the system by anyone. This is done through the registrant sending a `MsgCreateOracleScript` message to the chain.

A `MsgCreateOracleScript` message contains various parameters of the oracle script that is to be registered. These parameters include:

- `name`: Name of the oracle script.
- `description`: A description of the oracle script.
- `schema`: The oracle script's schema which details the inputs and outputs of this oracle script.
- `source_code_url`: The URL for the source code of the oracle script.
- `code`: The Owasm-compiled binary of the oracle script.
- `owner`: The owner of the oracle script. The owner will have edit rights. If omitted, the oracle script's parameters will no longer be able to be edited after being registered.
- `sender`: The message sender account.

In order to send a `MsgCreateOracleScript` message, we can use either bandchain.js or pyband

An example on how to send a `MsgCreateOracleScript` message via bandchain.js can be seen below.

```
import { Client, Wallet, Message, Coin, Transaction, Fee } from '@bandprotocol/bandchain.js'
import fs from 'fs'
import path from 'path'

const grpcURL = 'https://laozi-testnet6.bandchain.org/grpc-web'
const client = new Client(grpcURL)

// Setup the client
async function createOracleScript() {
  // Setup the wallet
  const { PrivateKey } = Wallet
  const mnemonic = process.env.MNEMONIC
  const privateKey = PrivateKey.fromMnemonic(mnemonic)
  const publicKey = privateKey.toPubkey()
  const sender = publicKey.toAddress().toAccBech32()

  // Setup the transaction's properties
  const chainId = await client.getChainId()
  const execPath = path.resolve(__dirname, 'hello_world.wasm')
  const code = fs.readFileSync(execPath)

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('0')

  const requestMessage = new Message.MsgCreateOracleScript(
    'Hello World!', // oracle script name
    code, // oracle script code
    sender, // owner
    sender, // sender
    '', // description
    '{repeat:u64}/{response:string}', // schema
    'https://ipfs.io/ipfs/QmSSrgJ6QuFDJHyC2SyTgnHKRBhPdLHUD2tJJ86xejrCfn' // source code url
  )

  // Construct the transaction
  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(350000)

  const txn = new Transaction()
  txn.withMessages(requestMessage)
  await txn.withSender(client, sender)
  txn.withChainId(chainId)
  txn.withFee(fee)
  txn.withMemo('')

  // Sign the transaction
  const signDoc = txn.getSignDoc(publicKey)
  const signature = privateKey.sign(signDoc)
  const txRawBytes = txn.getTxData(signature, publicKey)

  // Broadcast the transaction
  const sendTx = await client.sendTxBlockMode(txRawBytes)

  return sendTx
}

;(async () => {
  console.log(await createOracleScript())
})()
```

An example on how to send a `MsgCreateDataSource` message via pyband can also be seen below.

```
import os

from pyband import Client, Transaction
from pyband.wallet import PrivateKey
from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgCreateOracleScript
from google.protobuf.json_format import MessageToJson


def main():
    # Setup Client
    grpc_url = "laozi-testnet6.bandchain.org"
    c = Client(grpc_url)

    # Setup Wallet
    mnemonic = os.getenv("MNEMONIC")
    private_key = PrivateKey.from_mnemonic(mnemonic)
    public_key = private_key.to_public_key()
    sender_addr = public_key.to_address()
    sender = sender_addr.to_acc_bech32()

    # Prepare Transaction Properties
    deploy_msg = MsgCreateOracleScript(
        name="Hello World!",
        description="",
        schema="{repeat:u64}/{response:string}",
        source_code_url="https://ipfs.io/ipfs/QmSSrgJ6QuFDJHyC2SyTgnHKRBhPdLHUD2tJJ86xejrCfn",
        code=open("hello_world.wasm", "rb").read(),
        owner=sender,
        sender=sender,
    )

    account = c.get_account(sender)
    account_num = account.account_number
    sequence = account.sequence

    fee = [Coin(amount="0", denom="uband")]
    chain_id = c.get_chain_id()

    # Construct a Transaction
    txn = (
        Transaction()
        .with_messages(deploy_msg)
        .with_sequence(sequence)
        .with_account_num(account_num)
        .with_chain_id(chain_id)
        .with_gas(250000)
        .with_fee(fee)
    )

    # Sign the Transaction
    sign_doc = txn.get_sign_doc(public_key)
    signature = private_key.sign(sign_doc.SerializeToString())
    tx_raw_bytes = txn.get_tx_data(signature, public_key)

    # Broadcast the transaction
    tx_block = c.send_tx_block_mode(bytes(tx_raw_bytes))

    print(MessageToJson(tx_block))


if __name__ == "__main__":
    main()
```

After a successful transaction broadcast, the newly created oracle script ID can be found in the response json. The registrant can also view the created oracle script details on CosmoScan. An example of a successful transaction will return a response similar to the one shown below.

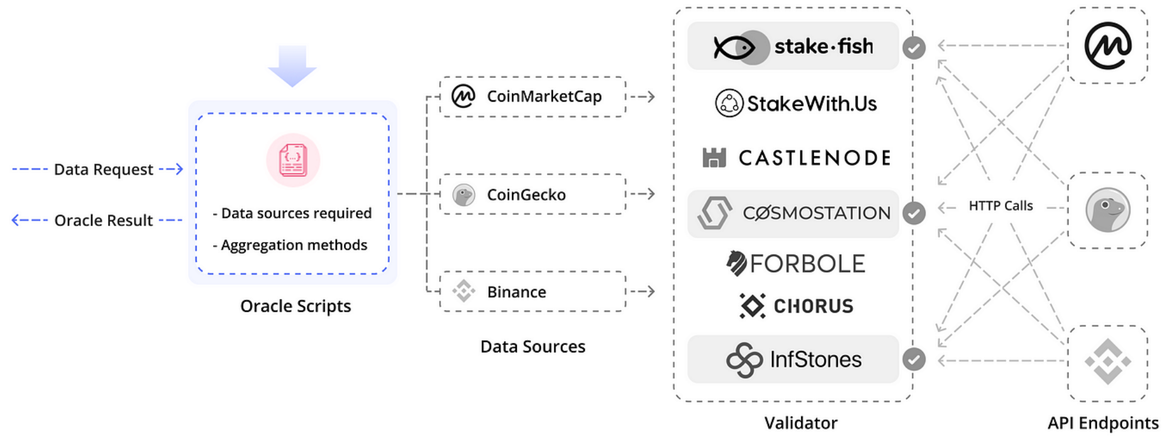

```
{
  "height": "7440523",
  "txhash": "FEDE0E7482CA6AB3A08E4643B2ADA03B0E6E961EE8747F41A1BF891BEDFE3C23",
  "data": "0A220A202F6F7261636C652E76312E4D73674372656174654F7261636C65536372697074",
  "rawLog": "[{\"events\":[{\"type\":\"create_oracle_script\",\"attributes\":[{\"key\":\"id\",\"value\"
  "logs": [
    {
      "events": [
        {
          "type": "create_oracle_script",
          "attributes": [
            {
              "key": "id",
              "value": "202"
            }
          ]
        },
        {
          "type": "message",
          "attributes": [
            {
              "key": "action",
              "value": "/oracle.v1.MsgCreateOracleScript"
            }
          ]
        }
      ]
    }
  ],
  "gasWanted": "250000",
  "gasUsed": "246278"
}
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Data Source
Oracle Script
Introduction
Creating an Oracle Script
Deployment
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Introduction

When someone wants to request data from BandChain's oracle, they do not interact or call the data sources directly. Instead, they call an oracle script, which then proceeds to execute the necessary data sources.

An oracle script's job, unlike a data source, is to be responsible for compiling the results from various data sources to enable on-chain security. As such, an oracle script, similarly to a smart contract on other platforms such as Ethereum, Near and Solana, is executed on-chain rather than off-chain.



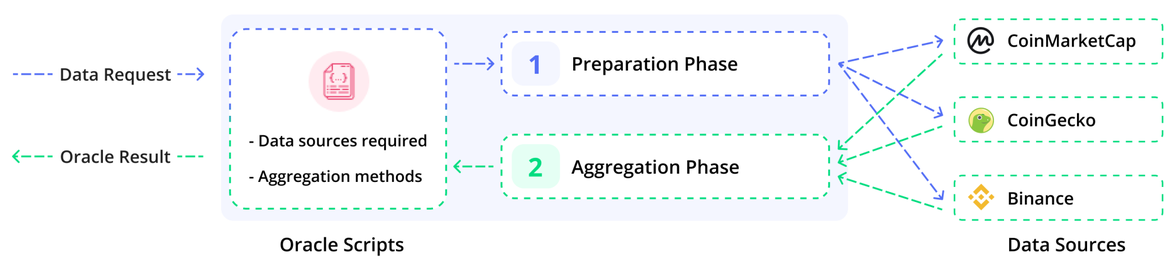
Oracle Script Specification

At a basic level, an oracle script itself is just an executable program that encodes two pieces of data:

- A set of raw data requests to the data sources it requires
- A method of aggregating those the raw data reports from the requests sent into a final result

Oracle scripts themselves are Turing-complete and can be written in any programming language that supports compilation into WebAssembly code. This composability and Turing-completeness make oracle scripts very similar to smart contracts.

Oracle Script Execution Flow



1. **Preparation Phase** During the preparation phase, the oracle script outlines the data sources that are required for its execution. It then sends out a request to BandChain's validators to retrieve the result from the required data sources. The content of this request consists of the data sources' execution steps and the parameters required by said data sources.
2. **Aggregation Phase** During the aggregation phase, the oracle script aggregates all the data reports returned by the validators. Each report contains the values which the validator received from said data sources. The script then proceeds to combine those values into a single final result. Note that the specifics of the aggregation process is entirely up to the design of the oracle script. BandChain does not enforce any regulations regarding the aggregation method used and entirely leaves that design decision to the creator of the script. Instead of a typical plain medianizer, the oracle script can encode custom conditions such as a data deviation rule to ensure that all data points returned will stay within certain percentage deviation from each other and to revert the transaction if otherwise.

👉 For more information on oracle scripts and its execution, please refer to the corresponding page on our wiki.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Data Source
Oracle Script
Introduction
Creating an Oracle Script
Deployment
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Creating an Oracle Script

In this section, we will take a look at how to create an oracle script.

Prerequisites

Rust Installation

While there are many ways to install Rust on your system. The official and recommended way to install Rust is using Rustup.

Rustup installs `rustc`, `cargo`, `rustup` and other standard tools to Cargo's bin directory. On Unix it is located at `$HOME/.cargo/bin` and on Windows at `%USERPROFILE%\.cargo\bin`. This is the same directory that cargo install will install Rust programs and Cargo plugins.

After installing Rust you can check the current version by typing `rustc --version` or `rustc -V` on your terminal to verify the success of the installation.

Note: If wasm32-unknown-unknown hasn't been added as a target, you can add it using the command below.

```
rustup target add wasm32-unknown-unknown
```

Writing the Oracle Script

File structure

Let's start by creating a Rust directory structure like in the example below.

```
.
├── hello_world
    ├── Cargo.toml
    └── src
        └── lib.rs
```

Adding Dependencies

As `Cargo.toml` is the manifest file for Rust's package manager: Cargo, this file contains metadata such as the name, version and dependencies of the package. By default, Cargo checks dependencies on crates.io. Therefore, when adding a crate, we only need to add the crate name and version to the `Cargo.toml`.

When creating an oracle script, two main dependencies are required: owasm-kit and obi.

An example is shown below:

```
[package]
name = "hello-world"

version = "0.1.0"
authors = ["Band Protocol <dev@bandprotocol.com>"]
edition = "2018"

[lib]
crate-type = ["cdylib"]

[dependencies]
owasm-kit = { version = "0.1.13" }
obi = { version = "0.0.2" }
```

Writing the Oracle Script

As mentioned in the introduction, an oracle script execution flow can be categorized into two main phases, the preparation phase and the execution phase. However, we also do need to define the oracle scripts input and outputs.

Input/Output

An oracle script's input and output can be defined in a struct. In the example below, we can see that this specific oracle scripts takes in an input `repeat` as a `u64` and returns an output `response` as a `string`

```
#[derive(OBIDecode, OBISchema)]
struct Input {
    repeat: u64,
}

#[derive(OBIEncode, OBISchema)]
struct Output {
    response: String,
}
```

Preparation Phase

The function below shows an example preparation phase for requesting data from data source `0327`. As `0327` does not require any inputs, an empty byte will be passed. However, in other data sources that do require an input, the corresponding calldata should be sent instead.

```
const DATA_SOURCE_ID: i64 = 327;
const EXTERNAL_ID: i64 = 0;

#[no_mangle]
fn prepare_impl(_input: Input) {
    oei::ask_external_data(
        EXTERNAL_ID,  // The assigned external ID for this data source
        DATA_SOURCE_ID,  // The data source to call by ID
        b"",  // Calldata to be sent to the data source
    )
}
```

Execution Phase

The function below shows an example of the execution phase for the data received from `0327`. This example retrieves the data reports and duplicates the majority result of the data report `x` times where `x` is defined by `repeat` as given in the input.

```
#[no_mangle]
fn execute_impl(input: Input) -> Output {
    let raw_result = ext::load_input::<String>(EXTERNAL_ID);  // Raw results from the given external ID
    let result: Vec<String> = raw_result.collect();
    let majority_result: String = ext::stats::majority(result).unwrap();  // Majority result
    Output {
        response: majority_result.repeat(input.repeat as usize),
    }
}
```

lib.rs

```
use obi::{OBIDecode, OBIEncode, OBISchema};
use owasm_kit::{execute_entry_point, prepare_entry_point, oei, ext};

#[derive(OBIDecode, OBISchema)]
struct Input {
    repeat: u64,
}

#[derive(OBIEncode, OBISchema)]
struct Output {
    response: String,
}

const DATA_SOURCE_ID: i64 = 327;
const EXTERNAL_ID: i64 = 0;

#[no_mangle]
fn prepare_impl(_input: Input) {
    oei::ask_external_data(
        EXTERNAL_ID,
        DATA_SOURCE_ID,
        b"",
    )
}

#[no_mangle]
fn execute_impl(input: Input) -> Output {
    let raw_result = ext::load_input::<String>(EXTERNAL_ID);
    let result: Vec<String> = raw_result.collect();
    let majority_result: String = ext::stats::majority(result).unwrap();
    Output {
        response: majority_result.repeat(input.repeat as usize),
    }
}

prepare_entry_point!(prepare_impl);
execute_entry_point!(execute_impl);
```

Compling the Oracle Script

To compile the oracle script, the following command can be run

```
RUSTFLAGS='-C link-arg=-s' cargo build --release --target wasm32-unknown-unknown
```

After the compilation is complete, the `.wasm` file can be found in the sub-directory: `./target/wasm32-unknown-unknown/release/*.wasm`.

More Examples

Below is another example of an oracle script that queries a token's total supply.

Query for token total supply

lib.rs

```
use obi::{OBIDecode, OBIEncode, OBISchema};
use owasm_kit::{execute_entry_point, ext, oei, prepare_entry_point};

#[derive(OBIDecode, OBISchema)]
struct Input {
    rpc: String,
    to: String,
}

#[derive(OBIEncode, OBISchema)]
struct Output {
    total_supply: String,
}

const DATA_SOURCE_ID: i64 = 98;
const EXTERNAL_ID: i64 = 0;

#[no_mangle]
fn prepare_impl(input: Input) {
    oei::ask_external_data(
        EXTERNAL_ID,
        DATA_SOURCE_ID,
        format!("{} {}", input.rpc, input.to).as_bytes(),
    );
}

#[no_mangle]
fn execute_impl(_input: Input) -> Output {
    Output {
        total_supply: ext::load_majority::<String>(EXTERNAL_ID).unwrap(),
    }
}

prepare_entry_point!(prepare_impl);
execute_entry_point!(execute_impl);
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Data Source
Introduction
Creating a Data Source
Deployment
Oracle Script
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

Deployment

On BandChain, a data source can be registered into the system by anyone. This is done through the registrant sending a `MsgCreateDataSource` message to the chain.

