Apla Blockchain Platform Guide

Apla

Feb 20, 2019

Concepts

CHAPTER 1

About this documentation

This documentation contains the description of the Apla blockchain platform based on the source code of GenesisKernel blockchain.

1.1 About the platform

Apla blockchain is a secure, simple and compliant blockchain infrastructure for a fast-growing global collaborative economy segment. Small and medium sized enterprises will benefit from reducing operational costs and go-to-market time, fundraising solutions at an early development stage, automation of business processes, integrated settlement system, anti-money laundering compliant infrastructure, trustless cooperation, business scalability and global reach of their products and services to end customers. A detailed description of the Apla platform can be found in the Apla White Paper (www.apla.io)

The use of the Apla platform, including the license terms, is subject to the terms and conditions set forth in the Apla legal documentation that can be downloaded from the www.apla.io website.

CHAPTER 2

Contents

2.1 Apla overview

2.1.1 Distinctive features

Apla blockchain is a secure, simple and compliant blockchain infrastructure for a fast-growing global collaborative economy segment. Small and medium sized enterprises will benefit from reducing operational costs and go-to-market time, fundraising solutions at an early development stage, automation of business processes, integrated settlement system, anti-money laundering compliant infrastructure, trustless cooperation, business scalability and global reach of their products and services to end customers. A detailed description of the Apla platform can be found in the Apla White Paper (www.apla.io)

The use of the Apla platform, including the license terms, is subject to the terms and conditions set forth in the Apla legal documentation that can be downloaded from the www.apla.io website.

The platform is built based on the GenesisKernel software copyrighted by EGAAS S.A., the Luxembourg Corporation.

2.1.2 Apla architecture

Network

The platform blockchain platform is built based on a peer-to-peer network.

Full nodes of the network store the up-to-date version of the blockchain and the database, in which the current state of the platform is recorded.

The network users receive data by requesting it from databases of full nodes using the software client (or REST API commands). New data is sent to the network in the form of transactions signed by users. Such transactions are in essence commands for modification of information in the database. Transactions are aggregated in blocks, which are then added to the blockchain on the network nodes. After a new block is added to the blockchain, each full node processes the transactions in this block, thus making changes to data in its database accordingly.

Validating nodes

The network's full nodes that have the right to form blocks, are called *validating nodes*. The number of validating nodes is limited and is defined with the system parameter.

Transactions

A transaction is created by the software client and includes data for execution of a special program controller – contract ("smart contract"), called by a user.

A transaction is signed by the private key of an account holder. Both the key and the signing function can be stored in a browser, in the software client, on a SIM card, or on a specialized physical device. In the current implementation, private keys are kept in the Molis software client encrypted by the AES algorithm. Transactions are signed using the ECDSA algorithm.

Each transaction has the following format:

- Type - ID of the executed contract.
- Data - parameters passed to the contract.
- KeyID - ID of the user who sent the transaction.
- PublicKey - user's public key (optional).
- BinSignatures - transaction signature.
- Time - transaction timestamp.
- EcosystemD - ID of the ecosystem, where the transaction was initiated.
- okenEcosystem - ID of the ecosystem, the tokens of which should be used for the transaction payments.
- MaxSum - maximum transaction fee.
- PayOver - additional payment for priority processing in the transaction queue.

Network protocol

A transaction is sent by a user to one of the validating nodes, where it undergoes a basic verification to ensure the correctness of its format and then is added to the transaction queue. This transaction is also sent to other validating nodes on the network, where it's also added to the transactions queue.

The Node, at a particular moment of time that has the right to generate a new block (according to the full_nodes system parameter), retrieves transactions from the queue and sends them to the block generator. Simultaneously with the formation of a new block, the processing of transactions which are added to this block is carried out: each transaction is sent to the virtual machine that executes a corresponding contract with parameters, passed in the transaction, resulting in modification of the information in the database.

A new block is checked for errors, and if it is recognized as valid, it is sent to other validating nodes on the network.

Validating nodes add this newly received block to the blocks queue. After having been validated, a new block is added to the blockchain, and the transactions in this block are processed, thus updating the database.

Block and transaction verification

The verification of a new block, carried out by a validating node after it has created a new block, and the verification of such block on all other validating nodes after they receive this block, includes the following checks:

- The first byte should be 0; if not, the received data is not considered a block

- Received block's generation timestamp should be before the current time
- The block's generation timestamp should correspond to the time interval when the validating node had the right to sign a new block
- The new block's number should be greater than that of the last block in the existing chain
- The total fee limit for transactions in the block should not be exceeded
- The block must be correctly signed with the key of the Node that created it; the following data should be signed: BlockID, Hash of the previous block, Time, Position in full_nodes, MrklRoot from all transactions in the block.
- Each transaction in the block is checked for correctness in the following ways:
  - Each transaction's hash must be unique;
  - The limit of transaction signed with one key should not be exceeded (max_block_user_tx);
  - The transaction size should not be exceeded (max_tx_size)
  - The time when the transaction was sent should not be greater than the time of the block formation and not less than the block formation time minus 86400 seconds;
  - Transactions should be correctly signed;
  - The tokens which are assigned to be used for payment of transaction fees should exist in the sys_currencies list;
  - The user who executed the contract should have a sufficient number of tokens in their account to pay for resources required for execution of the transaction.

Database

The platform's unified database, copies of which are stored and maintained up-to-date on every full node of the network, is used for storing large volumes of data (registers) and quick retrieval of data by contracts and interfaces. In the formation of a new block and its addition to the blockchain, all full nodes of the platform carry out a simultaneous update of database tables. Thus, the database stores the current (up-to-date) state of the blockchain, which ensures the equivalence of data on all full nodes and unambiguousness of contract execution on any validating node. When a new full node is added to the network, the up-to-date status of its database is reached by way of subsequent execution of all transactions recorded in the blocks of the blockchain.

The platform uses PostgreSQL as its database management system.

2.1.3 Ecosystems

The data space of Apla is divided into many relatively independent clusters – *ecosystems*, in which the activities of the network's users are implemented. An ecosystem is an autonomous software environment that consists of a certain number of applications and users, who create these applications and work with them. Any holder of an account can create a new ecosystem.

The software basis of an ecosystem is a collection of applications, which are systems of interfaces, contracts, and database tables. The specific ecosystem to which application elements belong is indicated by prefixes in their name (for example, @1name), where the ecosystem's ID is indicated after the "@" sign. When addressing application elements within the current ecosystem, the prefix can be omitted.

The Molis software client provides access to database management tools, contracts editor, interface editor, and other functions required for the creation of applications in an ecosystem, without resorting to any additional software modules.

A person can become a user of the platform only after receiving a private key for accessing one of the ecosystems (by default, ecosystem #1). A user can be a member of any number of ecosystems. Switching between ecosystems is carried out using a specialized menu of the software client.

Integrated Development Environment

The Molis software client includes a full-scale integrated development environment (IDE) for creation of blockchain applications. Working with this IDE does not require the software developers to have profound knowledge of blockchain technology. The IDE is comprised of:

- Ecosystem parameters table
- Contracts editor
- Database tables administration tools
- Interface editor and a visual interface designer
- Language resource editor
- Application import / export service

Applications on the platform

An Apla application is a system of tables, contracts and interfaces with configured access rights. Such applications perform useful functions or implement various services.

Each ecosystem creates its own set of tables for the development of applications. This, however, does not exclude the possibility of accessing tables from other ecosystems by specifying those ecosystems' prefixes in table names. Tables are not in any way bound (nor belong) to specific contracts, and can be used by all applications. The permissions for entering data into tables are set by way of configuring the access rights. Specialized contracts – smart laws – can be used for rights management.

The design and creation of applications on the platform does not require the software developers to know the structure of the network and its protocols, nor to understand the algorithm of blockchain formation and synchronization of databases on full nodes. Work in the Molis software client, including the creation of application elements, reading data from tables, execution of contracts and displaying results on the screen, looks and feels like operations with modules of a software environment on a local computer.

Ecosystem's tables

An unlimited number of tables can be created for each ecosystem on the platform's database. As mentioned earlier, tables belonging to a specific ecosystem can be identified by a prefix that contains the ecosystem ID, which is not displayed in the software client while working within that specific ecosystem. Making records in tables of other ecosystem's tables is possible in cases where the access rights are configured to allow such actions.

Tools for table administration

Tools for administration of an ecosystem's tables are available from the Tables menu of the administrative tools in the Molis software client. The following functions are implemented:

- Viewing the list of tables and their contents.
- Creation of new tables.

- Adding new table columns and specifying the data type in columns: Text, Date/Time, Varchar, Character, JSON, Number, Money, Double, Binary.
- Management of permissions for entering data and changing the table structure.

Operations with data in tables

To organize the work with the database, the Simvolio contract language and the Protypo template language both have the DBFind function, which provides for retrieving values and data arrays from tables. The contract language has a function for adding rows to tables, DBInsert, and a function for changing values in existing entries, DBUpdate (when a value is changed, only the data in the database table is rewritten, whereas the blockchain is appended with a new transaction while preserving all previous transactions). Data in tables can be modified but not deleted.

In order to minimize the time of contracts execution, the DBFind functions cannot address more than one table at the same time, thus the requests with JOIN are not supported. That is why it is not advisable to normalize the application tables, but rather include all available information to the rows, thus duplicating data available in other tables. This, however, is not just a coercive measure, but a necessary requirement for blockchain applications, where what is saved (signed by a private key) should be a full, complete, up-to-date for a specific moment in time set of data (document), which cannot be modified due to the change of values in other tables (which is inevitable in relational databases).

Ecosystem parameters

The ecosystem parameters are available for viewing and editing from the Ecosystem parameters section in the administrative tools of the Molis software client. Ecosystem parameters can be divided into the following groups:

- General parameters: name of the ecosystem (ecosystem_name), its description (ecosystem_description), account of its founder (founder_account), and other information,
- Access parameters, which define exclusive rights to access application elements (changing_tables, changing_contracts, changing_page, changing_menu, changing_signature, changing_language)
- Technical parameters: for example, user stylesheets (stylesheet),
- User parameters of the ecosystem, where constants or lists (separated by commas), required for the work of applications are stored.

Rights to edit can be specified for every ecosystem's parameter.

In order to retrieve values of certain ecosystem parameters, both the contracts language Simvolio and the template language Protypo have the EcosysParam function, where an ecosystem parameter name can be specified as an argument. To retrieve an element from a list (entered as an ecosystem parameter and separated by commas), you should specify you desired element's counting number as a second argument for the function.

Parameters of the platform ecosystem

All parameters of the blockchain platform are stored in the parameters table of the platform configuration ecosystem:

- Time period for creation of a block by a validating node,
- Source codes of pages, contracts, tables, and menus of new ecosystems,
- List of validating nodes,
- Maximum transaction and block sizes, and the maximum number of transactions in one block,
- Maximum number of transactions sent by the same account in one block,
- Maximum amount of Fuel spent on one transaction and one block,

- Fuel to APL exchange rate, and other parameters.

Managing the parameters of the platform configuration ecosystem on the program level is the same as managing the parameters of any other ecosystem. Unlike in other ecosystems, where all rights to manage ecosystem parameters belong to the ecosystem founder, changing the parameters of the platform configuration ecosystem can only be performed using the UpdSysContract contract, the management of which is defined in the platform's Legal System. Contracts (smart laws) of the Legal System are created before the network is launched and implement the rights and standards, stipulated in the "Platform's Legal System" section of the White Paper.

2.1.4 Access rights control mechanism

Apla has a multi-level access rights management system. Access rights can be configured to create and change any element of an application: contracts, database tables, interface pages, and ecosystem parameters. Permissions to change access rights can be configured as well.

By default, all rights in a Apla ecosystem are managed by its founder (this is defined in the MainCondition contract, which every ecosystem has by default). However, after specialized smart laws are created, access rights control can be transferred to all ecosystem members or a group of such members.

Controlled operations

Permissions can be defined in the Permissions field of contracts, tables and interface (pages, menus, and page blocks) editors, available from the Molis administrative tools section.

Permissions for the following operations can be configured:

- Table column permission – permission to change values in the table column.
- Table Insert permission – permission to add a new row to the table.
- Table New Column permission – permission to add a new column.
- Conditions for changing of Table permissions – permission to change the Table column, Table Insert, and Table New Column permissions.
- Conditions for change smart contract – permission to edit the smart contract.
- Conditions for change page – permission to edit the interface page.
- Conditions for change menu – permission to edit the menu.
- Conditions for change of ecosystem parameters – permission to change a certain parameter in the ecosystem configuration table.

Ways to manage permissions

Rules, that define the access rights, should be entered in the *Permissions* fields as arbitrary expressions in Simvolio language. Access will be granted in the event that at the moment of request the expression was true. If the *Permissions* field is left blank, it is automatically set to *false*, and the execution of related actions is blocked.

The easiest way to define permissions is to enter a logical (boolean) expression in the *Permissions* field. For example, $member == 2263109859890200332, where the ID of a certain ecosystem member is given.

The most versatile and recommended method for defining permissions is the use of the *ContractConditions* function, to which a contract name can be passed as a parameter. This contract should include the conditions, in which formulation of the table values (for example, user roles tables) and ecosystem parameters can be used.

Another method of permissions management is the use of the ContractAccess function. The list of contracts that are eligible to implement a corresponding action can be passed to the ContractAccess function as

parameters. For example, if we take the table that lists the accounts in the ecosystem's tokens, and put `ContractAccess("TokenTransfer")` function in the *Permissions* field of the amount column, then the operation of changing the values in the amount column will be allowed only to the *TokenTransfer* contract (all contracts that perform token transfer operations between accounts, will be able to perform such operations only by calling the *TokenTransfer* contract). Conditions for accessing the contracts themselves can be managed in the conditions section. They can be rather complex and can include many other contracts.

Exclusive rights

To resolve conflict situations or those critical for the operation of an ecosystem, the Ecosystem parameters table has a number of special parameters (*changing_smart_contracts, changing_tables, changing_pages*), where the conditions for obtaining exclusive rights to access any smart contracts, tables and pages are defined. These rights are set using special smart contracts, for example, executing a voting of ecosystem members or requesting the availability of a number of signatures of different user roles.

2.1.5 Off-Blockchain Servers

The platform allows for creation of Off-Blockchain Servers (OBS), which have the full set of functions of standard ecosystems, but work outside the blockchain. In OBS full-scale applications can be created using the contract and template languages, database tables and other software client functions. Contracts from blockchain ecosystems can be called using API.

Requests to web-resources

The main difference between OBS and standard ecosystems is the possibility to make requests from its contracts to any web-resources via HTTP/HTTPS using the HTTPRequest function. Arguments passed to this function should be: URL, request method (GET or POST), header, and request parameters.

Rights to read data

Since data in OBS is not saved to the blockchain (which, however, is available for reading), they have an option to configure rights to read tables. Read rights can be set for separate columns, and for any rows using a special contract.

Using OBS

OBS can be used for the creation of registration forms and sending verification information to users' emails or phones, storing data out of public access, and writing and testing the work of applications with their further export and import to blockchain ecosystems. Also, in OBS you can schedule contract execution, which allows for the creation of oracles, which are used for receiving data from the web and sending it to the blockchain.

Creating an OBS

OBS can be created on any full node on the network. Node Administrator defines the list of ecosystems that are allowed to use the functions of OBS, and assigns a user who will have the rights of the ecosystem founder and will be able to: install applications, accept new members, and configure resource access rights.

2.2 About Apla blockchain

This section explains how Apla works with blockchain.

2.2.1 Tip of the iceberg

If you are interested in developing Apla apps, using them, or managing ecosystems, then you probably don't need to know anything about the Apla blockchain at all.

In Apla, the blockchain and the blockchain network are hidden from ecosystem members, administrators, and even app developers. Instead, Apla provides interfaces for all these groups of users. These interfaces provide access to the top layer of a blockchain: its tamper-proof distributed *global state*.

App developers

In technical terms, the *global state* is a set of data. Apla's implementation of its global state is a database. From the app developer standpoint, applications interact with this database by querying, inserting, and updating the database tables.

Apla applications are a collection of contracts and pages that interact with tables.

Under the hood, contracts are executed by writing transactions to the blockchain. These transactions invoke contract code, which is executed by the blockchain network nodes, which leads to changes in the global state, the database. For an app developer, a contract is a function. When it is executed, data is written to the database. Pages are like scripts. Page code is a set of *Protypo* functions. Some of these functions display page elements, other fetch data from the database. No knowledge of transactions, block generation or consensus algorithms is required from an application developer to work with Apla blockchain.

Ecosystem members

In Apla, apps written by app developers work inside autonomous software environments called *ecosystems*. An ecosystem typically serves a certain purpose and combines many apps created to support different aspects of this purpose.

To get access to the apps of an ecosystem, a user must become an *ecosystem member*. One user can be a member of many ecosystems.

Ecosystem members can view and modify the database from application pages, like they would do in a common web application: by filling forms, pressing buttons, and navigating pages.

Ecosystem and platform apps

Apps can be divided by scope into *ecosystem apps* and *platform apps*.

Ecosystem apps implement some certain functionality or business process specific to an ecosystem. An ecosystem app is available only in its ecosystem.

Platform apps are available in all ecosystems. Any app can be developed as a platform app. Apla developers provide platform apps that support core functionality for ecosystem governance, such as apps for votings, notifications, and ecosystem member roles management.

2.2.2 Under the hood

The layers

Under the top layer that is visible to ecosystem members and app developers, lies the "engine" of Apla, its node network and blockchain protocols.

You can think of Apla as having several layers:

- User interfaces layer

  Ecosystem members interact with applications via pages and page elements.

- Apps layer

  App developers interact with global state (database tables) via contract and page code.

- Global state layer

  Global state (database) is updated and synchronized based on the operations written to the distributed ledger of operations (blockchain).

- Blockchain layer

  Distributed ledger of operations is updated with new blocks. New blocks hold the operations (transactions) that must be performed on the global state.

- Node network layer

  Network nodes implement Apla blockchain protocol. They distribute transactions in the node network, validate these transactions and generate new blocks. Blocks are, in turn, distributed and validated by network nodes. The distributed ledger is kept synchronized for all nodes in a network. If conflicts occur, nodes agree upon which block chains are considered valid and rollback the invalid block chains.

- Transactions layer

  Transactions that are basis for block generation and blockchain protocol itself are a result of actions performed at the top layer. As explained in *The implementation*, transactions are automatically generated by Apla frontend, Molis client.

  When a user or a developer makes an action such as clicking a button on a page or executing a contract from the code editor, Molis converts this action into a transaction and sends it to the network node that it is connected to.

Thus, the top layer is connected to the bottom layer and the transaction flow goes in the opposite direction:

- A user action in the user interface creates a transaction.
- The transaction gets included in a block.
- The block is included in the blockchain.
- Changes in the blockchain cause changes in the global state, the action is applied to the database.
- The database changes are displayed in the app.

The implementation

Two main components of Apla are its backend, go-apla, and Molis client, apla-front.

Molis client:

- Provides a user interface for Apla.

- Provides an IDE for app development.
- Stores private keys of user accounts and performs authorization.
- Requests app page data from the database, and displays app pages to users.
- Sends transactions to the backend via *REST API*.

    Transactions are created automatically for user actions that require a transaction. For example, when an app developer executes a contract from the IDE, Molis converts this action into a transaction.

The backend:

- Keeps the global state (the database) of the node.
- Implements all Apla blockchain protocols.
- Executes contract code in a *virtual machine*.
- Executes page code in a *template engine*. The result is page data that can be used by Molis client.
- Implements *REST API*.

2.3 Blockchain in general terms

This section explains, in general terms, how blockchain technology works.

2.3.1 Blockchain

A blockchain is a distributed *ledger* of *operations* performed on a *global state* by *nodes* of a blockchain network.

The ledger is cryptographically protected from falsification. Any attempts to retrospectively alter the ledger require significant effort and are evident to the blockchain network members.

2.3.2 Nodes

Members of a blockchain network are called *nodes*. Each node owns a copy of the global state and a copy of the ledger. There is no central server or authority that holds master copies. Instead, nodes use the ledger to agree upon what the global state must be.

2.3.3 Global state

The *global state* is a set of data. Most common implementation of a global state is a database.

The global state is mutable. It can be changed with operations that nodes write to the ledger. For example, a database entry with the asset owner name can be changed, so that a new person owns the asset.

2.3.4 Ledger

The *ledger* (or *blockchain*) is a list of operations that were performed on a global state.

The ledger is immutable. The blockchain technology makes it so that operations written to the ledger cannot be modified or removed at a later date. For example, there is no way to back-out the operation that states that the asset owner name was changed at some point.

2.3.5 Transactions and blocks

The operations that nodes write to the ledger are called *transactions*.

Every blockchain network node can write transactions to the ledger. When a node writes a transaction to the ledger, it signs the transaction with node's private key.

Note: For the sake of simplicity, it is assumed here that transactions are generated by nodes and are signed with a node's private key. This is a basic transaction flow. In Apla, the transaction flow is more complex, as explained in *About Apla blockchain*.

A selected node (*validating node*) checks that the transaction is valid and can be written to the ledger. If it is so, this node generates a new *block* in the blockchain and signs it. A block usually holds several transactions.

This block is passed to other network nodes that validate the block. If the block is valid, they include it in their ledger and perform all the transactions from this block on their copy of the global state. Once all nodes have validated the block, the network have reached *consensus*. This block is considered valid by all the nodes of a blockchain network, and the global state is the same for all the nodes.

Note: Apla uses a proof-of-authority consensus mechanism. Only the selected nodes called *validating nodes* can generate new blocks.

Other blockchain-based projects use different consensus mechanisms. For example, bitcoin cryptocurrency uses prof-of-work mechanism, where a node that solved a complex equation is awarded a right to create a new block.

2.3.6 Example

For example, three friends own a collection of expensive automobiles, which are stored in a hangar. Each friend has a list of automobiles in the hangar and a log of operations for the hangar.

In blockchain terms, three friends are network nodes, the hangar is a global state, the automobiles in the hangar are data stored in the global state, the log of operations is a ledger, and entries in this log are transactions.

Now, for example, one of the friends buys a new automobile and decides to add it to the hangar. He sends a message to his friends. The message says that he has added a Buick 1961 to the hangar. One of the friends (appointed for this task) validates this claim, for example, by contacting the hangar personnel and checking if the new automobile is really there. If this message is valid, he adds a new entry to his copy of the operations log and adjusts his list of automobiles. He also tells all other friends what he has changed in his operations log. Other friends check that the change was correct and adjust their list of automobiles in the hangar. When all three friends have accepted the change in their ledger and adjusted their list of automobiles, the change was finalized.

In blockchain terms, one of the network nodes generated a transaction and sent it to other nodes. One of the validating nodes then validated this transaction and added a new block to the ledger. This block was sent to other network nodes who validated it, added it to the blockchain, and changed their global state accordingly. Once all nodes have made the changes to the ledger and to the global state, it is said that the network have reached the consensus.

2.4 Proof-of-Authority consensus

This section describes Proof-of-Authority consensus and its implementation in Apla.

2.4.1 What is Proof-of-Authority consensus

In blockchain platforms, consensus mechanisms can be divided into *permissionless* (Bitcoin, Etherium) and *permissioned* (Apla, Etherium Private).

In a permissioned blockchain, all nodes are pre-authenticated. This advantage allows to use consensus types that provide high transaction rate in addition to other benefits. One of these consensus types is *Proof-of-Authority* (PoA) consensus.

Proof-of-Authority (PoA) is a new consensus algorithms family that provides high performance and fault tolerance. In PoA, rights to generate new blocks are awarded to nodes that have proven their authority to do so. To gain this authority and a right to generate new blocks, a node must pass a preliminary authentication.

2.4.2 Advantages of PoA consensus

Compared to other consensus types that require a proof of spent computational resources (Proof-of-Work) or an existing "share" (Proof-of-Stake), PoA consensus has several notable advantages:

- High-performance hardware is not required. Compared to PoW consensus, PoA consensus does not require nodes to spend computational resources for solving complex mathematical tasks.
- The interval of time at which new blocks are generated is predictable. For PoW and PoS consensuses, this time varies.
- High transaction rate. Blocks are generated in a sequence at appointed time interval by authorized network nodes. This increases the speed at which transactions are validated.
- Tolerance to compromised and malicious nodes, as long as 51% of nodes are not compromised. Apla implements a ban mechanism for nodes and means of revoking block generation rights.

2.4.3 PoA consensus and common attack vectors

Denial-of-service attacks

An attacker sends a large number of transactions and blocks to a targeted network node in an attempt to disrupt its operation and make it unavailable.

The PoA mechanism makes it possible to defend against this attack:

- Because network nodes are pre-authenticated, block generation rights can be granted only to nodes that can withstand DoS attacks.
- If a node is unavailable for a certain period, it can be excluded from the list of validating nodes.

51% attack

In PoA consensus, the 51% attack requires an attacker to obtain control over 51% of network nodes. This is different from the 51% attack for the Proof-of-Work consensus types where an attacker needs to obtain 51% of network computational power. Obtaining control of the nodes in a permissioned blockchain network is much harder than obtaining computational power.

For example, in a PoW consensus type network, an attacker can increase computation power (performance) of the controlled network segment thus increasing the controlled percentage. This makes no sense for PoA consensus, because the computational power of the node has no effect on the blockchain network decisions.

2.4.4 How PoA consensus works in Apla

Validating nodes

In Apla, only selected nodes called *validating nodes* can generate new blocks. These nodes maintain the blockchain network and the distributed ledger.

The list of validating nodes is kept in the blockchain registry. The order of nodes in this list determines the sequence in which nodes generate new blocks.

The leader node

The following formula determines the current *leader node*, a node that must generate a new block at the current time.

```
leader = ((time - first) / step) % nodes
```

`leader`
Current leader node.

`time`
Current time (UNIX).

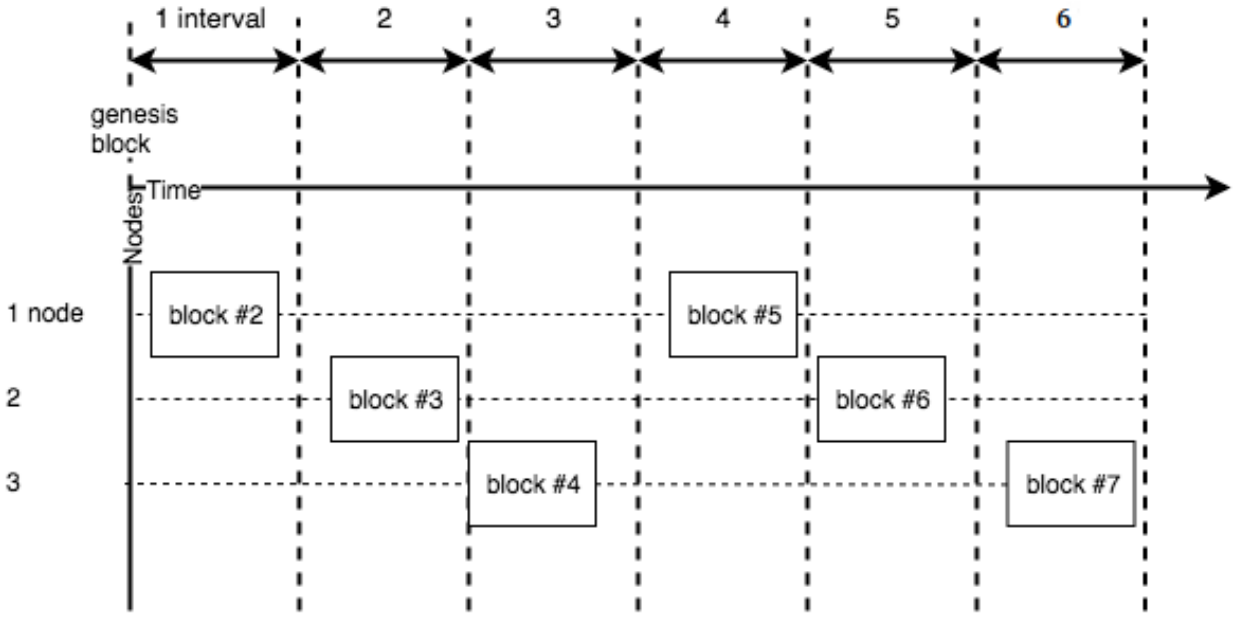
`first`
First block generation time (UNIX).

`step`
Number of seconds in the block generation interval.

`nodes`
Number of nodes at the current block generation interval.

Generation of new blocks

The new block is generated by a *leader node* of the current time interval. At each time interval, the leader role is passed to the next validating node from the *list of validating nodes*.



A new block is created

The leader node generates the new block as follows:

1. Collects all new transactions from its transaction queue.
2. Executes transactions one by one. Transactions that are invalid or cannot be executed are rejected.
3. Checks compliance to *block generation limits*.
4. Creates a block with valid transactions and signs it with node's private key (ECDSA algorithm).
5. Sends this block to other validating nodes.

