

Oracle Guide

For Dapp Developers

Aggregation types :

- Aggregation Number - 0, Aggregation Type - Mean Calculates Mean of incoming integer and double values. Returns encoded integer/double values to the client contract.
- Aggregation Number - 1, Aggregation Type - Standard Deviation Calculates standard deviation for incoming integer and double values. Remove responses which are outside the range of standard deviation. Then calculates mean on the remaining ones. Returns encoded integer/double values to the client contract.
- Aggregation Number - 2, Aggregation Type - Boolean Positive Responses ["success", "yes", "1", "positive", "successful", "true"] Negative Responses = ["failure", "no", "0", "negative", "unsuccessful", "false"] Check if there are more positive responses or more negative responses in the api responses. If there are more number of positive responses, oracle returns 1, else if there are more negative responses, oracle returns 0. If these values are equal, it will return 2 if both values are equal. Returns encoded integer values to the client contract.
- Aggregation Number - 3, Aggregation Type - Max Calculates the maximum value from the data coming from the api's. Returns encoded integer/double values to the client contract.
- Aggregation Number - 4, Aggregation Type - Min Calculates the maximum value from the data coming from the api's. Returns encoded integer/double values to the client contract.
- Aggregation Number - 5, Aggregation Type - Sum Calculates the sum of the data coming from the api's. Returns encoded integer/double values to the client contract.

- Aggregation Number - 6, Aggregation Type - First Returns the response from the first API. Returns encoded integer/double/string values to the client contract.
- Aggregation Number - 7, Aggregation Type - Last Returns the response from the last API. Returns encoded integer/double/string values to the client contract.
- Aggregation Number - 8, Aggregation Type - Median Calculates the median value from the data coming from the api's. Returns encoded integer/double values to the client contract.
- Aggregation Number - 9, Aggregation Type - Mode Calculates the mode from the data coming from the api's. If there are more than one modes, the greater value will be returned. Returns encoded integer/double values to the client contract.
- Aggregation Number - 10, Aggregation Type - Count
Calculates the count of a particular string (provided by the user) that is repeating.
Returns encoded string values to the client contract.

Response Types

:

- Response Number: 0, Type: Boolean
- Response Number: 1, Type: Integer
- Response Number: 2, Type: Double
- Response Number: 3, Type: String

Preferred API

:

If more than 50% API's fail to return any data, then the preferred API's data will be returned. If preferred API fails as well, then, the remaining data will go through the provided aggregation type.

Accepted Aggregation Types and Response types combinations :

Aggregation type Response type

0 (Mean) 1,2

1 (Standard Deviation) 1,2

2 (Boolean) 0,

- 3 (Max) 1,2
- 4 (Min) 1,2
- 5 (Sum) 1,2
- 6 (First) 0,1,2,3
- 7 (Last) 0,1,2,3
- 8 (Median) 1,2
- 9 (Mode) 1,2
- 10 (Count) 3,

Commands

These commands are for already deployed client contract (code provided below) that is deployed on the alacrity chain. // currently this contract works for the double response type only. We can call `addrequest` action in client contract to fetch data of an API. Data from oracle is being stored in table `btc balances`.

```
$ alaccli push action client addrequest
`["[API_array_of_objects]", "response_type", "aggregation_type", "prefered_api",
"string_to_count"]' -p
clientcontract@active
```

```
$ alaccli get table client client
btcbalances
```

Command explanation

:

API_Array_of_objects example :

```
[{ json_field: 'message'
endpoint: "http://www.mocky.io/v2/5e0333f13100006c006b2b31"

}, {}, {}
....]
```

Json_field example :

Used to Parse value from json using path path is string of json fields separated with '\n' required to obtain value from json Ex: {

"foo": { "bar": 1 }, "Arr": ["Key1" : "99.666", "Key2" : 99.556] } To parse value of the key "bar"

from given json response, json_field path will look like: "foo.bar". To parse value of "Key2" from the array "Arr", json_field will look like : "Arr[0].Key2".

Response_type : It can be 0/1/2/3 which represent Boolean/Integer/Double/String.

Aggregation_type : It can be 0/1/2/.... upto 10 which are explained in detail above.

Preferred_Api : If you have provided 5 apis in Api_Data, then its value can range from 1,2.....upto 5. Or it can be sent as an empty string ("").

String_to_Count : User can pass a string. Ex: ("yellow") This is used only with aggregation type 10, otherwise it can be sent as an empty string ("").