A `MsgCreateDataSource` message contains various parameters of the data source that is to be registered. These parameters include:

- `name`: Name of the data source.
- `description`: A description of the data source .
- `executable`: Contents of the executable to be run by block validators upon receiving a data request for this data source.
- `fee`: A stipulated per-query fee that those who use this data source are required to pay.
- `treasury`: A treasury address of which the stipulated fee is to be paid to.
- `owner`: The owner address of this data source. The owner will have edit rights. If omitted, the data source parameters will no longer be able to be edited after being registered.
- `sender`: The message sender account.

In order to send a `MsgCreateDataSource` message, we can use either bandchain.js or pyband

An example on how to send a `MsgCreateDataSource` message via **BandChain.js** can be seen below.

```
import { Client, Wallet, Message, Coin, Transaction, Fee } from '@bandprotocol/bandchain.js'
import fs from 'fs'
import path from 'path'

// Setup the client
const grpcURL = 'https://laozi-testnet6.bandchain.org/grpc-web'
const client = new Client(grpcURL)

async function createDataSource() {
  // Setup the wallet
  const { PrivateKey } = Wallet
  const mnemonic = process.env.MNEMONIC
  const privateKey = PrivateKey.fromMnemonic(mnemonic)
  const publicKey = privateKey.toPubkey()
  const sender = publicKey.toAddress().toAccBech32()

  // Setup the transaction's properties
  const chainId = await client.getChainId()
  const execPath = path.resolve(__dirname, 'hello_world.py')
  const file = fs.readFileSync(execPath, 'utf8')
  const executable = Buffer.from(file).toString('base64')

  let feeCoin = new Coin()
  feeCoin.setDenom('uband')
  feeCoin.setAmount('50000')

  const requestMessage = new Message.MsgCreateDataSource(
    'Hello World!', // Data source name
    executable, // Data source executable
    sender, // Treasury address
    sender, // Owner address
    sender, // Sender address
    [feeCoin], // Fee
    '' // Data source description
  )

  // Construct the transaction
  const fee = new Fee()
  fee.setAmountList([feeCoin])
  fee.setGasLimit(60000)

  const txn = new Transaction()
  txn.withMessages(requestMessage)
  await txn.withSender(client, sender)
  txn.withChainId(chainId)
  txn.withFee(fee)
  txn.withMemo('')

  // Sign the transaction
  const signDoc = txn.getSignDoc(publicKey)
  const signature = privateKey.sign(signDoc)
  const txRawBytes = txn.getTxData(signature, publicKey)

  // Broadcast the transaction
  const sendTx = await client.sendTxBlockMode(txRawBytes)

  return sendTx
}

;(async () => {
  console.log(await createDataSource())
})()
```

An example on how to send a `MsgCreateDataSource` message via pyband can also be seen below.

```
import os

from pyband import Client, Transaction
from pyband.wallet import PrivateKey
from pyband.proto.cosmos.base.v1beta1.coin_pb2 import Coin
from pyband.proto.oracle.v1.tx_pb2 import MsgCreateDataSource
from google.protobuf.json_format import MessageToJson


def main():
    # Setup the client
    grpc_url = "laozi-testnet6.bandchain.org"
    c = Client(grpc_url)

    # Setup the wallet
    mnemonic = os.getenv("MNEMONIC")
    private_key = PrivateKey.from_mnemonic(mnemonic)
    public_key = private_key.to_public_key()
    sender_addr = public_key.to_address()
    sender = sender_addr.to_acc_bech32()

    # Prepare the transaction's properties
    deploy_msg = MsgCreateDataSource(
        name="Hello World!",
        description="",
        executable=open("hello_world.py", "rb").read(),
        fee=[Coin(amount="0", denom="uband")],
        treasury=sender,
        owner=sender,
        sender=sender,
    )

    account = c.get_account(sender)
    account_num = account.account_number
    sequence = account.sequence

    fee = [Coin(amount="50000", denom="uband")]
    chain_id = c.get_chain_id()

    # Construct the transaction
    txn = (
        Transaction()
        .with_messages(deploy_msg)
        .with_sequence(sequence)
        .with_account_num(account_num)
        .with_chain_id(chain_id)
        .with_gas(60000)
        .with_fee(fee)
    )

    # Sign the Transaction
    sign_doc = txn.get_sign_doc(public_key)
    signature = private_key.sign(sign_doc.SerializeToString())
    tx_raw_bytes = txn.get_tx_data(signature, public_key)

    # Broadcast the transaction
    tx_block = c.send_tx_block_mode(bytes(tx_raw_bytes))

    print(MessageToJson(tx_block))


if __name__ == "__main__":
    main()
```

After the transaction is successfully broadcasted. The newly created data source ID can be found in the response json. The registrant can also view the created data source details on CosmoScan. An example of a successful transaction will return a response similar to the one shown below.


```
{
  "height": "7232244",
  "txhash": "B7A039E4AE9567A7380B5241B72FF69D38DB6831BE5023F05140616AD71FFA62",
  "data": "0A200A1E2F6F7261636C652E76312E4D73674372656174654461746153 6F75726365",
  "rawLog": "[{\"events\":[{\"type\":\"create_data_source\",\"attributes\":[{\"key\":\"id\",\"value\":\'
  "logs": [
    {
      "events": [
        {
          "type": "create_data_source",
          "attributes": [
            {
              "key": "id",
              "value": "327"
            }
          ]
        },
        {
          "type": "message",
          "attributes": [
            {
              "key": "action",
              "value": "/oracle.v1.MsgCreateDataSource"
            }
          ]
        }
      ]
    }
  ],
  "gasWanted": "100000",
  "gasUsed": "55306"
}
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Band Standard Dataset
Band VRF
Introduction
Getting Started
VRF integration
Example Use Cases
VRF Supported Blockchains
Pricefeed Module
cw-band
Develop
Nodes & Validators
Staking

VRF Supported Blockchains

`VRFprovider` and `Bridge` contracts have reviewed and audited by PeckShield

Audit report

Our deployed `Bridge`, `VRFProvider`, `MockVRFConsumer` and `VRFLens` contracts on each chain are shown in the table below.

Mainnet

Chain \ Contract	Bridge	VRFProvider	
BSC	0xB1bc084DFdf558C1DFE9332c955ffC60807Db350	0xaEC7D640E4F1970615e685286202486c26b7Bcb3	0x0753
Cronos	0xB1bc084DFdf558C1DFE9332c955ffC60807Db350	0xaEC7D640E4F1970615e685286202486c26b7Bcb3	0x0cc0

Testnet

Chain \ Contract	Bridge	VRFProvider	
Goerli	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0xBCA1F17f6c01FA81f214F0e11e76e85C2261188c	0x
Cronos	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0x4BB1773b0e784cFEc78C152b1F78c4B5Dcb9D12A	0x
OKC	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0xb9EB7Dc3E79Dc98E78ecd067064D77a2cB67401e	0x
Oasis	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0x6c51E9a7680244F7ed20aeE24E7055D28DA91969	0x
BSC	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0x72d1dFBb367326DFCd919B9E52755AB3687126B4	0x
OP Goerli	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0xABde7B7A97D532E01bf988d39Ce1638A56c9b2b0	0x
Polygon	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0x3006cbB2d33AcE7D2e390D722367B0D82081AD24	0x
Avalanche	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0x16665448a08f68D82215CCFdceF88A9ba1589Ae7	0x
Shibuya	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0x14919325f2d97a05d146b7b4c9374b265e722f00	0x
Klaytn	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0xf1f3554b6f46d8f172c89836fbed1ea8551eabad	0x
Godwoken	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0xF1F3554b6f46D8f172c89836FBeD1ea8551eabad	0x
TCGverse	0xFCb11e9560F9a646C0A917De27C9600cDf210A19	0xF22bA22A57d387F3F55B4d7643092338cCDf99D5	0x
Meter	0xb865B52A5DC922C6cb7926B234725F37716Ed97d	0x1cF350DA842D4816c2978691D93e4670EEd7e10D	0x

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Developer Guides
Developer Tools
Building on BandChain
Supported Blockchains
API Endpoints
Band CLI & REST Endpoints
Nodes & Validators
Staking

API Endpoints

Laozi Mainnet

Type	Value
Chain ID	laozi-mainnet
RPC	http://rpc.laozi1.bandchain.org
REST	https://laozi1.bandchain.org/api/
GRPC	https://laozi1.bandchain.org
GRPC_WEB	https://laozi1.bandchain.org/grpc-web

Laozi Testnet 6

Type	Value
Chain ID	band-laozi-testnet6
RPC	https://rpc.laozi-testnet6.bandchain.org
REST	https://laozi-testnet6.bandchain.org/api/
GRPC	https://laozi-testnet6.bandchain.org
GRPC_WEB	https://laozi-testnet6.bandchain.org/grpc-web
Faucet	https://laozi-testnet6.bandchain.org/faucet

Band CLI & REST Endpoints

You can get the RPC and REST endpoints from here.

Get Data Source by ID

CLI: `bandd query oracle data-source [id] --node ${RPC}`

Path: `/api/oracle/v1/data_sources/{id}` **Method**: `GET`

```
$ bandd query oracle data-source 1 --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/data_sources/1"
{
  "data_source": {
    "owner": "band1m5lq9u533qaya4q3nfyl6ulzqkpkhge9q8tpzs",
    "name": "d1",
    "description": "d1",
    "filename": "c56de9061a78ac96748c83e8a22330accf6ee8ebb499c8525613149a70ec49d0",
    "treasury": "band1m5lq9u533qaya4q3nfyl6ulzqkpkhge9q8tpzs",
    "fee": [
      {
        "denom": "uband",
        "amount": "1"
      }
    ]
  }
}
```

Get Oracle Script by ID

CLI: `bandd query oracle-script [id] --node ${RPC}`

Path: `/api/oracle/v1/oracle_scripts/{id}` **Method**: `GET`

```
$ bandd query oracle oracle-script 37 --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/oracle_scripts/37"
{
  "oracle_script": {
    "owner": "band1lv90l7xf3jyneh046clyfujz2se6xkqgcvzs3p",
    "name": "Band Standard Dataset (Crypto)",
    "description": "",
    "filename": "f0b7d894b25eca60e8519ca3d37b25ba1df2b01058ccf81ee825fa68293fcf16",
    "schema": "{symbols:[string],multiplier:u64}/{rates:[u64]}",
    "source_code_url": ""
  }
}
```

Get Data Oracle Request by ID

CLI: `bandd query request [id] --node ${RPC}`

Path: `/api/oracle/v1/requests/{id}` **Method**: `GET`

```
$ bandd query oracle request 238769 --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/requests/238769"
{
  "request": {
    "oracle_script_id": "37",
    "calldata": "AAAAAQAAAANCVEMAAAAAAA9CQA==",
    "requested_validators": [
      "bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus",
      "bandvaloper1zrl8gmuj3vug7qy7yazzaenl25fvd3s3ussk40"
    ],
    "min_count": "2",
    "request_height": "358324",
    "request_time": "1632326855",
    "client_id": "from_scan",
    "raw_requests": [
      {
        "external_id": "6",
        "data_source_id": "61",
        "calldata": "QlRD"
      },
      {
        "external_id": "0",
        "data_source_id": "57",
        "calldata": "QlRD"
      },
      {
        "external_id": "3",
        "data_source_id": "62",
        "calldata": "QlRD"
      },
      {
        "external_id": "5",
        "data_source_id": "60",
        "calldata": "aHVvYmlwcm8gQlRD"
      },
      {
        "external_id": "2",
        "data_source_id": "59",
        "calldata": "QlRD"
      },
      {
        "external_id": "4",
        "data_source_id": "60",
        "calldata": "YmluYW5jZSBCVEM="
      },
      {
        "external_id": "9",
        "data_source_id": "60",
        "calldata": "Yml0dHJleCBCVEM="
      },
      {
        "external_id": "7",
        "data_source_id": "60",
        "calldata": "a3Jha2VuIEJUQw=="
      },
      {
        "external_id": "8",
        "data_source_id": "60",
        "calldata": "Yml0ZmluZXggQlRD"
      },
      {
        "external_id": "1",
        "data_source_id": "58",
        "calldata": "QlRD"
      }
    ],
    "ibc_channel": null,
    "execute_gas": "50000"
  },
  "reports": [
    {
      "validator": "bandvaloper1zrl8gmuj3vug7qy7yazzaenl25fvd3s3ussk40",
      "in_before_resolve": true,
      "raw_reports": [
        {
          "external_id": "0",
          "exit_code": 0,
          "data": "NDMyODEuMTQ0NQo="
        },
        {
          "external_id": "1",
          "exit_code": 0,
          "data": "NDMyMTIuMjEK"
        },
        {
          "external_id": "6",
          "exit_code": 0,
          "data": "NDMxOTEuOAo="
        },
        {
          "external_id": "3",
          "exit_code": 0,
          "data": "NDMyMDUuMjY0OQo="
        },
        {
          "external_id": "2",
          "exit_code": 0,
          "data": "NDM0MjgK"
        },
        {
          "external_id": "9",
          "exit_code": 0,
          "data": "NDMxNzUuMjA3Cg=="
        },
        {
          "external_id": "8",
          "exit_code": 0,
          "data": "NDMxMDYuMAo="
        },
        {
          "external_id": "7",
          "exit_code": 0,
          "data": "NDMxNzAuMgo="
        },
        {
          "external_id": "4",
          "exit_code": 0,
          "data": "NDMzOTMuNDQ3OAo="
        },
        {
          "external_id": "5",
          "exit_code": 0,
          "data": "NDM1OTUuMDk0Ngo="
        }
      ]
    },
    {
      "validator": "bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus",
      "in_before_resolve": true,
      "raw_reports": [
        {
          "external_id": "2",
          "exit_code": 0,
          "data": "NDM0NDAK"
        },
        {
          "external_id": "3",
          "exit_code": 0,
          "data": "NDMyMDUuMjY0OQo="
        },
        {
          "external_id": "0",
          "exit_code": 0,
          "data": "NDMyODEuMTQ0NQo="
        },
        {
          "external_id": "1",
          "exit_code": 0,
          "data": "NDMyMTIuMjEK"
        },
        {
          "external_id": "7",
          "exit_code": 0,
          "data": "NDMxNzAuMgo="
        },
        {
          "external_id": "6",
          "exit_code": 0,
          "data": "NDMxOTEuOAo="
        },
        {
          "external_id": "8",
          "exit_code": 0,
          "data": "NDMxMDYuMAo="
        },
        {
          "external_id": "9",
          "exit_code": 0,
          "data": "NDMxNzUuMjA3Cg=="
        },
        {
          "external_id": "5",
          "exit_code": 0,
          "data": "NDMzNzkuMjk3NAo="
        },
        {
          "external_id": "4",
          "exit_code": 0,
          "data": "NDMzNjQuMzkzMgo="
        }
      ]
    }
  ],
  "result": {
    "client_id": "from_scan",
    "oracle_script_id": "37",
    "calldata": "AAAAAQAAAANCVEMAAAAAAA9CQA==",
    "ask_count": "2",
    "min_count": "2",
    "request_id": "249802",
    "ans_count": "2",
    "request_time": "1632326855",
    "resolve_time": "1632326863",
    "resolve_status": "RESOLVE_STATUS_SUCCESS",
    "result": "AAAAAQAAAAoPcQKq"
  }
}
```

Get Data Requests by Query Info

Path: `/api/oracle/v1/request_search?oracle_script_id={oracleScriptID}&calldata={calldataHex}&min_count={minCount}&ask_count={askCount}` **Method**: `GET`