The new block is validated

Other validating nodes:

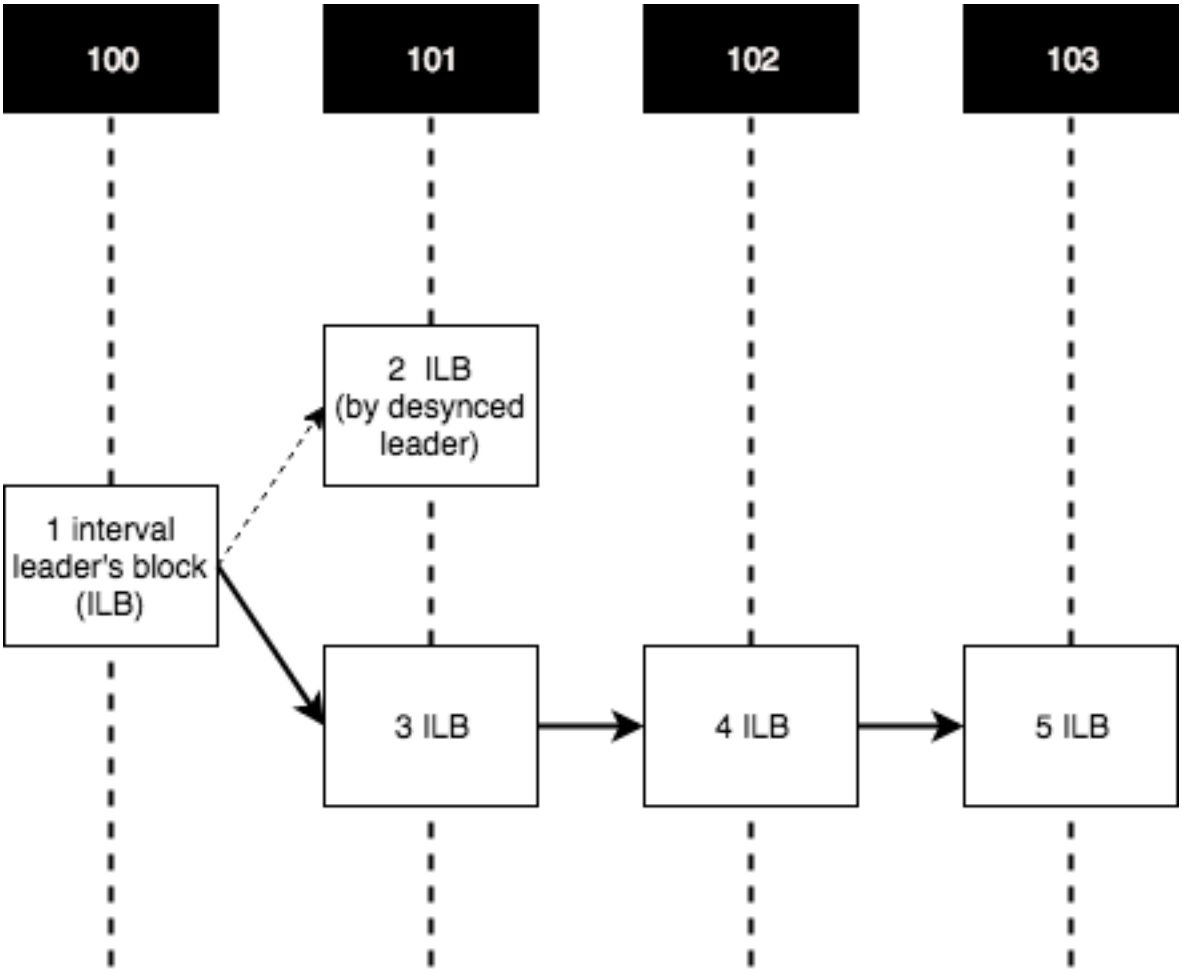
1. Receive the new block and validate that:
    - The new block was generated by the leader node of a current interval.
    - There are no other blocks generated by the leader node of a current interval.
    - The block is generated and signed correctly.
2. Execute transactions from the block one by one. Check that transactions are executed successfully and within block generation limits.
3. Add or reject the block, depending on the previous step:
    - If block validation is successful, add the new block to the node's blockchain.
    - If block validation failed, reject the block and send a *bad block* transaction. If the validating node that created this invalid block continues to generate such blocks, it can be banned or excluded from the list of validating nodes.

Forks

A *fork* is an alternate version of the blockchain. A fork contains one or more blocks that were generated independently from the rest of the blockchain.

Forks usually occur when a part of the network becomes desynchronized. Factors that influence the probability of forks are high network latency, intentional or unintentional time limits violation, time desynchronization at nodes. If network nodes have a significant geographic distribution, block generation interval must be increased.

Forks are resolved by following the *longest blockchain* rule. When two versions of the blockchain are detected, validating nodes rollback the shorter version and accept the longer one.



2.5 FAQ

1. In a few words, how would you describe Apla?
    - A blockchain platform, designed to build digital ecosystems on the basis of an integrated application development environment with a multi-level system for the management of access rights to data, interfaces and smart contracts.
2. Does Apla platform work on the Bitcoin, Ethereum, or some other blockchain?
    - No, Apla is built on the basis of being its own original blockchain.
3. What are the main differences between Apla and other public blockchain platforms with built-in mechanisms for the execution of smart contracts, such as Ethereum, Qtum, and those still being designed, including Tezos and EOS?
    - Apla features unique functions that cannot be found in the aforementioned blockchain platforms:
        - Integrated application development environment implemented in a single client software;
        - Specialized template language for the design of interfaces, harmonized with the contract-building language;
        - Multi-level system for the management of access rights to data, contracts, and interfaces where rights can be granted to persons, member roles, and contracts;

  - Ecosystems – autonomous software environments for the creation of blockchain applications and user interactions with them;
  - Legal system – a set of regulations, codified in smart laws (specialized smart contracts), which regulate relations between the platform users, define the protocol parameters changing procedures, that are used to solve problems.

4. Does Apla have its own cryptocurrency?

- Yes, Apla uses its own tokens, APL.

1. What is a validating node?
   - A validating node is a network node that is authorized to check transactions and create blocks.
2. Who can maintain a validating node?
   - Any network node with sufficient processing power and fault tolerance can become a validating node. Apla uses a Proof of Authority (PoA) consensus, where a node can become a validating node based on the voting of ecosystems. Only those ecosystems that are proved by investors as genuinely functioning (by platform token owners) can participate in such votings. With this algorithm it is most likely that the validating nodes will be run by major ecosystems, since it is in their best interest to maintain the network operation.
3. What are platform ecosystems?
   - Ecosystems are virtually autonomous software environments for the creation of blockchain applications and the user operations within them.
4. Who can create an ecosystem?
   - Any user of the platform can create a new ecosystem.
5. How can a user become a member of an ecosystem?
   - Registration in the platform network is made in any of its existing ecosystems; there can be different procedures for membership admission, which are defined by the ecosystems' policies: from posting information about a new ecosystem in a specialized catalog to sending out public keys.
6. Can one user create more than one ecosystem?
   - Yes, every user can create any number of ecosystems, and be a member of any number of ecosystems at the same time.
7. What is a platform application?
   - An application is an integral software product that implements a function or a service. Applications are comprised of database tables, contracts and interfaces.
8. Which programming language is used for the creation of applications?
   - Contracts are written using the Simvolio language, which was developed by the platform team (see contract language description).
   - Interfaces are written using Protypo – an original interface template language (see template language description).
9. Which software is used for creating applications and user interaction with them?
   - Applications are written and executed in Molis – the single software client; no other software is required.
10. Can platform contracts access data using third-party API interfaces?
    - No, contracts can directly access only the data stored in the blockchain. Specialized oracles are used to work with external data sources.

11. Can a contract saved in the blockchain be edited later?
    - Yes, contracts are editable. Rights to edit contracts are established by their creators, who can deny any changes or grant rights to make changes to contracts, to specific persons, or configure a complex set of conditions in a specialized smart law.
    - The Molis software client provides access to all contract versions.
12. What is a smart law?
    - A smart law is a contract that is created to control and restrict the operation of regular contracts, and thus the activities of the ecosystems' members.
    - A set of smart laws can be regarded as an ecosystem's legal system.
13. Can a contract call/execute another contract?
    - Yes, contracts can call other contracts by way of directly addressing another contract and providing parameters to it, or by way of calling a contract by link (name) (see contract language description).
14. Is a master contract required for applications to work?
    - No, it's not. Contracts are autonomous program modules that execute some functions. Each contract is configured to receive specific data, properly check these data, and execute some action, which will be recorded as a transition in the database.
15. Can applications be localized to different languages?
    - Yes, the software client has a built-in mechanism for localization support, allowing for the creation of interfaces in any language.
16. Can interfaces be created without using the Protypo template language?
    - Yes, the platform API can be used for that.
17. Are interface pages stored in the blockchain?
    - Yes, pages and contracts are stored in the blockchain, which protects them from falsification.
18. What types of databanks can be used for operation of contracts?
    - The Molis software client includes instruments for the creation of database tables (PostgreSQL is used at the moment, but we may change that later), and the Simvolio contracts language has all the functions required for reading and writing of data; Protypo template language includes the functions for reading data from tables.
19. How is the access to data in tables regulated?
    - Rights to add a column, a row, or to edit data in a column can be provided to ecosystem members, roles, or specific contracts (with the prohibition to contracts, other than those created to carry out specified operations).
20. Can applications inside an ecosystem exchange data with applications from another ecosystem?
    - Yes, data exchange can be organized through global (available for all ecosystems) tables.
21. Should all applications in a new ecosystem be written from scratch?
    - No, each new ecosystem has a number of applications available out-of-the-box: a mechanism for the management of members and roles in an ecosystem, an application for configuration and emission of tokens, a voting system, a social news system with incentives for activity, and a messenger for ecosystem members. These applications can be edited and configured to meet the specific requirements of any ecosystem.
22. Is there any payment for the operation of applications?
    - Yes, the use of resources of validating nodes should be paid for in platform tokens.

23. Who pays for the operation of applications?

    - An account (binding account), which the tokens for payment of resources are debited from, is set by the contract creator on its activation. It can be defined using ecosystem's smart laws whether or not the ecosystem members will pay for work with the application, and if yes, than what way of payment it will be (contributions or otherwise).

24. How are applications within ecosystems protected from exploit of their vulnerabilities?

    - The platform team understands that there is no way to completely avoid mistakes in the program code of applications, especially given that applications can be written by any user. That's why we decided to create a mechanism that eliminates the consequences of vulnerability exploitation. The platform has a legal system (a set of smart laws), that allow for stopping the operation of an attacking application and make a number of transactions for restoring to the status quo. The rights to execute such contracts and voting procedures to grant these rights are defined in the smart laws of the platform's legal system.

25. Which new functions are planned to be implemented in Apla in the future?

    - Visual interface designer,
    - Visual smart contract designer,
    - Support of hybrid (SQL and NoSQL) databases,
    - Parallel multi-threaded processing of transactions coming from different ecosystems,
    - Execution of resource-intensive calculations on the client side,
    - Hosting for ecosystems and a computing power exchange,
    - Partial nodes that store only a part of blocks on the server,
    - Semantic reference (ontology) for the unification of operations within the data in the platform.

26. Are there any proofs of Apla operability?

    - A number of proof of concept projects have been implemented on the platform during the last few months: a polling and voting system for a political party (Netherlands), new businesses registration (UAE), trading financial instruments (Luxembourg), register of property (India), and a contracts management system (UAE).

27. Does Apla have any obvious drawbacks?

    - The biggest drawback of the platform, compared to, say, Ethereum, is that Apla is just in the launch mode. But this drawback will transform into a big advantage over time.

28. What does the future of Apla look like?

    - The Genesis platform was designed based on the assumption that the full effect of blockchain technology can only be achieved when all activities, operations, registers and contracts are on the same blockchain network. Just as there can't be many co-existing Internets, there ultimately can't be many co-existing blockchain networks. We see the Genesis platform as a unified platform, which in the future will run the operations of all governments in the world.

2.6 Terms and Definitions

2.6.1 General Blockchain Technology Terms

- *blockchain* - an information system that stores data, secures this data from falsification and loss, transfers and processes data inside the system while preserving data reliability; the protection of data is achieved by: (1) writing the data into a chain of cryptographically linked blocks, (2) decentralized storage of copies of the chains

of blocks on nodes of a peer-to-peer network, and (3) synchronization of chains of blocks on all full nodes using a consensus algorithm; preserving data reliability when performing operations with data within the network is ensured by storing the algorithms of data transfer and processing (contracts) inside the blockchain; the chain of blocks itself is also referred to as the blockchain.

- *peer to peer network* - a computer network, consisting of equally privileged nodes (without a central server).
- *block* - a collection of transactions grouped by a validating node into a special data structure after their format and signatures are verified; a block contains a hash pointer as a link to the previous block, which is one of the measures that ensure the cryptographic security of the blockchain; a block is added to the blockchain after the consensus with other validating nodes on the network is achieved.
- *hash* - a uniquely reproducible cryptographic representation of a file or any other set of digital data; it ensures the invariability of data – any modification of data changes its hash.
- *block validation* - verification of the correctness of a block's structure, its creation time, its compatibility with the previous block, the transaction signatures, and the correspondence of transactions to the data in the blockchain.
- *validating node* - a node on the network that has the right to create and validate blocks.
- *consensus* - an agreement between validating nodes on the procedure of adding new blocks to the blockchain or an algorithm of such agreement.
- *transaction* - a single operation of data transfer on a blockchain network, or a record of such transaction in the blockchain.
- *token* - a unit that represents a share of rights, fixed as a set of identifiable numerical records in the register that includes a mechanism for exchanging shares of rights between these records.
- *identification* - a cryptographic procedure for recognition of a user in the system.
- *unique identification* - a procedure that links a user with a unique person; it requires legal and organizational efforts to implement biometric or other procedures for association of usernames with real persons.
- *private key* - a string of characters kept in secret by its owner and used for accessing this person's Virtual Account on the network and signing transactions.
- *public key* - a string of characters which can be used to check the authenticity of a signature made with a private key; public keys can be uniquely derived from private keys, but not vice versa.
- *digital signature* - an attribute of a digital document or message, obtained as a result of cryptographic data processing; a digital signature is used to check the integrity (absence of modifications) and authenticity of a document (verify sender's identity).
- *contract* - a program that performs operations with data stored in the blockchain; all contracts are stored in the blockchain.
- *transaction fee* - payment to a validating node for execution of a transaction.
- *double spend* - an attack on a blockchain network with the purpose of using the same tokens for two different transactions; this attack is implemented by forming and maintaining a fork (alternate version) of the blockchain; such an attack can be executed only in the event the attacker controls 50% or more of the network's validating power.
- *encryption* - transformation of digital data in such a way that only the party that possesses the appropriate decryption key is able to read it.
- *private blockchain* - a blockchain network where all nodes and access rights to data are under the centralized control of a single organization (government, corporation, or private individual).
- *public blockchain* - a blockchain network which is not controlled by any organization, all decisions are made by reaching a consensus among the network's participants, and data is available to everyone for read access.

- *delegated proof of stake (DPoS)* - a blockchain network consensus algorithm, where validating nodes are assigned by delegates (usually, token owners), who vote using their shares of rights.

2.6.2 Terms of the platform

- *testnet* - a version of the network that is used for testing software.
- *mainnet* - main version of the network.
- *Platform token* - tokens of the platform, which are used for payments for the use of network resources (fees).
- *Platform transaction* - commands that call contracts and pass parameters to them; the result of execution of a transaction by a node is the update of the platform's database.
- *fuel* - a conventional unit used to calculate the fee for execution of certain operations on the network; fuel exchange rate is decided on by the voting of validating nodes.
- *account* - a storage record for tokens, which can be accessed with a pair of keys – a private and a public one.
- *address* - a character-coded identifier of a user on the network, regarded as the name of this user's virtual account.
- *associated virtual account* - a virtual account from which the payment for execution of a contract is debited; an association of a contract with a virtual account is established upon contract creation and can be changed at any time; by default (before an association is established) payment is debited from the virtual account of the user who executed the contract.
- *Molis* - a software client used to connect to the network; Molis enables users to work with their virtual accounts, build ecosystems, and create applications in an integrated development environment (creating and editing tables, interface pages, and contracts).
- *web-Molis* - a fully-functional software client that works as a web-application.
- *platform ecosystem* - a relatively closed programming environment, which includes large numbers of applications and users who create and/or use these applications; ecosystem members can initiate the emission of the ecosystem's own token, use a system of smart contracts to establish the rules of interaction between its members, and set members' permissions to access the ecosystem's elements.
- *ecosystem parameters* - a set of configurable ecosystem attributes (name, description, logo, name of the ecosystem's token and its emission parameters, etc.); these attributes are stored and can be edited in a dedicated configuration table.
- *ecosystem members* - users who have access to functions and applications of a particular ecosystem.
- *dedicated ecosystem* - an ecosystem that has all the functions of a standard ecosystem, but works outside the blockchain (no data is saved in the blockchain); in dedicated ecosystems contracts are able to access any web resources over HTTP/HTTPS protocols, and rights to read data can be configured.
- *delegated Proof of Value of Ecosystem (DPoV(E))* - consensus algorithm, where validating nodes are assigned by the voting of ecosystems whose significance to the platform is confirmed (Valued Ecosystems), since it is in their best interest to maintain the smooth operation of the network; the approval of ecosystems that satisfy a number of formal indicators (number of transactions, number of members) to become Valued Ecosystems is implemented by the voting of token owners (in order to avoid fake ecosystems with bot-generated activities from taking part in the approval of Validating Nodes).
- *Simvolio* - a script language for building contracts; Simvolio contains functions for processing data received from interface pages, and for performing operations with values in database tables; contracts can be created and edited in the editor of the Molis software client.
- *Protypo* - a template language that includes functions required for obtaining values from database tables, and conditional statements/operators for building interface pages and forwarding user input data to contracts.

- *integrated development environment* - a set of software tools used for creating applications; the Molis software client's integrated development environment includes a contract editor, pages editor, tools for work with database tables, language resource editor, and application export and import functions; the integrated development environment will soon be complemented with visual editors based on semantic tools.
- *interface designer* - a tool in the Molis software client used for creating interfaces of application pages by arranging basic application elements (HTML containers, form fields, buttons, etc.) directly on the screen.
- *visual interface editor* - a tool in the Molis software client used for creating interfaces of application pages, which includes an interface designer and a generator of page code in Protypo language.
- *visual contract editor* - a tool in the Molis software client used for creating contracts using a visual interface.
- *language resources* - a module of the Molis software client used for localization of application interfaces; it associates a label on a page in an application with a text value in a selected language.
- *export of applications* - saving the source code of an application (any set of its tables, pages, and contracts) as a file.
- *import of applications* - uploading an application (all tables, pages, and contracts included in an exported file) into an ecosystem.
- *smart law* - a record in the blockchain that contains regulatory information, which is used for controlling the operation of contracts and management of access rights to registers; smart laws are specialized smart contracts.
- *legal system* - a set of regulations established in smart laws; a legal system regulates the relations between the platform users, defines procedures for changing protocol parameters and includes mechanisms that provide solutions to various challenges.
- *application* - a functionally complete software product created in the Molis software client's integrated development environment; an application consists of database tables, contracts, and interface pages.
- *application interface page* - a program code, written using the Protypo template language, that forms an interface on the screen.
- *interface block* - a program code, written using the Protypo template language, that can be included in application interface pages.
- *contract association* - linking a contract with a Virtual Account, from which the fee for performing contract operations will be debited.
- *access rights* - conditions for obtaining access to creating and editing tables, contracts and interface pages; access rights to tables can be specifically set for adding rows and columns, and for editing values in columns;
- *full node* - a node on the platform network that stores the full up-to-date version of the blockchain.
- *partial node* - a node on the platform network that stores only the blocks with data related to one ecosystem.
- *concurrent transactions processing* - a method for increasing the processing speed of transactions by simultaneously processing data from different ecosystems.

2.7 Application development tutorial

This tutorial explains how to write a simple application for Apla.

2.7.1 The goal

The goal of this tutorial is to create an application for Apla.

The application starts simple and grows in complexity as the tutorial progresses.

The final version of the app stores simple messages (strings) in a table with a timestamp and a message sender's account identifier. A user can access the list of messages and add new messages from the app's page. The app's page can be accessed from the ecosystem menu.

2.7.2 Part 1: The environment

quick-start

This tutorial uses Apla quick-start. The quick start provides docker containers for five network nodes. Each node has its own backend, database, and client. The manage.sh script can be used to control the nodes.

For installation instructions, see the quick-start README on GitHub.

Molis

Molis is a single unified client for Apla. Molis provides funcionality for all user and ecosystem roles. Application developers develop and test applications in Molis. Ecosystem administrators use Molis to manage ecosystems. Users can interact with ecosystem apps via Molis.

For this tutorial, you'll be writing contract code, page template code, and performing all other actions in the Molis client. Molis also provides a way to retrieve, save, and execute contract code, manage data structures (tables), assign access rights, and create applications.

Molis client is installed as a part of quick-start. Each node has its own Molis client instance.

2.7.3 Part 2: What is an app

App components

Apla apps are a combination of their *resources*: contracts, tables, and user interfaces. United into a single app, resources usually implement a certain functionality or a business process. An app and all its resources (except data stored in tables) can be exported and imported to other ecosystems.

- Contracts

  Contracts are like functions. They take input parameters, validate these parameters, and perform defined actions.

  Contracts are written in *Simvolio language*.

- Tables

  Tables hold information that is used by contracts and user interfaces.

  Tables are regular database tables: you can query, update, and insert. These actions are performed with three *Simvolio language* functions: *DBFind*, *DBInsert*, and *DBUpdate*.

- User interfaces

  User interfaces are pages and menus that will be displayed to users by Molis.

  The interfaces are written in *Protypo language*. Molis provides a visual editor for constructing interfaces.

The blockchain

All operations with contracts, tables, and interfaces are stored in the blockchain. A transaction is created for an operation. This transaction is then validated and executed in a virtual machine on all blockchain network nodes. Thus the state of resources is the same for all network nodes.

For example, if you change the contract code and save the changes, Molis creates a transaction that introduces the contract code change. After the transaction is validated and included in the blockchain, the new contract code becomes available to all nodes in the network. Same principle applies to tables and data stored in them, and to user interfaces.

Resource Access

The architecture of Apla apps is designed to be modular. Contracts, tables, and interfaces can be used by many different apps.

An app is a collection of its resources: contracts, pages, and tables. All resources of all apps within one ecosystem are available to each other. One resource can be used by many apps. Resources do not need to belong to a same app to be accessible.

For example, a dashboard page can use many tables that store information about ecosystem members and business processes; a contract can update several tables that are used by many ecosystem apps.

Access to resources is managed with access rights, which are implemented with contracts.

2.7.4 Part 3: The contract

You now have your network of five nodes and a basic understanding of what is an app and how apps work. Your first application will start as a simple "Hello, World!" application.

The spec

The application stores a single string in a table. It doesn't have any user interface.

Founder's account

The "root" privileges for an ecosystem are available to accounts with the *Admin* role. By default, this role has access to all operations. In a new ecosystem, Admin role is assigned to the *founder's account*. You must use this account to introduce major changes to the ecosystem, such as creating new apps and tables.

To login to the ecosystem with founder's account:

1. Make sure that quick-start is running. See quick-start README for more information.
2. Run `$ sudo ./manage.sh start-clients`

   This command starts Molis clients for all nodes.
3. One of the started clients is for the founder's account. This account lets you select roles after logging in. Choose *Admin*.

   Password for all accounts is `default`.

New app

Once you are logged as ecosystem's founder, you can create a new app.

To create a new app:

1. Go to the *Admin* tab.
2. From the list on the left, select *Application*.
3. In the *Applications* view, select *Create*.
4. Specify the name of your app in the *Name* field.
5. In the *Change conditions* specify `true`.

   The `true` value will make it possible for anyone to change the app.

   Another option is to specify `ContractConditions("MainCondition")`. This will forbid application changes to anyone except the founder.
6. Your app will appear in the list of apps. Click *select* to make it active.

Note: Selecting apps in the *Admin* tab makes it easier to navigate resources related to the selected app. It has no effect on the ecosystem. All ecosystem apps will still be available, no matter which one is selected.

New table

To store data, the application needs a table. Create this table from Molis.

To create a table:

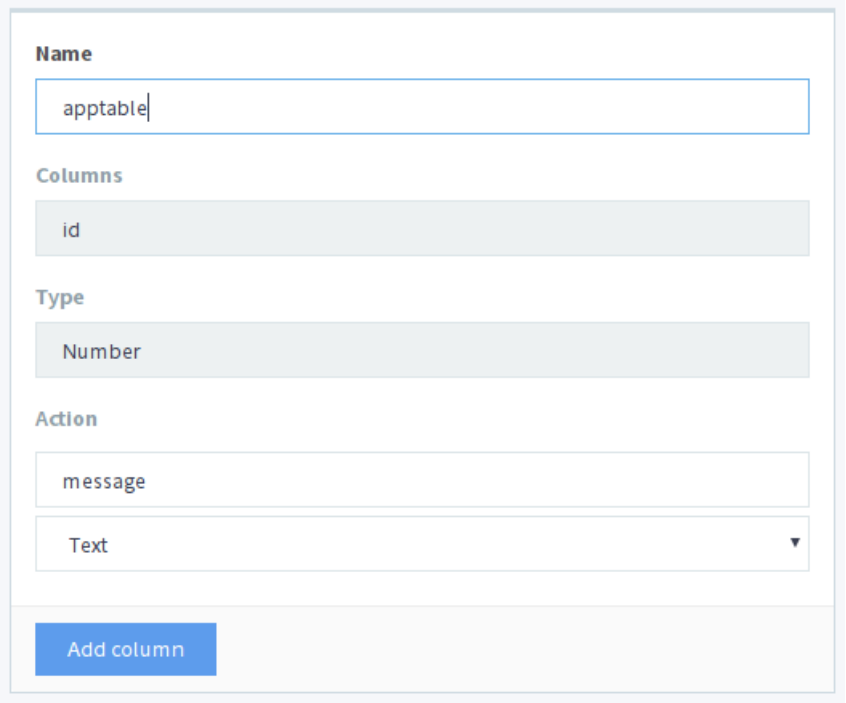
1. On the *Admin tab*, select *Resources* > *Tables*.

   This will display all tables for the selected app. The list will be empty, because your app doesn't have any tables yet.
2. Click *Create*.

   Molis will display the *Create table* view.
3. Specify a name for your table in the *Name* field.

   This tutorial uses `apptable` name for the table.
4. Add a column. Name it `message` and set its type to `Text`.

   As a result, the table must have two columns: `id` (predefined), and `message`. You will add more columns later.



5. For write permissions, specify `true` in every field.

   This will allow anyone to perform inserts and updates on the table, and to add columns.

   As an option, you can restrict writing permissions to the founder account. In this case, specify `ContractConditions("MainCondition")` in this parameter.

The contract

Contract code sections

Every contract has three sections:

- `data`

  Declares the input data (names and types of variables).

- `conditions`

  Validates the input data.

- `action`

  Performs actions defined by the contract logic.

Creating a new contract

1. On the *Admin tab*, select *Resources* > *Contracts*.

   This will display all contracts for the selected app. The list for your new app will be empty.

2. Click *Create*.

   A new contract template will open in the editor.

An empty contract template looks like this:

```
contract ... {
    data {

    }
    conditions {

    }
    action {

    }
}
```

Contract name

To start, give a name to your contract.

```
contract AppContract {
```

Data section

Fill the `data` section. The app must write strings to the table, so a `string` type variable is needed.

In the example below, `Message` is the name of the variable, `string` is its type.

```
data {
    Message string
}
```

Condition section

Fill the `conditions` section. The single validation condition is that the specified string must not be empty. If `Message` length is `0`, the contract will generate an alert with the defined message upon execution.

```
conditions {
    // avoid writing empty strings
    if Size($Message) == 0 {
        error "Message is empty"
    }
}
```

Action section

Fill the `action` section. The single action is writing the message to the table.

```
action {
    DBInsert("apptable", {message: $Message})
}
```

Full contract code

Below is the full contract code for this part.

All Apla contracts are constructed like this and always contain `data`, `conditions`, and `action` sections.

```
contract AppContract {
    data {
        Message string
    }
    conditions {
        // avoid writing empty strings
        if Size($Message) == 0 {
            error "Message is empty"
        }
    }
    action {
        DBInsert("apptable", {message: $Message})
    }
}
```

Save & execute

The contract is ready for testing:

1. In the Editor menu, click *Save*.

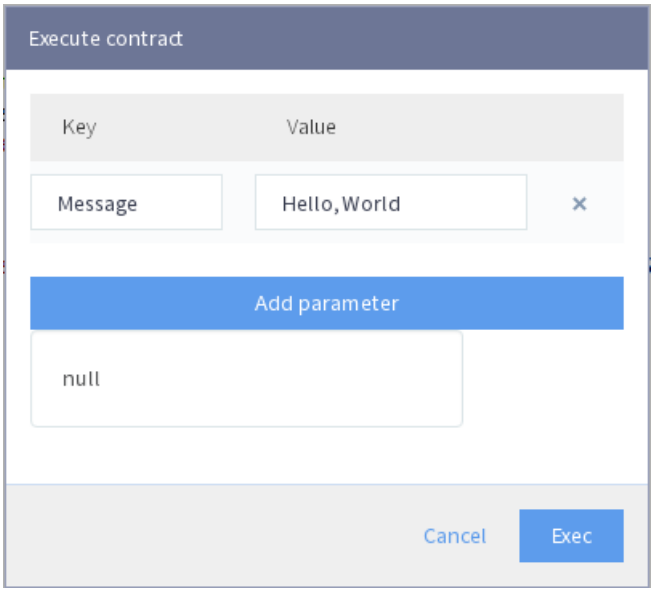
   This updates the contract code. The updated version becomes available to all the network nodes.

2. In the Editor menu, click *Execute*.

   This displays the *Execute contract* view.

3. In the *Execute contract* view, enter the input parameters for the contract.

   The contract has one parameter, `Message`, so specify `Message` in *Key* and `Hello, World!` in *Value*.



4. Click *Exec*.

The results will be displayed on the right.

If the string was added successfully, the results will contain the block number of the transaction that introduced the change, and the error code.

```
{
   "block": "31",
   "error": null
}
```

2.7.5 Part 4: The interface

After the contract is working, it's time to expand it into something more useful. In this part, you'll be implementing the UI and extra functionality.

The spec

The app stores strings in a table, like entries in a log. Every string has an author and a timestamp.

A user can view the stored list of strings from the application page, which is a simple table at this point.

The app does not provide a way to add new strings from the UI yet.

New columns

Just like before, edit the table from the *Admin* > *Resources* > *Tables* view.

