# 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 theExample 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 theoracle_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 thechannel_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 andIbcPacketReceiveMsg .

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 packet 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.Previous Introduction Next Example Use Cases