```
$ curl -X GET "${REST}/oracle/v1/request_search?oracle_script_id=37&calldata=000000010000000342544300000
{
  "request": {
    "request": {
      "oracle_script_id": "37",
      "calldata": "AAAAAQAAAANCVEMAAAAAAA9CQA==",
      "requested_validators": [
        "bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus",
        "bandvaloper1zrl8gmuj3vug7qy7yazzaenl25fvd3s3ussk40"
      ],
      "min_count": "2",
      "request_height": "358324",
      "request_time": "1632326855",
      "client_id": "from_scan",
      "raw_requests": [
        {
          "external_id": "6",
          "data_source_id": "61",
          "calldata": "QlRD"
        },
        {
          "external_id": "0",
          "data_source_id": "57",
          "calldata": "QlRD"
        },
        {
          "external_id": "3",
          "data_source_id": "62",
          "calldata": "QlRD"
        },
        {
          "external_id": "5",
          "data_source_id": "60",
          "calldata": "aHVvYmlwcm8gQlRD"
        },
        {
          "external_id": "2",
          "data_source_id": "59",
          "calldata": "QlRD"
        },
        {
          "external_id": "4",
          "data_source_id": "60",
          "calldata": "YmluYW5jZSBCVEM="
        },
        {
          "external_id": "9",
          "data_source_id": "60",
          "calldata": "Yml0dHJleCBCVEM="
        },
        {
          "external_id": "7",
          "data_source_id": "60",
          "calldata": "a3Jha2VuIEJUQw=="
        },
        {
          "external_id": "8",
          "data_source_id": "60",
          "calldata": "Yml0ZmluZXggQlRD"
        },
        {
          "external_id": "1",
          "data_source_id": "58",
          "calldata": "QlRD"
        }
      ],
      "ibc_channel": null,
      "execute_gas": "50000"
    },
    "reports": [
      {
        "validator": "bandvaloper1zrl8gmuj3vug7qy7yazzaenl25fvd3s3ussk40",
        "in_before_resolve": true,
        "raw_reports": [
          {
            "external_id": "0",
            "exit_code": 0,
            "data": "NDMyODEuMTQ0NQo="
          },
          {
            "external_id": "1",
            "exit_code": 0,
            "data": "NDMyMTIuMjEK"
          },
          {
            "external_id": "6",
            "exit_code": 0,
            "data": "NDMxOTEuOAo="
          },
          {
            "external_id": "3",
            "exit_code": 0,
            "data": "NDMyMDUuMjY0OQo="
          },
          {
            "external_id": "2",
            "exit_code": 0,
            "data": "NDM0MjgK"
          },
          {
            "external_id": "9",
            "exit_code": 0,
            "data": "NDMxNzUuMjA3Cg=="
          },
          {
            "external_id": "8",
            "exit_code": 0,
            "data": "NDMxMDYuMAo="
          },
          {
            "external_id": "7",
            "exit_code": 0,
            "data": "NDMxNzAuMgo="
          },
          {
            "external_id": "4",
            "exit_code": 0,
            "data": "NDMzOTMuNDQ3OAo="
          },
          {
            "external_id": "5",
            "exit_code": 0,
            "data": "NDM1OTUuMDk0Ngo="
          }
        ]
      },
      {
        "validator": "bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus",
        "in_before_resolve": true,
        "raw_reports": [
          {
            "external_id": "2",
            "exit_code": 0,
            "data": "NDM0NDAK"
          },
          {
            "external_id": "3",
            "exit_code": 0,
            "data": "NDMyMDUuMjY0OQo="
          },
          {
            "external_id": "0",
            "exit_code": 0,
            "data": "NDMyODEuMTQ0NQo="
          },
          {
            "external_id": "1",
            "exit_code": 0,
            "data": "NDMyMTIuMjEK"
          },
          {
            "external_id": "7",
            "exit_code": 0,
            "data": "NDMxNzAuMgo="
          },
          {
            "external_id": "6",
            "exit_code": 0,
            "data": "NDMxOTEuOAo="
          },
          {
            "external_id": "8",
            "exit_code": 0,
            "data": "NDMxMDYuMAo="
          },
          {
            "external_id": "9",
            "exit_code": 0,
            "data": "NDMxNzUuMjA3Cg=="
          },
          {
            "external_id": "5",
            "exit_code": 0,
            "data": "NDMzNzkuMjk3NAo="
          },
          {
            "external_id": "4",
            "exit_code": 0,
            "data": "NDMzNjQuMzkzMgo="
          }
        ]
      }
    ],
    "result": {
      "client_id": "from_scan",
      "oracle_script_id": "37",
      "calldata": "AAAAAQAAAANCVEMAAAAAAA9CQA==",
      "ask_count": "2",
      "min_count": "2",
      "request_id": "249802",
      "ans_count": "2",
      "request_time": "1632326855",
      "resolve_time": "1632326863",
      "resolve_status": "RESOLVE_STATUS_SUCCESS",
      "result": "AAAAAQAAAAoPcQKq"
    }
  }
}
```

Get Data Content by Hash

Path: `/oracle/v1/data/{data_hash}` **Method**: `GET`

```
$ curl -X GET "${REST}/oracle/v1/data/32ee6262d4a615f2c3ca0589c1c1af79212f24823453cb3f4cfff85b8d338045"
{
  "data": "IyEvdXNyL2Jpbi9lbnYgcHl0aG9uMwoKaW1wb3J0IGpzb24KaW1wb3J0IHVybGxpYi5yZXF1ZXN0CmltcG9ydCBzeXMK
}
```

Get Active Status of a Validator

CLI: `bandd query oracle validator [validator-address] --node ${RPC}`

Path: `/api/oracle/v1/validators/{validator-address}` **Method**: `GET`

```
$ bandd query oracle validator bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus --node ${RPC}
$ curl -X GET ${REST}/oracle/v1/validators/bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus
{
  "status": {
    "is_active": true,
    "since": "2021-09-07T14:52:54.598085215Z"
  }
}
```

Check Grant of Account on This Validator

Path: `/api/oracle/v1/reporter/{validator_address}/{reporter_address}` **Method**: `GET`

```
$ curl -X GET "${REST}/oracle/v1/reporter/bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus/band1rgcrk
{
  "is_reporter": true
}
```

Get Active Validators

CLI: `bandd query oracle active-validators --node ${RPC}`

Path: `/api/oracle/v1/active_validators` **Method**: `GET`

```
$ bandd query oracle active-validators --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/active_validators"
{
  "validators": [
    {
      "address": "bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus",
      "power": "10132920646"
    },
    {
      "address": "bandvaloper1ajna89zrh4u2kuvc6qqyak7fn9dqphlhuvg0jk",
      "power": "10111034000"
    },
    {
      "address": "bandvaloper1nlepx7xg53fsy6vslrss6adtmtl8a33kusv7fa",
      "power": "10110700071"
    },
    {
      "address": "bandvaloper1274qgg28xkz6f3upx05ftr9zepgmtfgts392dy",
      "power": "10109100000"
    },
    {
      "address": "bandvaloper1jengg99ssg9xq9dycemt782syyr4wwdn4d68s7",
      "power": "10103500000"
    },
    {
      "address": "bandvaloper1zkf9qzs7ayf3uqksxqwve8q693dsdhxk800wvw",
      "power": "10101000000"
    },
    ...
  }
}
```

Get Pending Requests

bandd: `bandd query oracle pending-requests [validator-address]`

Path: `/api/oracle/v1/pending_requests` **Method**: `GET`

```
$ bandd query oracle pending-requests bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/pending_requests/bandvaloper1kfj48adjsnrgu83lau6wc646q2uf65rf84tzus"
{
  "request_ids": [
    "249764"
  ]
}
```

Get Latest Price on Standard Price Reference Database

Path: `/oracle/v1/request_prices?ask_count={askCount}&min_count={minCount}&symbols={symbol-1}&symbols={symbol-2}&...&symbols={symbols-n}` **Method**: `GET`

```
$ curl -X GET "${REST}/oracle/v1/request_prices?symbols=BTC&symbols=ETH&ask_count=4&min_count=3"
{
  "price_results": [
    {
      "symbol": "BTC",
      "multiplier": "100",
      "px": "4822795",
      "request_id": "235866",
      "resolve_time": "1631766659"
    },
    {
      "symbol": "ETH",
      "multiplier": "100",
      "px": "303321",
      "request_id": "245820",
      "resolve_time": "1632163802"
    }
  ]
}
```

Get BandChain Oracle Governance Parameters

CLI: `bandd query oracle params --node ${RPC}`

Path: `/oracle/v1/params` **Method**: `GET`

```
$ bandd query oracle params --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/params"
{
  "params": {
    "max_raw_request_count": "12",
    "max_ask_count": "16",
    "max_calldata_size": "256",
    "max_report_data_size": "512",
    "expiration_block_count": "100",
    "base_owasm_gas": "20000",
    "per_validator_request_gas": "30000",
    "sampling_try_count": "3",
    "oracle_reward_percentage": "70",
    "inactive_penalty_duration": "600000000000",
    "ibc_request_enabled": true
  }
}
```

Get BandChain Oracle Counts

CLI: `bandd query oracle counts --node ${RPC}`

Path: `/oracle/v1/counts` **Method**: `GET`

```
$ bandd query oracle counts --node ${RPC}
$ curl -X GET "${REST}/oracle/v1/counts"
{
  "data_source_count": "98",
  "oracle_script_count": "61",
  "request_count": "249073"
}
```

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Joining Mainnet
Getting Started
Installation
Become a Validator
Joining Testnet
Yoda
Staking

Getting Started

> **NOTE**
> This guide includes full instructions for joining the mainnet either as an archive/full node or a pruned node

Overview

- Hardware Requirements
- Setup Node
- Setup Cosmovisor
- Sync Options
  - Block Sync
  - State Sync
  - Snapshot - ChainLayer
  - Snapshot - HighStakes
- Setup daemon service
- Setup Yoda

Hardware Requirements

You have to have at least 32 GB of RAM and 4 CPU Cores to run a node in BandChain Laozi mainnet.

Note: Storage size for validators will depend on the level of pruning.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Joining Mainnet
Joining Testnet
Getting Started
Installation
Become a Validator
Yoda
Staking

Installation

Step 1: Node Installation

Step 1.1: Node Configuration

This step provides procedures to install BandChain's executable and sync blocks with other peers.

Assuming to run on Ubuntu 22.04 LTS allowing connection on port 26656 for P2P connection.

Before beginning instructions, the following variables should be set to be used in further instructions. **Please make sure that these variables are set every time when using the new shell session.**

```
# Chain ID of testnet
export CHAIN_ID="band-laozi-testnet6"
# Wallet name to be used as validator's account, please change this into your name (no whitespace).
export WALLET_NAME=<YOUR_WALLET_NAME>
# Name of your validator node, please change this into your name.
export MONIKER=<YOUR_MONIKER>
# Seed and persistent peers for P2P communication
export SEEDS="da61931cbbbb2b62dbe7c470d049126cf365d257@35.213.165.61:26656,fffd730672f04d5dc065fa9afce2e
# URL of genesis file for Laozi testnet
export GENESIS_FILE_URL=https://raw.githubusercontent.com/bandprotocol/launch/master/band-laozi-testnet6
# Data sources/oracle scripts files
export BIN_FILES_URL=https://raw.githubusercontent.com/bandprotocol/launch/master/band-laozi-testnet6/fi
# Faucet endpoint
export FAUCET_URL=https://laozi-testnet6.bandchain.org/faucet
```

Step 1.2: Build BandChain Core

The following applications are required to build and run the BandChain node.

- make, gcc, g++ (can be obtained from the build-essential package on linux)
- wget, curl for downloading files

```
# install required tools
sudo apt-get update && \
sudo apt-get upgrade -y && \
sudo apt-get install -y build-essential curl wget jq
```

- Install Go 1.19.1

```
# Install Go 1.19.1
wget https://go.dev/dl/go1.19.1.linux-amd64.tar.gz
tar xf go1.19.1.linux-amd64.tar.gz
sudo mv go /usr/local/go

# Set Go path to $PATH variable
echo "export PATH=$PATH:/usr/local/go/bin:~/go/bin" >> $HOME/.profile
source ~/.profile
```

Go binary should be at `/usr/local/go/bin` and any executable compiled by go install command should be at `~/go/bin`

Step 1.3: Clone & Install BandChain Laozi

```
cd ~
# Clone BandChain Laozi version v2.5.1
git clone https://github.com/bandprotocol/chain
cd chain
git fetch && git checkout v2.5.1

