The following structures are used for marshalling purposes:

BalanceGetResult // balance get result
    APIResponse status // status
        i8 code // status code
        string message // status communication
    Amount amount // balance amount
        i32 integral // the integral
        i64 fraction // the fraction

```
typedef string Address
typedef string TransactionInnerId
typedef string Currency
```

TransactionGetResult // transaction get result
    APIResponse status // status
        i8 code // status code
        string message // status communication
    bool found // transaction found boolean flag
    Transaction transaction // transaction structure
        TransactionInnerId innerId // transaction inner ID
        Address source // source address
        Address target // destination address
        Amount amount // transaction amount
            i32 integral // the integral
            i64 fraction // the fraction
        Amount balance // balance
            i32 integral // the integral
            i64 fraction // the fraction
        Currency currency // transaction currency code
        SmartContract smartContract // smart contract structure
            required string address // smart contract address
            string sourceCode // smart contract's source code
            binary byteCode // smart contract's byte code
            binary contractState // smart contract state
            string hashState // smart contract hash state

```
        string method                   // smart contract method
        list<string> params             // set of smart contract parameters

typedef list<Transaction> Transactions

TransactionsGetResult                   // list of transactions get result
    APIResponse status                  // status
        i8 code                         // status code
        string message                  // status communication
    bool result                         // result success boolean flag
    Transactions transactions           // set of transactions

TransactionFlowResult                   // transaction processing result
    APIResponse status                  // status
        i8 code                         // status code
        string message                  // status communication

typedef binary PoolHash
typedef i64 PoolNumber
typedef i64 Time
typedef i32 Count

Pool                                    // pool structure
    PoolHash hash                       // transaction pool hash
    PoolHash prevHash                   // previous transaction pool hash
    Time time                           // pool generation time
    i32 transactionsCount               // transaction count in a pool
    PoolNumber poolNumber               // transaction pool number

typedef list<Pool> Pools

PoolListGetResult                       // transaction pool list get result
    APIResponse status                  // status
        i8 code                         // status code
        string message                  // status communication
    bool result                         // result success boolean flag
    Pools pools                         // set of pools
```

PoolInfoGetResult // pool information get result
    APIResponse status // status
        i8 code // status code
        string message // status communication
    bool isFound // pool transactions found boolean flag
    Pool pool // pool content

PoolTransactionsGetResult // transaction pool get result
    APIResponse status // status
        i8 code // status code
        string message // status communication
    Transactions transactions // set of transactions

CumulativeAmount
    i64 integral // the integral
    i64 fraction // the fraction

```
typedef map<Currency, CumulativeAmount> Total;
```

PeriodStats // period statistics
    Time periodDuration // period
    Count poolsCount // number of pools
    Count transactionsCount // number of transactions
    Total balancePerCurrency // balance per currency
    Count smartContractsCount // number of smart contracts

```
typedef list<PeriodStats> StatsPerPeriod
```

StatsGetResult // period statistics get result
    APIResponse status // status
        i8 code // status code
        string message // status communication
    StatsPerPeriod stats // statistics content

```
typedef string NodeHash
```

NodesInfoGetResult // nodes information get result

```
    APIResponse status              // status
        i8 code                     // status code
        string message              // status communication
    Count count                     // number of nodes
    NodesHashes nodesHashes         // list of node hashes

SmartContractGetResult              // smart contract get result
    APIResponse status              // status
        i8 code                     // status code
        string message              // status communication
    SmartContract smartContract     // smart contract

typedef list<SmartContract> SmartContractsList

SmartContractsListGetResult         // smart contracts list get result
    APIResponse status              // status
        i8 code                     // status code
        string message              // status communication
    SmartContractsList smartContractsList    // smart contracts list

typedef list<string> SmartContractAddressList

SmartContractAddressesListGetResult      // smart contracts addresses list get result
    APIResponse status              // status
        i8 code                     // status code
        string message              // status communication
    SmartContractAddressList addressesList   // addresses list
```

Functions of API

BalanceGet

Getting the wallet balance with the address in currency (CS by default).

Parameters:

1: Address address
2: Currency currency = 'cs'

Type of result: BalanceGetResult

Operation algorithm:

The address field is re-coded: if the address string length is 64 symbols, then we re-code from 16 representation into string, or else we compute the public key hash in respect of the same based on the blake2 algorithm.