Add the following columns to the `apptable` table:

- `author` of type `Number` with *Update* set to `true`.

  This field will store the identifier of the author's account.

- `timestamp` of type `Date/Time` with *Update* set to `true`.

Updated contract

Update the contract code to handle author IDs and timestamps.

Author IDs are identifers of the ecosystem accounts. Timestamps are the date and time of the contract execution in the Unix time format.

Both of these values are provided by the *predefined variables*. Since there is no need to input or validate the predefined variables, changes are needed only in the action section.

Change the contract so that the author's ID and the timestamp are written to the table when a message is added. The author's ID is defined by `$key_id`, the timestamp is defined by `$time`.

```
action {
    DBInsert("apptable", {message: $Message, author: $key_id, timestamp: $time})
}
```

The page

For this part, the application's interface is a simple page that displays information stored in the table.

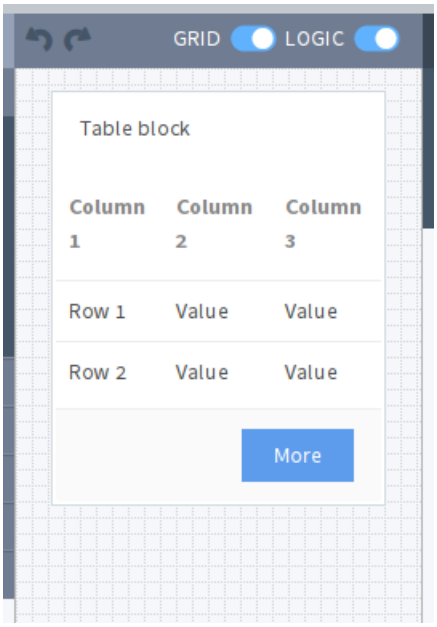
Just like all other resources, UI pages can be created in Molis:

1. Navigate to *Admin* > *Resources* > *Pages*.
2. Click *Create*.

   A visual editor will open in the new tab.

Designer's view

The default page is empty. Fortunately, you can use predefined structures to fill the page quickly.



Create a basic table with header:

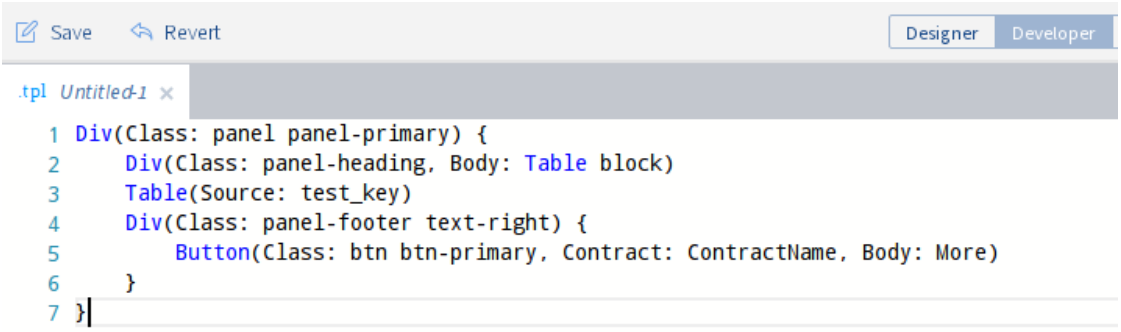
1. In the view selector on the right, click *Designer*.

   The view will switch to the visual editor.
2. From the menu on the left, select *Table With Header* and drag it to the page.

   A table with several elements will appear.

Developer's view

User interfaces for Apla are written in *Protypo*. You'll need to write code for the page, so switch to the developer's view.



To switch to the developer's view:

1. In the view selector on the right, click *Developer*.

   The view will switch to the code editor with the page code.

Get data from the table

At the moment, the page template does nothing. Change the code, so that the page displays data from the `apptable` table.

1. To request data from a table, use the *DBFind* function.

   The function call in the following exaple gets data from the `apptable` table, puts it into the `src_table` source, and orders it by the timestamp field. The `src_table` source is later used as a source of data for the table view on the page.

   ```
   DBFind(Name: apptable, Source: src_table).Columns(Columns: "author,
   ↪timestamp,message").Order(timestamp)
   ```

2. To display data from the `src_table` source, specify it as a source along with a list of headers in the `Table` function.

   ```
   Table(Columns: "AUTHOR=author,TIME=timestamp,MESSAGE=message", Source:␣
   ↪src_table)
   ```

3. In the view selector on the right, Click *Preview* to check that the data is displayed correctly.

Full page code

Below is the full page code for this part. This basic page will be expanded later.

```
DBFind(Name: apptable, Source: src_table).Columns(Columns: "author,timestamp,message
↪").Order(timestamp)

Div(Class: panel panel-primary) {
    Div(Class: panel-heading, Body: Table block)
    Table(Columns: "AUTHOR=author,TIME=timestamp,MESSAGE=message", Source: src_table)
    Div(Class: panel-footer text-right) {
        Button(Class: btn btn-primary, Contract: ContractName, Body: More)
    }
}
```

Save the page

Click *Save* to save the page:

1. Specify `AppPage` or any other name for a page in the *Name* field.
2. Leave the *Menu* option at `default_menu`.
3. In *Change Conditions* specify `true`.
4. Click *Confirm*.

2.7.6 Part 5: The app

In the previous parts you've created a contract, a table to store data, and a basic UI page to display this data.

In this part, you'll be finalizing the app, so it looks and behaves like an actual application.

The spec

The app stores messages in a table, like entries in a log. Every message has an author and a timestamp.

A user can view the stored messages by opening the application page from the ecosystem menu. The default table view holds 25 mesages and provides a way to browse more.

A user can add new messages from the UI page, one message at a time.

The menu

A page is always linked to a menu. For example, the the `default_page` page that is displayed on the *Home* tab is linked to the default ecosystem menu, `default_menu`.

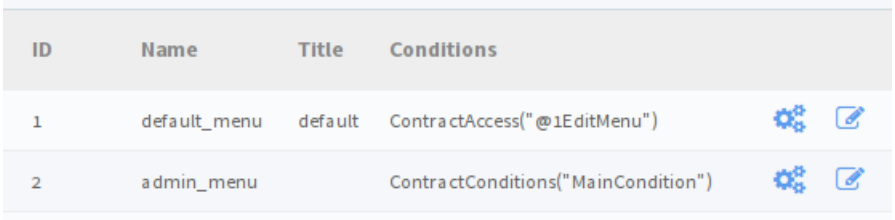
Because the tutorial app is small (just one page), there is no need to create an individual menu for it. A new menu item in the default menu will be enough.

Note: You can define what menu is displayed for the page by editing page properties in *Admin* > *Resources* > *Pages*. For example, if your app has several pages, you may want to create a menu to navigate between these pages and assign it to all pages of your app.

Add a menu item

Just like all other resources, menus can be created and edited in Molis:

1. Navigate to *Admin* > *Menu*.



2. Click the edit button next to the `default_menu` entry.

A visual editor will open in the new tab displaying Protypo template for the default ecosystem menu.

3. Add a new menu item to the end of the template. This menu item will open the app's page. The icon is from the FontAwesome icon set.

```
MenuItem(Title:Messages, Page:AppPage, Icon:"fa fa-envelope")
```

4. Click *Save*.

Test the new menu item

Check that the new menu item works:

1. Open the *Home* tab.
2. Click *Refresh* in the menu.

   A new item titled *Messages* will appear.

3. Click *Messages*.

   The app's page will open.

Sending messages

Buttons in Protypo can execute contracts and open pages, depending on the arguments.

The *Button* function has two arguments for contracts:

- `Contract`

  Name of the contract that must be activated.

- `Params`

  Input parameters for the contract.

Form

To send data to contracts, add a form to the app's page. This form must have an input field for the message, and a button that will activate the AppContract contract.

Below is an example of such form. It is enclosed in its own *Div*. Place it after the Div element that holds the table view. The *Input* field of this form has a defined name, `message_input`. This name is used by the button to send `Message` parameter value to the contract. Finally, *Val* function is used to obtain the value of the input field.

```
Div(Class: panel panel-primary) {
  Form() {
        Input(Name: message_input, Class: form-control, Type: text, Placeholder:
↪"Write a message...", )
        Button(Class: btn btn-primary, Body: Send, Contract: AppContract, Params:
↪"Message=Val(message_input)")
  }
}
```

Test this new functionality by sending messages. You may notice that the table doesn't refresh when a new message is sent. This is addressed *later in this tutorial*.

Table navigation

The default table view on the page will display only 25 first entries. Add a simple navigation that will allow users to navigate all table entries.

Navigation buttons

The navigation will use two buttons. Each button will reload the app's page and pass parameters to it.

- *Previous* button will show previous 25 entries. If there are no additional entries, the button will not be displayed.
- *Next* button will show next 25 entries. If there are no additional entries, the button will not be displayed.

Variables

This navigation requires two variables to store the table view state:

- `#table_view_offset#`

  This variable stores the current table view offset.

  Navigation buttons will pass this as a parameter when reloading the page.

- `#record_count#`

  This variable stores the total number of entries in the table.

  This value will be calculated.

Record count

To calculate `#record_count#`, modify the existing *DBFind* function call. The variable specified in the `.Count()` call will store the record count.

```
DBFind(Name: apptable, Source: src_table).Columns(Columns: "author,timestamp,
↪message").Order(timestamp).Count(record_count)
```

Table offset

The table view offset must be passed to the page when it is opened. If `#table_view_offset#` is not passed, it is assumed to be `0`.

Add the following code to the top of the page template. This code uses conditionals. *GetVar* function checks if the variable is set. *SetVar* function sets the variable.

```
If(GetVar(table_view_offset)){
}.Else{
    SetVar(table_view_offset, 0)
}
```

Modify the *DBFind* function call again. This time it must use the new table view offset.

```
DBFind(Name: apptable, Source: src_table).Columns(Columns: "author,timestamp,
↪message").Order(timestamp).Count(record_count).Offset(#table_view_offset#)
```

Button code

Buttons in Protypo can execute contracts and open pages, depending on the arguments.

If you haven't already done so, open the page in the editor, and delete the existing *More* button.

Afterwards, locate the *Div* function call that defines the footer, `Div(Class: panel-footer text-right)`. Add the button code to it.

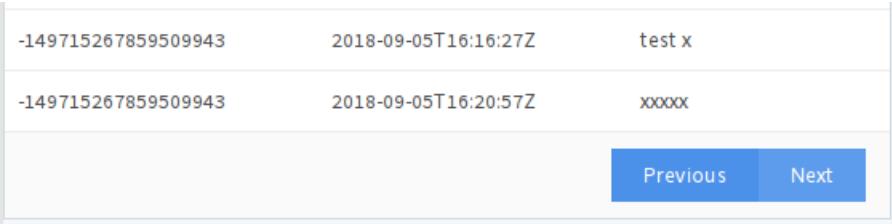
```
Div(Class: panel-footer text-right) {

}
```

The *Previous* button will be displayed only if there is at least one step to go back to. The new table view offset for the page, `offset_previous` is calculated when the button is added. Parameters are passed to the reopened page in the `PageParams` parameter.

```
If(#table_view_offset# >= 25) {
    SetVar(offset_previous, Calculate(#table_view_offset# - 25))
    Button(Class: btn btn-primary, Body: Previous, Page: AppPage, PageParams:
↪"table_view_offset=#offset_previous#")
}
```

The *Next* button will be displayed only if the total record count is more than what is displayed on the page. The new table view offset for the page, `offset_next` is calculated when the button is added. Parameters are passed to the reopened page in the `PageParams` parameter.

```
If(#record_count# >= Calculate(#table_view_offset# + 25)) {
    SetVar(offset_next, Calculate(#table_view_offset# + 25))
    Button(Class: btn btn-primary, Body: Next, Page: AppPage, PageParams:
↪"table_view_offset=#offset_next#")
}
```



After the buttons are added, save the page and test it from the *Home* > *Messages* menu item.

Page refresh

One final functionality that must be implemented is the automatic update of the table located on the page. When a user sends a new message, it must be displayed in the table.

You can implement this by making the *Send* button re-open the current page in addition to executing the contract. The `#table_view_offset#` parameter must be passed to the page without changes.

Add `Page` and `PageParams` arguments to *Send* button code like demonstrated below.

```
Button(Class: btn btn-primary, Body: Send, Contract: AppContract, Params:
↪"Message=Val(message_input)", Page:AppPage, PageParams:"table_view_offset=#table_
↪view_offset#")
```

Full page code

This part introduced many changes to the application page template. Below is the full code for the app page.

```
If(GetVar(table_view_offset)){
}.Else{
    SetVar(table_view_offset, 0)
}

DBFind(Name: apptable, Source: src_table).Columns(Columns: "author,timestamp,message
↪").Order(timestamp).Count(record_count).Offset(#table_view_offset#)

Div(Class: panel panel-primary) {
 Div(Class: panel-heading, Body: Table block)
 Table(Columns: "AUTHOR=author,TIME=timestamp,MESSAGE=message", Source: src_table)
 Div(Class: panel-footer text-right) {

  If(#table_view_offset# >= 25) {
    SetVar(offset_previous, Calculate(#table_view_offset# - 25))
    Button(Class: btn btn-primary, Body: Previous, Page: AppPage, PageParams:"table_
↪view_offset=#offset_previous#")
  }

  If(#record_count# >= Calculate(#table_view_offset# + 25)) {
    SetVar(offset_next, Calculate(#table_view_offset# + 25))
    Button(Class: btn btn-primary, Body: Next, Page: AppPage, PageParams:"table_view_
↪offset=#offset_next#")
  }

 }
}

Div(Class: panel panel-primary) {
  Form() {
        Input(Name: message_input, Class: form-control, Type: text, Placeholder:
↪"Write a message...", )
        Button(Class: btn btn-primary, Body: Send, Contract: AppContract, Params:
↪"Message=Val(message_input)", Page:AppPage, PageParams:"table_view_offset=#table_
↪view_offset#")
  }
}
```

2.7.7 Conclusion

This tutorial stops at the point where you have the basic application for your ecosystem. It doesn't explain other important topics for application developers like layout styles, access rights management and interaction between apps and resources. Please consult the rest of the documentation for more information about these advanced topics.

2.8 Smart contracts

A smart contract (hereinafter, "contract") is a basic element of applications, which performs a single action (typically, makes a record in a database table), initiated from the user interface by a user or by another contract. All operations with data in applications are formed as a system of contracts, interacting with each other through database tables or by call functions in a contract body.

Contracts are written using an original (developed by the team of platform developers) Turing-complete script language called Simvolio, with compilation into bytecode. The language includes a set of functions, operators and constructions that can be used for the implementation of data processing algorithms and operations with the database.

Contracts can be edited, but only if editing was not forbidden by way of putting `false` in the contract editing rights. Operations with data in the blockchain are performed by the most up-to-date (current) version of the contract. The complete history of changes made to contracts is stored in the blockchain and available from the software client.

2.8.1 Structure of the contract

Contracts are declared with the contract keyword, followed by the new contract's name. The contract's body must be enclosed in curly brackets. Every contract consists of three sections:

1. **data** - declares the input data (names and types of variables).
2. **conditions** - verifies the correctness of input data.
3. **action** - includes the actions performed by the contract.

```
contract MyContract {
    data {
        FromId int
        ToId   int
        Amount money
    }
    func conditions {
        ...
    }
    func action {
    }
}
```

Data section

The contract input data, as well as the parameters of the form for the reception of the data are described in the `data` section.

The data are listed line by line: first, the variable name is specified (only variables, but not arrays are transferred), then the type and the parameters for the building of the interface form are indicated optionally through a gap in double quotation marks:

- *optional* - form element that does not require to be filled.

```
contract my {
  data {
      Name string
      RequestId int
      Photo file "optional"
      Amount money
```

(continues on next page)

(continued from previous page)

```
      Private bytes
  }
  ...
}
```

Conditions section

Validation of the data obtained is performed in the Conditions section. The following commands are used to warn of the presence of errors: `error`, `warning`, `info`. In fact, all three of them generate an error that stops the contract operation, but each of them displays a different message in the interface: *critical error*, *warning*, and *informative error*. For instance,

```
if fuel == 0 {
      error "fuel cannot be zero!"
}
if money < limit {
      warning Sprintf("You don't have enough money: %v < %v", money, limit)
}
if idexist > 0 {
      info "You have already been registered"
}
```

Action section

The action section contains the contract's main program code that retrieves additional data and records the resulting values to database tables. For example,

```
    action {
DBUpdate("keys", $key_id, {"-amount": $amount})
DBUpdate("keys", $recipient, {"+amount": $amount,pub: $Pub})
    }
```

Price function

A contract can contain the **price** function. This function determines the extra fuel cost for executing the contract, in fuel units.

This function can return a value of *int* and *money* types. The returned value will be added to the cost of the contract execution and multiplied by **fuel_rate** coefficient.

```
contract MyContract {
  action {
            DBUpdate("keys", $key_id, {"-amount": $amount})
            DBUpdate("keys", $recipient, {"+amount": $amount,pub: $Pub})
  }
  func price int {
       return 10000
  }
}
```

2.8.2 Variables in the contract

Contract input data, declared in the data section, is passed to other sections though variables with the `$` sign followed by data names. The `$` sign can be used to declare additional variables; such variables will be considered global for this contract and all nested contracts.

A contract can access predefined variables that contain data about the transaction, from which this contract was called.

- `$time` – transaction time, int.
- `$ecosystem_id` – ecosystem ID, int.
- `$block` – number of the block, in which this transaction is included, int.
- `$key_id` – ID of the account that signed the transaction if the contract is outside of ecosystem with `ecosystem_id == 0`.
- `$type` identifier of an external contract from where the current contract was called.
- `$block_key_id` – address of the node that formed the block, in which this transaction is included.
- `$block_time` – time, when the block with the transaction containing the current contract was formed.
- `$original_contract` – name of the contract, which was initially called for transaction processing. If this variable is an empty string, it means that the contract was called in the process of verification of a condition. To check whether this contract was called by another contract or directly from a transaction, the values of **$original_contract** and **$this_contract** are to be compared. If they are equal, it means that the contract was called from the transaction.
- `$this_contract` – name of the currently executed contract.
- `$guest_key` – guest account ID.
- `$stack` – stack of contract calls. It is of the *array* type and contains strings with names of the called contract. Array element 0 is the currently executed contract. The last element is the name of the original contract that was called when transaction was processed.
- `$auth_token` is the authorization token, which can be used in OBS contracts, for example, when calling contracts though API with the `HTTPRequest` function.

```
var pars, heads map
heads["Authorization"] = "Bearer " + $auth_token
pars["obs"] = "false"
ret = HTTPRequest("http://localhost:7079/api/v2/node/mycontract", "POST",␣
↪heads, pars)
```

Predefined variables are accessible not only in contracts, but also in Permissions fields, (where conditions for access to application elements are defined), where they are used in construction of logical expressions. When used in Permissions fields, variables related to block formation (`$time`, `$block`, etc.) always equal zero.

Predefined variable `$result` is used to return a value from a nested contract.

```
contract my {
  data {
      Name string
      Amount money
  }
  func conditions {
      if $Amount <= 0 {
```

(continues on next page)

(continued from previous page)

```
            error "Amount cannot be 0"
        }
        $ownerId = 1232
    }
    func action {
        var amount money
        amount = $Amount - 10
        DBUpdate("mytable", $ownerId, {name: $Name,amount: amount})
        DBUpdate("mytable2", $citizen, {amount: 10})
    }
}
```

2.8.3 Nested Contracts

A nested contract can be called from the *conditions* and *action* sections of the enclosing contract. A nested contract can be called directly with parameters specified in parenthesis after its name (`NameContract(Params)`), or using the *CallContract* function, for which the contract name is passed using a string variable.

2.8.4 File Upload

To upload files from `multipart/form-data` forms, the contract fields with type `file` must be used. Example:

```
contract Upload {
    data {
        File file
    }
    ...
}
```

The *UploadBinary* system contract is intended to upload and store files. To request a download link for a file from the template designer, there is a special template designer function – *Binary*.

2.8.5 Using JSON in PostgreSQL queries

JSON type can be specified as column type. In this case, use the following syntax: **columnname->fieldname** to address record fields. The obtained value will be recorded in the column with name **columnname.fieldname**. Syntax **columnname->fieldname** can be used in parameters *Columns,One,Where* when using **DBFind**.

```
var ret map
var val str
var list array
ret = DBFind("mytable").Columns("myname,doc,doc->ind").WhereId($Id).Row()
val = ret["doc.ind"]
val = DBFind("mytable").Columns("myname,doc->type").WhereId($Id).One("doc->type")
list = DBFind("mytable").Columns("myname,doc,doc->ind").Where("doc->ind = ?", "101")
val = DBFind("mytable").WhereId($Id).One("doc->check")
```

2.8.6 Date/time operations in PostgreSQL queries

Functions do not allow direct possibilities to select, update, etc.. but they allow you to use the capabilities and functions of PostgreSQL when you get values and a description of the where conditions in the samples. This includes,

among other things, the functions for working with dates and time. For example, you need to compare the column *date_column* and the current time. If *date_column* has the type timestamp, then the expression will be the following `date_column> now ()`. And if *date_column* stores time in Unix format as a number, then the expression will be `to_timestamp (date_column)> now ()`.

```
to_timestamp(date_column) > now()
date_initial < now() - 30 * interval '1 day'
```

Following Simvolio functions work with date and time in SQL format:

- *BlockTime*
- *DateTime*
- *UnixDateTime*

2.8.7 Contract Editor

Contracts can be created and edited in a special editor which is a part of the Molis software client. Each new contract has a typical structure created in it by default with three sections: `data, conditions, action`. The contracts editor helps to:

- Write the contract code (highlighting key words of the Simvolio language.
- Format the contract source code.
- Bind the contract to an account, from which the payment for its execution will be charged.
- Define permissions to edit the contract (typically, by specifying the contract name with the permissions stipulated in a special function ContractConditions or by way of direct indication of access conditions in the Change conditions field).
- View the history of changes made to the contract with the option to restore previous versions.

2.9 Simvolio Contracts Language

Apla contracts are written in a Turing-complete script language called Simvolio, with compilation into bytecode. The language includes a set of functions, operators and constructions that can be used for implementation of data processing algorithms and operations with the database.

2.9.1 Basic elements and constructions of the language

Data Types and Variables

Data type must be defined for every variable. In obvious cases, data types are converted automatically. The following data types can be used:

- `bool` - Boolean, can be true or false.
- `bytes` - a sequence of bytes.
- `int` - a 64-bit integer.
- `array` - an array of values of arbitrary types.
- `map` - an associative array of values of arbitrary data types with string keys.

- `money` - an integer of the big integer type; values are stored in the database without decimal points, which are added when displaying values in the user interface in accordance with the currency configuration settings.
- `float` - a 64-bit number with a floating point.
- `string` - a string; must be defined in double quotes or back quotes: “This is a string” or ‘This is a string‘.
- `file` - associative array with a set of keys and values:
  - `Name` - file name (`string` type).
  - `MimeType` - mime-type of the file (`string` type).
  - `Body` - file contents (`bytes` type).

All identifiers, including the names of variables, functions, contracts, etc. are case sensitive (MyFunc and myFunc are different names).

Variables are declared with the **var** keyword, followed by names and types of variables. Variables declared inside curly brackets must be used within the same pair of curly brackets. When declared, variables have default values: for *bool* type it is *false*, for all numeric types – zero values, for strings – empty strings. Examples of variables declaration:

```
func myfunc( val int) int {
    var mystr1 mystr2 string, mypar int
    var checked bool
    ...
    if checked {
         var temp int
         ...
    }
}
```

Arrays

The language supports two array types:

- `array` - a simple array with numeric index starting from zero.
- `map` - an associative array with string keys.

When assigning and retrieving array elements, index must be put in square brackets. Multiple indexes are not supported in arrays. For example, you cannot address an array element as *myarr[i][j]*.

```
var myarr array
var mymap map
var s string

myarr[0] = 100
myarr[1] = "This is a line"
mymap["value"] = 777
mymap["param"] = "Parameter"

s = Sprintf("%v, %v, %v", myarr[0] + mymap["value"], myarr[1], mymap["param"])
// s = 877, This is a line, Parameter
```

You can also define arrays of `array` type by specifying elements in `[]`. For `map` type arrays, use `{}`.

```
var my map
my={"key1": "value1", key2: i, "key3": $Name}
```

(continues on next page)

(continued from previous page)

```
var mya array
mya=["value1", {key2: i}, $Name]
```

You can use such initializaton in the expressions. For example, this can be used in the function parameters.

```
DBFind...Where({id: 1})
```

For associative arrays, you must specify a key. Keys are specified as a string in double quotes (`""`). If key name contrains only letters, numbers and underscore, then double quotes can be omitted.

```
{key1: "value1", key2: "value2"}
```

An array can contain strings, numbers, variable names of any type, and variable names with `$` sign. Nested arrays are supported. A different map or array can be specified as a value.

Expressions cannot be used as array elements. Use a variable to store the expression result and specify this variable instead.

```
[1+2, myfunc(), name["param"]] // don't do this
[1, 3.4, mystr, "string", $ext, myarr, mymap, {"ids": [1,2, i], company: {"Name":
↪"MyCompany"}} ] // this is ok

var val string
val = my["param"]
MyFunc({key: val, sub: {name: "My name", "color": "Red"}})
```

If and While Statements

The contract language supports the standard **if** conditional statement and the **while** loop, which can be used in functions and contracts. These statements can be nested in each other.

A keyword must be followed by a conditional statement. If the conditional statement returns a number, then it is considered as *false* when its value = zero.

For example, *val == 0* is equivalent to *!val*, and *val != 0* is the same as just *val*. The **if** statement can have an **else** block, which executes in case the **if** conditional statement is false. The following comparison operators can be used in conditional statements: `<`, `>`, `>=`, `<=`, `==`, `!=`, as well as `||` (OR) and `&&` (AND).

```
if val > 10 || id != $citizen {
    ...
} else {
    ...
}
```

The **while** statement is intended for implementation of loops. A **while** block will be executed while its condition is true. The **break** operator is used to end a loop inside a block. To start a loop from the beginning, the **continue** operator must be used.

```
while true {
    if i > 100 {
        break
    }
    ...
    if i == 50 {
        continue
```

(continues on next page)

(continued from previous page)

```
    }
    ...
}
```

Apart from conditional statements, the language supports standard arithmetic operations: +, −, ⋆, /.

Variables of **string** and **bytes** types can be used as a condition. In this case, the condition will be true when the length of the string (bytes) is greater than zero, and false for an empty string.

Functions

Functions of the contracts language perform operations with data received in the data section of a contract: reading and writing database values, converting value types, and establishing connections between contracts.

Functions are declared with the **func** keyword, followed by the function name and a list of parameters passed to it (with their types), all enclosed in curly brackets and separated by commas. After the closing curly bracket the data type of the value returned by the function must be stated. The function body must be enclosed in curly brackets. If a function does not have parameters, then the curly brackets are not necessary. To return a value from a function, the `return` keyword is used.

```
func myfunc(left int, right int) int {
    return left*right + left - right
}
func test int {
    return myfunc(10, 30) + myfunc(20, 50)
}
func ooops {
    error "Ooops..."
}
```

Functions don't return errors, because all error checks are carried out automatically. When an error is generated in any function, the contract stops its operation and displays a window with the error description.

An undefined number of parameters can be passed to a function. To do this, put `...` instead of the type of the last parameter. In this case, the data type of the last parameter will be *array*, and it will contain all, starting from this parameter, variables that were passed with the call. Variables of any type can be passed, but you should take care of possible conflicts related to data type mismatch.

```
func sum(out string, values ...) {
    var i, res int

    while i < Len(values) {
       res = res + values[i]
       i = i + 1
    }
    Println(out, res)
}

func main() {
   sum("Sum:", 10, 20, 30, 40)
}
```

Let's consider a situation, where a function has many parameters, but we need only some of them when calling it. In this case, optional parameters can be declared in the following way: `func myfunc(name string).Param1(param string).Param2(param2 int) {...}`. You can specify only the parameters you need with the call in arbitrary order: `myfunc("name").Param2(100)`. In the function body you can address these

variables as usual. If an extended parameter is not specified with the call, it will have the default value, for example, an empty string for a string and zero for a number. You can specify several extended parameters and use `...`: `func DBFind(table string).Where(request string, params ...)` and call `DBFind("mytable").Where("id > ? and type = ?", myid, 2)`

```
func DBFind(table string).Columns(columns string).Where(format string, tail ...)
         .Limit(limit int).Offset(offset int) string  {
   ...
}

func names() string {
   ...
   return DBFind("table").Columns("name").Where("id=?", 100).Limit(1)
}
```

2.9.2 Simvolio functions by purpose

Retrieving values from the database:

- *AppParam*
- *DBFind*
- *DBRow*
- *DBSelectMetrics*
- *EcosysParam*
- *GetHistory*
- *GetHistoryRow*
- *GetColumnType*
- *GetDataFromXLSX*
- *GetRowsCountXLSX*
- *GetBlock*
- *LangRes*

Changing values in tables:

- *DBInsert*
- *DBUpdate*
- *DBUpdateExt*
- *DelColumn*
- *DelTable*

Array operations:

- *Append*
- *Join*
- *Split*
- *Len*
- *Row*
- *One*
- *GetMapKeys*
- *SortedKeys*

Operations with contracts and conditions:

- *CallContract*
- *ContractAccess*
- *ContractConditions*
- *EvalCondition*

- *GetContractById*
- *RoleAccess*
- *GetContractByName*
- *TransactionInfo*
- *Throw*
- *ValidateCondition*

Operations with account addresses:

- *AddressToId*
- *IdToAddress*
- *PubToID*

Operations with values of variables:

- *DecodeBase64*
- *EncodeBase64*
- *Float*
- *HexToBytes*
- *FormatMoney*
- *Random*
- *Int*
- *Hash*
- *Sha256*
- *Str*
- *UpdateLang*

Mathematical operations:

- *Floor*
- *Log*
- *Log10*
- *Pow*
- *Round*
- *Sqrt*

Operations with JSON:

- *JSONEncode*
- *JSONEncodeIndent*
- *JSONDecode*

Operations with strings:

- *HasPrefix*
- *Contains*
- *Replace*
- *Size*
- *Sprintf*
- *Substr*
- *ToLower*
- *ToUpper*
- *TrimSpace*

Operations with bytes:

- *StringToBytes*
- *BytesToString*

Operations with date and time in SQL format:

- *BlockTime*
- *DateTime*
- *UnixDateTime*

Operations with system parameters:

- *SysParamString*
- *SysParamInt*
- *DBUpdateSysParam*
- *UpdateNotifications*
- *UpdateRolesNotifications*

Functions for OBS mode:

- *HTTPRequest*
- *HTTPPostJSON*

Functions for OBS Master mode:

- *CreateOBS*
- *ListOBS*
- *RunOBS*
- *StopOBS*
- *RemoveOBS*

2.9.3 Simvolio functions reference

AppParam

Returns the value of a specified application parameter (from the application parameters table *app_params*).