# Install binaries to $GOPATH/bin
make install
```

Step 1.4: Initialize the BandChain and download the genesis file

```
cd $HOME

# Initialize configuration and genesis state
bandd init --chain-id $CHAIN_ID "$MONIKER"

# Replace genesis file with our genesis file
wget $GENESIS_FILE_URL -O $HOME/.band/config/genesis.json

# Download data sources / oracle scripts files, and store in $HOME/.band/files
wget -qO- $BIN_FILES_URL | tar xvz -C $HOME/.band/

# Create new account
bandd keys add $WALLET_NAME
```

Step 1.5: Setup seeds and minimum gas price

```
# Add seeds to config.toml
sed -E -i \
  "s/seeds = \".*\"/seeds = \"${SEEDS}\"/" \
  $HOME/.band/config/config.toml

# Add minimum gas price
sed -E -i \
  "s/^minimum-gas-prices *=.*/minimum-gas-prices = \"0.0025uband\"/" \
  $HOME/.band/config/app.toml
```

Step 2: Setup Cosmovisor

PRE-REQUISITE READINGS
- Cosmoviser

Cosmovisor is a small process manager for Cosmos SDK application binaries that monitors the governance module via `stdout` for incoming chain upgrade proposals

Step 2.1: Setup environment variables

Add required environment variables for Cosmovisor into your profile

```
cd ~
echo "export DAEMON_NAME=bandd" >> ~/.profile
echo "export DAEMON_HOME=$HOME/.band" >> ~/.profile
source ~/.profile
```

Step 2.2: Install and provide binaries

Install Cosmovisor and provide bandd binary to Cosmovisor

```
# Install Cosmovisor
go install github.com/cosmos/cosmos-sdk/cosmovisor/cmd/cosmovisor@v1.0.0

# Setup folder and provide bandd binary for Cosmovisor Genesis
mkdir -p $HOME/.band/cosmovisor/genesis/bin
mkdir -p $HOME/.band/cosmovisor/upgrades
cp $HOME/go/bin/bandd $HOME/.band/cosmovisor/genesis/bin

# Setup folder and provide bandd binary for Cosmovisor Upgrades
mkdir -p $HOME/.band/cosmovisor/upgrades/v2_5/bin
cp $HOME/go/bin/bandd $DAEMON_HOME/cosmovisor/upgrades/v2_5/bin
```

Step 3: Sync with the network

There are two main ways to sync a node on the BandChain; Blocksync and State Sync. However, we recommend using State Sync as it's faster.

Block Sync | State Sync

Block Sync

Blocksync is faster than traditional consensus and syncs the chain from genesis by downloading blocks and verifying against the Merkle tree of validators. For more information see Tendermint's Fastsync Docs

When syncing via Blocksync, node operators will need to provide the binary of each upgrade version for Cosmovisor to switch when it reaches the upgrade heights.

You can see the detail of genesis and each upgrade in the table below.

Upgrade name	Upgrade Height	Upgrade detail	Go version	Bandd version	Binary path
`genesis`	`0`	-	`1.16.7`	`v2.3.6`	`~/.band/cosmovisor/genesis/bin`
`v2_4`	`427000`	link	`1.19.1`	`v2.4.1`	`~/.band/cosmovisor/upgrades/v2_4/bin`
`v2_5`	`5557800`	link	`1.19.1`	`v2.5.1`	`~/.band/cosmovisor/upgrades/v2_5/bin`

Before doing the next step, you have to build and provide each correct bandd binary version to Cosmovisor in the binary path so that Cosmovisor can automatically switch it correctly.

Step 4: Setup daemon service

We do recommend running the Bandchain node as a daemon, which can be set up using `systemctl`.

Step 4.1: Create BandChain service

Run the following command to create a new daemon for `cosmovisor` that runs bandd (This script work on non-root user).

```
# Write bandd service file to /etc/systemd/system/bandd.service
export USERNAME=$(whoami)
sudo -E bash -c 'cat << EOF > /etc/systemd/system/bandd.service
[Unit]
Description=BandChain Node Daemon
After=network-online.target

[Service]
Environment="DAEMON_NAME=bandd"
Environment="DAEMON_HOME=${HOME}/.band"
Environment="DAEMON_RESTART_AFTER_UPGRADE=true"
Environment="DAEMON_ALLOW_DOWNLOAD_BINARIES=false"
Environment="UNSAFE_SKIP_BACKUP=true"
User=$USERNAME
ExecStart=${HOME}/go/bin/cosmovisor start
Restart=always
RestartSec=3
LimitNOFILE=4096

[Install]
WantedBy=multi-user.target
EOF'
```

Step 4.2: Register and start bandd service

In this step, we will register and start bandd service

```
# Register bandd to systemctl
sudo systemctl enable bandd
# Start bandd daemon
sudo systemctl start bandd
```

Once `bandd` service has been started, logs can be queried by running `journalctl -u bandd.service -f` command. You will see your node beginning to sync.

Step 5: Setup Yoda

PRE-REQUISITE READINGS
- Yoda

Since a subset of validators who are selected for a data request must send the data they received as a transaction of MsgReportData to BandChain.

Yoda is a program used by BandChain's validator nodes to help automatically query data from data providers by executing data source script, then submitting the result to fulfill the request. Read more on the Yoda section.

Step 5.1: Installation

Before setting up Yoda, the Lambda function executor need to be set up to execute data sources. If this step has not been done yet, please follow the instructions on the following pages (select either one of these methods):

- AWS Lambda Function
- Google Cloud Function

To check Yoda version, use the following command.

```
yoda version
# v2.5.1
```

Step 5.2: Set the Yoda configurations

Use the command below to config your Yoda, replacing `$VARIABLES` with their actual values.

```
rm -rf ~/.yoda # clear old config if exist
yoda config chain-id $CHAIN_ID
yoda config node http://localhost:26657
yoda config broadcast-timeout "5m"
yoda config rpc-poll-interval "1s"
yoda config max-try 5
yoda config validator $(bandd keys show $WALLET_NAME -a --bech val)
```

Then, add multiple reporter accounts to allow Yoda to submit transactions concurrently.

```
yoda keys add REPORTER_1
yoda keys add REPORTER_2
yoda keys add REPORTER_3
yoda keys add REPORTER_4
yoda keys add REPORTER_5
```

Lastly, configure the Lambda Executor endpoint to helps running data source scripts and return results to Yoda. More details about the executor can be found in this section.

```
export EXECUTOR_URL=<YOUR_EXECUTOR_URL>
yoda config executor "rest:${EXECUTOR_URL}?timeout=10s"
```

Step 5.3: Start Yoda

To start Yoda, it's also recommended to use `systemctl`.

```
# Write yoda service to /etc/systemd/system/yoda.service
export USERNAME=$(whoami)
sudo -E bash -c 'cat << EOF > /etc/systemd/system/yoda.service
[Unit]
Description=Yoda Daemon
After=network-online.target

[Service]
User=$USERNAME
ExecStart=/home/$USERNAME/go/bin/yoda run
Restart=always
RestartSec=3
LimitNOFILE=4096

[Install]
WantedBy=multi-user.target
EOF'
```

The first time running Yoda, you will need to register and start `yoda` services by running the following commands.

```
# Register yoda to systemctl
sudo systemctl enable yoda
# Start yoda daemon
sudo systemctl start yoda
```

After `yoda` service has been started, logs can be queried by running `journalctl -u yoda.service -f` command. The log should be similar to the following log example below. Once verified, you can stop tailing the log by typing `Control-C`.

```
... systemd[...]: Started Yoda Daemon.
... yoda[...]: I[...] ⭐  Creating HTTP client with node URI: tcp://localhost:26657
... yoda[...]: I[...] 🚀  Starting WebSocket subscriber
... yoda[...]: I[...] 👂  Subscribing to events with query: tm.event = 'Tx'...
```

Step 5.4: Wait for the latest blocks to be synced

It is imperative to exercise caution and allow adequate time for the newly started BandChain node to synchronize its blocks until it has reached the latest block. The latest block can be verified on CosmoScan.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Joining Mainnet
Getting Started
Installation
Become a Validator
Joining Testnet
Yoda
Staking

Become a Validator

This guide will show you how to register the running node as a validator. So that the program can fulfill the data on BandChain.

Step 1: Fund the Validator Account

```
bandd keys show $WALLET_NAME
```

Then fund tokens into this account ready for staking.

Step 2: Stake Tokens with the Validator Account

```
bandd tx staking create-validator \
    --amount 1000000uband \
    --commission-max-change-rate 0.01 \
    --commission-max-rate 0.2 \
    --commission-rate 0.1 \
    --from $WALLET_NAME \
    --min-self-delegation 1 \
    --moniker "$MONIKER" \
    --pubkey $(bandd tendermint show-validator) \
    --chain-id $CHAIN_ID
```

Registered validators can be found on CosmoScan.

Step 3: Register Reporters and Become Oracle Provider

Yoda contains multiple reporters. You will need to register the reporters to help the validator submit transactions of reporting data.

Firstly, reporter accounts must be created on BandChain by supplying a small amount of BAND tokens.

```
# Send 1uband from a wallet to each reporter.
bandd tx multi-send 1uband $(yoda keys list -a) \
  --from $WALLET_NAME \
  --chain-id $CHAIN_ID
```

Secondly, register reporters to the validator, so that oracle requests for validator can be assigned to the reporters.

```
bandd tx oracle add-reporters $(yoda keys list -a) \
  --from $WALLET_NAME \
  --chain-id $CHAIN_ID
```

Finally, activate the validator to become an oracle provider

```
bandd tx oracle activate \
  --from $WALLET_NAME \
  --chain-id $CHAIN_ID
```

If all procedures are successful, then the oracle provider status for the validator should be `active`.

```
bandd query oracle validator $(bandd keys show -a $WALLET_NAME --bech val)

# {
#   "is_active": true,
#   "since": ...
# }
```

And now you have become a validator on BandChain Laozi mainnet.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Joining Mainnet
Joining Testnet
Getting Started
Installation
Become a Validator
Yoda
Staking

Getting Started

> **NOTE**
> This guide includes full instructions for joining the testnet either as an archive/full node or a pruned node

Overview

- Hardware Requirements
- Setup Node
- Setup Cosmovisor
- Sync Options
  - Block Sync
  - State Sync
  - Snapshot - ChainLayer
  - Snapshot - HighStakes
- Setup daemon service
- Setup Yoda

Hardware Requirements

You have to have at least 16 GB of RAM and 4 CPU Cores to run a node in BandChain Laozi testnet.

Note: Storage size for validators will depend on the level of pruning.

Installation

Step 1: Node Installation

Step 1.1: Node Configuration

This step provides procedures to install BandChain's executable and sync blocks with other peers.

Assuming to run on Ubuntu 22.04 LTS allowing connection on port 26656 for P2P connection.

Before beginning instructions, the following variables should be set to be used in further instructions. **Please make sure that these variables are set every time when using the new shell session.**

```
# Chain ID of Laozi Mainnet
export CHAIN_ID=laozi-mainnet
# Wallet name to be used as validator's account, please change this into your name (no whitespace).
export WALLET_NAME=<YOUR_WALLET_NAME>
# Name of your validator node, please change this into your name.
export MONIKER=<YOUR_MONIKER>
# URL of genesis file for Laozi Mainnet
export GENESIS_FILE_URL=https://raw.githubusercontent.com/bandprotocol/launch/master/laozi-mainnet/genes
# Data sources/oracle scripts files
export BIN_FILES_URL=https://raw.githubusercontent.com/bandprotocol/launch/master/laozi-mainnet/files.ta
```

The following applications are required to build and run the BandChain node.

- make, gcc, g++ (can be obtained from the build-essential package on linux)
- wget, curl for downloading files

```
# install required tools
sudo apt-get update && \
sudo apt-get upgrade -y && \
sudo apt-get install -y build-essential curl wget jq
```

- Install Go 1.19.1

```
# Install Go 1.19.1
wget https://go.dev/dl/go1.19.1.linux-amd64.tar.gz
tar xf go1.19.1.linux-amd64.tar.gz
sudo mv go /usr/local/go

# Set Go path to $PATH variable
echo "export PATH=$PATH:/usr/local/go/bin:~/go/bin" >> $HOME/.profile
source ~/.profile
```

Go binary should be at /usr/local/go/bin and any executable compiled by go install command should be at ~/go/bin

Step 1.2: Clone & Install BandChain Laozi

```
cd ~
# Clone BandChain Laozi version v2.5.1
git clone https://github.com/bandprotocol/chain
cd chain
git fetch && git checkout v2.5.1

# Install binaries to $GOPATH/bin
make install
```

Step 1.3: Initialize the BandChain and download the genesis file

```
cd $HOME

# Initialize configuration and genesis state
bandd init --chain-id $CHAIN_ID "$MONIKER"

# Replace genesis file with our genesis file
wget $GENESIS_FILE_URL -O $HOME/.band/config/genesis.json

# Download data sources / oracle scripts files, and store in $HOME/.band/files
wget -qO- $BIN_FILES_URL | tar xvz -C $HOME/.band/

# Create new account
bandd keys add $WALLET_NAME
```

Step 1.4: Configure general settings

This can be done by editing fields in `$HOME/.band/config/config.toml` and `$HOME/.band/config/app.toml`.

For a list of seeds and peers, please see here.

For minimum-gas-prices, you can set any number but the recommendation is 0.0025uband (according to this proposal).

```
# List of seeds and persistent peers you want to add
# e.g. SEEDS="8d42bdcb6cced03e0b67fa3957e4e9c8fd89015a@34.87.86.195:26656,543e0cab9c3016a0e99775443a17b
export SEEDS="<SEED>,<SEED>,..."
export PERSISTENT_PEERS="<PERSISTENT_PEER>,<PERSISTENT_PEER>,..."