We get the balance from the blockchain based on the so-computed address by searching for the element containing such address in the balance cache. If the cache balance is found not in the last block, then we:

Download the pool from the storage based on the last block's hash value and then, while the public key length is valid and hash value is not equal to the hash value of the last verified block:

Determine the transaction balance in respect of all transactions of the block (if address source);

Add the transaction price to the result in respect of all transactions of the block (if address target);

If the balance found refers to the address source, then we add the address source balance found to the result and terminate a loop;

Download the pool from storage based on the hash value of the previous unviewed block.

If the wallet address found refers to the address source, then we add cache balance to the result.

We save the balance of this wallet address in the balances cache.

TransactionGet

Getting transaction content with transactionId.

Parameters:
1: TransactionId transactionId

Type of result: TransactionGetResult

Operation algorithm:

We start from dividing the transactionId string into hash and index. Then we get transaction from storage based on its ID structure. Unless the pool hash is empty, we believe that the transaction has been found. We convert the transaction from the csdb::Transaction structure to the api::Transaction structure. The result status we set is SUCCESS.

TransactionsGet

Getting a set of wallet transactions with address starting from offset to the number of limit.

Parameters:

1: Address address
2: i64 offset
3: i64 limit

Type of result: TransactionsGetResult

Operation algorithm:

The address field is re-coded: if the address string length is 64 symbols, then we re-code from 16 representation into string, or else we compute the public key hash in respect of the same based on the blake2s algorithm.
We determine an empty array of transactions.
Download pool from storage based on the last block's hash value. While the amount of block sequence is equal to the required public key amount, we perform the following:
If the number of transactions in a pool is more than zero, then we:
Determine that the current transaction index is equal to the number of transactions (1);
In respect of each transaction in a pool, if the wallet address corresponds to the transaction sender or recipient, then:
If the offset input is equal to 0, then we add a transaction to the array of transactions, or else we reduce the offset parameter;
If the number of transactions in the array is equal to the limit parameter, then we terminate a loop;
If the current transaction index is equal to 0, then we terminate a loop;
Reduce the current transaction index.
If the number of transactions in the array of transactions is equal to the limit parameter, then we terminate a loop;
Download the pool from the block with the previous hash.
We convert the transaction from the csdb::Transaction structure to the api::Transaction structure.
The result status we set is SUCCESS.

TransactionFlow

Processing the transaction with specified inputs.

Parameters:

1: Transaction transaction

Type of result: TransactionFlowResult

Operation algorithm:

Address source is recoded: if the address string length is 64 symbols, then we re-code from 16 representation into string, or else we compute the public key hash in respect of the same based on the blake2s algorithm.
We determine the transaction object with field values from the inbound transaction.
If the transaction specifies the smart contract address, then we:
Update the smart contract cache:
Obtain the last block’s hash value.
Set the current hash's variable equal to the last block’s hash value.
While the current hash's variable is not equal to the hash value of the block last viewed by the smart contract, we:
Download the block containing the current hash to the pool.
Retrieve the smart contract in respect of all transactions.
Complete class structures from the smart contract. If the current transaction unrolls the smart contract, then we cache transaction ID, sender, recipient or else save the smart contract address and ID.
Move the current hash's variable to the previous block.
We set the hash value of the block last viewed by the smart contract equal to the last block’s hash value.
We retrieve the transaction's smart contract from the cache memory of the transactions being unrolled by sender.
We retrieve the transaction's smart contract from the cache memory of executable transactions by sender.
If the smart contract unrolling is required, then we retrieve from the smart contract its byte code, source code, address target, hash state, method and parameters or else we quit the function in the FAILURE status.
If the smart contract is specified, we add the serialized structure of the smart contract being unrolled to the structure of the transaction being processed.
We submit the transaction for processing.
Unless the smart contract is specified, we quit the function in the SUCCESS status.
We open transport protocol for connections.
We execute the smart contract's byte code using the executor.
We add to the structure of the transaction being processed the new smart contract having its status as “executed” during the previous step of the smart contract, its method, parameters, address and submit the transaction for processing.
We quit the function in the SUCCESS status.

PoolListGet

Getting the content of blocks starting from the offset block, with the number of blocks being equal to limit.

Parameters:
1: i64 offset
2: i64 limit

Type of result: PoolListGetResult

Operation algorithm:

We set a limit on the values of offset and limit parameters equal to 100.
We allocate space to the pool of return result in the amount of limit.
We retrieve the last block's hash value on the blockchain as the current hash and the number of blocks on the blockchain.
In respect of all blocks starting from the offset block, with the number of blocks being equal to limit, we perform the following actions in reverse order:
We search for a block with the current hash value in the pool cache.
If the current block is located at the end of the pool cache, then we:
Download that block and convert it to the API pool format.
If the block falls within the range of blocks to be returned, then we add it to the result pool.
Add the hash value and pool to the pool cache.
Retrieve the previous block's hash value.
Or else we add the current block being viewed to the result and set a hash value equal to the previous block's hash value.