Syntax

```
AppParam(app int, name string, ecosystemid int) string
```

app
: Application identifer.

name
: Application parameter name.

ecosystemid
: Ecosystem identifier.

Example

```
AppParam(1, "app_account", 1)
```

DBFind

Receives data from a database table in accordance with the specified request.

Returs an *array* comprised of *map* associative arrays that contain data from the database record. To get the first *map* element (first record from the request), use the `.Row()` tail function. To get a value from the first *map* element (a value from the specific column), use the `.One(column string)`.

Syntax

```
DBFind(table string)
    [.Columns(columns array|string)]
    [.Where(where map)]
    [.WhereId(id int)]
    [.Order(order string)]
    [.Limit(limit int)]
    [.Offset(offset int)]
    [.Ecosystem(ecosystemid int)] array
```

table
: Table name.

olumns
: List of returned columns. If not specified, all columns will be returned.

  This value can be specified as an array or as a string with comma separators.

where
: Search condition.

  Example: `.Where({name: "John"})` or `.Where({"id": {"$gte": 4}})`.

  This parameter must contain an associative array with search conditions. This array may contain nested elements.

  Following syntactic constructions are possible:

  - `{"field1": "value1", "field2" : "value2"}`

    This is equivalent to `field1 = "value1" AND field2 = "value2"`.

  - `{"field1": {"$eq":"value"}}`

    This is equivalent to `field = "value"`.

  - `{"field1": {"$neq": "value"}}`

    This is equivalent to `field != "value"`.

  - `{"field1: {"$in": [1,2,3]}`

    This is equivalent to `field IN (1,2,3)`.

  - `{"field1": {"$nin" : [1,2,3]}`

    This is equivalent to `field NOT IN (1,2,3)`.

  - `{"field": {"$lt": 12}}`

    This is equivalent to `field < 12`.

  - `{"field": {"$lte": 12}}`

    This is equivalent to `field <= 12`.

  - `{"field": {"$gt": 12}}`

    This is equivalent to `field > 12`.

- `{"field": {"$gte": 12}}`

  This is equivalent to `field >= 12`.

- `{"$and": [<expr1>, <expr2>, <expr3>]}`

  This is equivalent to `expr1 AND expr2 AND expr3`.

- `{"$or": [<expr1>, <expr2>, <expr3>]}`

  This is equivalent to `expr1 OR expr2 OR expr3`.

- `{field: {"$like": "value"}}`

  This is equivalent to `field like '%value%'` (substring search).

- `{field: {"$begin": "value"}}`

  This is equivalent to `field like 'value%'` (begins with `value`).

- `{field: {"$end": "value"}}`

  This is equivalent to `field like '%value'` (ends with `value`).

- `{field: "$isnull"}`

  This is equivalent to `field is null`.

Make sure that you don't overwrite associative array keys. For example, if you want to make the `id>2 and id<5` query, you cannot specify `{id:{"$gt": 2}, id:{"$lt": 5}}`, because the first `id` element will be overwritten by the second one. Instead, you can use the following constructions:

```
{id: [{"$gt": 2}, {"$lt": 5}]}
```

```
{"$and": [{id:{"$gt": 2}}, {id:{"$lt": 5}}]}
```

id
Search by identifier. For example, `.WhereId(1)`.

order
A field that will be used for sorting. By default, values are sorted by *id*.

If sorting is done using only one field, then it can be specified as a string. Otherwise, pass an array of strings and objects:

Descending order: `{"field": "-1"}`. Eqivalent of `field desc`.

Ascending order: `{"field": "1"}`. Eqivalent of `field asc`.

limit
Maximum number of returned values (default = 25, maximum = 250).

offset
Offset for returned values.

ecosystemid
Ecosystem ID.

By default, values are taken from the table in the current ecosystem.

Example

```
var i int
ret = DBFind("contracts").Columns("id,value").Where("id> ? and id < ?", 3, 8).Order(
↪"id")
while i < Len(ret) {
    var vals map
    vals = ret[0]
    Println(vals["value"])
    i = i + 1
}

var ret string
ret = DBFind("contracts").Columns("id,value").WhereId(10).One("value")
if ret != nil {
 Println(ret)
}
```

DBRow

Returns an associative array *map* with data obtained from a database table in accordance with the specified query.

Syntax

```
DBRow(table string)
    [.Columns(columns array|string)]
    [.Where(where map)]
    [.WhereId(id int)]
    [.Order(order array|string)]
    [.Ecosystem(ecosystemid int)] map
```

table
: Table name.

columns
: List of returned columns. If not specified, all columns will be returned.

  This value can be specified as an array or as a string with comma separators.

where
: Search condition.

  Example: `.Where({name: "John"})` or `.Where({"id": {"$gte": 4}})`.

  For more information, see *DBFind*.

id
: Search by identifier. For example, `.WhereId(1)`.

order
: A field that will be used for sorting. By default, values are sorted by *id*.

  For more information, see *DBFind*.

ecosystemid
: Ecosystem identifier. By default, current ecosystem identifer.

Example

```
var ret map
ret = DBRow("contracts").Columns(["id","value"]).Where({id: 1})
Println(ret)
```

DBSelectMetrics

Returns aggregated data for a metric.

Metrics are updated each time 100 blocks are generated. Aggregated data is stored in 1-day periods.

Syntax

```
DBSelectMetrics(metric string, timeInterval string, aggregateFunc string) array
```

metric
Metric name.

ecosystem_pages
Number of pages in ecosystems.

Returned values: *key* is an ecosystem ID, *value* is the number of pages in this ecosystem.

ecosystem_members
Number of members in ecosystems.

Returned values: *key* is an ecosystem ID, *value* is the number of members in this ecosystem.

ecosystem_tx
Number of transactions in ecosystems.

Returned values: *key* is an ecosystem ID, *value* is the number of transactions in this ecosystem.

timeInterval
Time interval for aggregated metric data. Examples: `1 day`, `30 days`.

aggregateFunc
Aggregation function. Examples: `max`, `min`, `avg`.

Example

In the example below, `row` contains a map with `key` and `value` keys. The `key` key contains ecosystem IDs, the `value` key contains metric value.

```
var rows array
rows = DBSelectMetrics("ecosystem_tx", "30 days", "avg")

var i int
while(i < Len(rows)) {
   var row map
   row = rows[i]
   i = i + 1
}
```

EcosysParam

Returns the value of a specified parameter from the ecosystem settings (*parameters* table).

Syntax

```
EcosysParam(name string) string
```

name
Name of the received parameter.

num
Position of the parameter.

Example

```
Println( EcosysParam("gov_account"))
```

GetHistory

Returns the history of changes of a record from the specified table.

Syntax

```
GetHistory(table string, id int) array
```

table
Table name.

id
Identifier of a record.

Return value

Return an array of associative arrays of *map* type. These arrays contain the history of changes of a record in the specified table.

Each associative array contains fields of a record before the next change was made.

The resulting list is sorted in the order from recent changes to earlier ones.

The *id* field in the resulting table points to the id in the *rollback_tx* table. The *block_id* field contains the block number. The *block_time* field contains the block timestamp.

Example

```
var list array
var item map
list = GetHistory("blocks", 1)
if Len(list) > 0 {
   item = list[0]
}
```

GetHistoryRow

Returns a single snapshot from the history of changes of a record in a specified table.

Syntax

```
GetHistoryRow(table string, id int, rollbackId int) map
```

table
: Table name.

id
: Record identifier.

RollbackId
: Identifier of the id record in the *rollback_tx* table.

Return value

Returns a single history record with the *rollbackId* identifier from the *rollback_tx* table.

GetColumnType

Returns the type of a column in a specified table.

Syntax

```
GetColumnType(table, column string) string
```

table
: Table name.

column
: Column name.

Returned value

Following column types can be returned: *text, varchar, number, money, double, bytes, json, datetime, double*.

Example

```
var coltype string
coltype = GetColumnType("members", "member_name")
```

GetDataFromXLSX

Returns data from XLSX spreadsheets.

Syntax

```
GetDataFromXLSX(binId int, line int, count int, sheet int) string
```

binId
: Identifier of an XLSX spreadsheet from the *binary* table.

line
: Spreadsheet line, starting from 0.

count
: Number of lines to return.

sheet
: List number in XLSX file, starting from 1.

Example

```
var a array
a = GetDataFromXLSX(binid, 12, 10, 1)
```

Returned value

Returned value is an array that contains arrays with spreadsheet cell data.

GetRowsCountXLSX

Returns the number of rows in a specified XLSX file.

Syntax

```
GetRowsCountXLSX(binId int, sheet int) int
```

binId
: Identifier of an XLSX spreadsheet from the *binary* table.

sheet
: List number in XLSX file, starting from 1.

Example

```
var count int
count = GetRowsCountXLSX(binid, 1)
```

LangRes

Returns a language resource with name label for language *lang*, specified as a two-character code, for instance, *en, fr, ru*; if there is no language resource for a selected language, the result will be returned in English.

Syntax

```
LangRes(label string, lang string) string
```

label
: Language resource name.

lang
: Two-character language code.

Example

```
warning LangRes("confirm", $Lang)
error LangRes("problems", "de")
```

GetBlock

Returns information about a block.

Syntax

```
GetBlock(blockID int64) map
```

blockID
: Block identifier.

Returned value

Returned value is a *map* associative array that contains the following data:

- *id*

  Block identifier.

- *time*

  Block generation time, in unixtime format.

- *key_id*

Node key of a node that generated the block.

Example

```
var b map
b = GetBlock(1)
Println(b)
```

DBInsert

Adds a record to a specified *table* and returns the **id** of the inserted record.

Syntax

```
DBInsert(table string, params map) int
```

tblname
Name of the table in the database.

params
Associative array where keys are field names and values are values to insert.

Example

DBInsert("mytable", {name: "John Smith", amount: 100})

DBUpdate

Changes column values in the table for the record with a specified **id**. If a record with this identifier does not exist, the operation will result with an error.

Syntax

```
DBUpdate(tblname string, id int, params map)
```

tblname
Name of the table in the database.

id
Identifier **id** of the changeable record.

params
Associative array where keys are field names and values are values to update.

Example

```
DBUpdate("mytable", myid, {name: "John Smith", amount: 100})
```

DBUpdateExt

The function updates columns in a record whose column matches the search condition.

Syntax

```
DBUpdateExt(tblname string, where map, params map)
```

tblname
Name of the table in the database.

where
Search condition.

Examples: `{name: "John"}`, `{"id": {"$gte": 4}}`, `{id: $key_id, ecosystem: $ecosystem_id}`.

For more information about search condition syntax, see *DBFind*.

params
Associative array where keys are field names and values are values to update.

Example

```
DBUpdateExt("mytable", {id: $key_id, ecosystem: $ecosystem_id}, {name: "John Smith",␣
↪amount: 100})
```

DelColumn

Deletes a column in the specified table. The table must have no records in it.

Syntax

```
DelColumn(tblname string, column string)
```

tblname
Name of the table in the database.

column
Name of the column that must be deleted.

```
DelColumn("mytable", "mycolumn")
```

DelTable

Deletes the specified table. The table must have no records in it.

Syntax

```
DelTable(tblname string)
```

tblname
Name of the table in the database.

Example

```
DelTable("mytable")
```

Append

Inserts *val* of any *type* to an *src* array.

Syntax

```
Append(src array, val anyType) array
```

src
An array.

val
Value to append.

Example

```
var list array
list = Append(list, "new_val")
```

Join

Merges the elements of the *in* array into a string with the specified *sep* separator.

Syntax

```
Join(in array, sep string) string
```

in
Name of the *array* type array, the elements of which you want to merge.

sep
Separator string.

Example

```
var val string, myarr array
myarr[0] = "first"
myarr[1] = 10
val = Join(myarr, ",")
```

Split

Splits the *in* string into elements using *sep* as a separator, and puts them into an array.

Syntax

```
Split(in string, sep string) array
```

in
: is the initial string.

sep
: is the separator string.

Example

```
var myarr array
myarr = Split("first,second,third", ",")
```

Len

Returns the number of elements in the specified array.

Syntax

```
Len(val array) int
```

val
: Array of the *array* type.

Example

```
if Len(mylist) == 0 {
  ...
}
```

Row

Returns the first *map* associative array from the *list* array. If the *list* is empty, then the result will be an empty *map*. This function is mostly used with the DBFind function. The *list* parameter must not be specified in this case.

Syntax

```
Row(list array) map
```

list

- a map array, returned by the **DBFind** function.

Example

```
var ret map
ret = DBFind("contracts").Columns("id,value").WhereId(10).Row()
Println(ret)
```

One

The function returns the value of the *column* key from the first associative array in the *list* array. If the *list* list is empty, then nil is returned. This function is mostly used with the DBFind function. The *list* parameter must not be specified in this case.

Syntax

```
One(list array, column string) string
```

list

- a map array, returned by the **DBFind** function.

column

- name of the returned key.

Example

```
var ret string
ret = DBFind("contracts").Columns("id,value").WhereId(10).One("value")
if ret != nil {
   Println(ret)
}
```

GetMapKeys

Returns an array of keys from the *val* associative array.

Syntax

```
GetMapKeys(val map) array
```

val
: An array.

Example

```
var val map
var arr array
val["k1"] = "v1"
val["k2"] = "v2"
```

SortedKeys

Returns a sorted array of keys from the *val* associative array.

Syntax

```
SortedKeys(val map) array
```

val
: An array.

Example

```
var val map
var arr array
val["k1"] = "v1"
val["k2"] = "v2"
arr = SortedKeys(val)
```

CallContract

Calls a contract by its name. All the parameters specified in the section *data* of the contract must be listed in the transmitted array. The function returns the value that was assigned to **$result** variable in the contract.

Syntax

```
CallContract(name string, params map)
```

name
: Name of the contract being called.

params
: Associative array with input data for the contract.

Example

```
var par map
par["Name"] = "My Name"
CallContract("MyContract", par)
```

ContractAccess

Checks whether the name of the executed contract matches with one of the names listed in the parameters. Typically used to control access of contracts to tables. The function is specified in the *Permissions* fields when editing table columns or in the *Insert* and *New Column* fields in the *Table permission* section.

Syntax

```
ContractAccess(name string, [name string]) bool
```

name
: Contract name.

Example

```
ContractAccess("MyContract")
ContractAccess("MyContract","SimpleContract")
```

ContractConditions

Calls the **conditions** section from contracts with specified names.

For such contracts, the *data* block must be empty. If the conditions *conditions* is executed without errors, then *true* is returned. If an error is generated during execution, the parent contract will also end with this error. This function is usually used to control access of contracts to tables and can be called in the *Permissions* fields when editing system table.

Syntax

```
ContractConditions(name string, [name string]) bool
```

name
: Contract name.

Example

```
ContractConditions("MainCondition")
```

EvalCondition

Takes from the *tablename* table the value of the *condfield* field from the record with the *'name'* field, which is equal to the *name* parameter and checks if the condition from the field *condfield* is made.

Syntax

```
EvalCondition(tablename string, name string, condfield string)
```

tablename
Name of the table.

name
Value for searching by the field 'name'.

condfield
Name of the field where the condition to be checked is stored.

Example

```
EvalCondition(`menu`, $Name, `condition`)
```

GetContractById

The function returns the contract name by its identifier. If the contract can't be found, an empty string will be returned.

Syntax

```
GetContractById(id int) string
```

id
Contract identifier in the *contracts* table.

Example

```
var name string
name = GetContractById($IdContract)
```

GetContractByName

eturns a contract identifier in the *contracts* by its name. If the contract does not exist, a zero value will be returned.

Syntax

```
GetContractByName(name string) int
```

name
ontract identifier in the *contracts* table.

Example

```
var id int
id = GetContractByName(`NewBlock`)
```

RoleAccess

Checks if the contract caller's role identifier matches one of the identifiers specified in parameters.

Use this function to control contract access to tables and other data.

Syntax

```
RoleAccess(id int, [id int]) bool
```

id
Role identifier.

Example

```
RoleAccess(1)
RoleAccess(1, 3)
```

TransactionInfo

Searches a transaction by the specified hash and returns information about the executed contract and its parameters.

Syntax

```
TransactionInfo(hash: string)
```

hash
Transaction hash in a hex string format.

Return value

The function returns a string in the json format:

```
{"contract":"ContractName", "params":{"key": "val"}, "block": "N"}
```

contract
Contract name.

params
Parameters passed to the contract.

block
Block ID where this transaction was processed.

Example

```
var out map
out = JSONDecode(TransactionInfo(hash))
```

Throw

Generates an error of type *exception*, but adds an *id* field to it.

Syntax

```
Throw(ErrorId string, ErrDescription string)
```

ErrorId
Error identifier.

ErrDescription
Error description.

Results

The result of such transaction has this format:

```
{"type":"exception","error":"Error description","id":"Error ID"}
```

Example

```
Throw("Problem", "There is a problem")
```

ValidateCondition

Attempts to compile the condition specified in the *condition* parameter. If a mistake occurs during the compilation process, the mistake will be generated and the calling contract will terminate. This function is designed to check the correctness of the conditions when they change.

Syntax

```
ValidateCondition(condition string, state int)
```

condition
Condition to validate.

state
State identifier. Specify 0 if checking for global conditions.

Example

```
ValidateCondition(`ContractAccess("@1MyContract")`, 1)
```

AddressToId

Returns the identification number of the citizen by the string value of the address of his account. If a wrong adress is specified, then 0 returns.

Syntax

```
AddressToId(address string) int
```

address
Account address in the `XXXX-...-XXXX` format or in the number form.

Example

```
wallet = AddressToId($Recipient)
```

IdToAddress

Returns the address of a account based on its ID number. If a wrong ID is specified, returned is 'invalid'.

Syntax

```
IdToAddress(id int) string
```

id
ID, numerical.

Example

```
$address = IdToAddress($id)
```

PubToID

Returns the account address by the public key in hexadecimal encoding.

Syntax

```
PubToID(hexkey string) int
```

hexkey
: Public key in hexadecimal form.

Example

```
var wallet int
wallet = PubToID("fa5e78.....34abd6")
```

DecodeBase64

Decodes a string in base64 encoding.

Syntax

```
DecodeBase64(input string) string
```

input
: Input string in base64 encoding.

Example

```
val = DecodeBase64(mybase64)
```

EncodeBase64

Encodes a string in base64 encoding and returns a string in the encoded format.

Syntax

```
EncodeBase64(input string) string
```

input
: Input string.

Example

```
var base64str string
base64str = EncodeBase64("my text")
```

Float

Converts an integer *int* or *string* to a floating-point number.

Syntax

```
Float(val int|string) float
```

val
: An integer or a string.

Example

```
val = Float("567.989") + Float(232)
```

HexToBytes

Converts a string with hexadecimal encoding to a *bytes* value (sequence of bytes).

Syntax

```
HexToBytes(hexdata string) bytes
```

hexdata
: A string containing a hexadecimal notation.

Example

```
var val bytes
val = HexToBytes("34fe4501a4d80094")
```

FormatMoney

Returns a string value of exp/10^digit. If *digit* parameter is not specified, it is taken from the **money_digit** ecosystem parameter.

Syntax

```
FormatMoney(exp string, digit int)
```

exp
: Numeric value as a string.

digit
: Exponent of the base 10 in the `exp/10^digit` expression. This value can be positive or negative. Positive value determines the number of digits after the comma.

Example

```
s = FormatMoney("123456723722323332", 0)
```

Random

Returns a random number in the range between min and max (min <= result < max). Both min and max must be positive numbers.

Syntax

```
Random(min int, max int) int
```

min
: Minimum value for the random number.

max
: Upper bound for random numbers. Generated random numbers will be smaller than this value.

Example

```
i = Random(10,5000)
```

Int

Converts a string value to an integer.

Syntax

```
Int(val string) int
```

val
: String containing a number.

Example

```
mystr = "-37763499007332"
val = Int(mystr)
```

Hash

Accepts a byte array or a string and returns a hash that was generated by the system cryptoprovider.

Syntax

```
Hash(val interface{}) string, error
```

val
: A string or a byte array.

Example

```
var hash string
hash = Hash("Test message")
```

Sha256

Feturns **SHA256** hash of a specified string.

Syntax

```
Sha256(val string) string
```

val
: Incoming line for which the **Sha256** hash must be calculated.

Example

```
var sha string
sha = Sha256("Test message")
```

Str

Converts a numeric *int* or *float* value to a string.

Syntax

```
Str(val int|float) string
```

val
: An integer or a floating-point number.

Example

```
myfloat = 5.678
val = Str(myfloat)
```

UpdateLang

Updates the language source in the memory. Is used in the transactions that change language sources.

Syntax

```
UpdateLang(name string, trans string)
```

name
Name of the language source.

trans
Source with translations.

Example

```
UpdateLang($Name, $Trans)
```

JSONEncode

Converts a number, a string, or an array to a string in JSON format.

Syntax

```
JSONEncode(src int|float|string|map|array) string
```

src
Data to convert.

Example

```
var mydata map
mydata["key"] = 1
var json string
json = JSONEncode(mydata)
```

JSONEncodeIndent

Converts a number, a string, or an array to a string in JSON format with the specified indentation.

Syntax

```
JSONEncodeIndent(src int|float|string|map|array, indent string) string
```

src
Data to convert.

indent
String that will be used as indentation.

Example

```
var mydata map
mydata["key"] = 1
var json string
json = JSONEncodeIndent(mydata, "\t")
```

JSONDecode

Converts a string in JSON format to a number, a string, or an array.

Syntax

```
JSONDecode(src string) int|float|string|map|array
```

src
String with data in JSON format.

Example

```
var mydata map
mydata = JSONDecode(`{"name": "John Smith", "company": "Smith's company"}`)
```

HasPrefix

Checks if the string begins with a specified substring.

Syntax

```
HasPrefix(s string, prefix string) bool
```

s
A string.

prefix
Prefix to check.

Return value

Returns `true` if the string begins with a specified substring.

Example

```
if HasPrefix($Name, `my`) {
...
}
```

Contains

Checks if a string contains the specified substring.

Syntax

```
Contains(s string, substr string) bool
```

s
: A string.

substr
: A substring.

Return value

Returns `true` if the string *s* containts the *substr* substring.

Example

```
if Contains($Name, `my`) {
...
}
```

Replace

Replaces in the *s* string all occurrences of the *old* string with the *new* string and returns the result.

Syntax

```
Replace(s string, old string, new string) string
```

s
: A string.

old
: Searched string.

new
: Replacement string.

Example

```
s = Replace($Name, `me`, `you`)
```

Size

Returns the size of the specified string.

Syntax

```
Size(val string) int
```

val
: A string.

Example

```
var len int
len = Size($Name)
```

Sprintf

The function creates a string based on the specified template and parameters.

Available wildcards:

- `%d` (number)
- `%s` (string)
- `%f` (float)
- `%v` (for any types)

Syntax

```
Sprintf(pattern string, val ...) string
```

pattern
: Template for the string.

Example

```
out = Sprintf("%s=%d", mypar, 6448)
```

Substr

Returns the substring obtained from a specified string starting from the offset *offset* (calculating from 0) and with maximum length of *length*.

In case of wrong offsets or length an empty string is returned.

If the sum of the *offset* and *length* is more than string size, then the substring will be returned starting from the offset to the end of the string.

Syntax

```
Substr(s string, offset int, length int) string
```

val
A string.

offset
Offset from the beginning of the string.

length
Maximum substring length.

Example

```
var s string
s = Substr($Name, 1, 10)
```

ToLower

Returns the specified string in a lower case.

Syntax

```
ToLower(val string) string
```

val
Input string.

Example

```
val = ToLower(val)
```

ToUpper

Returns the specified string in an upper case.

Syntax

```
ToUpper(val string) string
```

val
Input string.

Example

```
val = ToUpper(val)
```

TrimSpace

Removes leading and trailing spaces, tab, and newline symbols from a string.

Syntax

```
TrimSpace(val string) string
```

val
Input string.

Example

```
val = TrimSpace(val)
```

Floor

Returns the largest integer smaller than or equal to the specified number.

Syntax

```
Floor(x float|int|string) int
```

x
A number.

Example

```
val = Floor(5.6) // returns 5
```

Log

Returns the natural logarithm of the specified number.

Syntax

```
Log(x float|int|string) float
```

x
: A number.

Example

```
val = Log(10)
```

Log10

Returns the base-10 logarithm of the specified number.

Syntax

```
Log10(x float|int|string) float
```

x
: A number.

Example

```
val = Log10(100)
```

Pow

Returns the specified base to the specified power (x^y).

Syntax

```
Pow(x float|int|string, y float|int|string) float
```

x
: Base number.

y
: Exponent.

Example

```
val = Pow(2, 3)
```

Round

Returns the value of the specified number rounded to the nearest integer.

Syntax

```
Round(x float|int|string) int
```

x
A number.

Example

```
val = Round(5.6)
```

Sqrt

Returns the square root of the specified number.

```
Sqrt(x float|int|string) float
```

x
A number.

Example

```
val = Sqrt(225)
```

StringToBytes

Converts a string to *bytes*.

Syntax

```
StringToBytes(src string) bytes
```

src
Input string.

Example

```
var b bytes
b = StringToBytes("my string")
```

BytesToString

Converts *bytes* to a string.

Syntax

```
BytesToString(src bytes) string
```

src
: Input string.

Example

```
var s string
s = BytesToString($Bytes)
```

SysParamString

Returns the value of the specified system parameter.

Syntax

```
SysParamString(name string) string
```

name
: Parameter name.

Example

```
url = SysParamString(`blockchain_url`)
```

SysParamInt

Returns the value of the specified system parameter in the form of a number.

Syntax

```
SysParamInt(name string) int
```

name
: System parameter name.

Example

```
maxcol = SysParam(`max_columns`)
```

DBUpdateSysParam

Updates the value and the condition of the system parameter. If you do not need to change the value or condition, then specify an empty string in the corresponding parameter.

Syntax

```
DBUpdateSysParam(name, value, conditions string)
```

name
System parameter name.

value
New value for the parameter.

conditions
New condition for changing the parameter.

Example

```
DBUpdateSysParam(`fuel_rate`, `400000000000`, ``)
```

UpdateNotifications

Obtains a list of notifications for the specified keys from the database, and sends the obtained notifications to Centrifugo.

Syntax

```
UpdateNotifications(ecosystemID int, keys int ...)
```

ecosystemID
Ecosystem identifier.

key
List of keys, separated by commas. As an alternative, you can specify one array with a list of keys.

Example

```
UpdateNotifications($ecosystem_id, $key_id, 23345355454, 35545454554)
UpdateNotifications(1, [$key_id, 23345355454, 35545454554] )
```

UpdateRolesNotifications

Obtains a list of notifications for all keys with specified role identifiers from the database, and sends the obtained notifications to Centrifugo.

Syntax

```
UpdateRolesNotifications(ecosystemID int, roles int ...)
```

ecosystemID
Ecosystem identifier.

roles
List of role identifiers, separated by commas. As an alternative, you can specify one array with a list of keys.

Example

```
UpdateRolesNotifications(1, 1, 2)
```

HTTPRequest

Sends an HTTP request to a specified address.

Note: This function can be used only in Off-Blockchain Server (OBS) contracts.

Syntax

```
HTTPRequest(url string, method string, heads map, pars map) string
```

url
Address, to which the request will be sent.

method
Request type (GET or POST)

heads
An array of data for header formation.

pars
Parameters.

Example

```
var ret string
var pars, heads, json map
heads["Authorization"] = "Bearer " + $auth_token
pars["obs"] = "true"
```

(continues on next page)

(continued from previous page)

```
ret = HTTPRequest("http://localhost:7079/api/v2/content/page/default_page", "POST",␣
↪heads, pars)
json = JSONToMap(ret)
```

HTTPPostJSON

This function is similar to the *HTTPRequest* function, but it sends a *POST* request and parameters are passed in one string.

Note: This function can be used only in Off-Blockchain Server (OBS) contracts.

Syntax

```
HTTPPostJSON(url string, heads map, pars string) string
```

url
: Address, to which the request will be sent.

heads
: Data array for header formation.

pars
: Parameters as a json string.

Example

```
var ret string
var heads, json map
heads["Authorization"] = "Bearer " + $auth_token
ret = HTTPPostJSON("http://localhost:7079/api/v2/content/page/default_page", heads, `{
↪"obs":"true"}`)
json = JSONToMap(ret)
```

BlockTime

Returns generation time of a block in SQL format.

Use this function instead of the `NOW()` function.

Syntax

```
BlockTime()
```

Example

```
var mytime string
mytime = BlockTime()
DBInsert("mytable", myid, {time: mytime})
```

DateTime

Converts unixtime to a string in the *YYYY-MM-DD HH:MI:SS* format.