# Add seeds and persistent peers to config.toml
sed -E -i \
  "s/seeds = \".*\"/seeds = \"${SEEDS}\"/" \
  $HOME/.band/config/config.toml

sed -E -i \
  "s/persistent_peers = \".*\"/persistent_peers = \"${PERSISTENT_PEERS}\"/" \
  $HOME/.band/config/config.toml

# Add minimum gas price
sed -E -i \
  "s/^minimum-gas-prices *=.*/minimum-gas-prices = \"0.0025uband\"/" \
  $HOME/.band/config/app.toml
```

Step 2: Setup Cosmovisor

> PRE-REQUISITE READINGS
> - Cosmoviser

Cosmovisor is a small process manager for Cosmos SDK application binaries that monitors the governance module via `stdout` for incoming chain upgrade proposals

Step 2.1: Setup environment variables

Add required environment variables for Cosmovisor into your profile

```
cd ~
echo "export DAEMON_NAME=bandd" >> ~/.profile
echo "export DAEMON_HOME=$HOME/.band" >> ~/.profile
source ~/.profile
```

Step 2.2: Install and provide binaries

Install Cosmovisor and provide bandd binary to Cosmovisor

```
# Install Cosmovisor
go install github.com/cosmos/cosmos-sdk/cosmovisor/cmd/cosmovisor@v1.0.0

# Setup folder and provide bandd binary for Cosmovisor Genesis
mkdir -p $HOME/.band/cosmovisor/genesis/bin
mkdir -p $HOME/.band/cosmovisor/upgrades
cp $HOME/go/bin/bandd $HOME/.band/cosmovisor/genesis/bin

# Setup folder and provide bandd binary for Cosmovisor Upgrades
mkdir -p $HOME/.band/cosmovisor/upgrades/v2_5/bin
cp $HOME/go/bin/bandd $DAEMON_HOME/cosmovisor/upgrades/v2_5/bin
```

Step 3: Sync with the network

There are three main ways to sync a node on the BandChain; Blocksync, State Sync, and Snapshots.

> INFO
> It is highly recommended to use **State Sync** or **Snapshots** as it's faster. However, if you want to use Blocksync, please read the following section.

Block Sync | State Sync | Snapshot - ChainLayer | Snapshot - HighStakes

Block Sync

Blocksync is faster than traditional consensus and syncs the chain from genesis by downloading blocks and verifying against the Merkle tree of validators. For more information see Tendermint's Fastsync Docs

When syncing via Blocksync, node operators will need to provide the binary of each upgrade version for Cosmovisor to switch when it reaches the upgrade heights.

You can see the detail of genesis and each upgrade in the table below.

Upgrade name	Upgrade Height	Upgrade detail	Go version	Bandd version	Binary path
`genesis`	`0`	-	`1.16.7`	`v2.3.6`	`~/.band/cosmovisor/genesis/bin`
`v2_4`	`11525000`	link	`1.19.1`	`v2.4.1`	`~/.band/cosmovisor/upgrades/v2_4/bin`
`v2_5`	`16562500`	link	`1.19.1`	`v2.5.1`	`~/.band/cosmovisor/upgrades/v2_5/bin`

Before doing the next step, you have to build and provide each correct bandd binary version to Cosmovisor in the binary path so that Cosmovisor can automatically switch it correctly.

Step 4: Setup daemon service

We do recommend running the Bandchain node as a daemon, which can be set up using `systemctl`.

Step 4.1: Create BandChain service

Run the following command to create a new daemon for `cosmovisor` that runs bandd (This script work on non-root user).

```
# Write bandd service file to /etc/systemd/system/bandd.service
export USERNAME=$(whoami)
sudo -E bash -c 'cat << EOF > /etc/systemd/system/bandd.service
[Unit]
Description=BandChain Node Daemon
After=network-online.target

[Service]
Environment="DAEMON_NAME=bandd"
Environment="DAEMON_HOME=${HOME}/.band"
Environment="DAEMON_RESTART_AFTER_UPGRADE=true"
Environment="DAEMON_ALLOW_DOWNLOAD_BINARIES=false"
Environment="UNSAFE_SKIP_BACKUP=true"
User=$USERNAME
ExecStart=${HOME}/go/bin/cosmovisor start
Restart=always
RestartSec=3
LimitNOFILE=4096

[Install]
WantedBy=multi-user.target
EOF'
```

Step 4.2: Register and start bandd service

In this step, we will register and start bandd service

```
# Register bandd to systemctl
sudo systemctl enable bandd
# Start bandd daemon
sudo systemctl start bandd
```

Once `bandd` service has been started, logs can be queried by running `journalctl -u bandd.service -f` command. You will see your node beginning to sync.

Step 5: Setup Yoda

> PRE-REQUISITE READINGS
> - Yoda

Yoda is a program used by BandChain's validator nodes to help automatically query data from data providers by executing data source script, then submitting the result to fulfill the request.

Since a subset of validators who are selected for a data request must send the data they received as a transaction of MsgReportData to BandChain.

Step 5.1: Installation

Before setting up Yoda, the Lambda function executor need to be set up to execute data sources. If this step has not been done yet, please follow the instructions on the following pages (select either one of these methods):

- AWS Lambda Function
- Google Cloud Function

To check Yoda version, use the following command.

```
yoda version
# v2.5.1
```

Step 5.2: Set the Yoda configurations

Use the command below to config your Yoda, replacing `$VARIABLES` with their actual values.

```
rm -rf ~/.yoda # clear old config if exist
yoda config chain-id $CHAIN_ID
yoda config node http://localhost:26657
yoda config broadcast-timeout "5m"
yoda config rpc-poll-interval "1s"
yoda config max-try 5
yoda config validator $(bandd keys show $WALLET_NAME -a --bech val)
```

Then, add multiple reporter accounts to allow Yoda to submit transactions concurrently.

```
yoda keys add REPORTER_1
yoda keys add REPORTER_2
yoda keys add REPORTER_3
yoda keys add REPORTER_4
yoda keys add REPORTER_5
```

Lastly, configure the Lambda Executor endpoint to helps running data source scripts and return results to Yoda. More details about the executor can be found in this section.

```
export EXECUTOR_URL=<YOUR_EXECUTOR_URL>
yoda config executor "rest:${EXECUTOR_URL}?timeout=10s"
```

Step 5.3: Start Yoda

To start Yoda, it's also recommended to use `systemctl`.

```
# Write yoda service to /etc/systemd/system/yoda.service
export USERNAME=$(whoami)
sudo -E bash -c 'cat << EOF > /etc/systemd/system/yoda.service
[Unit]
Description=Yoda Daemon
After=network-online.target

[Service]
User=$USERNAME
ExecStart=/home/$USERNAME/go/bin/yoda run
Restart=always
RestartSec=3
LimitNOFILE=4096

[Install]
WantedBy=multi-user.target
EOF'
```

The first time running Yoda, you will need to register and start `yoda` services by running the following commands.

```
# Register yoda to systemctl
sudo systemctl enable yoda
# Start yoda daemon
sudo systemctl start yoda
```

After `yoda` service has been started, logs can be queried by running `journalctl -u yoda.service -f` command. The log should be similar to the following log example below. Once verified, you can stop tailing the log by typing `Control-C`.

```
... systemd[...]: Started Yoda Daemon.
... yoda[...]: I[...] ⭐  Creating HTTP client with node URI: tcp://localhost:26657
... yoda[...]: I[...] 🚀  Starting WebSocket subscriber
... yoda[...]: I[...] 👂  Subscribing to events with query: tm.event = 'Tx'...
```

Step 5.4: Wait for the latest blocks to be synced

It is imperative to exercise caution and allow adequate time for the newly started BandChain node to synchronize its blocks until it has reached the latest block. The latest block can be verified on CosmoScan.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Joining Mainnet
Joining Testnet
Getting Started
Installation
Become a Validator
Yoda
Staking

Become a Validator

This step provides procedures to register the node as a validator.

Step 1: Create a new Account to be Used as Validator

```
# Request new coins from faucet
curl --location --request POST "${FAUCET_URL}" \
--header 'Content-Type: application/json' \
--data-raw "{
 \"address\": \"$(bandd keys show $WALLET_NAME -a)\"
}"
```

Step 2: Stake Tokens with the Validator Account

```
bandd tx staking create-validator \
    --amount 1000000uband \
    --commission-max-change-rate 0.01 \
    --commission-max-rate 0.2 \
    --commission-rate 0.1 \
    --from $WALLET_NAME \
    --gas-prices 0.0025uband \
    --min-self-delegation 1 \
    --moniker "$MONIKER" \
    --pubkey $(bandd tendermint show-validator) \
    --chain-id $CHAIN_ID
```

After becoming a validator, the validator node will be shown on Block Explorer here.

Step 3: Register Reporters and Become Oracle Provider

Now, Yoda has multiple reporters. To grant the reporters be able to report data for the validator, the following commands should be run.

Firstly, reporter accounts must be created on Bandchain by supplying some small amount of BAND tokens.

```
# Send 1uband from a wallet to each reporter.
bandd tx multi-send 1uband $(yoda keys list -a) \
  --from $WALLET_NAME \
  --gas-prices 0.0025uband \
  --chain-id $CHAIN_ID
```

Secondly, grant all reporters for the validator, so that oracle requests for validator can be sent by the reporters.

```
bandd tx oracle add-reporters $(yoda keys list -a) \
  --from $WALLET_NAME \
  --gas-prices 0.0025uband \
  --chain-id $CHAIN_ID
```

Finally, activate the validator to become an oracle provider

```
bandd tx oracle activate \
  --from $WALLET_NAME \
  --gas-prices 0.0025uband \
  --chain-id $CHAIN_ID
```

If all procedures are successful, then the oracle provider status for the validator should be `active`.

```
bandd query oracle validator $(bandd keys show -a $WALLET_NAME --bech val)