Custom Command Examples :

In this example, dapp developer requests for price of USD against BTC, with aggregation type as standard deviation and response type as integer. And dapp dev has not selected any preferred API.

```
alacli push action oracle addrequest  
'["[{ "json_field": 'USD', "endpoint": 'https://min-api.cryptocompare.com/data/price?fsym=BTC&tsyms=USD,CAD' }, { "json_field": 'USD', "endpoint":  
: 'https://min-api.cryptocompare.com/data/price?fsym=BTC&tsyms=USD,CAD'  
}, ], "1", "1", "", "" ]' -p clientcontract@active
```

Timeline for receiving the aggregated response :

The request when hit by the client contract will be handled by assigned oracle and a standby oracle, the assigned oracle will have a maximum timeframe of 20 seconds to respond with the correct response, if it fails to send the same, then the standby oracle will send the correct response between 20-40 seconds. If both the oracles fail to send the aggregated response. Then after 40 seconds an empty response will be returned.

Code for unpacking responses :

The response that client contract will be receiving is going to be encoded in the response type that it provides.

For unpacking Integer :

```
const auto rate = unpack<integer>(response);
```

For unpacking Double :

```
const auto rate =
```

```
unpack<double>(response); For unpacking String :
```

```
const auto rate = unpack<string>(response); For
```

```
unpacking Boolean : same as unpacking integer value.
```

Basic Smart Contract that the Dapp Developer needs to have in order to successfully send and receive response :

Client.cpp

```
#include "client.hpp"
```

```
namespace alaio {
```

```
client::client( name s, name code, datastream<const char*> ds )
```

```
: contract(s, code, ds),
```

```
_config_singleton(get_self(), get_self().value) {
```

```
config = config_singleton.get_or_create(_self,
```

```
configuration{
```

```
.next_request_id = 0 });
```

```
} client::~~client() {
```

```
config_singleton.set( config, get_self() ); } void client::reply( const alaio::name& caller,
```

```
uint64_t request_id, const std::vector<char>& response )
```

```
{
```

```
check( caller == get_self(), "received reply from another caller" ); // respond only to replies to
```

```

self const auto rate = unpack<double>(response);

    btc_balances_table balances( get_self(), get_self().value );
    balances.emplace( get_self(), [&]( auto& bal )
    {
    bal.id = request_id;
    bal.amount = rate; }); }

void client::transfer( const std::vector<api>& apis, uint16_t response_type, uint16_t
aggregation_type, uint16_t preferred_api, std::string string_to_count )
{
    uint64_t request_id = _config.next_request_id;
    request_id += microseconds(current_time()).count();

    _config.next_request_id++;

    make_request( request_id, apis, response_type, aggregation_type, preferred_api,
string_to_count);

} void client::make_request( uint64_t request_id, const std::vector<api>& apis, uint16_t
response_type, uint16_t aggregation_type, uint16_t preferred_api, std::string string_to_count
)
{
    action(
permission_level{ get_self(), active_permission }, oracle_account, request_action,
std::make_tuple(request_id, get_self(), apis, response_type, aggregation_type,
preferred_api, string_to_count)

).send(); } extern "C" { void apply(uint64_t receiver, uint64_t
code, uint64_t action) {

if (code == receiver) { } else if (code == oracle_account.value && action ==
reply_action.value) { // listen for reply notification by oracle contract
    execute_action( name(receiver), name(code), &client::reply ); }

execute_action( name(receiver), name(code), &client::transfer ); } }
} /// namespace alai0