Syntax

```
DateTime(unixtime int) string
```

Example

```
DateTime(1532325250)
```

UnixDateTime

Converts a string in the *YYYY-MM-DD HH:MI:SS* format to unixtime.

Syntax

```
UnixDateTime(datetime string) int
```

Example

```
UnixDateTime("2018-07-20 14:23:10")
```

CreateOBS

Creates a child OBS (Off-Blockchain Server).

Syntax

```
CreateOBS(OBSName string, DBUser string, DBPassword string, OBSAPIPort int)
```

OBSName
: Name for the created OBS.

DBUser
: Role name for the database.

DBPassword
Password for this role.

OBSAPIPort
Port for API requests.

ListOBS

Returns a list of child OBSs (Off-Blockchain Servers).

This function can be used only in OBS Master mode.

Syntax

```
ListOBS()
```

Returned value

An associative array where keys are OBS names and values are process statuses.

RunOBS

Runs a process for an OBS.

This function can be used only in OBS Master mode.

Syntax

```
RunOBS(OBSName string)
```

OBSName
Name for an OBS.

Can contain only letters and numbers. Space symbols cannot be used.

StopOBS

Stops the process of a specified OBS.

This function can be used only in OBS Master mode.

Syntax

```
StopOBS(OBSName string)
```

OBSName
Name for a OBS.

Can contain only letters and numbers. Space symbols cannot be used.

RemoveOBS

Removes the process of the specified OBS. Stops and removes linked processes.

This function can be used only in OBS Master mode.

Syntax

```
RemoveOBS(OBSName string)
```

OBSName
: Name for a OBS.

  Can contain only letters and numbers. Space symbols cannot be used.

2.9.4 System Contracts

System contracts are created by default during product installation. All of these contracts are created in the first ecosystem, that's why you need to specify their full name to call them from other ecosystems, for instance, `@1NewContract`.

NewEcosystem

This contract creates a new ecosystem. To get an identifier of the newly created ecosystem, take the *result* field, which will return in txstatus.

Parameters:

- *Name string "optional"* - name for the ecosystem. This parameter can be set and/or chanted later.

EditEcosystemName

This contract changes ecosystem name in the *1_ecosystems* table. This table exists only in the first ecosystem.

Parameters:

- *SystemID* - ,
- *NewName* -

MoneyTransfer

This contract transfers money from the current account in the current ecosystem to a specified account.

Parameters:

- *Recipient string* - recipient's account in any format – a number or `XXXX-....-XXXX`.
- *Amount string* - transaction amount in qAPL.
- *Comment string "optional"* - comments.

NewContract

This contract creates a new contract in the current ecosystem.

Parameters:

- *Value string* - text of the contract, there must be only one contract on the upper level.
- *Conditions string* - contract change conditions.
- *Wallet string "optional"* - identifier of user's id where contract must be tied.
- *TokenEcosystem int "optional"* - identifier of the ecosystem, which currency will be used for transactions when the contract is activated.

EditContract

Edits the contract in the current ecosystem.

Parameters:

- *Id int* - ID of the contract to be edited.
- *Value string "optional"* - text of the contract or contracts.
- *Conditions string "optional"* - rights for contract change.

BindWallet

Binds a contract to an account in the current ecosystem. Contracts can be bound only from the account that was specified when the contract was created. After the contract is bound to an account, this account will pay for the execution of this contract.

Parameters:

- *Id int* - identifier of the contract to bind.

UnbindWallet

Unbinds a contract from an account in the current ecosystem. Only the account that is bound to the contract can unbind it. After the contract is unbound, a user that executes the contract will pay for the execution.

Parameters:

- *Id int* - identifier of the bound contract.

NewParameter

This contract adds a new parameter to the current ecosystem.

Parameters:

- *Name string* - parameter name.
- *Value string* - parameter value.
- *Conditions string* - rights for parameter change.

EditParameter

This contract changes an existing parameter in the current ecosystem.

Parameters:

- *Name string* - name of the parameter to be changed.
- *Value string* - new value.
- *Conditions string* - new condition for parameter change.

NewMenu

This contract adds a new menu in the current ecosystem.

Parameters:

- *Name string* - menu name.
- *Value string* - menu text.
- *Title string "optional"* - menu header.
- *Conditions string* - rights for menu change.

EditMenu

This contract changes an existing menu in the current ecosystem.

Parameters:

- *Id int* - ID of the menu to be changed.
- *Value string "optional"* - new text of menu.
- *Title string "optional"* - menu header.
- *Conditions string "optional"* - new rights for page change.

AppendMenu

This contract adds text to an existing menu in the current ecosystem.

Parameters:

- *Id int* - complemented menu identifier.
- *Value string* - text to be added.

NewPage

This contract adds a new page in the current ecosystem.

Parameters:

- *Name string* - page name.
- *Value string* - page text.
- *Menu string* - name of the menu, attached to this page.

- *Conditions string* - rights for changing this page.
- *ValidateCount int "optional"* - number of nodes that is required for page validation. If this parameter is not specified, then *min_page_validate_count* ecosystem parameter value is used. This value cannot be less than *min_page_validate_count* and greater than *max_page_validate_count*.
- *ValidateMode int "optional"* - mode of page validity checks. A value of 0 means that a page is checked upon loading the page. A value of 1 means that a page is checked upon loading and leaving the page.

EditPage

This contract changes an existing page in the current ecosystem.

Parameters:

- *Id int* - ID of the page to be changed.
- *Value string "optional"* - new text of the page.
- *Menu string "optional"* - name of the new menu on the page.
- *Conditions string "optional"* - new rights for page change.
- *ValidateCount int "optional"* - number of nodes that is required for page validation. If this parameter is not specified, then *min_page_validate_count* ecosystem parameter value is used. This value cannot be less than *min_page_validate_count* and greater than *max_page_validate_count*.
- *ValidateMode string "optional"* - mode of page validity checks. A value of 0 means that a page is checked upon loading the page. A value of 1 means that a page is checked upon loading and leaving the page.

AppendPage

The contract adds text to an existing page in the current ecosystem.

Parameters:

- *Id int* - ID of the page to be changed.
- *Value string* - text that needs to be added to the page.

NewBlock

This contract adds a new page block with a template to the current ecosystem.

Parameters:

- *Name string* - block name.
- *Value string* - block text.
- *Conditions string* - rights for block change.

EditBlock

This contract changes an existing block in the current ecosystem.

Parameters

- *Id int* - ID of the block to be changed.
- *Value string* - new text of a block.

- *Conditions string* - new rights for change.

NewTable

This contract adds a new table in the current ecosystem.

Parameters:

- *Name string* - table name in Latin script.
- *Columns string* - array of columns in JSON format `[{"name":"...", "type":"...","index": "0", "conditions":"..."},...]`, where
  - *name* - column name in Latin script.
  - *type* - type `varchar,bytea,number,datetime,money,text,double,character`.
  - *index* - non-indexed field - "0"; create index - "1".
  - *conditions* - condition for changing data in a column; read access rights must be specified in the JSON format. For example, `{"update":"ContractConditions(`MainCondition`)", "read":"ContractConditions(`MainCondition`)"}`
- *Permissions string* - access conditions in JSON format `{"insert": "...", "new_column": "...", "update": "..."}`.
  - *insert* - rights to insert records.
  - *new_column* - rights to add columns.
  - *update* - rights to change rights.

EditTable

This contract changes access permissions to tables in the current ecosystem.

Parameters:

- *Name string* - table name.
- *Permissions string* - access permissions in JSON format `{"insert": "...", "new_column": "...", "update": "..."}`.
  - *insert* - condition to insert records.
  - *new_column* - condition to add columns.
  - *update* - condition to change data.

NewColumn

This contract adds a new column to a table in the current ecosystem.

Parameters:

- *TableName string* - table name in.
- *Name* - column name in Latin script.
- *Type* - type `varchar,bytea,number,money,datetime,text,double,character`.
- *Index* - non-indexed field - "0"; create index - "1".

- *Permissions* - condition for changing data in a column; read access rights must be specified in the JSON format. For example, `{"update":"ContractConditions(`MainCondition`)", "read":"ContractConditions(`MainCondition`)"}`

EditColumn

This contract changes the rights to change a table column in the current ecosystem.

Parameters:

- *TableName string* - table name in Latin script.
- *Name* - column name in Latin script.
- *Permissions* - condition for changing data in a column; read access rights must be specified in the JSON format. For example, `{"update":"ContractConditions(`MainCondition`)", "read":"ContractConditions(`MainCondition`)"}`.

NewLang

This contract adds language resources in the current ecosystem. Permissions to add resources are set in the *changing_language* parameter in the ecosystem configuration.

Parameters:

- *Name string* - name of the language resource in Latin script.
- *Trans* - language resources as a string in JSON format with two-character language codes as keys and translated strings as values. For example: `{"en": "English text", "ru": " "}`.
- *[Lang string]* - optional parameter that specifies the language for error messages generated during the contract execution.

EditLang

This contract updates the language resource in the current ecosystem. Permissions to make changes are set in the *changing_language* parameter in the ecosystem configuration.

Parameters:

- *Id int*- language resource ID.
- *Name string* - name of the language resource.
- *Trans* - language resources as a string in JSON format with two-character language codes as keys and translated strings as values. For example `{"en": "English text", "ru": " "}`.
- *[Lang string]* - optional parameter that specifies the language for error messages generated during the contract execution.

NewSign

This contract adds the signature confirmation requirement for a contract in the current ecosystem.

Parameters:

- *Name string* - name of the contract, where an additional signature confirmation will be required.
- *Value string* - description of parameters in a JSON string, where

  - *title* - message text.
  - *params* - array of parameters that are displayed to users, where **name** is the field name, and **text** is the parameter description.
- *Conditions string* - condition for changing the parameters.

Example of *Value*:

```
{"title": "Would you like to sign?", "params":[{"name": "Recipient", "text":
"Wallet"},{"name": "Amount", "text": "Amount(EGS)"}]}
```

EditSign

The contract updates the parameters of a contract with a signature in the current ecosystem.

Parameters:

- *Id int* - identifier of the signature to be changed.
- *Value string* - a string containing new parameters.
- *Conditions string* - new condition for changing the signature parameters.

Import

This contract imports data from a *.sim file into the ecosystem.

Parameters:

- *Data string* - data to be imported in text format; this data is the result of export from an ecosystem to a .sim file.

NewCron

The contract adds a new task in cron to be launched by timer. The contract is available only in OBS.

Parameters:

- *Cron string* - a string that defines the launch of the contract by timer in the *cron* format.
- *Contract string* - name of the contract to launch in OBS; the contract must not have parameters in its `data` section.
- *Limit int* - an optional field, where the number of contract launches can be specified (until contract is executed this number of times).
- *Till string* - an optional string with the time when the task must be ended (this feature is not yet implemented).
- *Conditions string* - rights to modify the task.

EditCron

This contract changes the configuration of a task in cron for launch by timer. The contract is available only in OBS.

Parameters:

- *Id int* - task ID.
- *Cron string* - a string that defines the launch of the contract by timer in the *cron* format; to disable a task, this parameter must be either an empty string or absent.

- *Contract string* - name of the contract to launch in OBS; the contract must not have parameters in its data section.
- *Limit int* - an optional field, where the number of contract launches can be specified (until contract is executed this number of times).
- *Till string* - an optional string with the time of task must be ended (this feature is not yet implemented).
- *Conditions string* - new rights to modify the task.

NewAppParam

Adds a new app parameter to the current ecosystem.

Parameters:

- *App int* - application identifier.
- *Name string* - parameter name.
- *Value string* - parameter value.
- *Conditions string* - rights to change the parameter.

EditAppParam

Changes an existing app parameter in the current ecosystem.

Parameters:

- *Id int* - parameter indentifier.
- *Value string* - new value for the parameter.
- *Conditions string* - new rights to change the parameter.

NewDelayedContract

Adds a new task to the delayed contract scheduler.

The delayed contracts scheduler runs contracts that are required for the currently generated block.

Parameters:

- *Contract string* - contract name.
- *EveryBlock int* - the contract will be executed every this amount of blocks.
- *Conditions string* - rights for changing the task.
- *BlockID int "optional"* - block number where the contract must be executed. If this value is not specified, then it is calculated automatically by adding *EveryBlock* to the current block number.
- *Limit int "optional"* - maximum number of task executions. If this value is not specified, then the task will be executed for an unlimited amount of time.

EditDelayedContract

Changes a task in the delayed contract scheduler.

Parameters:

- *Id int* - task identifier.
- *Contract string* - contract name.
- *EveryBlock int* - the contract will be executed every this amount of blocks.
- *Conditions string* - rights for changing the task.
- *BlockID int "optional"* - block number where the contract must be executed. If this value is not specified, then it is calculated automatically by adding *EveryBlock* to the current block number.
- *Limit int "optional"* - maximum number of task executions. If this value is not specified, then the task will be executed for an unlimited amount of time.
- *Deleted int "optional"* - task toggle. A value of *1* disables the task. A value of *0* enables the task.

UploadBinary

The contract adds/rewrites a static file in X_binaries. When calling a contract via HTTP API, `multipart/form-data` must be used; the `DataMimeType` parameter will be used with the form data.

Parameters:

- *Name string* - static file name.
- *Data bytes "file"* - content of the static file.
- *DataMimeType string "optional"* - mime type of the static file.
- *AppID int* - application identifier.
- *MemberID int "optional"* - ecosystem member identifier. Is 0 by default.

If the DataMimeType is not passed, then `application/octet-stream` is used by default. If MemberID is not passed, then the static file is considered a system file.

2.10 User Interfaces

2.10.1 Interfaces building

Integrated Development Environment of the Molis software client, which is created using the *JavaScript React library*, includes an interface editor and a virtual interface designer. Interface pages are the essential part of applications that provides for retrieval and display of data from database tables, creation of forms for receipt of user input data, passing data to contracts, and navigation between application pages. Interface pages, just as contracts, are stored in the blockchain, which ensures their protection from falsification when loading them in the software client.

Interface template engine

Interface elements (pages and menus) are formed on Validating Nodes in a so-called *template engine* from templates created by programmers in the interface editor of the Molis software client. All interface pages are built using the Protypo functional language developed by platform developers. Interfaces are requested from nodes on the network using the *content* API command. What the template engine sends as a reply to such request is not an HTML page, but a JSON code comprised of HTML tags that form a tree in accordance with the template structure. For testing purposes, a POST request can be sent to `api/v2/content` with the *template* parameter containing the name of a template to process.

Creating interface templates

Interfaces can be created and edited using a specialized editor, available in the **Interface** section of administrative tools in Molis. The editor provides for:

- Writing codes of interface pages with highlighting of keywords of the Protypo template language,
- Selecting a menu, which will be displayed on the page,
- Editing the page menu,
- Configuring permission to edit the page (typically, by way of specifying the name of the contract with permissions in the *ContractConditions* function, or by direct indication of access rights in the *Change conditions* field),
- Launching a visual interface designer,
- Page preview.

Visual interface designer

Visual Interface Designer allows for creating page designs without resorting to the interface source code in Protypo language. The Designer allows for setting the positions of form elements and text on the page using drag-and-drop, as well as configuring sizes and design of page blocks. The Designer provides a set of ready-to-use blocks for displaying typical data models: panels with headers, forms, and information panels. The program logics (receipt of data and conditional constructs) can be added in the page editor after the page design is created. (In the future, we plan to create a full-scale visual interface editor.)

Use of styles

By default, interface pages are displayed using Angular Bootstrap Angle classes. If needed, users can create their own styles. Storage of styles is implemented using a special stylesheet parameter of the ecosystem configuration table.

Page blocks

To use typical code fragments on multiple interface pages there is an option to create page blocks and embed them in the interface code using the Insert command. Such blocks can be created and edited on the Interface page of the administrative section in Molis. For blocks, just as for pages, permissions for editing can be defined.

Language resources editor

The Molis software client includes a mechanism for interface localization using a special function of the Protypo template language – LangRes, which substitutes the language resource labels on the page with corresponding text lines in the language selected by the user in the software client (or browser for the web-version of the client). A shorter syntax $lable$ can be used instead of the LangRes function. Translation of messages in pop-up windows, initiated by contracts, is carried out by the LangRes function of the Simvolio language.

Language resources can be created and edited in the Language resources section of the administrative tools of the Molis software client. A language resource consists of a label (name) and the translations of this name into different languages with the indication of corresponding two-character language identifiers (EN, FR, JP, etc.).

Rights to add and change language resources can be configured using the same way as for any other table in the languages table (Tables section of the Molis administrative tools).

2.10.2 Protypo template language

Protypo functions provide for implementation of the following operations:

- retrieving values from the database: DBFind,
- representation of data retrieved from the database as tables and diagrams,
- assignment and display of values of variables, operations with data: SetVar, GetVar, Data,
- display and comparison of date/time values: DateTime, Now, CmpTime,
- building forms with various sets of user data input fields: Form, ImageInput, Input, RadioGroup, Select,
- validation of data in the form fields by displaying error messages: Validate, InputErr,
- display of navigation elements: AddToolButton, LinkPage, Button,
- calling contracts: Button,
- creation of HTML page layout elements – various containers with an option to specify css classes: Div, P, Span, etc.,
- embedding images onto a page and uploading of images: Image and ImageInput,
- conditional display of page layout fragments: `If, ElseIf, Else`,
- creation of multi-level menus,
- interface localization.

Overview of Protypo

Page template language is a functional language that allows for calling functions using `FuncName(parameters)`, and for nesting functions into each other. Parameters can be specified without quote marks. Unnecessary parameters can be dropped.

```
Text FuncName(parameter number 1, parameter number 2) another text.
FuncName(parameter 1,,,parameter 4)
```

If a parameter contains a comma, it should be enclosed in quotes marks (back quotes or double quotes). If a function can have only one parameter, commas can be used in it without quotes. Also, quotes should be used in case a parameter has an unpaired closing parenthesis.

```
FuncName("parameter number 1, the second part of first paremeter")
FuncName(`parameter number 1, the second part of first paremeter`)
```

If you put a parameter in quotes, but a parameter itself includes quotes, then you can use different type of quotes or double them in the text.

```
FuncName("parameter number 1, ""the second part of first"" paremeter")
FuncName(`parameter number 1, "the second part of first" paremeter`)
```

In description of functions, every parameter has a specific name. You can call functions and specify parameters in the order they were declared, or specify any set of parameters in any order by their names: ''Parameter_name: Parameter_value''. This approach allows to safely add new function parameters without breaking the compatibility with current templates. For example, all of these calls are correct in terms of language use for a function described as ''FuncName(Class,Value,Body)'':

```
FuncName(myclass, This is value, Div(divclass, This is paragraph.))
FuncName(Body: Div(divclass, This is paragraph.))
FuncName(myclass, Body: Div(divclass, This is paragraph.))
FuncName(Value: This is value, Body:
     Div(divclass, This is paragraph.)
)
FuncName(myclass, Value without Body)
```

Functions can return text, generate HTML elements (for instance, ''Input''), or create HTML elements with nested HTML elements (''Div, P, Span''). In the latter case a parameter with a pre-defined name **Body** should be used to define nested elements. For example, two *div*, nested in another *div*, can look like this:

```
Div(Body:
   Div(class1, This is the first div.)
   Div(class2, This is the second div.)
)
```

To define nested elements, which are described in the *Body* parameter, the following representation can be used: `FuncName(...){...}`. Nested elements should be specified in curly braces.

```
Div(){
   Div(class1){
      P(This is the first div.)
      Div(class2){
          Span(This is the second div.)
      }
   }
}
```

If you need to specify the same function a number of times in a row, you can use points instead of writing the function name every time. For example, the following lines are equal:

```
Span(Item 1)Span(Item 2)Span(Item 3)
Span(Item 1).(Item 2).(Item 3)
```

The language allows for assigning variables using the **SetVar** function. To substitute values of variables use `#varname#`.

```
SetVar(name, My Name)
Span(Your name: #name#)
```

To substitute the language resources of the ecosystem, you can use the `$langres$`, where *langres* is the name of the language source.

```
Span($yourname$: #name#)
```

The following variables are predefined:

- `#key_id#` - current user account identifier,
- `#ecosystem_id#` - current ecosystem identifier.
- `#guest_key#` - guest wallet identifier.
- `#isMobile#` - is 1 if the client is running on a mobile device.

Passing parameters to a page using PageParams

There is a number of functions that support the **PageParams** parameter, which serves for passing parameters when redirecting to a new page. For example, `PageParams: "param1=value1,param2=value2"`. Parameter values can be both simple strings or rows with value substitution. When parameters are passed to a page, variables with parameter names are created; for example, `#param1#` and `#param2#`.

- `PageParams: "hello=world"` - the page will receive the hello parameter with world as value,
- `PageParams: "hello=#world#"` - the page will receive the hello parameter with the value of the world variable.

Additionally, the **Val** function allows for obtaining data from forms, which were specified in redirect. In this case,

- `PageParams: "hello=Val(world)"` - the page will receive the hello parameter with the value of the world form element.

Calling contracts

Protypo implements contract calling by clicking on a button in a form (*Button* function). Once this event is initiated, the data entered by the user in the fields of the interface forms is passed to the contract (if the names of form fields correspond to the names of variables in the data section of the called contract, data is transferred automatically). The Button function allows for opening a modal window for user verification of the contract execution (Alert), and initiation of redirect to a specified page after the successful execution of the contract, and passing certain parameters to this page.

2.10.3 Protypo functions by purpose

Operations with variables

- *GetVar*
- *SetVar*
- *VarAsIs*

Navigation

- *AddToolButton*
- *Button*
- *LinkPage*

Data operations

- *Calculate*
- *CmpTime*
- *DateTime*
- *Money*

Displaying data

- *Code*
- *CodeAsIs*
- *Chart*
- *ForList*
- *Hint*
- *Image*
- *MenuGroup*
- *MenuItem*
- *QRcode*
- *Table*

Receiving data

- *Address*
- *AddressToId*
- *AppParam*
- *Data*
- *DBFind*
- *EcosysParam*
- *GetHistory*
- *GetColumnType*
- *JsonToSource*
- *ArrayToSource*
- *LangRes*
- *Range*
- *SysParam*
- *Binary*
- *TransactionInfo*

Elements of data formatting

- *Div*
- *Em*
- *P*
- *SetTitle*
- *Label*

- *Span*
- *Strong*

Elements of forms

- *Form*
- *ImageInput*
- *Input*
- *InputErr*
- *RadioGroup*
- *Select*
- *InputMap*
- *Map*

Operations with code

- *If*
- *And*
- *Or*
- *Include*

2.10.4 Protypo functions reference

Address

This function returns the account address in the `1234-5678-...-7990` format given the numerical value of the address; if the address is not specified, the address of the current user will be taken as the argument.

Syntax

```
Address (account)
```

Address

account
Account identifier.

Example

```
Span(Your wallet: Address(#account#))
```

AddressToId

Returns the account identifier for the specified account address in the `1234-5678-...-7990` format.

Syntax

```
AddressToId(Wallet)
```

AddressToId

Wallet
Account address in the XXXX-...-XXXX format or as a number.

Example

```
AddressToId(#wallet#)
```

AddToolButton

Adds a button to the buttons panel. Creates **addtoolbutton** element.

Syntax

```
AddToolButton(Title, Icon, Page, PageParams)
    [.Popup(Width, Header)]
```

AddToolButton

Title
Button title.

Icon
Icon for the button.

Page
Page name for the jump.

PageParams
Parmeters that are passed to the page.

Popup
Outputs a modal window.

Header
Window header.

Width
Window width in percent.

Range of values for this parameter is from 1 to 100.

Example

```
AddToolButton(Help, help, help_page)
```

And

This function returns the result of execution of the **and** logical operation with all parameters listed in parentheses and separated by commas. The parameter value will be `false` if it equals an empty string (`""`), zero or *false*. In all other cases the parameter value is `true`. The function returns 1 if true or 0 in all other cases. The element named `and` is created only when a tree for editing is requested.

Syntax

```
And(parameters)
```

Example

```
If(And(#myval1#,#myval2#), Span(OK))
```

AppParam

Outputs the value of an app parameter. The value is taken from the app_param table of the current ecosystem. If there is a language resource with the given name, then its value will be substituted automatically.

Syntax

```
AppParam(App, Name, Index, Source)
```

AppParam

App
Application identifier.

Name
Parameter name.

Index
This parameter can be used when the parameter value is a list of items separated by commas.

Index of a parameter element, starting from 1. For example if `type = full,light` then `AppParam(1, type, 2)` returns `light`.

This parameter cannot be used with *Source* parameter.

Source
This parameter can be used when the parameter value is a list of items separated by commas.

Creates a *data* object. Elements of this object are values of the specified parameter. The object can be used as a data source in *Table* and *Select* functions.

This parameter cannot be used with *Index* parameter.

Example

```
AppParam(1, type, Source: mytype)
```

ArrayToSource

Creates an **arraytosource** element and populates it with *key* - *value* pairs that were passed in a JSON array. The resulting data is put into the *Source* element, which can later be used in functions that use source inputs (such as *Table*).

Syntax

```
ArrayToSource(Source, Data)
```

ArrayToSource

Source
Data source name.

Data
A JSON array or a name of a variable (`#name#`) that holds a JSON array.

Example

```
ArrayToSource(src, #myjsonarr#)
ArrayToSource(dat, [1, 2, 3])
```

Binary

Returns a link to a static file that is stored in the *binaries* table.

Syntax

```
Binary(Name, AppID, MemberID)[.ById(ID)][.Ecosystem(ecosystem)]
```

Binary

Name
File name.

AppID
Application identifier.

MemberID
Account identifier. The default value is 0.

ID
Static file identifier.

ecosystem
Ecosystem identifier. If this parameter is not specified, binary file is requested from the current ecosystem.

Example

```
Image(Src: Binary("my_image", 1))
Image(Src: Binary().ById(2))
Image(Src: Binary().ById(#id#).Ecosystem(#eco#))
```

Button

Creates a **button** HTML element. This element creates a button, which executes a contract or opens a page.

Syntax

```
Button(Body, Page, Class, Contract, Params, PageParams)
    [.CompositeContract(Contract, Data)]
    [.Alert(Text, ConfirmButton, CancelButton, Icon)]
    [.Popup(Width, Header)]
    [.Style(Style)]
    [.ErrorRedirect((ErrorID,PageName,PageParams)]
```

Button

Body
Child text or elements.

Page
Name of the page to redirect to.

Class
Classes for the button.

Contract
Name of the contract to execute.

Params
List of values to pass to the contract. By default, values of contract parameters (data `section`) are obtained from HTML elements (for example, input fields) with similarly-named identifiers (`id`). If the element identifiers differ from the names of contract parameters, then the assignment in the `contractField1=idname1, contractField2=idname2` format should be used. This parameter is returned to *attr* as an object `{field1: idname1, field2: idname2}`.

PageParams
Parameters for redirection to a page in the following format: `contractField1=idname1, contractField2=idname2`. In this case, variables with parameter names `#contractField1#` and `#contractField2` are created on the target page, and are assigned the specified values (see the parameter passing specifications in the "*Passing Parameters to a Page Using PageParams*" section above).

CompositeContract
Used for adding extra contracts for a button. CompositeContract can be used several times.

Name
Contract name.

Data
Contract parameters as a JSON array.

Alert
Displays a message.

Text
Message text.

ConfirmButton
Confirm button caption.

CancelButton
Cancel button caption.

Icon
Icon.

Popup
Outputs a modal window.

Header
Window header.

Width
Window width in percent.

Range of values for this parameter is from 1 to 100.

Style
Specifies CSS styles.

Style
CSS styles.

ErrorRedirect
Specifies a redirect page. This redirect page is used when the *Throw* function generates an error during the contract execution. There may be several *ErrorRedirect* calls. As a result, an *errredir* attribute is returnes with *ErrorID* list of keys and parameters as values.

ErrorID
Error identifier.

PageName
Name of the redirect page.

PageParams
Parameters passed to this page.

Example

```
Button(Submit, default_page, mybtn_class).Alert(Alert message)
Button(Contract: MyContract, Body:My Contract, Class: myclass, Params:"Name=myid,
↪Id=i10,Value")
```

Calculate

This function returns the result of an arithmetic expression passed in the **Exp** parameter. The following operations can be used: +, -, *, /, and parenthesis ().

Syntax

```
Calculate(Exp, Type, Prec)
```

Calculate

Exp
Arithmetic expression. Can contain numbers and *#name#* variables.

Type
Result data type: **int, float, money**. If not specified, then the result type will be *float* in case there are numbers with a decimal point, or *int* in all other cases.

Prec
The number of significant digits after the point can be specified for *float* and *money* types.

Example

```
Calculate( Exp: (342278783438+5000)\*(#val#-932780000), Type: money, Prec:18 )
Calculate(10000-(34+5)\*#val#)
Calculate("((10+#val#-45)\*3.0-10)/4.5 + #val#", Prec: 4)
```

Chart

Creates an HTML diagram.

Syntax

```
Chart(Type, Source, FieldLabel, FieldValue, Colors)
```

Chart

Type
Diagram type.

Source
Name of the data source, for example, a source taken from the *DBFind* command.

FieldLabel
Name of a field that will be used for headers.

FieldValue
Name of a field that will be used for values.

Colors
List of used colors.

Example

```
Data(mysrc,"name,count"){
    John Silver,10
    "Mark, Smith",20
    "Unknown ""Person""",30
}
Chart(Type: "bar", Source: mysrc, FieldLabel: "name", FieldValue: "count", Colors:
↪"red, green")
```

CmpTime

This function compares two time values in the same format.

Supports unixtime, `YYYY-MM-DD HH:MM:SS`, and any arbitrary format, if the sequence is followed from years to seconds, for example `YYYYMMDD`).

Syntax

```
CmpTime(Time1, Time2)
```

Return values

- `-1` - Time1 < Time2,
- `0` - Time1 = Time2,
- `1` - Time1 > Time2.

Example

```
If(CmpTime(#time1#, #time2#)<0){...}
```

Code

Creates a **code** element for displaying the specified code.