# {
#   "is_active": true,
#   "since": ...
# }
```

And now you have become a validator on Bandchain Laozi testnet.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Yoda
Staking

Yoda

Yoda is a program that is used by BandChain's validator nodes to automatically fulfill data for oracle requests.

Since a subset of validators who are selected for a data request must return the data they received from running the specified data source(s), each of them have to send a `MsgReportData` transaction to BandChain in order to fulfill their duty.

Although the transaction can be sent manually by user, it is not convenient, and would be rather time-consuming. Furthermore, most data providers already have APIs that can be used to query data automatically by another software. Therefore, we have developed Yoda to help validators to automatically query data from data providers by executing data source script, then submit the result to fulfill the request.

Yoda's execution flow consists of the follwing steps

1. Listening on incoming oracle requests and filter them.
2. Loading data source script based on received requests.
3. Execute the data source to get a result from data provider.
4. Collect result into messages and store in local pending list.
5. Send transaction based on pending list to fulfill oracle requests.

Yoda usually run as a process alongside the main BandChain daemon process. It holds another wallet (private key) account called a `reporter account`, which is owned by validator account. The reporter account is used to help validator submit transactions of reporting data.

The reason of using reporter account instead of validator account is to maintain security of validator's private key. If Yoda's server is compromised, then reporter's private key may be exposed but not validator's private key. Moreover, most validators use hardware wallets, which is not designed for server that runs 24/7.

It's mechanism is designed as event-driven approach, which handle tasks concurrently to improve performance.

Yoda's diagram

The image above represents high-level mechanism of Yoda.

The following section describes mechanism of the Yoda in each step.

1. Subscribe to Events and Filter Only Oracle Requests that should be responsible to

Firstly, Yoda listen to newly created transaction events occurred on BandChain via Tendermint's RPC Websocket. Then, it is extracted to check whether there is any event of newly created oracle request inside the transaction. If the event is found, then it is filtered to only select request that is assigned to the validator that yoda is responsible for.

2. Extract Oracle Request Events and Load Data Source Scripts

After oracle request events have been filtered, they are extracted to collect list of data source and calldata that need to be executed for this request.

Then, We have to also provide information for verifying request, which are as follow

- Chain ID of the BandChain
- Validator address
- Oracle request ID
- Data source's external ID (indicated by request)
- Signature of above content signed by Yoda's reporter account
- Public key of Yoda's reporter account for verify signature

These information can be used by data provider to ensure that incoming requests do really need the result to fulfill oracle requests and not to be used elsewhere or by someone else other than assigned validators.

3. Execute Data Source on Executor

Yoda executes those data source scripts by sending their script files along with necessary information for verifying oracle request to Yoda's executor environment, which helps running data source scripts and returns results to Yoda. More information about data source executor can be found in this section.

4. Collect Results from Executor, Add to Local Pending List

Once results have been returned from Yoda's executor, Yoda constructs `MsgReportData` based on given answers and append to a pending message queue. As submitting a transaction takes some time and it is impossible to sign multiple transactions at the same time, the pending list can be accumulated.

5. Pack unsend ReportMsg(s) to a transaction and send to BandChain node

Once Yoda's reporter account is ready to sign a transaction, as a transaction can contains multiple messages, all messages in the pending list will be taken to construct the new transaction, then Yoda sign the transaction and send to BandChain to fulfill target oracle requests.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Staking
Introduction
Delegate
Undelegate
Redelegate
Claim Rewards
Reinvest
Staking FAQ

Introduction

What is staking

In a Proof-of-Stake (PoS) blockchain protocol, block creation is reliant on validators. These entities bond the native asset (in this case, BAND) to the network and take part in the consensus protocol by broadcasting cryptographic signatures, or votes, to agree upon the next block. In exchange for their contributions to the network, validators are awarded transaction fees. These fees are then distributed proportionately among the stakeholders who have bonded their BAND. Staking is an integral part of PoS protocols as it motivates honest participation in the network and bolsters the network's security.

Validators are required to operate and maintain a software referred to as a node (see Running a node for further details). However, you have the option to delegate your BAND to a validator and receive a portion of the staking rewards in line with your staked amount.

Risks of Slashing

There must be some penalty imposed on the validators for any intentional or unintentional deviation from the sanctioned protocol. Some evidence is immediately admissible, such as a double-sign at the same height and round, or a violation of "prevote-the-lock" (a rule of the Tendermint consensus protocol). Such evidence will result in the validator losing its good standing and its bonded atoms as well its proportionate share of tokens in the reserve pool – collectively called its "stake" – will get slashed.

The slashing mechanism is a fundamental component of PoS protocols, including the Band Protocol. It acts as a preventive measure against malicious activities and violations within the network. Validators, by staking their BAND tokens, are motivated to operate in the best interest of the network. However, if they contravene these expectations, they stand to lose a part or all of their staked tokens

There are two primary conditions under which a validator can get slashed.

1. Double Signing:

Double signing refers to the act of a validator signing two blocks at the same height on two different chains. This is considered a severe violation of the network protocol, and it can cause instability and uncertainty in the blockchain.

To deter this, Band Protocol imposes a significant penalty. If a validator is caught double signing, they face a slashing penalty of 5% of their staked and delegated BAND tokens.