```

Client.hpp

```
#pragma  
once
```

```
#include <alaiolib/asset.hpp>  
#include <alaiolib/alaio.hpp>  
#include <alaiolib/singleton.hpp>  
#include <alaiolib/system.hpp>  
#include <alaiolib/time.hpp>
```

```
namespace alaio {
```

```
static constexpr alaio::name oracle_account = "alaio"_n; static_assert(oracle_account,  
"oracle_account is empty. Please provide valid oracle account name");
```

```
static constexpr alaio::name active_permission = "active"_n;  
static constexpr alaio::name request_action = "addrequest"_n;  
static constexpr alaio::name reply_action = "reply"_n;
```

```
class [[alaio::contract]] client : public contract {
```

```
private:
```

```
struct api {
```

```
std::string endpoint; uint16_t  
request_type; uint16_t  
response_type; std::string  
parameters; std::string json_field;
```

```
}; struct [[alaio::table]]
```

```
btc_balances {
```

```
uint64_t id;  
double  
amount;
```

```
uint64_t primary_key() const { return id; }; typedef alaio::multi_index<
```

```
"btcbalances"_n, btc_balances > btc_balances_table;
```

```

    struct [[alaio::table("config")]] configuration {
uint64_t next_request_id = 0; }; typedef alaio::singleton< "config"_n,
    configuration > configuration_singleton;
    void make_request( uint64_t request_id, const std::vector<api>& apis, uint16_t
response_type, uint16_t aggregation_type, uint16_t preferred_api, std::string string_to_count );

    configuration_singleton _config_singleton;
    configuration _config;

public:
    client( name s, name code, datastream<const char*> ds );
    ~client();

    void reply( const alaio::name& caller, uint64_t request_id, const std::vector<char>&
response );

[[alaio::action]] void transfer( const std::vector<api>& apis, uint16_t
response_type, uint16_t aggregation_type, uint16_t preferred_api, std::string
string_to_count );
};

} /// namespace
alaio

```

How Dapp Developers can Retrieve the unique request id

:

By making the request via alajs, they can retrieve the unique request id associated with this request can be retrieved inside the response of the action performed.

```

const result = await
api.transact({
    actions: [{
        account: 'myclient',
        name: 'addreq',

```



```
    authorization: [{
      actor: "myclient",
      permission: 'active', }],

    data: { apis:
      response_type:
      Aggregation_type:
      preferred_api:
      string_to_count: }, } ],
    {
      blocksBehind: 3, expireSeconds: 30, }); let reqId =
      result.processed.action_traces[0].inline_traces[0].act.data.request_id // Variable
      reqid will receive the request id.
```

For Block Producers

:

Setting up the Oracle Application

Get the oracle app from : github url

```
$ cd oracle-app
```

Prerequisites In order for this application to work you need to run ALA node that has the MongoDB plugin activated. Check out MongoActionReader section of demux-js-ala github to get information on Mongo requirements for demux.

How to run Install all

required packages \$ npm

```
install
```

Create default.json in config/

```
directory $ mkdir config $ cd
```

```
config
```

Create file default.json with the following structure:

```
{
  "private_key":
  "{PRIVATE_ACTIVE_KEY}", "ala_data": {
    "endpoint": "http://alacrity.network.url:80",
    "oracle_contract_name": "oracle", "oracle_account":
```

```
"<account, which is running this app>" }, "mongo": {  
"historyDBName": "ALA",  
"applicationDBName": "oracle",  
"endpoint": "<your mongo endpoint>" } }
```

Config description:

ala_data->'endpoint' - alacrity rpc api endpoint 'oracle_contract_name' - account that has oracle contract deployed on 'oracle_account' - oracle account which is running the app and will sign 'reply' actions to oracle contract. 'historyDBName' - name of mongo database created by mongoddb plugin

'applicationDBName' - name of mongo database which will be used for storing application data mongo->'endpoint' - address of mongo which is used for storing oracle contract actions

Export private key for oracle contract

```
$ export PRIVATE_ACTIVE_KEY=<oracle active private  
key>
```

Run application by executing

```
$ npm start
```

In case you want application to start processing blocks from current head block (instead of last block processed by application) then you should launch it using

```
$ npm start -- --skip-missed
```

Setting up the oracle contract

Every block producer must execute 'setoracle' action in order to receive requests. When 'addrequest' is executed, active oracle is chosen based on block time and pending request count. If request will be completed in the first 20 seconds, successful requests of assigned oracle will increase otherwise it's failed request will increase and request will be assigned to standby oracle.

```
$ alaccli push action alaio setoracle `["bp_account"]` -p bp_account@active
```

Rewards -

Penalties

Rewards/Penalties for oracle requests will be handled as per the tokenomics. Rewards can be claimed along with the block producing rewards and penalty for failed request will be adjusted from the oracle rewards. If the penalty is greater than the oracle rewards, then it will be subtracted from block producing rewards. If it is still greater than the block rewards, it will be claimed in the next cycle of oracle rewards and block rewards.

Failed Request - Successful requests

The request when hit by the client will be handled by assigned oracle and a standby oracle. The assigned oracle will have a time period of 20 seconds to respond with the correct response, if it fails to send a response then the standby oracle will stand a chance to send the correct response for the next 20 seconds i.e. between 20-40 seconds. Time period for successful request is 20 seconds. Any response which will be returned within this timeframe will be considered as successful. And it will be considered unsuccessful otherwise.