This function replaces variables (e.g. `#name#`) with their values.

Syntax

```
Code(Text)
```

Code

Text
Source code.

Example

```
Code( P(This is the first line.
    Span(This is the second line.))
)
```

CodeAsIs

Creates a **code** element for displaying the specified code.

This function does not replace variables with their values. For example, `#name#` will be displayed as is.

Syntax

```
CodeAsIs(Text)
```

CodeAsIs

Text
Source code.

Example

```
CodeAsIs( P(This is the #test1#.
    Span(This is the #test2#.))
)
```

Data

Creates a **data** element and fills it with specified data and put into the *Source*, that then should be specified in *Table* and other commands resivieng *Source* as the input data. The sequence of column names corresponds to that of *data* entry values.

Syntax

```
Data(Source,Columns,Data)
    [.Custom(Column){Body}]
```

Data

Source
Data source name. You can specify any name, which can be included in other commands later as a data source (e.g. *Table*).

Columns
List of columns, separated by commas.

Data
Data.

One record per line. Column values must be separated by commas. Data should be in the same order as set in *Columns*.

For values with commas, put the value in double quotes (`"example1, example2", 1, 2`). For values with quotes, put the value in double double quotes (`"""example", "example2""", 1, 2`).

Custom
Allows for assigning calculated columns for data. For example, you can specify a template for buttons and additional page layout elements. These fields are usually assigned for output to *Table* and other commands that use received data.

If you want to assign several calculated columns, use multiple *Custom* tail functions.

Column
Column name. A unique name must be assigned.

Body
A code fragment. You can obtain values from other columns in this entry using `#columnname#`, and then use these values in the code fragment.

Example

```
Data(mysrc,"id,name"){
"1",John Silver
2,"Mark, Smith"
3,"Unknown ""Person"""
 }.Custom(link){Button(Body: View, Class: btn btn-link, Page: user, PageParams: "id=
↪#id#"}
```

DateTime

Displays time and date in the specified format.

Syntax

```
DateTime(DateTime, Format)
```

DateTime

DateTime
Time and date in unix time or in a standard format `2006-01-02T15:04:05`.

Format
Format template: `YY` 2-digit year format, `YYYY` 4-digit year format, `MM` - month, `DD` - day, `HH` - hours, `MM` - minutes, `SS` – seconds. Example: `YY/MM/DD HH:MM`.

If the format is not specified, the *timeformat* parameter value set in the *languages* table will be used. If this parameter is absent, the `YYYY-MM-DD HH:MI:SS` format will be used instead.

Example

```
DateTime(2017-11-07T17:51:08)
DateTime(#mytime#,HH:MI DD.MM.YYYY)
```

DBFind

Creates a **dbfind** element, fills it with data from the *table* table, and puts it to the *Source* structure. The *Source* structure can be then used in *Table* and other commands that receive *Source* as input data. The sequence of records in *data* must correspond to the sequence of column names.

Syntax

```
DBFind(table, Source)
    [.Columns(columns)]
    [.Where(conditions)]
    [.WhereId(id)]
    [.Order(name)]
    [.Limit(limit)]
    [.Offset(offset)]
    [.Count(countvar)]
    [.Ecosystem(id)]
    [.Cutoff(columns)]
    [.Custom(Column){Body}]
    [.Vars(Prefix)]
```

DBFind

table
Table name.

Source
Data source name.

Columns

columns
List of columns to be returned. If not specified, all columns will be returned. If there are columns of JSON type, you can address the record fields using the following syntax: **columnname->fieldname**. In this case, the resulting column name will be **columnname.fieldname**.

Where

conditions
Data search conditions. For example, `.Where(name = '#myval#')`.

If there are columns of JSON type, you can address record fields using the following syntax: **columnname->fieldname**.

WhereId
Search by ID. For example, `.WhereId(1)`.

id
Record identifier.

Order
Sorting by field.

For more information about sorting syntax, see *DBFind*.

name
Field name.

Limit

limit
Number of returned rows. Default value is 25, maximum value is 250.

Offset

offset
Offset for returned rows.

Count
Total number of rows for the specified *Where* condition.

In addition to being stored in a variable, the total count is also returned in the *count* parameter of the *dbfind* element.

If *Where* and *WhereID* were not specified, then the total number of rows in a table will be returned.

countvar
Name of a variable that will hold the row count.

Ecosystem

id
Ecosystem ID. By default, data is taken from the specified table in the current ecosystem.

Cutoff
Is used for trimming and displaying a large volume of text data.

columns
List of columns separated by commas that must be processed by the *Cutoff* tail function.

As a result, column value is replaced by a JSON obkect with two fields: *link* and *title*. If the value in a column is longer than 32 symbols, then a link to a full text and first 32 symbols are returned. If the value is 32 symbols and shorter, then the link is empty, and the title holds the full column value.

Custom
Allows for assigning calculated columns for data. For example, you can specify a template for buttons and additional page layout elements. These fields are usually assigned for output to *Table* and other commands that use received data.

If you want to assign several calculated columns, use multiple *Custom* tail functions.

Column
Column name. A unique name must be assigned.

Body
A code fragment. You can obtain values from other columns in this entry using `#columnname#`, and then use these values in the code fragment.

Vars
Generates a set of variables with values from the first row obtained by the query. When specifying this function, the *Limit* parameter automatically becomes equal to 1 and only one record is returned.

Prefix
Prefix that is added to variable names. The format is *#prefix_columnname#*, where the column name follows the underscore sign. If there are columns containing JSON fields, then the resulting variable will be in the following format *#prefix_columnname_field#*.

Example

```
DBFind(parameters,myparam)
DBFind(parameters,myparam).Columns(name,value).Where(name='money')
DBFind(parameters,myparam).Custom(myid){Strong(#id#)}.Custom(myname){
   Strong(Em(#name#))Div(myclass, #company#)
}
```

Div

Creates a **div** HTML element.

Syntax

```
Div(Class, Body)
    [.Style(Style)]
    [.Show(Condition)]
    [.Hide(Condition)]
```

Div

Class
Classes for this *div*.

Body
Child elements.

Style
Specifies CSS styles.

Style
CSS styles.

Show

Defines conditions to show this block.

Condition
See *Hide* below.

Hide
Defines conditions to hide this block.

Condition

Sequence of `InputName=Value` expressions. *Condition* is true when all expressions that it contains are true. An expression is true when `InputName` input has the `Value` text. If several *Show* or *Hide* calls are specified, then at least one of the *Condition* parameters must be true.

Example

```
Div(class1 class2, This is a paragraph.).Show(inp1=test,inp2=none)
```

EcosysParam

This function gets a parameter value from the parameters table of the current ecosystem. If there is a language resource for the resulting name, it will be translated accordingly.

Syntax

```
EcosysParam(Name, Index, Source)
```

EcosysParam

Name
Parameter name.

Index
In cases where the requested parameter is a list of elements separated by commas, you can specify an index starting from 1. For example, if `gender = male,female`, then `EcosysParam(gender, 2)` will return `female`.

This parameter cannot be used with *Source* parameter.

Source
This parameter can be used when the parameter value is a list of items separated by commas.

Creates a *data* object. Elements of this object are values of the specified parameter. The object can be used as a data source in *Table* and *Select* functions.

This parameter cannot be used with *Index* parameter.

```
Address(EcosysParam(founder_account))
EcosysParam(gender, Source: mygender)

EcosysParam(Name: gender_list, Source: src_gender)
Select(Name: gender, Source: src_gender, NameColumn: name, ValueColumn: id)
```

Em

Creates an **em** HTML element.

Syntax

```
Em(Body, Class)
```

Em

Body
hild text or elements.

Class
Classes for this *em*.

Example

```
This is an Em(important news).
```

ForList

Displays a list of elements from the *Source* data source in the template format set out in *Body*, and creates the **forlist** element.

Syntax

```
ForList(Source, Index){Body}
```

ForList

Source
Data source from *DBFind* or *Data* functions.

Index
Variable for the iteration counter. Count starts from 1.

This parameter is optional. If it is not specified, the iteration count value is written to the *[Source]_index* variable.

Body
A template to insert the elements in.

```
ForList(mysrc){Span(#mysrc_index#. #name#)}
```

Form

Creates a **form** HTML element.

Syntax

```
Form(Class, Body) [.Style(Style)]
```

Form

Body
Child class or elements.

Class
Classes for this *form*.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
Form(class1 class2, Input(myid))
```

GetColumnType

Returns the type of a column in a specified table.

Following column types can be returned: *text, varchar, number, money, double, bytes, json, datetime, double*.

Syntax

```
GetColumnType(Table, Column)
```

GetColumnType

Table
Table name.

Column
Column name.

Example

```
SetVar(coltype,GetColumnType(members, member_name))Div(){#coltype#}
```

GetHistory

Creates a **gethistory** element and popuates it with the history of changes of a record from the specified table. The resulting data is put into the *Source* element, which can later be used in functions that use source inputs (such as *Table*).

The resulting list is sorted in the order from recent changes to earlier ones.

The *id* field in the resulting table points to the id in the *rollback_tx* table. The *block_id* field contains the block number. The *block_time* field contains the block timestamp.

Syntax

```
GetHistory(Source, Name, Id, RollbackId)
```

GetHistory

Source
Name for the data source.

Name
Table name.

Id
Identifier of a record.

RollbackId
Optional parameter. If specified, only one record with the specified identifier will be returned from the *rollback_tx* table.

Example

```
GetHistory(blocks, BlockHistory, 1)
```

GetVar

This function returns the value of the current variable if it exists, or returns an empty string if a variable with this name is not defined. An element with **getvar** name is created only when a tree for editing is requested. The difference between `GetVar(varname)` and `#varname#` is that in case *varname* does not exist, *GetVar* will return an empty string, whereas *#varname#* will be interpreted as a string value.

Syntax

```
GetVar(Name)
```

GetVar

Name
Variable name.

Example

```
If(GetVar(name)){#name#}.Else{Name is unknown}
```

Hint

Creates a **hint** element to display hints.

Syntax

```
Hint(Icon,Title,Text)
```

Hint

Icon
Icon name.

Title
Hint title.

Text
Hint text.

Example

```
Hint(myicon, My Header, This is a hint text)
```

If

Conditional statement.

Child elements of the first *If* or *ElseIf* with fulfilled *Condition* are returned. Otherwise, child elements of *Else* are returned.

Syntax

```
If(Condition){ Body }
    [.ElseIf(Condition){ Body }]
    [.Else{ Body }]
```

If

Condition

A condition is considered to be not fulfilled if it equals an *empty string*, *0* or *false*. In all other cases the condition is considered fulfilled.

Body
Child elements.

Example

```
If(#value#){
   Span(Value)
}.ElseIf(#value2#){Span(Value 2)
}.ElseIf(#value3#){Span(Value 3)}.Else{
   Span(Nothing)
}
```

Image

Creates an **image** HTML element.

Syntax

```
Image(Src, Alt, Class)
    [.Style(Style)]
```

Image

Src
Image source, file or `data:...`.

Alt
Alternative text for the image.

lass
List of classes.

Example

```
Image(\images\myphoto.jpg)
```

ImageInput

Creates an **imageinput** element for image upload. In the third parameter you can specify either image height or aspect ratio to apply: *1/2*, *2/1*, *3/4*, etc. The default width is 100 pixels with *1/1* aspect ratio.

Syntax

```
ImageInput(Name, Width, Ratio, Format)
```

ImageInput

Name
Element name.

Width
Width of the cropped image.

Ratio
Aspect ratio (width to height) or height of the image.

Format
Format of the uploaded image.

Example

```
ImageInput(avatar, 100, 2/1)
```

Include

Inserts a template with a specified name to the page code.

Syntax

```
Include(Name)
```

Include

Name

Template name.

Example

```
Div(myclass, Include(mywidget))
```

Input

Creates an **input** HTML element.

Syntax

```
Input(Name, Class, Placeholder, Type, Value, Disabled)
    [.Validate(validation parameters)]
    [.Style(Style)]
```

Input

Name
Element name.

Class
Classes for this *input*.

Placeholder
The *placeholder* element for this *input*.

Type
Type of the *input*.

Value
Element value.

Disabled
If the *input* is disabled or not.

Validate
Validation parameters.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
Input(Name: name, Type: text, Placeholder: Enter your name)
Input(Name: num, Type: text).Validate(minLength: 6, maxLength: 20)
```

InputErr

Creates an **inputerr** element with validation error texts.

Syntax

```
InputErr(Name,validation errors)]
```

InputErr

Name
Name of the corresponding *Input* element.

validation errors
One or more parameters for validation error messages.

Example

```
InputErr(Name: name,
    minLength: Value is too short,
    maxLength: The length of the value must be less than 20 characters)
```

InputMap

Creates a text input field for an address. Provides an ability to select coordinates on a map.

Syntax

```
InputMap(Name, Type, MapType, Value)
```

InputMap

Name
Element name.

Value
Default value.

This value is an object in the string format. For example, `{"coords":[{"lat":number, "lng":number},]}` or `{"zoom":int, "center":{"lat":number,"lng":number}}`. The *address* field can be used to save the address value for cases when InputMap is created with a predefined *Value*, so that address field is not empty.

Type
Use `polygon` value for this parameter.

MapType
Map type.

This parameter can have the following values: `hybrid`, `roadmap`, `satellite`, `terrain`.

Example

```
InputMap(Name: Coords,Type: polygon, MapType: hybrid, Value: `{"zoom":8, "center":{
↪"lat":55.749942860682545,"lng":37.6207172870636}}`)
```

JsonToSource

Creates a **jsontosource** element and populates it with *key* - *value* pairs that were passed in a JSON oblect. The resulting data is put into the *Source* element, which can later be used in functions that use source inputs (such as *Table*).

The records in the resulting data is sorted by JSON keys, in alphabetical order.

Syntax

```
JsonToSource(Source, Data)
```

JsonToSource

Source
Data source name.

Data
A JSON oblect or a name of a variable (`#name#`)that holds a JSON array.

Example

```
JsonToSource(src, #myjson#)
JsonToSource(dat, {"param":"value", "param2": "value 2"})
```

Label

Creates a **label** HTML element.

Syntax

```
Label(Body, Class, For)
    [.Style(Style)]
```

Label

Body
Child text or elements.

Class
Classes for this *label*.

For
This label's *for* value.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
Label(The first item).
```

LangRes

Returns a specified language resource. In case of request to a tree for editing it returns the **langres** element. A short notation in the `$langres$` format can be used.

Syntax

```
LangRes(Name, Lang)
```

LangRes

Name
Name of language resource.

Lang
Two-character language identifier.

By default, the language defined in the *Accept-Language* request is returned.

Lcid identifiers can be specified, for example, *en-US,en-GB*. In this case, if the requested values will not be found, for example, for *en-US*, then the language resource will be looked for in *en*.

Example

```
LangRes(name)
LangRes(myres, fr)
```

LinkPage

Creates a **linkpage** element – a link to a page.

Syntax

```
LinkPage(Body, Page, Class, PageParams)
    [.Style(Style)]
```

LinkPage

Body
Child elements or text.

Page
Page to redirect to.

Class
Classes for this button.

PageParams
Redirection parameters.

Style
Specifies CSS styles.

Style
CSS styles

Example

```
LinkPage(My Page, default_page, mybtn_class)
```

Map

Creates a visual representation of a map and displays coordinates in an arbitrary format.

Syntax

```
Map(Hmap, MapType, Value)
```

Map

Hmap
HTML element height on a page.

The default value is 100.

Value
Map value, an object in the string format.

For example: `{"coords":[{"lat":number,"lng":number},]}` or `{"zoom":int, "center":{"lat":number,"lng":number}}`. If `center` is not specified, then map window will be automatically adjusted for the specified coordinates.

MapType
Map type.

This parameter can have the following values: `hybrid`, `roadmap`, `satellite`, `terrain`.

Example

```
Map(MapType:hybrid, Hmap:400, Value:{"coords":[{"lat":55.58774531752405,"lng":36.
↪97260184619233},{"lat":55.58396161622043,"lng":36.973803475831005},{"lat":55.
↪585222890513975,"lng":36.979811624024364},{"lat":55.58803635636347,"lng":36.
↪978781655762646}],"area":146846.65783403456,"address":"Unnamed Road, Moscow, Russia,
↪ 143041"})
```

MenuGroup

Creates a nested submenu in the menu and returns the **menugroup** element. The *name* parameter will also return the value of *Title* before replacement with language resources.

Syntax

```
MenuGroup(Title, Body, Icon)
```

MenuGroup

Title
Menu item name.

Body
Child elements in submenu.

Icon
Icon.

Example

```
MenuGroup(My Menu){
    MenuItem(Interface, sys-interface)
    MenuItem(Dahsboard, dashboard_default)
}
```

MenuItem

Creates a menu item and returns the **menuitem** element.

Syntax

```
MenuItem(Title, Page, Params, Icon, Obs)
```

MenuItem

Title
Menu item name.

Page
Page to redirect to.

Params
Parameters, passed to the page in the *var:value* format, separated by commas.

Icon
Icon.

Obs
This parameter that defines the transition to an off-blockchain server. If `Obs: true`, then the link redirects to an OBS; if `Obs: false`, then the link redirects to the blockchain; if the parameter was not specified, then it is defined based on where the menu was loaded.

Example

```
MenuItem(Interface, interface)
```

Money

Returns a string value of `exp/10^digit`. If *Digit* parameter is not specified, it is taken from the **money_digit** ecosystem parameter.

Syntax

```
Money(Exp, Digit)
```

Money

Exp
Numeric value as a string.

Digit
Exponent of the base 10 in the `exp/10^digit` expression. This value can be positive or negative. Positive value determines the number of digits after the comma.

Example

```
Money(Exp, Digit)
```

Or

Returns a result of the **IF** logical operation with all parameters specified in parentheses and separated by commas. The parameter value is considered `false` if it equals an empty string (`""`), 0 or `false`. In all other cases the parameter value is considered `true`. The function returns 1 for true or 0 in all other cases. Element named **or** is created only when the tree for editing is requested.

Syntax

```
Or(parameters)
```

Example

```
If(Or(#myval1#,#myval2#), Span(OK))
```

P

Creates a **p** HTML element.

Syntax

```
P(Body, Class)
    [.Style(Style)]
```

P

Body
hild text or elements.

Class
classes for this *p*.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
P(This is the first line.
  This is the second line.)
```

QRcode

Returns a *qrcode* element with a QR code for the specified text.

Syntax

```
QRcode(Text)
```

QRcode

Text
Text for the QR code.

Example

```
QRcode(#name#)
```

RadioGroup

Creates a **radiogroup** element.

Syntax

```
RadioGroup(Name, Source, NameColumn, ValueColumn, Value, Class)
    [.Validate(validation parameters)]
    [.Style(Style)]
```

RadioGroup

Name
Element name.

Source
Data source name from *DBFind* or *Data* functions.

NameColumn
Column name to use as a source of element names.

ValueColumn
Column name to use as a source of element values.

Columns created using *Custom* must not be used in this parameter.

Value
Default value.

Class
Classes for the element.

Validate
Validation parameters.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
DBFind(mytable, mysrc)
RadioGroup(mysrc, name)
```

Range

Creates a **range** element and fills it with integer values from *From* to *To* (*To* is not included) with a *Step* step. The resulting data is put into the *Source* element, which can later be used in functions that use source inputs (such as *Table*). Values are written to the *id* column. If invalid parameters are specified, an empty *Source* is returned.

Syntax

```
Range(Source,From,To,Step)
```

Range

Source
Data source name.

From
Starting value (i = From).

To
End value (i < To).

Step
Value change step. If this parameter is not specified a value of 1 is used.

Example

```
Range(my,0,5)
SetVar(from, 5).(to, -4).(step,-2)
Range(Source: neg, From: #from#, To: #to#, Step: #step#)
```

Select

Creates a **select** HTML element.

Syntax

```
Select(Name, Source, NameColumn, ValueColumn, Value, Class)
    [.Validate(validation parameters)]
    [.Style(Style)]
```

Select

Name
Element name.

Source
Data source name from *DBFind* or *Data* functions.

NameColumn
Column name to use as a source of element names.

ValueColumn
Column name to use as a source of element values.

Columns created using *Custom* must not be used in this parameter.

Value
Default value.

Class
Classes for the element.

Validate
Validation parameters.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
DBFind(mytable, mysrc)
Select(mysrc, name)
```

SetTitle

Sets the page title. The element **settitle** IS be created.

Syntax

```
SetTitle(Title)
```

SetTitle

Title
Page title.

Example

```
SetTitle(My page)
```

SetVar

Assigns a *Value* to a *Name* variable.

Syntax

```
SetVar(Name, Value)
```

SetVar

Name
Variable name.

Value
Value of the variable, which can contain a reference to another variable.

Example

```
SetVar(name, John Smith).(out, I am #name#)
Span(#out#)
```

Span

Creates a **span** HTML element.

Syntax

```
Span(Body, Class)
    [.Style(Style)]
```

Span

Body
Child class or elements.

Class
Classes for this *span*.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
This is Span(the first item, myclass1).
```

Strong

Creates a **strong** HTML element.

Syntax

```
Strong(Body, Class)
```

Strong

Body
Child class or elements.

Class
Classes for this *strong*.

Example

```
This is Strong(the first item, myclass1).
```

SysParam

Displays the value of a system parameter from the system_parameters table.

Syntax

```
SysParam(Name)
```

SysParam

Name
Parameter name.

Example

```
Address(SysParam(founder_account))
```

Table

Creates a **table** HTML element.

Syntax

```
Table(Source, Columns)
    [.Style(Style)]
```

Table

Source
Data source name as specified, for example, in the *DBFind* command.

Columns
Headers and corresponding column names, as follows: `Title1=column1,Title2=column2`.

Style
Specifies CSS styles.

Style
CSS styles.

Example

```
DBFind(mytable, mysrc)
Table(mysrc,"ID=id,Name=name")
```

TransactionInfo

The function searches a transaction by the specified hash and returns information about the executed contract and its parameters.

Syntax

```
TransactionInfo(Hash)
```

TransactionInfo

Hash
Transaction hash in a hex string format.

Return value

The function returns a string in the json format:

```
{"contract":"ContractName", "params":{"key":  "val"}, "block":  "N"}
```

Above,

- *contract* - contract name
- *params* - parameters passed to the contract
- *block* - block ID where this transaction was processed.

Example

```
P(TransactionInfo(#hash#))
```

VarAsIs

Assigns a value to a variable. The specified value is assigned as is, encountered variable names are not replaced with their values.

For a version with variables substitution, see *SetVar*.

Syntax

```
VarAsIs(Name, Value)
```

VarAsIs

Name
Variable name.

Value
Value. Variable names in value are not replaced.

For example, if *Value* is `example #varname#`, then the value of the variable will also be `example #varname#`.

Example

```
VarAsIs(name, I am #name#)
```

2.10.5 Styles for mobile app

Typography

Headings

- `h1` ... `h6`

Emphasis Classes

- `.text-muted`
- `.text-primary`
- `.text-success`
- `.text-info`
- `.text-warning`
- `.text-danger`

Colors

- `.bg-danger-dark`
- `.bg-danger`
- `.bg-danger-light`
- `.bg-info-dark`
- `.bg-info`
- `.bg-info-light`
- `.bg-primary-dark`
- `.bg-primary`
- `.bg-primary-light`
- `.bg-success-dark`
- `.bg-success`
- `.bg-success-light`
- `.bg-warning-dark`
- `.bg-warning`
- `.bg-warning-light`
- `.bg-gray-darker`
- `.bg-gray-dark`
- `.bg-gray`
- `.bg-gray-light`
- `.bg-gray-lighter`

Grid

- `.row`
- `.row.row-table`
- `.col-xs-1` ... `.col-xs-12` works only when the parent has `.row.row-table` class

Panel

- `.panel`
- `.panel.panel-heading`
- `.panel.panel-body`
- `.panel.panel-footer`

Form

- `.form-control`

Button

- `.btn.btn-default`
- `.btn.btn-link`
- `.btn.btn-primary`
- `.btn.btn-success`
- `.btn.btn-info`
- `.btn.btn-warning`
- `.btn.btn-danger`

Icons

- All icons from FontAwesome: `fa fa-<icon-name></icon-name>`.
- All icons from SimpleLineIcons: `icon-<icon-name>`.

2.11 Applications on Platform

An application on platform is a system of contracts, tables and interfaces that performs a certain function or provides a dedicated service. An application is not a autonomous program module - the only thing that unites the application elements is the performance of certain functions and exchange data. The boundaries of an application are not always strictly defined, since its elements can be simultaneously used in multiple applications.

The following is a typical functional fragment of an application:

1. Redirect to a page that displays:
    - information, received from database tables (*DBFind* function),
    - form fields for user input of new data,
    - button (*Button* function), which calls a contract.
2. On the contract call:
    - data from form fields is passed to the `data` section of the contract,
    - the `conditions` section checks user rights to launch this contract and the validity of data received from the page; if for some reason the contract cannot be executed, a message is displayed on the screen (`error`, `warning` or `info`), leaving the user on the same interface page,
    - the `action` section receives additional data from tables (DBFind function) and records data into the database (using *DBUpdate* and *DBInsert* functions).
3. After the contract has been successfully executed, the user is redirected to the page, whose name was specified in the *Page* parameter of the *Button* function that launched the contract, and the parameters listed in the *PageParams* are passed to the new page.

A page can have more than one button to call various contracts. Buttons that call contracts and redirect to pages can be built in rows of tables that display data about objects in the user interface. In this case, object identifiers are used in button parameters, which can be used to redirect to object-related pages (for example, object editing).

2.11.1 Tables and Data Storage

Applications use database tables that can be conditionally divided into two types:

1. Tables that store big arrays of structured data about objects (persons, organizations, property, etc.),
2. Document tables that store the current state of the processes implemented by a particular application (process type, stage, signatures, etc.) or data about its current operations (notifications, messages, records about voting and transactions).

Typically, register tables contain the most up-to-date information, which means that information about objects in the registers is updated as soon as a new piece of data is received. The work with documents is based on a completely different principle. Due to the fact that the *DBFind* functions of the Simvolio and Protypo languages can request information only from a single table (which means that they don't support *JOIN*), it is necessary to record exhaustive information (IDs, names, pictures) in the tables that store documents. For example, when requesting information from a table that stores messages, we need to receive not just the `id` of the sender, but all information required to display the message on an interface page, including the user name and avatar (userpic). After having being saved, the data in the document tables should not be modified. It should be noted that non-normalization of data in tables is not related only to the technical limitations of *JOIN* use, but is prescribed by the very ideology of the blockchain as a temporal database that is designed to store the complete history of events. This means that a document (for example, a message), which was signed and saved in a table, should not under any circumstances be modified in the future, even, for example, in case its author changes their name in the register of users (whereas such modification is inevitable under the relational data scheme).

2.11.2 Navigation

Users can switch between applications using sections, displayed in the software client as tabs, or using cascading menus inside these sections. After clicking on a section tab, users are redirected to the section's main page, which can be defined from the administrative tools.

Navigation within an application is carried out for the most part using the menus (each page has one menu liked to it). Transition between pages can be implemented using links (*LinkPage*) or button clicks (*Button*). In both cases parameters *PageParams* can be passed to the *Page* target page. Calling a new page is also possible after successful execution of a contract. In this case, the parameters of the *Button* function, which calls the contract, should include the name of the page to redirect to (*Page*) and the passed parameters *PageParams*.

Navigation in multi-user applications can be organized using ready-to-use *Notification* and *Roles* applications, which are installed by default in every ecosystem. The *Notification* application allows for display of messages to specific users or user roles in the Molis software client. A notification message consists of a header and a link to a page. Additionally, parameters can be passed using *PageParams* to the target page in the format, supported by the *LinkPage* and *Button* functions. Sometimes, a target page where, for example, a user needs to make some decision, can be accessed only from a notification, because there are no links from menus or other pages to this target page. Notifications are formed with contracts *Notifications_Single_Send* or *Notifications_Roles_Send*, which should be called from a contract that ends some functional stage of an application. After the user receives a notification, opens the target page and performs the required action (thus, executing a contract), the notification should be deactivated by calling the *Notifications_Single_Close* or *Notifications_Roles_Finishing* contract. The list of notifications and their statuses is available in the *Notification* application.

2.11.3 Interface Pages

Page structure is formed using the `If(Condition){Body } .ElseIf(Condition){Body} . Else{Body}` conditional statement, which can use the PageParam parameters sent to the page when it was called by the LinkPage or Button functions. If some code fragments need to be used in many pages, they should be recorded into page blocks. Such blocks can be included in a page using the *Include* function.

2.11.4 Variables, Column Names and Language Resources

The unification of names of variables (on pages and in contracts), identifiers of interface page form fields, table column names and language resource labels can help significantly speed up the development of applications and make the program code easier to read. Let's suppose we want to pass parameters from an interface page to a contract. In this case, if the name of the username variable in the data section of the contract corresponds to the name of the username field of an interface page, which was passed to this contract, then you don't need to specify this `username=username` pair in the *Params* parameters of the *Button* function. Using the same names for variables and column names makes it easier to use the DBInsert and DBUpdate functions; for example, `DBUpdate("member", $id, {username: $username})`. Using the same names for variables and language resource labels makes it easier to display the columns names of interface tables `Table(mysrc,"ID=id,$username$=username")`.

2.11.5 Access Rights

The most important element of an application is the system for the management of access rights to its resources. These access rights can be established on a number of levels:

1. Permission to call a specific contract by the current user. This permission can be configured in the `conditions` section of the contract by using a logical expression in the `If` statement, or with nested contracts; for example, *MainConditions* or *RoleConditions*, where typical rights or user role rights are defined.

2. Current user's permission to change (using the contracts) values in table columns, to add rows and columns to tables. Permissions can be set using the `ContractConditions` function in Permissions fields of table columns and in the *Write permissions / Insert / Update / New column* field on the table editing page. The conditions specified in the Update field specify the rights to change all columns of the table in general, the conditions in the fields Permissions impose additional restrictions for each column separately.

3. Permission only for specific contracts to change values in table columns or to add rows to tables. Contract names should be specified in the parameters of the *ContractAccess* function, which should be written in *Permissions* fields of table columns, and in the *Permissions / Insert* field on the table editing page.

4. Permission to edit application elements (contracts, pages, menu, and page blocks). Permissions can be set in the *Change conditions* fields in element editors. This is done using the *ContractConditions* function, to which the name of the contract that checks the permissions of the current user should be passed as a parameter.

2.11.6 Application Example: SendTokens

The application sends tokens from one user account to another. Information about the amounts of tokens on accounts is stored in the *keys* tables (*amount* column), which are installed in ecosystems by default. This example implies that the tokens have already been distributed to user accounts.

System Contract

The main contract for this application is the *TokenTransfer* contract, which has the exclusive permission to change values in the *amount* column of the *keys* table. In order to activate this permission, we should write the `ContractAccess("TokenTransfer")` function in the *Permissions* field of the *amount* column. From this moment, any operations with tokens can only be carried out by calling the TokenTransfer contract.

In order to prevent the execution of the TokenTransfer contract from within another contract without the knowledge of the account holder, TokenTransfer should be a contract with confirmation. This means that its `data` section should contain the `Signature string "optional hidden"` string, and the confirmation parameters should be set on the *Contracts with Confirmation* page in the Molis administrative tools, which includes: text and parameters that should be displayed to the user in a pop-up information window (for details, see the *Contracts with Confirmation* section).