What's critical to understand is that the penalty isn't limited to validators. If you've delegated your BAND tokens to a validator, and they engage in double signing, your staked tokens are also at risk. The 5% penalty also applies to the BAND tokens you've staked with that validator.

2. Downtime:

A robust and reliable network necessitates active and consistent participation from validators. Validators in the Band Protocol have a duty to be online and operational to validate transactions and add new blocks to the blockchain.

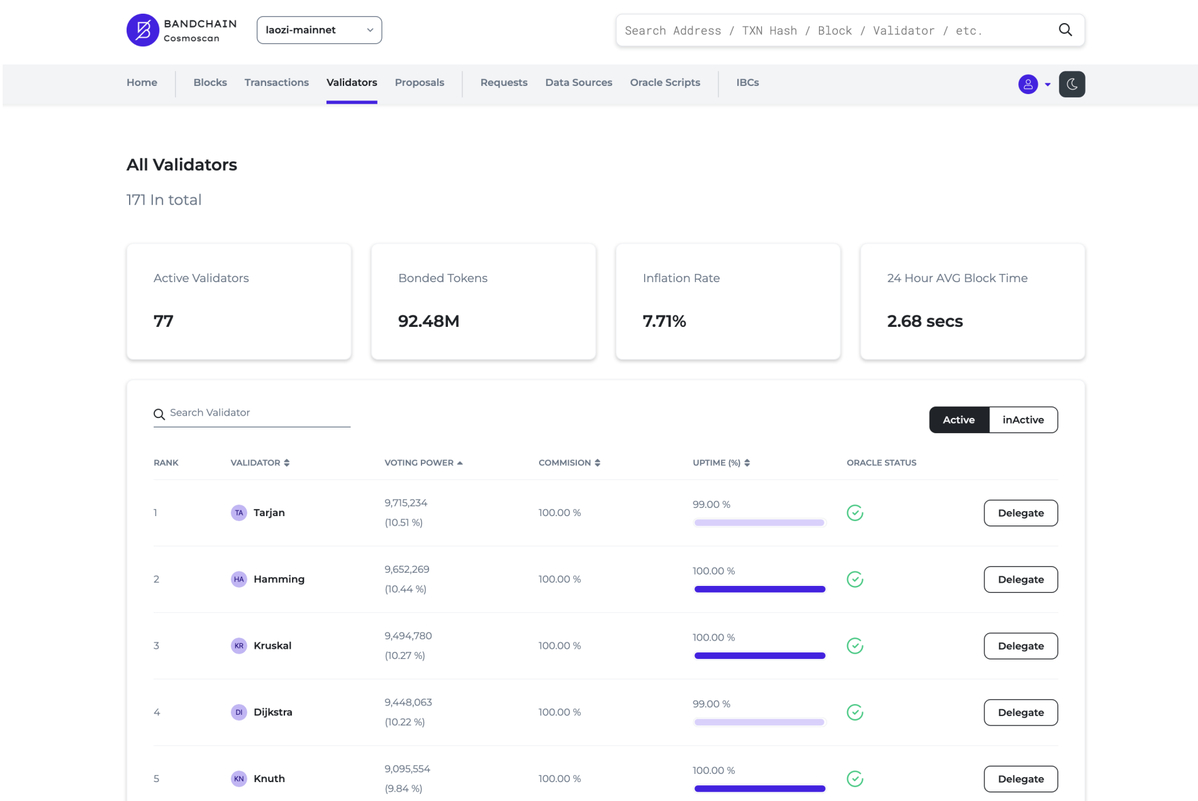
If a validator is offline and misses a certain number of blocks, in this case, 30,000 blocks, they face a downtime penalty. A slashing penalty of 0.01% is imposed, and the validator is put into **jail**. This means they are prevented from signing any further blocks until they can prove they're back online and operational.

Again, if you've delegated your tokens to a validator and they're jailed due to downtime, your staked tokens also face the 0.01% slashing penalty.

These slashing risks underline the importance of due diligence when choosing a validator. You need to assess their security measures, reliability, and track record to ensure you don't inadvertently put your staked tokens at risk. Moreover, it highlights the importance of understanding the technicalities and rules of the protocol before staking or delegating your BAND tokens.

How to choose a validator

On the Validators page, you will encounter a table that outlines various validators. This table is made up of 6 columns:



- **Column 1 — Rank:** Represents the sequence of validators based on the number of tokens bonded to each validator.
- **Column 2 — Validator:** Name of validator
- **Column 3 — Voting Power:** Denotes the number of tokens bonded to a particular validator.
- **Column 4 — Commission:** Every validator charge commission on the rewards earned by the delegators on their behalf.
- **Column 5 — Uptime:** Demonstrates the proportion of blocks a validator has been active within the last 100 blocks.
- **Column 6 — Oracle Status:** shows that is this validator working as an oracle provider or not.

There are three factors you should consider when choosing a validator:

1. **Voting Power:** It's critical to distribute Voting Power across all validators for network security. You can contribute to strengthening the BandChain network by choosing validators with lower voting power.
2. **Commission:** A lower commission implies that you'll retain more rewards.
3. **Uptime:** As we discussed earlier in the Staking Risks section, validators with lower uptime are at a greater risk of getting slashed.

In conclusion, the decision on which validator to stake with requires careful consideration of these factors.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Staking
Introduction
Delegate
Undelegate
Redelegate
Claim Rewards
Reinvest
Staking FAQ

Delegate

Participation in BandChain doesn't require you to be a validator. Instead, you can delegate BAND tokens to any validator and earn a share of the fees and rewards associated with block creation. However, it's important to note that staking with a validator involves certain risks. Make sure to familiarize yourself with the potential dangers related to validator staking

This tutorial will assist you in staking your BAND tokens easily and efficiently

Step 0 — Before you start

- Download and install the newest version of Ledger Live on your desktop machine.
- Update your Ledger device to the newest firmware.
- Install the latest version of the Cosmos App on your Ledger device.
- Have some BAND on BandChain mainnet

Step 1 — Navigate to Cosmoscan

To begin, assume that you have BAND on the BandChain, then navigate to the website cosmoscan.io. This website is our official block explorer and is a crucial tool for interacting with the BandChain.

Step 2 — Make sure you in the right site

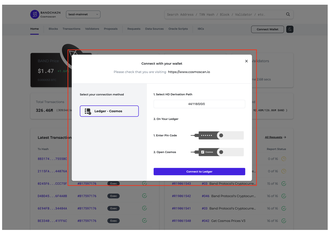
Once you're on the site, make sure the url is `https://www.cosmoscan.io` and ensure that you're operating on the mainnet. You can verify this by locating the network information on the top-left corner of the website interface. The mainnet is the primary network for transacting tokens.

Step 3 — Click to connect wallet

Next, you will need to connect your wallet to the website. click the button **Connect Wallet** on the right of navigation bar to initiate this process.

Step 4 — Connect with ledger

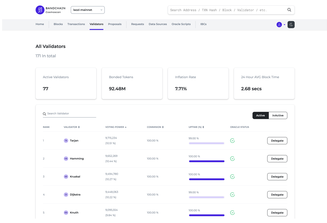
Upon clicking the **Connect Wallet** button, a pop-up window will appear. At the moment, our website only supports wallet connections via the Ledger device utilizing the Cosmos Application. Please ensure your Ledger device is properly set up, the Cosmos application is open on your device, and it's ready for the connection process.



Once you've successfully connected your Ledger device, you'll see an account button, as shown in the image below.

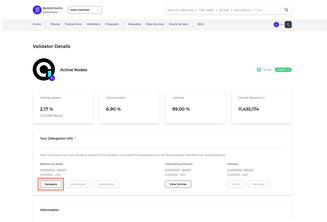
Step 5 — Navigate to Validators page

Proceed to the validators page by clicking on **Validator** located in the navigation bar. or navigate to Validators. Once on the validators page, you'll see a list of all validators that you can choose to delegate your tokens to.



Step 6 — Choose a validator

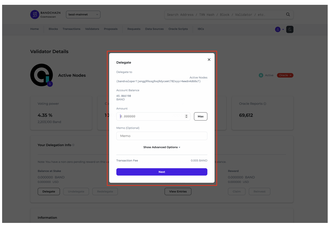
Choose a validator that you wish to delegate to. This decision should be based on various factors (check out How to choose a validator section) Upon clicking on a validator, you'll be presented with specific information about them. This typically includes their identity, performance statistics, commission rates, and more.



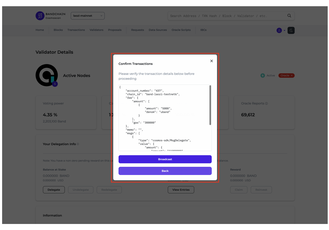
Step 7 — Delegate to validator

Scroll down to the **Your Delegation Info** section. This area provides detailed information about your current and past delegations with the selected validator. You'll notice a **Delegate** button within this section – click on it to initiate the delegation process.

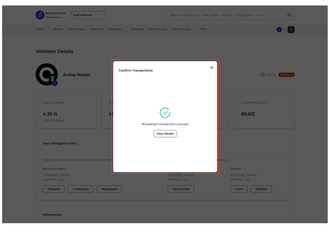
Upon clicking **Delegate**, a new window will pop up. Enter the amount of BAND tokens you wish to stake in the provided field, and then click **Next**.



At this point, you'll see transaction details for your delegation. Review the information, and if everything is ok, click **Broadcast**.

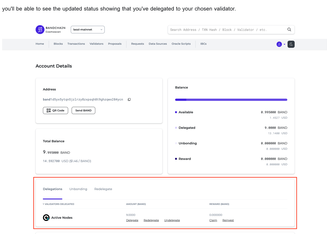


Voila! You have successfully delegated your BAND tokens. Congratulations on taking this step in staking and earning rewards on your tokens!



To verify your transaction, navigate back to your **Account Details** page. To view detailed information about your account, click the drop-down arrow, then click your address prefixed with **band** to navigate to your account details page.

you'll be able to see the updated status showing that you've delegated to your chosen validator.



Next — Manage your BAND stake

Once you have staked your BAND, you have 4 options to consider.

- Undelegate
- Redelegate
- Claim Rewards
- Reinvest