PoolInfoGet

Getting the block content with hash value and index number.

Parameters:

1: PoolHash hash
2: i64 index

Type of result: PoolInfoGetResult

Operation algorithm:

We download the block with the transmitted hash value to the pool.
We set the block found boolean flag in the result structure equal to the block validity boolean flag.
If the block has been found, then we download the block, convert and save it in the result structure pool.
We set the result status equal to SUCCESS.

PoolTransactionsGet

Getting a set of transactions starting from offset, with the number of blocks being equal to limit, from the block with hash value and index number.

Parameters:

1: PoolHash hash

2: i64 index
3: i64 offset
4: i64 limit

Type of result: PoolTransactionsGetResult

Operation algorithm:
We download the block with the transmitted hash value to the pool.
If the block is valid, then we retrieve transactions from the pool with offset biasing to the number of limit, save them to the transaction vector in the result structure.
We set the result status equal to SUCCESS.

StatsGet

Getting the period statistics collected.

Parameters:
None.

Type of result: StatsGetResult

Operation algorithm:
We get the current statistics (period, the number of pools, the number of transactions, balances per currency and the number of smart contracts).
We return such statistics as result (plus the SUCCESS status).

NodesInfoGet

Getting information about nodes.

Parameters:
None.

Type of result: NodesInfoGetResult

Operation algorithm:
We return the NOT_IMPLEMENTED status as result.

SmartContractGet

Getting a smart contract based on its address.

Parameters:

1:Address address

Type of result: SmartContractGetResult

Operation algorithm:

We update the smart contract caches:

We get the last block’s hash value.

We set the current hash's variable equal to the last block’s hash value.

While the current hash's variable is not equal to the hash value of the block last viewed by the smart contract, we:

Download the block containing the current hash to the pool.

Retrieve the smart contract in respect of all transactions.

Complete class structures from the smart contract. If the current transaction unrolls the smart contract, then we cache transaction ID, sender, recipient or else save the smart contract address and ID.

Move the current hash's variable to the previous block.

We set the hash value of the block last viewed by the smart contract equal to the last block’s hash value.

We download the smart contract performance transaction and retrieve the smart contract to the result structure.

We set the SUCCESS result status if the smart contract address is empty; if not, FAILURE.

SmartContractsListGet

Getting the list of smart contracts based on the source address of smart contracts.

Parameters:

1: Address deployer

Type of result: SmartContractsListGetResult

Operation algorithm:

The deployer field is re-coded: if the address string length is 64 symbols, then we re-code from 16 representation into string, or else we compute the public key hash in respect of the same based on the blake2 algorithm.

We update the smart contract caches:

We get the last block’s hash value.

We set the current hash's variable equal to the last block’s hash value.
While the current hash's variable is not equal to the hash value of the block last viewed by the smart contract, we:
Download the block containing the current hash to the pool.
Retrieve the smart contract in respect of all transactions.
Complete class structures from the smart contract. If the current transaction unrolls the smart contract, then we cache transaction ID, sender, recipient or else save the smart contract address and ID.
Move the current hash's variable to the previous block.
We set the hash value of the block last viewed by the smart contract equal to the last block’s hash value.
We retrieve smart contracts in respect of an array of transaction ID’s from source addresses and place them to the result structure.
We set the SUCCESS result status.

SmartContractAddressesListGet

Getting the list of smart contracts’ addresses based on the source address of smart contracts.

Parameters:
1: Address deployer

Type of result: SmartContractAddressesListGetResult

Operation algorithm:
The deployer field is re-coded: if the address string length is 64 symbols, then we re-code from 16 representation into string, or else we compute the public key hash in respect of the same based on the blake2 algorithm.
We update the smart contract caches:
We get the last block’s hash value.
We set the current hash's variable equal to the last block’s hash value.
While the current hash's variable is not equal to the hash value of the block last viewed by the smart contract, we:
Download the block containing the current hash to the pool.
Retrieve the smart contract in respect of all transactions.
Complete class structures from the smart contract. If the current transaction unrolls the smart contract, then we cache transaction ID, sender, recipient or else save the smart contract address and ID.
Move the current hash's variable to the previous block.
We set the hash value of the block last viewed by the smart contract equal to the last block’s hash value.
We retrieve smart contracts’ addresses in respect of an array of transaction IDs from source addresses and place them to the result structure.
We set the SUCCESS result status.