```
contract TokenTransfer {
data {
    Amount money
    Sender_AccountId int
    Recipient_AccountId int
    Signature string "optional hidden"
}
conditions {
    //check the sender
    $sender = DBFind("keys").Where("id=$", $Sender_AccountId)
    if(Len($sender) == 0){
        error Sprintf("Sender %s is invalid", $Sender_AccountId)
    }
    $vals_sender = $sender[0]

    //check the recipient
    $recipient = DBFind("keys").Where("id=$", $Recipient_AccountId)
    if(Len($recipient) == 0){
        error Sprintf("Recipient %s is invalid", $Recipient_AccountId)
    }
    $vals_recipient = $recipient[0]

    //check amount
    if $Amount == 0 {
        error "Amount is zero"
    }

    //check balance
    var sender_balance money
    sender_balance = Money($vals_sender["amount"])
    if $Amount > sender_balance {
        error Sprintf("Money is not enough %v < %v", sender_balance, $Amount)
    }
}
action {
    DBUpdate("keys", $Sender_AccountId, {"-amount": $Amount})
    DBUpdate("keys", $Recipient_AccountId, {"+amount": $Amount})
}
}
```

The following checks are carried out in the conditions section of the TokenTransfer contract: the accounts involved in the transaction should exist, the amount of tokens to be transferred should be non-zero, the amount of the transaction should be smaller or equal to the balance of the sender's account. The action section carries out the modification of values in the amount column of the sender's and receiver's accounts.

Token Sending Form

The token sending form contains fields to input the transaction amount and the recipient address.

```
Div(Class: panel panel-default){
  Form(){
    Div(Class: list-group-item text-center){
      Span(Class: h3, Body: LangRes(SendTokens))
    }
    Div(Class: list-group-item){
      Div(Class: row df f-valign){
```

(continues on next page)

(continued from previous page)

```
        Div(Class: col-md-3 mt-sm text-right){
          Label(For: Recipient_Account){
            Span(Body: LangRes(Recipient_Account))
          }
        }
        Div(Class: col-md-9 mb-sm text-left){
          Input(Name: Recipient_Account, Type: text, Placeholder: "xxxx-xxxx-xxxx-xxxx
↪")
        }
      }
      Div(Class: row df f-valign){
        Div(Class: col-md-3 mt-sm text-right){
          Label(For: Amount){
            Span(Body: LangRes(Amount))
          }
        }
        Div(Class: col-md-9 mc-sm text-left){
          Input(Name: Amount, Type: text, Placeholder: "0", Value: "5000000")
        }
      }
    }
    Div(Class: panel-footer clearfix){
      Div(Class: pull-right){
        Button(Body: LangRes(send), Contract: SendTokens, Class: btn btn-default)
      }
    }
  }
}
```

We could use the Button function to directly call the TokenTransfer transfer contract and pass the current user's (sender) account address to it, but for the purpose of demonstration of the work of contracts with confirmation we'll create an intermediary user contract SendTokens. It is important to note, that since the names of data in the data section of the contract and the names of the interface form fields are the same, we don't need to specify the Params parameters in the Button function.

The form can be placed on any page in the software client. After the contract execution has ended, the user will stay on the current page (because we didn't specify a target page Page in the Button function).

Custom Contracts

The TokenTransfer contract is defined as a contract with confirmation, and that is why in order to call it from another contract we need to place the Signature string "signature:TokenTransfer" in the data section of our custom contract. The conditions section of the SendTokens contract checks the availability of the account; the action section calls the TokenTransfer contract and passes parameters to it.

```
contract SendTokens {
    data {
        Amount money
        Recipient_Account string
        Signature string "signature:TokenTransfer"
    }

    conditions {
        $recipient = AddressToId($Recipient_Account)
        if $recipient == 0 {
```

(continues on next page)

(continued from previous page)

```
            error Sprintf("Recipient %s is invalid", $Recipient_Account)
        }
    }

    action {
        TokenTransfer("Amount,Sender_AccountId,Recipient_AccountId,Signature",
↪$Amount, $key_id, $recipient, $Signature)
    }
}
```

2.12 Compiler and virtual machine

This section reviews the code organization in the packages/script directory, which refers to program compilation and the operation of the Simvolio language virtual machine.

2.12.1 Source code storage and compilation

Contracts and functions are written in Simvolio language and stored in the contracts tables in ecosystems.

When a contract is executed, its source code is read from the database and compiled into bytecode.

When a contract is changed, its source code is updated and saved in the database. This source code is then compiled, and as a result the corresponding virtual machine bytecode is also changed.

The bytecode is not physically saved anywhere, so when a program is executed again, the source code is compiled anew.

The entire source code described in the contracts table for all ecosystems is compiled in a strict sequence into one virtual machine and the state of the virtual machine is the same on all nodes.

When you call a contract, the virtual machine does not change its state in any way. Any execution of a contract or a function call occurs on a separate runtime stack that is created with each external call.

Each ecosystem can have a so-called virtual ecosystem that works with its tables outside the blockchain, within one node, and cannot directly affect the blockchain or other virtual ecosystems. In this case, a node hosting such a virtual ecosystem compiles its contracts and creates its own virtual machine.

2.12.2 Virtual machine structures

VM structure

The virtual machine is organized in memory as follows.

```
type VM struct {
   Block
   ExtCost func(string) int64
   FuncCallsDB map[string]struct{}
   Extern bool
}
```

The VM structure has these elements:

- **Block** is a structure that contains a block (see below).

- **ExtCost** is a function that returns the cost of executing external golang functions.
- **FuncCallsDB** is a map of golang function names that return the cost of execution as the first parameter. These are the functions for working with databases that calculate the cost using EXPLAIN.
- **Extern** is a flag which states that the contract is external. When you create a VM, it is set to true and does not require the called contract to be present when compiling the code. That is, it allows calling the contract code that be determined in the future.

Blocks structure

A virtual machine is a tree of **Block** type objects.

A block is an independent unit containing some bytecode. In simple words, everything you put in curly brackets ({}) in the language is a block.

For example, the following code creates a block with a function. This block in turn contains a block with an *if* statement, which, in turn, has a block with a *while* statement in it.

```
func my() {
     if true {
          while false {
               ...
           }
     }
}
```

Blocks are organized in memory as follows.

```
type Block struct {
    Objects map[string]*ObjInfo
    Type int
    Owner *OwnerInfo
    Info interface{}
    Parent *Block
    Vars []reflect.Type
    Code ByteCodes
    Children Blocks
}
```

The Block structure has these elements:

- **Objects** is a map of internal objects of the **ObjInfo** pointer type. If, for example, there is a variable in the block, then it is possible to get information about it by its name.
- **Type** is the type of the block. For functions, **ObjFunc**, for contracts **ObjContract**.
- **Owner** is a pointer to the **OwnerInfo** structure. This structure contains information about the owner of the compiled contract. It is specified during compilation of contracts or is taken from the **contracts** table.
- **Info** contains information about the object and depends on the block type.
- **Parent** is a pointer to the parent block.
- **Vars** is an array with the current block variable types.
- **Code** is the bytecode itself, which will be executed when the control is transferred to this block. For example, in the case when a function is called or a loop body is executed.
- **Children** is an array of child blocks. For example it may contain nested functions, loops, and conditional operators.

ObjInfo structure

ObjInfo structure contains information about an internal object.

```
type ObjInfo struct {
   Type int
   Value interface{}
}
```

The ObjInfo structure has these elements:

- **Type** is the object type. It can have one of the following values:
  - **ObjContract** – contract
  - **ObjFunc** – function
  - **ObjExtFunc** – external golang function
  - **ObjVar** – variable
  - **ObjExtend** – the $name variable
- **Value** – contains the structure for each type.

ContractInfo structure

For the **ObjContract** type, the **Value** field contains the **ContractInfo** structure.

```
type ContractInfo struct {
    ID uint32
    Name string
    Owner *OwnerInfo
    Used map[string]bool
    Tx *[]*FieldInfo
    Settings map[string]interface{}
}
```

The ContractInfo structure has these elements:

- **ID** – contract identifier. This value is indicated in the blockchain when calling the contract.
- **Name** – contract name.
- **Owner** – additional information about the contract.
- **Used** – map of the contract names that are called inside this contract.
- **Tx** – data array described in the `data` section of the contract.
- **Settings** – map of the values that are described in the `settings` section of the contract.

FieldInfo structure

FieldInfo structure is used in **ContractInfo** structure and describes elements of `data` section in contracts.

```
type FieldInfo struct {
      Name string
      Type reflect.Type
```

(continues on next page)

(continued from previous page)

```
      Tags string
}
```

The FieldInfo structure has these elements:

- **Name** is the name of the field.
- **Type** is the type of the field
- **Tags** – additional tags for the field.

FuncInfo structure

For the **ObjFunc** type, the **Value** field contains the **FuncInfo** structure.

```
type FuncInfo struct {
     Params []reflect.Type
     Results []reflect.Type
    Names *map[string]FuncName
    Variadic bool
    ID uint32
}
```

The FuncInfo structure has these elements:

- **Params** – an array of parameter types.
- **Results** – an array of returned types.
- **Names** – map of data for tail functions. For example, `DBFind().Columns ()`.
- **Variadic** – true if the function can have a variable number of parameters.
- **ID** – function identifier.

FuncName structure

FuncName structure is used in **FuncInfo** structure and describes the data for a tail function.

```
type FuncName struct {
   Params []reflect.Type
   Offset []int
   Variadic bool
}
```

The FuncName structure has these elements:

- **Params** – an array of parameter types.
- **Offset** – an array of offsets for these variables. In fact, all parameters that are expressed in functions using the dot are variables that can be assigned initialization values.
- **Variadic** – true, if the tail function can description can have a variable number of parameters.

ExtFuncInfo structure

For the **ObjExtFunc** type, the **Value** field contains the **ExtFuncInfo** structure. It describes the golang functions.

```
type ExtFuncInfo struct {
   Name string
   Params []reflect.Type
   Results []reflect.Type
   Auto []string
   Variadic bool
   Func interface{}
}
```

The ExtFuncInfo structure has these elements:

- The **Name**, **Params**, **Results** parameters are the same as for the **FuncInfo** structure.
- **Auto** – an array of variables that are additionally passed to the golang functions, if any. For example, the *sc* variables of *SmartContract* type.
- **Func** – golang function.

VarInfo structure

For the **ObjVar** type, the **Value** field contains a **VarInfo** structure.

```
type VarInfo struct {
   Obj *ObjInfo
   Owner *Block
}
```

The ExtFuncInfo structure has these elements:

- **ObjInfo** – information about the type and value of the variable.
- **Owner** – pointer to the owner block.

Value for ObjExtend

For **ObjExtend** type, the **Value** field contains a string with the name of a variable or a function.

2.12.3 Virtual machine commands

ByteCode structure

The bytecode is a sequence of **ByteCode** type structures.

```
type ByteCode struct {
   Cmd uint16
   Value interface{}
}
```

This structure has the following fields:

- **Cmd** field stores the command identifier

- **Value** field contains the operand (value).

As a rule, commands perform operations to the top elements of the stack, and write the resulting value there if necessary.

Command identifiers

The identifiers of the virtual machine commands are described in the *packages/script/mds_list.go*.

- **cmdPush** – put a value from the *Value* field to the stack. For example, it is used to put numbers and lines to the stack.
- **cmdVar** – put the value of a variable to the stack. *Value* contains an indicator of the *VarInfo* structure with the information about the variable.
- **cmdExtend** – put the value of an external variable to the stack. *Value* contains a string with the variable name (begins with `$`).
- **cmdCallExtend** – call an external function (their names begin with **$**). The parameters of the function will be taken from the stack, and the result(s) of the function will be put to the stack. Value contains the name of the function (begins with `$`).
- **cmdPushStr** – put the string from *Value* to the stack.
- **cmdCall** – call the virtual machine function. *Value* contains the **ObjInfo** structure. This command is applicable both for *ObjExtFunc* golang and for *ObjFunc* Simvolio functions. When a function is called, its parameters are taken from the stack, and the resulting values are put to the stack.
- **cmdCallVari** – similar to the **cmdCall** command, calls the virtual machine function. This command is used to call functions with a variable number of parameters.
- **cmdReturn** – is used to exit the function. The returned values are placed to the stack. *Value* is not used.
- **cmdIf** – transfers control to the bytecode in the **Block** structure, an indicator to which is passed in the *Value* field. Control is only transferred if calling the *valueToBool* function with the edge stack element returns *true*. Otherwise, control is transferred to the next command.
- **cmdElse** – the command works in the same way as the **cmdIf** command, but the control is transferred to the specified block only if *valueToBool* with the edge stack element returns *false*.
- **cmdAssignVar** – gets a list of variables of type **VarInfo** from *Value*. These variables will get a value with the **cmdAssign** command.
- **cmdAssign** – assign values from the stack to the variables obtained by the **cmdAssignVar** command.
- **cmdLabel** – defines a label where the control is returned during the while loop.
- **cmdContinue** – the command passes control to the **cmdLabel** label. Performs a new iteration of the loop. *Value* is not used.
- **cmdWhile** – checks the top element of the stack with *valueToBool*. If the value is true, calls the **Block** from the *Value* field.
- **cmdBreak** – exits the loop.
- **cmdIndex** – puts a value from a *map* or *array* to the stack by an index. *Value* is not used. `(map | array) (index value) => (map | array [index value])`.
- **cmdSetIndex** – assigns the top value from the stack to the element of a map or an array. *Value* is not used. `(map | array) (index value) (value) => (map | array)`,
- **cmdFuncName** – adds parameters that are passed using sequential descriptions divided by dots. `func name Func (...) .Name (...)`.

- **cmdError** – a command is created that terminates a contract or function with an error that was specified in *error, warning* or *info*.

Commands for working on the stack

Below are the commands to work directly with the stack. The *Value* field is not used in these commands.

Note: in the current version, there is no fully automatic type conversion. For example, `string + float | int | decimal => float | int | decimal, float + int | str => float`, but `int + string => runtime error`.

- **cmdNot** – logic negation `(val) => (! ValueToBool (val))`
- **cmdSign** – change of sign. `(val) => (-val)`
- **cmdAdd** – addition. `(val1) (val2) => (val1 + val2)`
- **cmdSub** – subtraction. `(val1) (val2) => (val1-val2)`
- **cmdMul** – multiplication. `(val1) (val2) => (val1 * val2),`
- **cmdDiv** – division. `(val1) (val2) => (val1 / val2),`
- **cmdAnd** – logical AND `(val1) (val2) => (valueToBool (val1) && valueToBool (val2)),`
- **cmdOr** – logical OR. `(val1) (val2) => (valueToBool (val1) || valueToBool (val2)),`
- **cmdEqual** – equality comparison, bool is returned. `(val1) (val2) => (val1 == val2),`
- **cmdNotEq** – comparison for inequality, bool is returned. `(val1) (val2) => (val1! = val2),`
- **cmdLess** – comparison for being less, bool is returned. `(val1) (val2) => (val1 <val2)`
- **cmdNotLess** – the comparison for being greater or equal, bool is returned. `(val1) (val2) => (val1> = val2),`
- **cmdGreat** – comparison for being greater, bool is returned. `(val1) (val2) => (val1> val2),`
- **cmdNotGreat** – comparison for being less or equal, bool is returned. `(val1) (val2) => (val1 <= val2).`

Runtime structure

The execution of bytecode does not affect the virtual machine. This, for example, allows you to simultaneously run various functions and contracts within a single virtual machine. The **Runtime** structure is used to run functions and contracts, as well as any expressions and bytecode.

```
type RunTime struct {
   stack []interface{}
   blocks []*blockStack
   vars []interface{}
   extend *map[string]interface{}
   vm *VM
   cost int64
   err error
}
```

- **stack** – the stack on which the bytecode is executed.
- **blocks** – block calls stack.

- **vars** – stack of variables. When calling a bytecode in a block, its variables are added to this stack of variables. After exiting the block, the size of the variables stack returns to the previous value.
- **extend** – a pointer to a map with values of external variables ($name).
- **vm** – a virtual machine pointer.
- **cost** – the resulting cost of execution.
- **err** – the execution error, if occurred.

blockStack structure

This structure is used in the Runtime structure.

```
  type blockStack struct {
       Block *Block
       Offset int
  }


* **Block** - indicator of the block being executed.

* **Offset** - the offset of the last command executed in the bytecode of the␣
↪specified block.
```

RunCode function

Bytecode is executed in the **RunCode** function. It contains a loop that performs the appropriate actions for each bytecode command. Before processing the bytecode, necessary data must be initialized. Here, the new block is added to other blocks.

```
rt.blocks = append(rt.blocks, &blockStack{block, len(rt.vars)})
```

Next, the information about the parameters of the "tail" functions is obtained. These parameters are contained in the last element of the stack.

```
var namemap map[string][]interface{}
if block.Type == ObjFunc && block.Info.(*FuncInfo).Names != nil {
    if rt.stack[len(rt.stack)-1] != nil {
        namemap = rt.stack[len(rt.stack)-1].(map[string][]interface{})
    }
    rt.stack = rt.stack[:len(rt.stack)-1]
}
```

Next, all variables that are defined in the current block must be initialized with teir initial values.

```
start := len(rt.stack)
varoff := len(rt.vars)
for vkey, vpar := range block.Vars {
   rt.cost--
   var value interface{}
```

Since function variables are also variables, we need to take them from the last elements of the stack in the same order as they are described in the function itself.

```
if block.Type == ObjFunc && vkey < len(block.Info.(*FuncInfo).Params) {
  value = rt.stack[start-len(block.Info.(*FuncInfo).Params)+vkey]
} else {
```

Here we initialize local variables with initial values.

```
     value = reflect.New(vpar).Elem().Interface()
     if vpar == reflect.TypeOf(map[string]interface{}{}) {
        value = make(map[string]interface{})
     } else if vpar == reflect.TypeOf([]interface{}{}) {
        value = make([]interface{}, 0, len(rt.vars)+1)
     }
  }
  rt.vars = append(rt.vars, value)
}
```

Next, we need to update the values of the variable parameters that were transferred in the tail functions.

```
if namemap != nil {
  for key, item := range namemap {
    params := (*block.Info.(*FuncInfo).Names)[key]
    for i, value := range item {
       if params.Variadic && i >= len(params.Params)-1 {
```

If it is possible to transfer a variable number of parameters, then we combine them into one variable array.

```
              off := varoff + params.Offset[len(params.Params)-1]
              rt.vars[off] = append(rt.vars[off].([]interface{}), value)
          } else {
              rt.vars[varoff+params.Offset[i]] = value
       }
     }
   }
}
```

After that, all we are left to do is to move the stack by removing the values that were transferred as parameters of the function from the stack top. We have already copied their values into an array of variables.

```
if block.Type == ObjFunc {
     start -= len(block.Info.(*FuncInfo).Params)
}
```

After the bytecode commands execution loop is over, we must correctly clear the stack.

```
last := rt.blocks[len(rt.blocks)-1]
```

Remove the current block from the stack of blocks.

```
rt.blocks = rt.blocks[:len(rt.blocks)-1]
if status == statusReturn {
```

In case of a successful exit from the executed function, we add the return values to the previous end of the stack.

```
if last.Block.Type == ObjFunc {
   for count := len(last.Block.Info.(*FuncInfo).Results); count > 0; count-- {
     rt.stack[start] = rt.stack[len(rt.stack)-count]
     start++
```

(continues on next page)

(continued from previous page)

```
    }
    status = statusNormal
  } else {
```

As you can see, if we aren't executing a function, then we do not restore the stack state and exit the function as is. The reason for this is that loops and conditional constructions already executed inside a function are also bytecode blocks.

```
      return
    }
  }
  rt.stack = rt.stack[:start]
```

Other functions for working with VM

A virtual machine is created using the **NewVM** function. Three functions, **ExecContract**, **CallContract** and **Settings** are added to each virtual machine. The adding occurs using the **Extend** function.

```
for key, item := range ext.Objects {
    fobj := reflect.ValueOf(item).Type()
```

We go through all the transferred objects and look only for functions.

```
switch fobj.Kind() {
case reflect.Func:
```

According to the information received about the function, we fill the **ExtFuncInfo** structure and add it to the top-level map **Objects** by its name.

```
data := ExtFuncInfo{key, make([]reflect.Type, fobj.NumIn()), make([]reflect.Type,␣
↪fobj.NumOut()),
   make([]string, fobj.NumIn()), fobj.IsVariadic(), item}
for i := 0; i < fobj.NumIn(); i++ {
```

We have the so-called **Auto** parameters. Typically, this is the first parameter, for example sc *SmartContract* or rt *Runtime*. We cannot transfer them from the Simvolio language, but they are necessary for us when performing some golang functions. Therefore, we specify which variables will be automatically used at the time the function is called. In this case, the **ExecContract**, **CallContract** functions have such rt *Runtime* parameter.

```
if isauto, ok := ext.AutoPars[fobj.In(i).String()]; ok {
  data.Auto[i] = isauto
}
```

We fill in the information about the parameters:

```
  data.Params[i] = fobj.In(i)
}
```

and about the types of returned values

```
for i := 0; i < fobj.NumOut(); i++ {
   data.Results[i] = fobj.Out(i)
}
```

Adding a function to the root Objects will allow the compiler to find them later when used from contracts.

```
            vm.Objects[key] = &ObjInfo{ObjExtFunc, data}
        }
    }
```

2.12.4 Compilation

The functions located in the *compile.go* file are responsible for the compilation of the array of tokens obtained from the lexical analyzer. The compilation can be conditionally divided into two levels. At the top level, we process functions, contracts, blocks of code, conditional statements and loop statements, variable definitions, and so on. At the lower level, we compile expressions that are inside of code blocks or conditions in a loop and a conditional statement. In the beginning, let us consider a simpler lower level.

Translating expressions into a bytecode is done in the **compileEval** function. Since we have a virtual machine working with a stack, it is necessary to translate the usual infix record of expressions into a postfix notation or a reverse Polish notation. For example, 1 +2 should be converted to 12+, then you put 1 and 2 to the stack, and then we apply the addition operation for the last two elements in the stack and write the result to the stack. The translation algorithm itself can be found on the Internet – for example, https://master.virmandy.net/perevod-iz-infiksnoy-notatsii-v-postfiksnuyu-obratnaya-polskaya-zapis/. The global variable *opers = map [uint32] operPrior* contains the priorities of the operations that are necessary when translating into the reverse Polish notation. The following variables are defined at the beginning of the function:

- **buffer** – temporary buffer for bytecode commands,
- **bytecode** – final buffer of bytecode commands,
- **parcount** – temporary buffer for calculating parameters when calling functions,
- **setIndex** – the variable in the process of work is set to *true*, when we are assigning to the *map* or *array* element. For example, *a["my"] = 10*. In this case, we will need to use the special **cmdSetIndex command**.

Then there is a loop in which we get the next token and process it accordingly. For example, if braces are found:

```
case isRCurly, isLCurly:
     i--
    break main
case lexNewLine:
      if i > 0 && ((*lexems)[i-1].Type == isComma || (*lexems)[i-1].Type == lexOper) {
            continue main
      }
     for k := len(buffer) - 1; k >= 0; k-- {
           if buffer[k].Cmd == cmdSys {
               continue main
           }
     }
    break main
```

then we stop parsing the expression, and when moving the string, we look at whether the previous statement is an operation and whether we are inside the parentheses, otherwise we exit and the expression is parsed. In general, the algorithm itself corresponds to an algorithm for translating into a reverse Polish notation, taking into account that it is necessary to take the calls of functions, contracts, index calls, and other things that you will not meet in case of parsing, for example, for a calculator, into account. Consider the option of parsing the *lexIdent* type token. We are looking for a variable, function or contract with this name. If nothing is found and this is not a function or a contract call, then we indicate an error.

```
objInfo, tobj := vm.findObj(lexem.Value.(string), block)
if objInfo == nil && (!vm.Extern || i > *ind || i >= len(*lexems)-2 || (*lexems)[i+1].
↪Type != isLPar) {
```

(continues on next page)

(continued from previous page)

```
      return fmt.Errorf(`unknown identifier %s`, lexem.Value.(string))
}
```

We may have a situation where a contract called will be described later. In this case, if a function and a variable with the same name are not found, then we believe that we will have a contract call. In a language, the contracts and functions calls do not differ. But we need to call the contract through the **ExecContract** function, the one we use in the bytecode.

```
if objInfo.Type == ObjContract {
    objInfo, tobj = vm.findObj(`ExecContract`, block)
    isContract = true
}
```

In *count*, we will write down the number of variables so far and this value will also go to the stack with the number of function parameters. We simply increase this quantity by one unit in the last element of the stack at each subsequent detection of the parameter.

```
count := 0
if (*lexems)[i+2].Type != isRPar {
    count++
}
```

Since we have the *Used* list of called parameters for contracts, then we need to make the marks for the case of contract being called, and in case the contract is called without the *MyContract()* parameters, we have to add two empty parameters to the call **ExecContract**, which should get the minimum two parameters.

```
if isContract {
   name := StateName((*block)[0].Info.(uint32), lexem.Value.(string))
   for j := len(*block) - 1; j >= 0; j-- {
      topblock := (*block)[j]
      if topblock.Type == ObjContract {
            if topblock.Info.(*ContractInfo).Used == nil {
                 topblock.Info.(*ContractInfo).Used = make(map[string]bool)
            }
           topblock.Info.(*ContractInfo).Used[name] = true
       }
    }
    bytecode = append(bytecode, &ByteCode{cmdPush, name})
    if count == 0 {
       count = 2
       bytecode = append(bytecode, &ByteCode{cmdPush, ""})
       bytecode = append(bytecode, &ByteCode{cmdPush, ""})
     }
    count++

}
```

If we see that there is a square bracket next, then we add the **cmdIndex** command to get the value by the index.

```
if (*lexems)[i+1].Type == isLBrack {
     if objInfo == nil || objInfo.Type != ObjVar {
         return fmt.Errorf(`unknown variable %s`, lexem.Value.(string))
     }
    buffer = append(buffer, &ByteCode{cmdIndex, 0})
}
```

The **compileEval** function generates the bytecode of the expressions in blocks directly, but the **CompileBlock** function

forms both the object tree and the bytecode not related to the expressions. Compilation is also based on the work of the finite state machine, just as it was done for lexical analysis, but with the following differences. First, we do not operate with symbols but with tokens, and second, we describe all states and transitions in *states* variable immediately. It represents an array of maps with indices by type of tokens and each token has the structure of the *compileState* with a new state specified in the *NewState*, and in case it is clear what structure we have parsed, then the function of the handler in the *Func* field is specified.

Let us review the main state as an example.

```
{ // stateRoot
   lexNewLine: {stateRoot, 0},
   lexKeyword | (keyContract << 8): {stateContract | statePush, 0},
   lexKeyword | (keyFunc << 8): {stateFunc | statePush, 0},
   lexComment: {stateRoot, 0},
   0: {errUnknownCmd, cfError},
},
```

If we encounter line break or comments, then we stay in the same state. If we encounter the **contract** keyword, then we change the state to the *stateContract* and begin parsing this construction. If we encounter the **func** keyword, then we change to the *stateFunc state*. If other tokens are received, the error generation function will be called. Suppose that we have encountered the *func* keyword and we have changed the state to *stateFunc*.

```
{ // stateFunc
    lexNewLine: {stateFunc, 0},
    lexIdent: {stateFParams, cfNameBlock},
    0: {errMustName, cfError},
},
```

Since the name of the function must follow the **func** keyword, then when changing the string, we remain in the same state, and with all the other tokens we generate the corresponding error. If we get the function name in the token-identifier, then we go to *stateFParams* state in which we get the parameters of the function. In doing so, we call the **fNameBlock** function. It should be noted that the *Block* structure was created using the *statePush* flag, and here we take it from the buffer and fill with the data we need.