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Staking
Introduction
Delegate
Undelegate
Redelegate
Claim Rewards
Reinvest
Staking FAQ

Undelegate

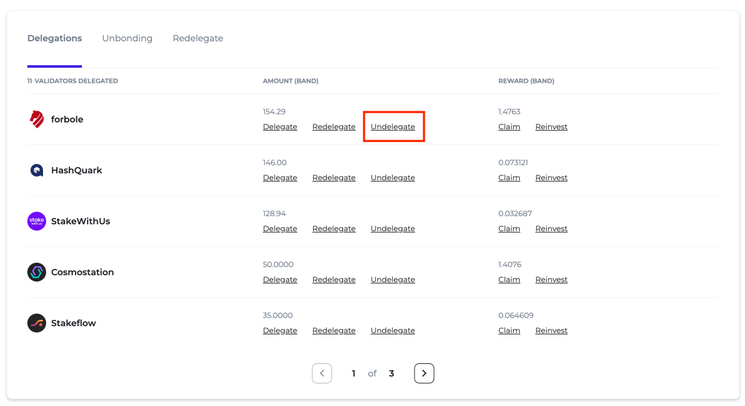
Once you delegate your BAND, you can later undelegate to receive your BAND back. However, there is a 21-day unbonding period before the BAND amount becomes available.

we will show you how to undelegate

> **INFO**
> This guide necessitates that you first familiarize yourself with the Delegate document before proceeding.

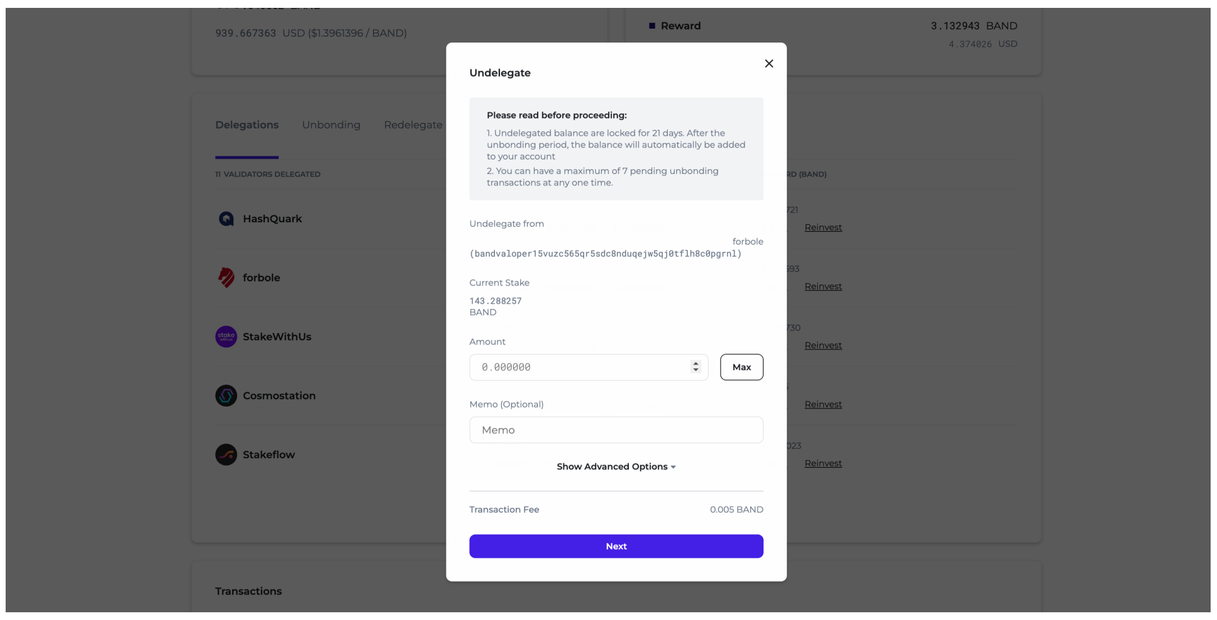
Step 1 — Select the Validator you want to undelegate

Navigate to your account details page. you will see the Delegation section. then click **Undelegate** on your selected validator



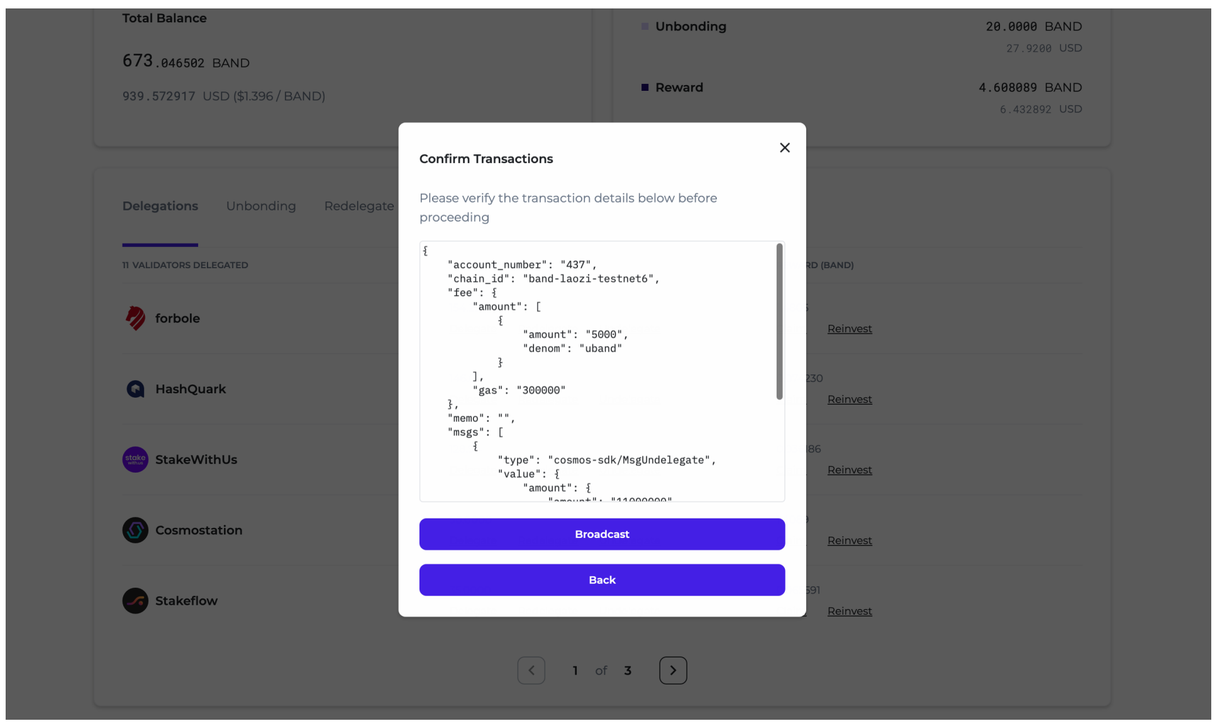
Step 2 — Enter BAND amount

Undelegate pop-up window will be shown. enter BAND amount that you want to undelegate and click **Next**

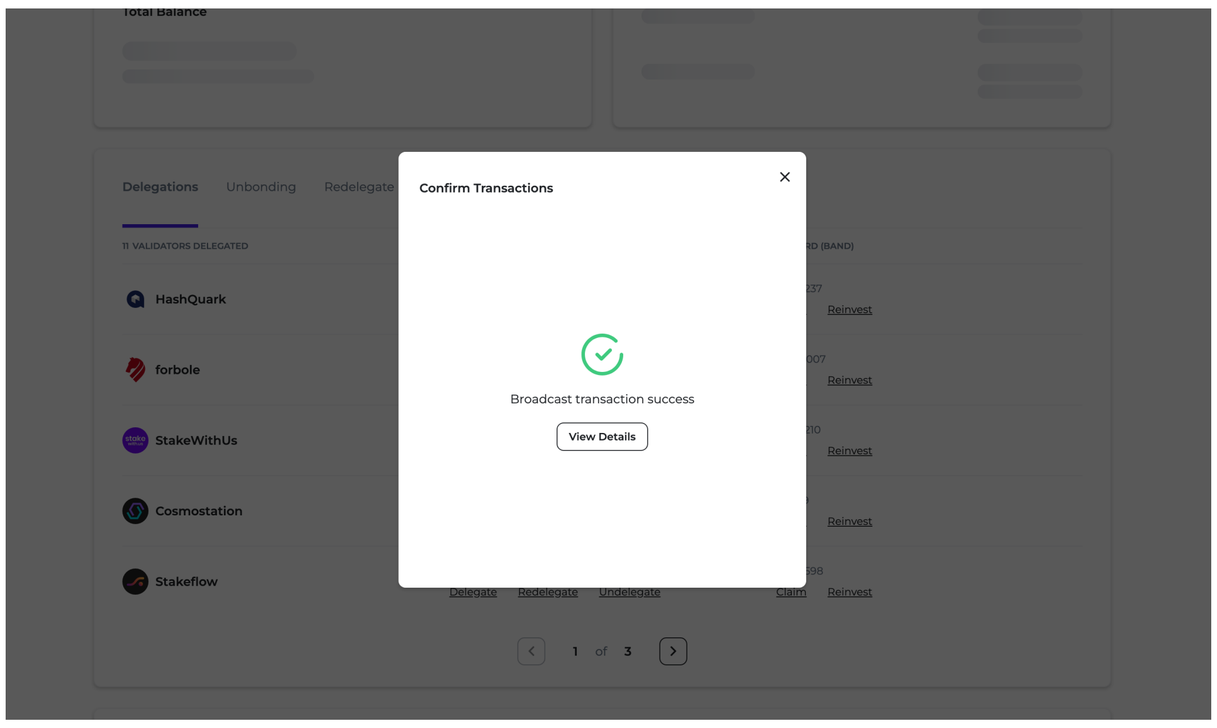


Step 3 — Check Transaction details

check if transaction detail is valid then click **Broadcast**

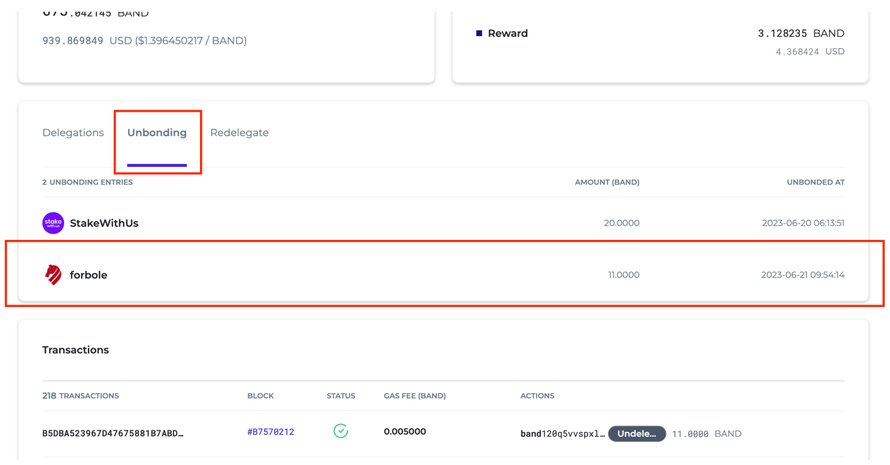


Well done! Your BAND has been successfully undelegated.



Step 4 — Check Unbonding tab

In the Unbonding tab, the date when your BAND will be available will be displayed.



Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Staking
Introduction
Delegate
Undelegate
Redelegate
Claim Rewards
Reinvest
Staking FAQ

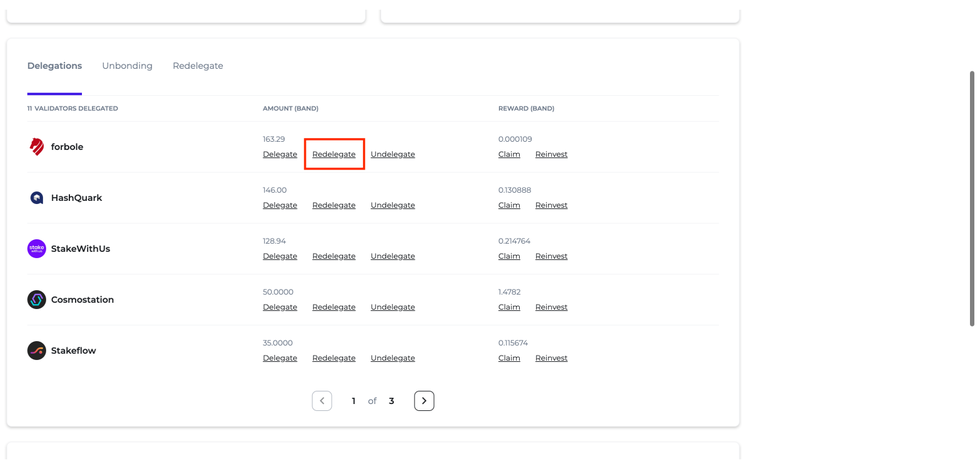
Redelegate

If you've already delegated your BAND tokens and then decide to switch validators, you have the option to **redelegate** your BAND to new validators.

> **INFO**
> This guide necessitates that you first familiarize yourself with the Delegate document before proceeding.

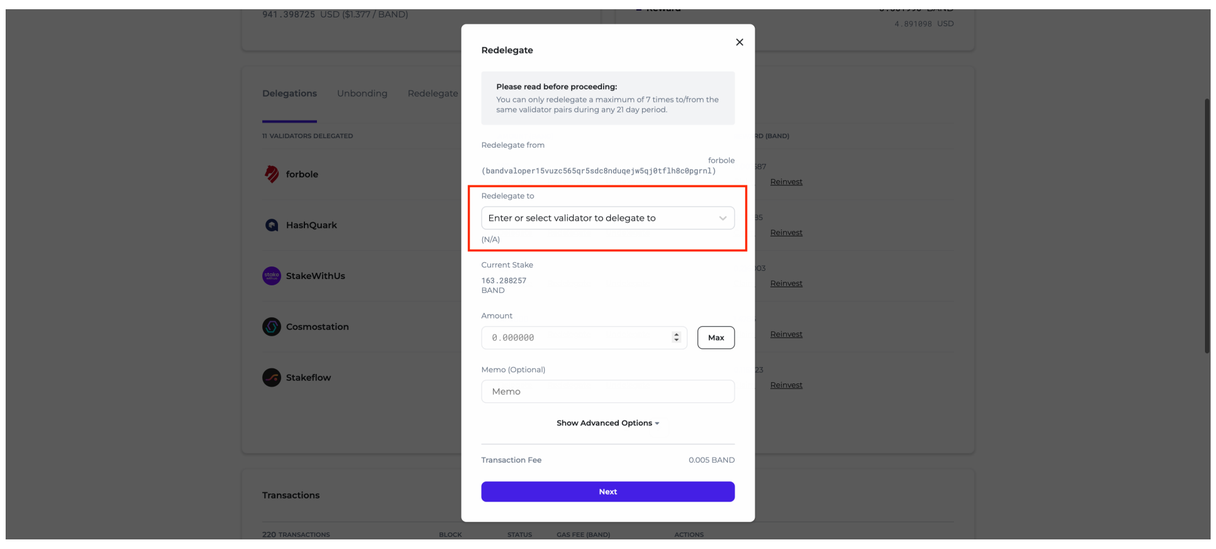
Step 1 — Select the Validator you want to redelegate

navigate to your account details page. you will see the section of Delegation. then click **Redelegate** on your selected validator

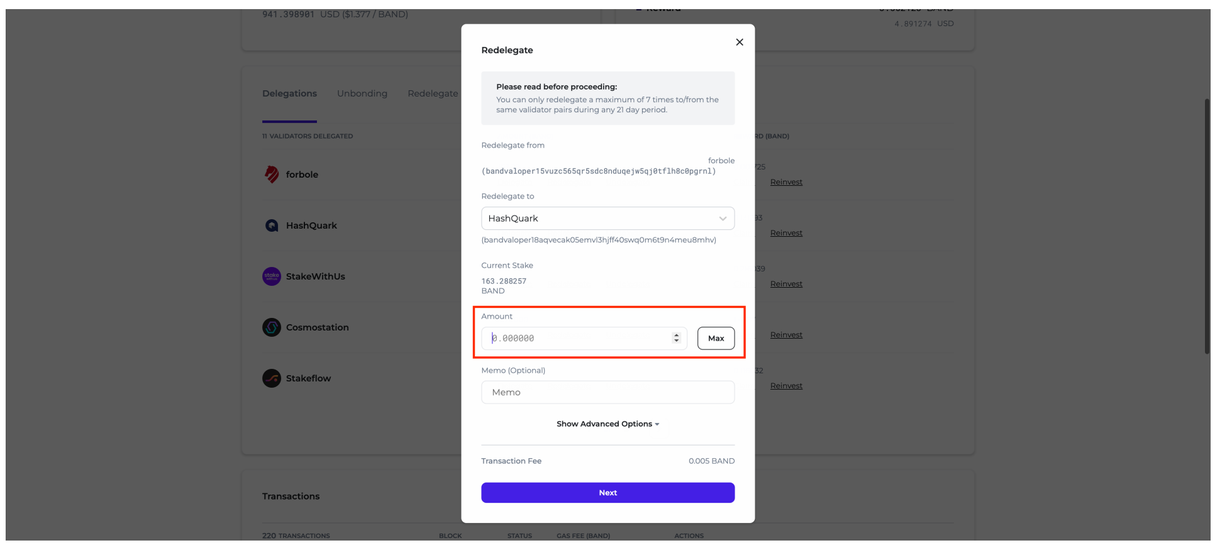


Step 2 — Select the Validator you want to delegate to

In the Redelegate pop-up window, you will see a dropdown menu labeled **Enter or select validator to delegate to** Click on the dropdown menu and select the new validator you want to delegate to.

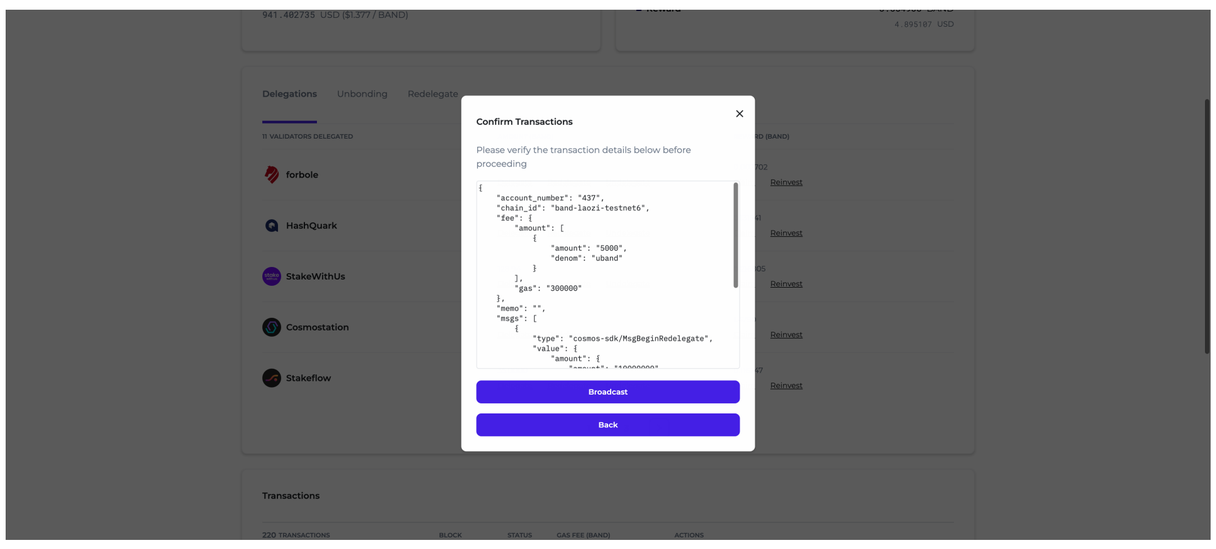


and Enter BAND amount you want to redalegate to and click Next to the next step

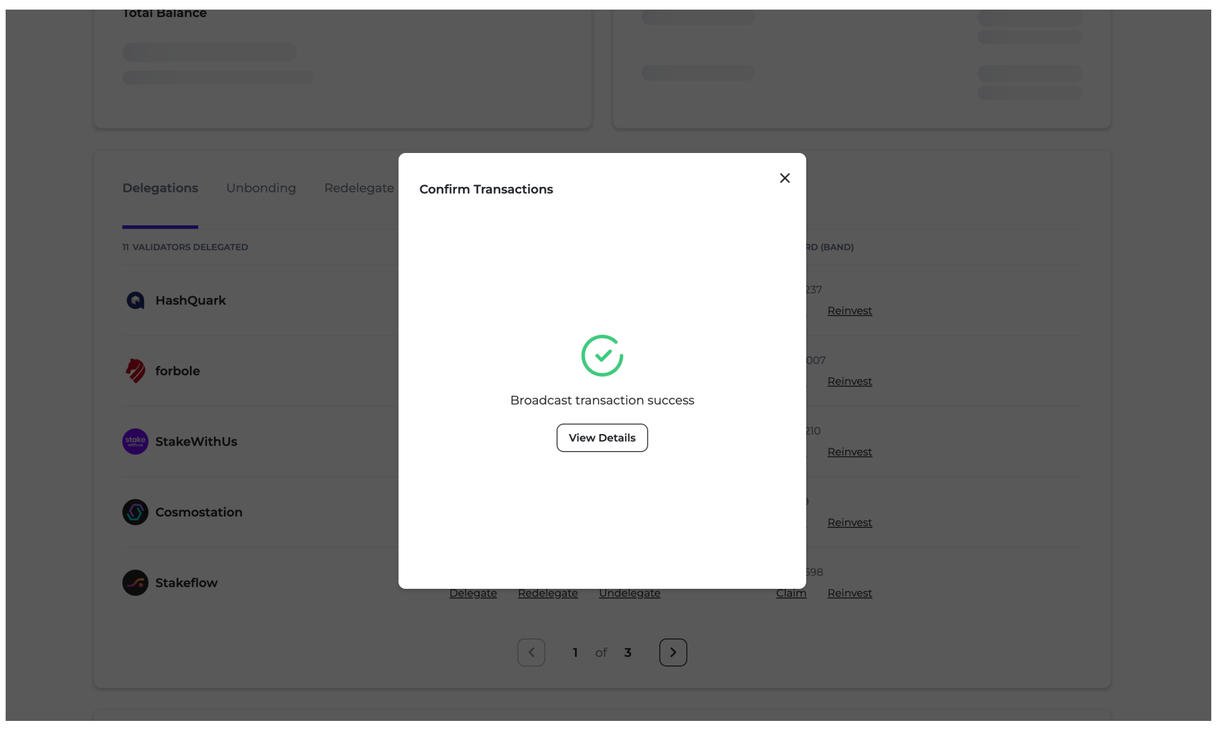


Step 3 — Check Transaction details

Check if transaction detail is valid then click **Broadcast**

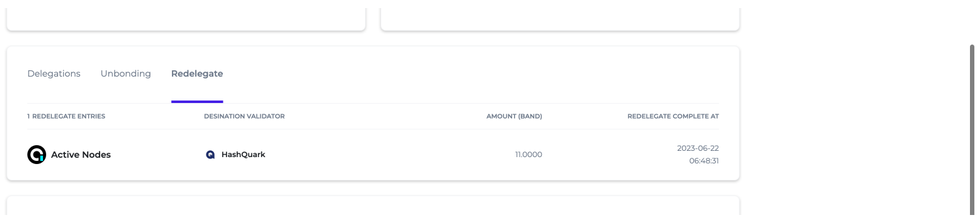


Congratulations! Your BAND has been successfully redelegated to the new validator



Step 4 — Check Redelegate tab

In the Redelegate tab, you can view the details of your redelegation.



Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Overview
Running a Node
Yoda
Staking

Overview

What is Node

A node in the context of the BandChain network refers to any computer equipped with the necessary software to interface with the network. These nodes, by connecting and communicating with one another, play a pivotal role in transmitting information, validating transactions, and maintaining the ledger's overall integrity. Although the terms 'node' and 'client' can be mistakenly interchanged, they represent two distinct components of the blockchain ecosystem. The 'node' denotes the connected computers, while the 'client' is the software running on these computers.

A specific type of node, known as a full-node, carries out more comprehensive tasks within the network. It does not only participate in the information exchange and transaction validation process; it stores and verifies the entire blockchain's state. By maintaining a complete copy of the blockchain, a full-node contributes significantly to the robustness, decentralization, and security of the entire network.

What is Validator

In the world of blockchain and decentralized finance, a validator plays a crucial role in maintaining the health, security, and smooth operation of the network. Essentially, a validator is a network participant that verifies and validates new transactions and adds them to the blockchain. They do this by running a full node that is constantly synced with the network and participating in the consensus protocol, which can vary from one blockchain to another. In proof-of-stake (PoS), validators are chosen based on the number of tokens they hold and are willing to **stake** as collateral.

Validators are responsible for performing two main functions on the network. First, they are responsible for proposing and committing new blocks to the blockchain. They participate in the block consensus protocol by broadcasting votes which contain cryptographic signatures signed by each validator's private key. This is similar to most other Cosmos-based delegated proof-of-stake blockchains.

Each validator will have a certain amount of BAND tokens bonded to them. The source of these tokens can either be their own holdings, or the tokens delegated to them by other token owners. In most cases, there will be a large number of parties with tokens staked to them. In that case, the top 100 validator candidates with the most token staked to them will become BandChain′s validators.

Running a Node

There are two networks you can join.If this is your initial experience operating a validator on BandChain, we advise you to begin by running a node on the testnet.

- Joining Mainnet
- Joining Testnet

An node is simply any computer running the software needed to connect with the BandChain network. Nodes connect with one another to send information back and forth to validate transactions and store data about the state of the blockchain.

Though the terms are often used interchangeably these connected computers are the nodes and the software that they run is called the client.

A full-node is a node that stores and verifies the entire state of a blockchain.

Band Protocol Documentation
Introduction
Core Concepts
Products
Develop
Nodes & Validators
Staking
Introduction
Delegate
Undelegate
Redelegate
Claim Rewards
Reinvest
Staking FAQ

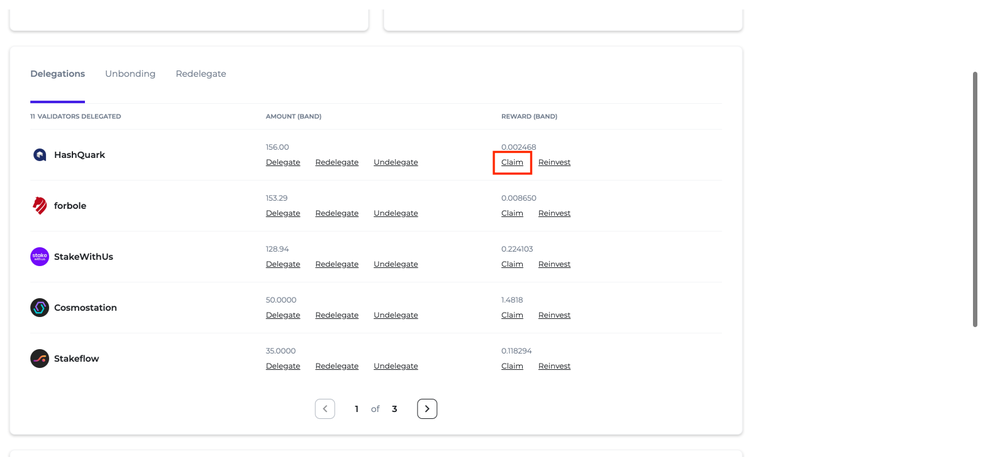
Claim Rewards

Delegator will get rewards for staking. This guide is intended to provide step-by-step instructions on how to claim these staking rewards.

> **INFO**
> This guide necessitates that you first familiarize yourself with the Delegate document before proceeding.

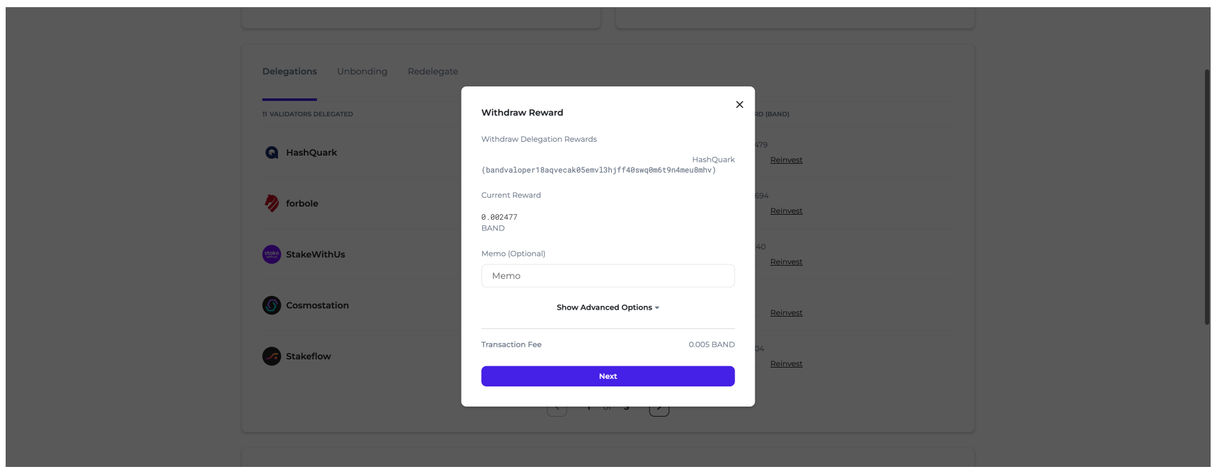
Step 1 — Navigate account details page

navigate to your account details page. you will see the section of Delegation. then click **Claim** on your selected validator



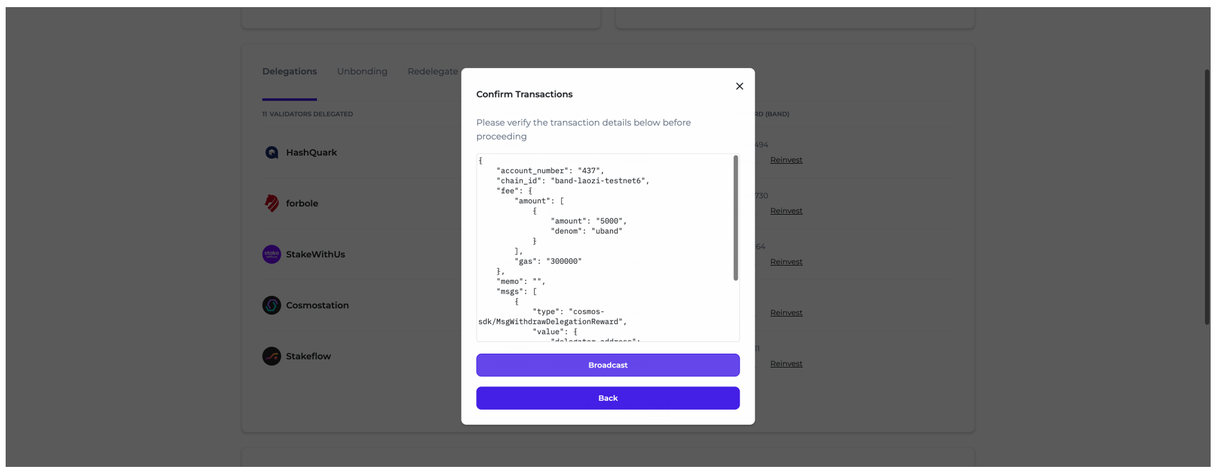
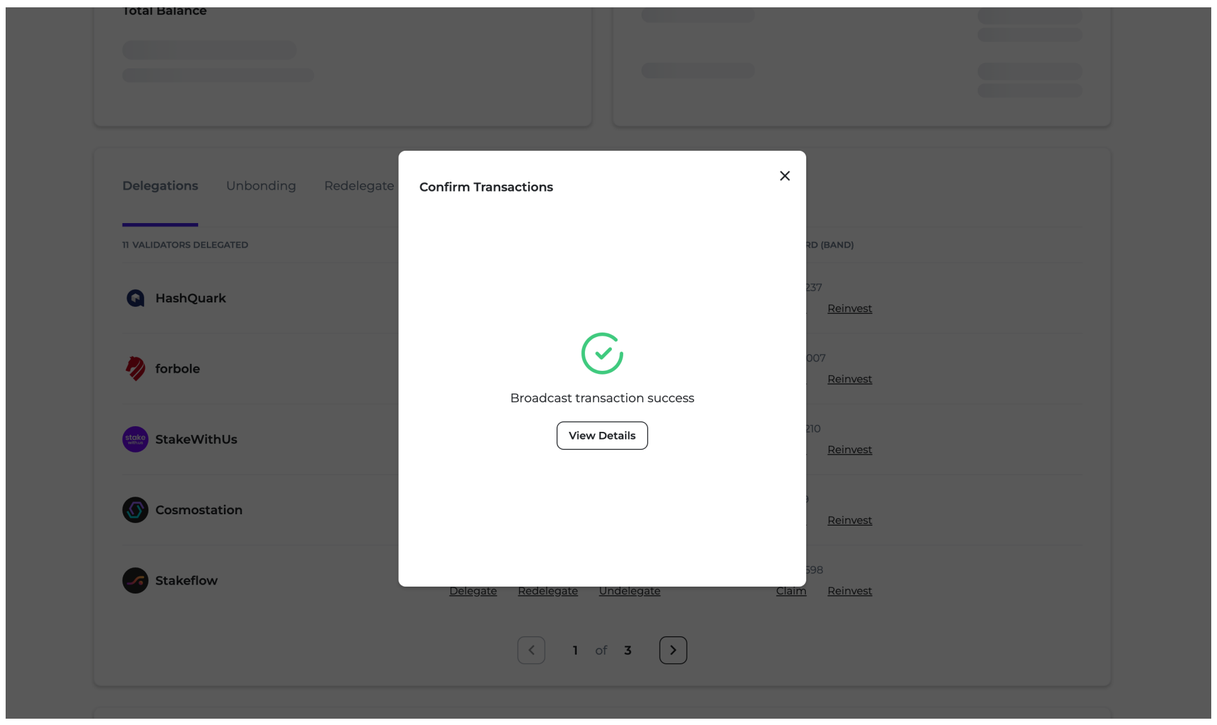
Step 2 — Click Next in pop-up window

In the Withdraw Reward pop-up window, you will see the amount of reward you will receive.



Step 3 — Check Transaction details

check if transaction detail is valid then click **Broadcast**



Congratulations! Your BAND has been successfully claimed reward