```
func fNameBlock(buf *[]*Block, state int, lexem *Lexem) error {
    var itype int

    prev := (*buf)[len(*buf)-2]
    fblock := (*buf)[len(*buf)-1]
   name := lexem.Value.(string)
   switch state {
     case stateBlock:
        itype = ObjContract
       name = StateName((*buf)[0].Info.(uint32), name)
       fblock.Info = &ContractInfo{ID: uint32(len(prev.Children) - 1), Name: name,
           Owner: (*buf)[0].Owner}
    default:
       itype = ObjFunc
       fblock.Info = &FuncInfo{}
     }
     fblock.Type = itype
    prev.Objects[name] = &ObjInfo{Type: itype, Value: fblock}
    return nil
}
```

The **fNameBlock** function is used for contracts and functions (including those nested in other functions and contracts). It fills the *Info* field with the appropriate structure and writes itself into the map *Objects* of the parent block. This is done so that we can then call this function or contract by the given name. Similarly, we create functions for all states

and variants. These functions are usually very small and perform some work on the formation of the virtual machine tree. As for the **CompileBlock** function, it simply goes through all the tokens and switches states according to those described in the *states*. Almost the whole additional processing code for additional flags.

- **statePush** – the *Block* object is added to the object tree,
- **statePop** – used when the block ends with closing curly braces,
- **stateStay** – indicates that when you change to a new state, you need to stay on the current token,
- **stateToBlock** – indicates the transition to *stateBlock* state. Used to handle while and if, when it is needed to go into * * **the processing** of the block inside curly brackets after the expression is processed,
- **stateToBody** – indicates the transition to *stateBody*,
- **stateFork** – saves the position of the token. Used when an expression starts in an identifier or a name with **$**. We can have either a function call or an assignment,
- **stateToFork** – used to get the token stored in the *stateFork* flag. This token will be passed to the processing function,
- **stateLabel** – serves for inserting the **cmdLabel** command. This flag is needed for the while construction,
- **stateMustEval** – checks for a conditional expression availability at the beginning of the if and while constructions.

Besides the **CompileBlock** function, you should also mention the **FlushBlock** function. The matter is that the tree of blocks is built independent of the existing virtual machine. More precisely, we take information about the functions and contracts existing in a virtual machine, but we gather the compiled blocks into a separate tree. Otherwise, if an error occurs during compilation, we will have to roll back the state of the virtual machine to the previous state. Therefore, we compile the tree separately, but have to call the **FlushContract** function after the compilation is successful. This function adds our finished block tree to the current virtual machine. At this point, the compilation stage is considered complete.

2.12.5 Lexical analysis

The lexical analyzer processes the incoming string and forms a sequence of tokens of the following types:

- **sys** - is the system token, for example: {}[](),.
- **oper** – operator +-/*
- **number** – number
- **ident** – identifier
- **newline** – line break
- **string** – string
- **comment** – comment

In this version, preliminarily with the help of *script/lextable/lextable.go*, a transition table (finite state machine) is constructed to parse the tokens, which is written to the *lex_table.go* file. Generally, you can get rid of the preliminary generation of this file and create a transfer table at startup immediately in memory (in init ()). The lexical analysis itself occurs in the *lexParser* function in *lex.go*.

lextable/lextable.go

Here we define the alphabet with which our language will operate and describe the finite state machine that changes from one state to another depending on the next received symbol.

states contains a JSON object containing a list of states.

In addition to the specific symbols, d is used to indicate all symbols that are not indicated in the state

n is 0x0a, s is a space, q is the backquotes, Q is double quotes, r is characters>= 128, a is A-Z and a-z, 1 is 1-9.

The names of the states are the keys, and the possible values are listed in the value object, and then there is a new state to make the transition into for each set, then the name of the token, if we need to return to the initial state and the third parameter is the service flags, which indicate what to do with the current symbol.

For example, we have the main state and the incoming character /. `"/": ["Solidus", "", "push next"],`

push gives the command to remember it in a separate stack, and **next** – go to the next character, while we change the state to **solidus**. After that, take the next character and look at the **solidus** state.

If we have / or * – then we go into the comment state, so they start with // or /*. It is clear that for each comment there are different subsequent states, since they end in different symbols.

And if we have the following character not/and not *, then we record everything put in our stack (/) as a token with oper type, clear the stack and return to the main state.

This module changes this state tree into a numeric array and writes it to the *lex_table.go file*.

In the first loop:

```
for ind, ch := range alphabet {
i := byte(ind)
```

we form the alphabet of allowed symbols. Further in *state2int*, we give each state its own sequence identifier.

```
state2int := map[string]uint{`main`: 0}
if err := json.Unmarshal([]byte(states), &data); err == nil {
for key := range data {
if key != `main` {
state2int[key] = uint(len(state2int))
```

When we go through all the states and for each set in the state and for each symbol in this set, we write a three-byte number [id of the new state (0 = main)] + [token type (0-no token)] + [flags]. The two-dimensionality of the *table* array consists in its division into states and 33 incoming symbols from the *alphabet* array located in the same order. That is, in the future we will work with this table in approximately the following way.

We are in the *main* state on the zero line of the *table*. We take the first character, look up its index in the *alphabet* array and take the value from the column with the given index. Further from the received value we receive flags in the lower byte, the second byte – indicates the type of the received token, if its parsing is finished, and in the third byte we receive the index of a new state where we should go. All this will be discussed in more detail in the **lexParser** function in the *lex.go file*.

If you want to add some new characters, you need to add them to the *alphabet* array and increase the *AlphaSize* constant. If you want to add a new combination of symbols, they should be described in the states, similar to the existing options. After this, run lextable.and update the *lex_table.go* file.

lex.go

The **lexParser** function produces lexical analysis directly and on the basis of an incoming string returns an array of received tokens. Let us consider the structure of a token.

```
type Lexem struct {
   Type uint32 // Type of the lexem
   Value interface{} // Value of lexem
   Line uint32 // Line of the lexem
   Column uint32 // Position inside the line
}
```

- **Type** – token type. It can be one of the following values: *lexSys, lexOper, lexNumber, lexIdent, lexString, lexComment, lexKeyword, lexType, lexExtend*,
- **Value** – the value of the token. The type of the value depends on the type. Let us consider it in more detail,
- **lexSys** – this includes brackets, commas, etc. In this case, *Type = ch << 8 | lexSys* – see the *isLPar . . . isRBrack constants, and the Value itself is uint32(ch)*,
- **lexOper** – values represent an equivalent sequence of characters in the form of uint32. For example, see the isNot . . . isOr constants,
- **lexNumber** – numbers are stored as *int64* or *float64*. If the number has a decimal point, then it is *float64*,
- **lexIdent** – identifiers are stored as strings,
- **lexNewLine** – the line break character. Also serves to count the line and token position,
- **lexString** – lines are stored as *string*,
- **lexComment** – comments are also stored as *string*,
- **lexKeyword** – the keywords store the corresponding index only – constants from *keyContract . . . keyTail*. In this case, *Type = KeyID << 8 | lexKeyword*. Also, it should be noted that the *true,false,nil* keywords are immediately converted to tokens of *lexNumber* type, with the appropriate bool and *intreface{}* types,
- **lexType** – in this case, the value contains the corresponding *reflect.Type* type value,
- **lexExtend** – identifiers starting with the dollar sign **$**. These variables and functions are passed from the outside and are therefore allocated to a special type of tokens. The value contains the name in the form of a string without the dollar sign in the beginning,
- **Line** – the string where the token is found,
- **Column** – the position of the token in the string.

Let us consider the **lexParser** function in detail. The *todo* function – based on the current state and the transmitted symbol, finds the symbol index in our alphabet and gets a new state, the token identifier, if any, and additional flags from the transition table. The parsing itself involves sequential calling of this function for each next character and switching to a new state. As soon as we see that a token is received, we create the corresponding token in the output maxim and continue parsing. It should be noted that in the process of parsing, we do not accumulate symbols of a token in a separate stack or array as we just save the offset, where our token begins. After the token is obtained, we shift the offset for the next token to the current parsing position.

Remaining is to review the flags that are used in the parsing:

- **push** – this flag means that we begin to accumulate symbols in a new token,
- **next** – the character must be added to the current token,
- **pop** – the receipt of the token is completed. As a rule, with this flag we have an identifier-type of the parsed token,
- **skip** – this flag is used to exclude a character from parsing. For example, the control slashes in the string are *n r "*. They are automatically replaced at the stage of this lexical analysis.

2.12.6 Simvolio language

Lexemes

Source code of a program must be in the UTF-8 encoding.

The following lexeme types are processed:

- **Keywords** - `action`, `break`, `conditions`, `continue`, `contract`, `data`, `else`, `error`, `false`, `func`, `if`, `info`, `nil`, `return`, `settings`, `true`, `var`, `warning`, `while`.
- **Numeric literals** - only decimal numeric literals are accepted. There are two base types: **int** and **float**. If a literal has a decimal point, it becomes *float*. The *int* type is an equivalent of **int64** in golang. The *float* type is equivalent of **float64** in golang.
- **Strings** - strings can be enclosed in the double quotes ("a string") or backquotes 'a string'). Strings of both types can include newline symbols. Strings in double quotes can contain double quote, newline symbols and carriage return symbols escaped with slashes (\). For example: `"This is a \"first string\".\r\nThis is a second string."`.
- **Comments** - there are two types of comments. Single-line comments use two slash symbols (//). For example, `// this is a comment`. Multi-line comments use slash and asterisk symbols and can span several lines. For example: `/* This is a multi-line comment */`.
- **Identifiers** - names of variables and functions that consist of a-z and A-Z letters, UTF-8 symbols, numbers and underscores. The names can begin from a letter, underscore, `@` or `$` symbol. Names that begin with `$` are the names of variables defined in the *data* section. Names that begin with `$` can also be used to define global variables in the common scope of *conditions* and *action*. Contracts from an ecosystem can be called using a `@` symbol. For example: `@1NewTable(...)`.

Types

Corresponding golang types are specified next to Simvolio types.

- **bool** - bool, default value is **false**.
- **bytes** - []byte{}, an empty array of bytes by default.
- **int** - int64, default value is **0**.
- **address** - uint64, default value is **0**.
- **array** - []interface{}, an empty array by default.
- **map** - map[string]interface{}, an empty associative array by default.
- **money** - decimal.Decimal, default value is **0**.
- **float** - float64, default value is **0**.
- **string** - string, an empty string by default.

Variables of these types are defined with the `var` keyword. For example:`var var1, var2 int`. When a variable is defined like this, it gets a default value of its type.

Initially, all variable values have the *interface{}* type, and then they are assigned the required golang type. Thus, for example *array* and *map* types are golang types *[]interface{}* and *map[string]interface{}*. Both types of array can contain elements of any type.

Expressions

Expressions can contain arithmetical operations, logical operations and function calls. All expressions are evaluated from the left to the right according to operation priorities. If operation priorities are equal, evaluation also goes from left to the right.

A list of operations from the top priority to the lowest priority:

- **Function calls and round parentheses**. When functions are called, passed parameters are evaluated from the left to the right.

- **Unary operations** - logical negation ! and arithmetical sign change −.
- **Multiplication and division** - arithmetical multiplication ⋆ and division /.
- **Addition and subtraction** - arithmetical addition + and subtraction −.
- **Logical comparisons** - >= > < <=.
- **Logical equality and inequality** - == !=.
- **Logical AND** - &&.
- **Logical OR** - ||.

When logical AND and OR are evaluated, both sides of expression are evaluated in any case.

Simvolio doesn't have a type check at the time of compilation. When operands are evaluated, an attempt is made to cast a type to a more complex one. Types in the order of complexity can be arranged as follows: *string, int, float, money*. Only a part of type casts is implemented. String types support the addition operation, which results in string concatenation. For example, **string + string = string, money - int = money, int * float = float** For functions, a type check is performed for *string* and *int* types at the moment of execution.

The **array** and **map** types can be addressed by index []. For the *array* type, an *int* value must be specified as index. For the *map* type, a variable or a *string* value must be specified. If a value is assigned to an *array* element with an index that is larger than the current maximum index, then empty elements will be added to the array. These elements will be initialized with *nil* value. For example:

```
var my array
my[5] = 0
var mymap map
mymap["index"] = my[3]
```

In expressions where a logical value is required (such as **if, while, &&, ||, !**) types are automatically cast to a logical value:

- **bytes** - true if size is not 0
- **int** - true if not 0
- **array** - true if not nil and size is not 0
- **map** - true if not nil and size is not 0
- **money** - true if not 0
- **float** - true if not 0
- **string** - true if size is not 0

```
var mymap map
var val string
if mymap && val {
...
}
```

Scope

Curly brackets specify a block that can contain local scope variables. By default the scope of a variable extends to its own block and all nested blocks. Inside a block, a new variable can be defined with a name of an existing variable. In this case an external variable with the same name becomes unavailable.

```
var a int
a = 3
{
   var a int
   a = 4
   Println(a) // 4
}
Println(a) // 3
```

Contract execution

When a contract is called, the parameters defined in the **data** section must be passed to it. Before executing a contract, the virtual machine receives these parameters and assigns them to the corresponding variables ($Param). After that, the predefined **conditions** function is called, followed by the **action** function call. If a contract defines a **rollback** function, then it will be called when a contract is rolled back.

Errors that occur during the contract execution can be divided into two types: generated errors and environment errors. Generated errors are produced with special commands: **error, warning, info** and when built-in functions return *err* not equal to *nil*.

Exceptions aren't handled in the language. Any error stops the execution of a contract. Because a separate stack and structures for keeping variable values are created when a contract is executed, when the contract execution is finished, this data will be automatically deleted by the golang garbage collector.

Backus–Naur form (BNF)

- <decimal digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
- <decimal number> ::= <decimal digit> {<decimal digit>}
- <symbol code> ::= ''' <any symbol> '''
- <real number> ::= ['-'] <decimal number'.'[<decimal number>]
- <integer number> ::= ['-'] <decimal number> | <symbol code>
- <number> := <integer number> | <real number>
- <letter> ::= 'A' | 'B' | . . . | 'Z' | 'a' | 'b' | . . . | 'z' | 0x80 | 0x81 | . . . | 0xFF
- <space> ::= 0x20
- <tabulation> ::= 0x09
- <newline> := 0x0D 0x0A
- <special symbol> ::= '!' | '"' | '$' | '"' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | '<' | '=' | '>' | '[' | '' | ']' | '_' | 'l' | '}' | '{' | <tabulation> | <space> | <newline>
- <symbol> ::= <decimal digit> | <letter> | <special symbol>
- <name> ::= (<letter> | '_') {<letter> | '_' | <decimal digit>}
- <function name> ::= <name>
- <variable name> ::= <name>
- <type name> ::= <name>
- <string symbol> ::= <tabulation> | <space> | '!' | '#' | . . . | '[' | ']' | . . .
- <string element> ::= {<string symbol> | '"' | 'n' | 'r' }

- <string> ::= “” { <string element> } “” | ‘‘ { <string element> } ‘‘
- <assignment operator> ::= ‘=’
- <unary operator> ::= ‘-‘
- <binary operator> ::= ‘==’ | ‘!=’ | ‘>’ | ‘<’ | ‘<=’ | ‘>=’ | ‘&&’ | ‘||’ | ‘*’ | ‘/’ | ‘+’ | ‘-‘
- <operator> ::= <assignment operator> | <unary operator> | <binary operator>
- <parameters> ::= <expression> {‘,’<expression>}
- <contract call> ::= <contract name> ‘(‘ [<parameters>] ‘)’
- <function call> ::= <contract call> [{‘.’ <name> ‘(‘ [<parameters>] ‘)’}]
- <block contents> ::= <block command> {<newline><block command>}
- <block> ::= ‘{‘<block contents>’}’
- <block command> ::= (<block> | <expression> | <variables definition> | <if> | <while> | break | continue | return)
- <if> ::= if <expression><block> [else <block>]
- <while> ::= while <expression><block>
- <contract> ::= contract <name> ‘{‘[<data section>] {<function>} [<conditions>] [<action>]’}’
- <data section> ::= data ‘{‘ {<data parameter><newline>} ‘}’
- <data parameter> ::= <variable name> <type name> “”{<tag>}””
- <tag> ::= optional | image | file | hidden | text | polymap | map | address | signature:<name>
- <conditions> ::= conditions <block>
- <action> ::= action <block>
- <function> ::= func <function name>’(‘[<variable description>{‘,’<variable description>}]’)’[{<tail>}] [<type name>] <block>
- <variable description> ::= <variable name> {‘,’ <variable name>} <type name>
- <tail> ::= ‘.’<function name>’(‘[<variable description>{‘,’<variable description>}]’)’
- <variables definition> ::= var <variable description>{‘,’<variable description>}

2.13 Daemons

This section describes how Apla nodes interact with each other from the technical standpoint.

2.13.1 About the backend daemons

Apla backend is go-apla. It operates at every network node. Backend daemons perform individual functions of the backend and support Apla blockchain protocols. The daemons distribute blocks and transactions in the blockchain network, generate new blocks, validate received blocks and transactions, and resolve blockchain forks if they occur.

Full node daemons

A *full node* (a node that can generate new blocks and send transactions) runs the following backend daemons:

- *BlockGenerator daemon*

  Generates new blocks.

- *BlockCollections daemon*

  Downloads new blocks from other nodes.

- *Confirmations daemon*

  Confirms that blocks present at the node are also present at the majority of other nodes.

- *Disseminator daemon*

  Distributes transactions and blocks to other full nodes.

- QueueParserBlocks daemon

  Processes blocks from the block queue. The block queue contains blocks from other nodes.

  Block processing logic is the same as *BlockCollections daemon*.

- QueueParserTx daemon

  Validates transactions from the transaction queue.

- Notificator daemon

  Sends notifications to users.

- Scheduler daemon

  Runs contracts on schedule.

Regular node daemons

A *regular node* (a node that only sends transactions) runs the following backend daemons:

- *BlockCollections daemon*
- *Confirmations daemon*
- *Disseminator daemon*
- QueueParserTx
- Notificator
- Scheduler

2.13.2 BlockCollections daemon

BlockCollections daemon downloads blocks and synchronizes the blockchain with other network nodes.

Blockchain synchronization

BlockCollections daemon synchronizes blockchain by determining the maximum block number in the blockchain network, requesting new blocks, and resolving forks in the blockchain.

Blockchain update check

BlockCollections daemon sends a request for the current block ID to all full nodes.

If the current block ID of the daemon's node is less than the current block ID of any full node, then the blockchain is considered outdated.

Downloading new blocks

The node that returned the maximum current block number is considered to be the most up-to-date node.

The daemon downloads all blocks that aren't already known from it.

Fork resolution

If a fork is detected in the blockchain, the daemon walks the fork backwards by downloading all blocks up to the common ancestor block.

When the common ancestor block is found, the rollback is performed on the daemon's node blockchain, and correct blocks are added to the blockchain up to the newest block.

Tables

BlockCollections daemon uses the following tables:

- block_chain (writes received blocks)
- config
- full_nodes
- main_lock
- node_public_key
- transactions
- transactions_status
- info_block

Database lock

Yes.

Requests

BlockCollections daemon makes the following requests to other daemons:

- *Type 10* to all full nodes (maximum block number).
- *Type 7* to a node with maximum block number (block data).

2.13.3 BlockGenerator daemon

BlockGenerator daemon generates new blocks.

Scheduling

BlockGenerator daemon schedules new block generation by analyzing the newest block in the blockchain.

New block can be generated if the following conditions are true:

- A node that generated the newest block is located next to the daemon's node in the list of validating nodes.
- Minimum amount of time has passed since the newest block was generated.

Block generation

When a new block is generated, the daemon includes all new transactions in it. These transactions can be received from other nodes (*Disseminator daemon*), or generated by daemon's node. The resulting block is saved in the local database.

Tables

BlockGenerator daemon uses the following tables:

- block_chain (saves new blocks)
- config
- system_recognized_states
- full_nodes
- main_lock
- node_public_key
- transactions
- transactions_status
- info_block
- incorrect_tx

Database lock

Yes.

Requests

BlockGenerator daemon makes no requests to other daemons.

2.13.4 Disseminator daemon

Disseminator daemon sends transactions and blocks to full nodes.

Regular node

When working at a regular node, the daemon sends transactions generated by its node to all full nodes.

Full node

When working at a full node, the daemon sends hashes of generated blocks and transactions to all full nodes.

The full nodes then respond with requests for transactions that are unknown to their nodes. The daemon sends full transaction data in response.

Tables

Disseminator daemon uses the following tables:

- config
- system_recognized_states
- full_nodes
- transactions

Database lock

No.

Requests

Disseminator daemon makes the following requests to other daemons:

- *Type 1* to full nodes (transaction and block hashes).
- *Type 2* from full nodes (transaction data).

2.13.5 Confirmations daemon

Confirmations daemon checks that all blocks from its node are present at the majority of other nodes.

Block confirmation

A block is considered confirmed when a number of nodes in a network have confirmed this block.

The daemon confirms all blocks, one by one, starting from the first block in the database that is not confirmed at the moment.

Each block is confirmed in this way:

- Confirmations daemon sends a request to all full nodes. This request contrains the ID of the block that is being confirmed.
- All full nodes respond with a hash of this block.
- If a hash from a response matches the hash of the block present at daemon's node, then the confirmations counter is increased. If hashes don't match, the disconfirmations counter is increased.

Tables

Confirmations daemon uses the following tables:

- confirmation
- info_block
- full_nodes

Database lock

No.

Requests

Confirmation daemon makes the following requests to other daemons:

- *Type 4* to full nodes (block hash request).

2.13.6 Tcpcerver protocol

A TCP server (tcpserver) works at full nodes. The TCP server uses a binary protocol over TCP to handle requests from BlockCollections, Disseminator, and Confirmation daemons.

Request types

Every request has a type definded by first two bytes of a request.

Type 1

Request sender

Disseminator daemon sends this request.

Request data

Transaction and block hashes.

Request handling

Block hashes are added to blocks queue.

Transaction hashes are analyzed and transactions that aren't already present at the node are selected.

Response

None. *Type 2* requests are made after handling this request.

Type 2

Request sender

Disseminator daemon sends this request.

Request data

Transaction data, including data size.

- *data_size* (4 bytes)

  Size of the transaction data, in bytes.

- *data* (data_size bytes)

  Transaction data.

Request handling

Transaction is validated and added to the transactions queue.

Response

None.

Type 4

Request sender

Confirmations daemon sends this request.

Request data

Block ID.

Response

Block hash.

If a block with this ID is not present, `0` value is returned.

Type 7

Request sender

BlockCollections daemon sends this request.

Request data

Block ID.

- *block_id* (4 bytes)

Response

Block data, including data size.

- *data_size* (4 bytes)

  Size of the block data, in bytes.

- *data* (data_size bytes)

  Block data.

If a block with this ID is not present, connection is closed.

Type 10

Request sender

BlockCollections daemon sends this request.

Request data

None.

Response

Block identifier.

- *block_id* (4 bytes)

2.14 REST API v2

All functions, available from the Molis software client, including authentication, receipt of data about ecosystems, error handling, operations with database tables, interface pages, and execution of contracts (network transactions) are available though REST API of the platform. Thus, by using REST API, developers can access any function of the platform without using the Molis software client.

Command calls are performed by addressing `/api/v2/command/[param]`, where **command** is a command name, and **param** is an additional parameter (for example, the name of the resource to change or receive). Request parameters must be sent with `Content-Type: x-www-form-urlencoded`. The server response will be sent in JSON format.

- *contract/{name}*
- *sendTX*
- *txstatus*
- *txinfo/{hash}*
- *txinfoMultiple/*
- */page/validators_count/{name}*
- *content/{menu|page}/{name}*
- *content/source/{name}*
- *content/hash/{name}*
- *content*
- *node/{name}*
- *maxblockid*
- *block/{id}*
- *avatar/{ecosystem}/{member}*
- *config/centrifugo*
- *updnotificator*

2.14.1 Error handling

Status 200 is returned in case of a successful request execution. In case of an error, apart from an error status, a JSON object is returned with the following fields:

- **error**

  Error identifier.

- **msg**

  Error text.

- **params**

  Array of additional parameters of the error, which can be put into the error message

Listing 1: Response example

```
400 (Bad Request)
Content-Type: application/json
{
    "err": "E_INVALIDWALLET",
    "msg": "Wallet 1111-2222-3333 is not valid",
    "params": ["1111-2222-3333"]
}
```

Error List

E_CONTRACT
There is no %s contract.

E_DBNIL
DB is nil.

E_ECOSYSTEM
Ecosystem %d doesn't exist.

E_EMPTYPUBLIC
Public key is undefined.

E_EMPTYSIGN
Signature is undefined.

E_HASHWRONG
Hash is incorrect.

E_HASHNOTFOUND
Hash has not been found.

E_HEAVYPAGE
This page is heavy.

E_INSTALLED
Platform is already installed.

E_INVALIDWALLET
Wallet %s is not valid.

E_LIMITTXSIZE
The size of transaction is too big.

E_NOTFOUND
Content page or menu has not been found.

E_NOTINSTALLED
Platform is not installed.

In this case, you need to run the installation with the *install* command.

The **E_NOTINSTALLED** error should be returned by any command except for install, in case the system is not yet installed.

E_PARAMNOTFOUND
Parameter has not been found.

E_QUERY
DB query is wrong.

E_RECOVERED
API was recovered.

Returned if there is a panic error.

This error means that you have encountered a bug that needs to be found and fixed.

E_SERVER
Server error.

Returned if there is an error in the golang library functions. The *msg* field contains the error text.

The **E_SERVER** error may appear in response to any command. If it appears as a result of incorrect input parameters, it can be changed to relevant errors. In the other case, this error reports of invalid operation or incorrect system configuration, which requires a more detailed investigation.

E_SIGNATURE
Signature is incorrect.

E_STATELOGIN
%s is not a membership of ecosystem %s.

E_TABLENOTFOUND
Table %s has not been found.

E_TOKEN
Token is not valid.

E_TOKENEXPIRED
Token is expired by %s.

E_UNAUTHORIZED
Unauthorized.

The **E_UNAUTHORIZED** error can be returned for any command except for *install, getuid, login* in cases where login was not performed or the session has expired.

E_UNDEFINEVAL
Value %s is undefined.

E_UNKNOWNUID
Unknown uid.

E_OBSCREATED
Off-Blockchain Server is already created.

2.14.2 Routes unavailable in OBS

Requests that are not available in OBS.

- txstatus
- txinfo
- txinfoMultiple
- appparam
- appparams
- history
- balance
- block
- maxblockid
- ecosystemparams
- ecosystemparam
- systemparams
- ecosystems

2.14.3 Authentication

The JWT token is used for authentication. After receiving a JWT token, you must put it in the header of every request: `Authorization: Bearer TOKEN_HERE`.

getuid

GET/ Returns a unique value, which needs to be signed with your private key and sent back to server using the **login** command.

A temporary JWT token is created, which needs to be passed to **Authorization** when calling **login**.

Request

```
GET
/api/v2/getuid
```

Response

- *uid*

  Signature line.

- *token*

  Temporary token to pass in login.

  The lifetime of a temporary token is 5 seconds.

- *network_id*

  NetworkID server identifier.

In cases where authorization is not required, the following information is returned:

- *expire*

  Number of seconds before the time runs out.

- *ecosystem*

  Ecosystem ID.

- *key_id*

  Wallet ID.

- *address*

  Wallet address in the `XXXX-XXXX-.....-XXXX` format.

Response example

```
200 (OK)
Content-Type: application/json
{
    "uid": "28726874268427424",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6I........AU3yPRp64SLO4aJqhN-
↪kMoU5HNYTDplQXbVu0Y"
    "network_id": "5324976953280995276"
}
```

Errors

E_SERVER

login

POST/ User authentication.

The **getuid** command should be called in the first place in order to receive a unique value and sign it. A temporary JWT token, which was received along with getuid, should be passed in the header.

In case of success, the token that was received in the response should be included in all queries in the *Authorization* header.

Request

```
POST
/api/v2/login
```

- *[ecosystem]*

  Ecosystem ID.

  If not specified, the command will work with the first ecosystem.

- *[expire]*

  Lifetime of the JWT token in seconds (36000 by default).

- *[pubkey]*

  Public hex key, if the blockchain already stores a key, then the wallet number should be passed with the *key_id* parameter.

- *[key_id]*

  Account id as a number or in the `XXXX-...-XXXX` format.

  Use this in cases where the public key is already stored in the blockchain. Can't be used together with *pubkey*.

- *signature*

  A uid signature received though getuid hex.

Response

- *token*

  JWT token.

- *ecosystem*

  Ecosystem ID.

- ***key_id*** Account ID.

- *address*

  Account address in the `XXXX-XXXX-.....-XXXX` format.

- *notify_key*

  Key for notifications.

- *isnode*

  Whether this user is the owner of this node.

  Values: true, false.

- *isowner*

  Whether this user is the owner of this ecosystem.

  Values: true, false.

- *obs*

  Does this ecosystem have an off-blockchain server.

  Values: true, false.

Response example

```
200 (OK)
Content-Type: application/json
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6I........AU3yPRp64SLO4aJqhN-
↪kMoU5HNYT8fNGODp0Y"
    "ecosystem":"1",
    "key_id":"12345",
    "address": "1234-....-3424"
}
```

Errors

E_SERVER, E_UNKNOWNUID, E_SIGNATURE, E_STATELOGIN, E_EMPTYPUBLIC

2.14.4 Service Commands

version

GET/ Returns the current server version.

This request does not require authorization.

Request

```
GET
/api/v2/version
```

Response example

```
200 (OK)
Content-Type: application/json
"0.1.6"
```

2.14.5 Data Request Functions

balance

GET/ Requests the balance of an account in the current ecosystem.

Request

```
GET
/api/v2/balance/{key_id}
```

- *key_id*

  Account id. Can be specified in any format: `int64, uint64, XXXX-...-XXXX`. This wallet will be searched for in the ecosystem where the user is currently logged in.

Response

- *amount*

  Account balance in minimum units.

- *money*

  Account balance in units.

Response example

```
200 (OK)
Content-Type: application/json
{
    "amount": "123450000000000000000",
    "money": "123.45"
}
```

Errors

E_SERVER, E_INVALIDWALLET

blocks

GET/ Returns the list of blocks with additional information about transactions in each block.

This request does not require authorization.

Request

```
GET
/api/v2/blocks
```

Response

- Block number
  - List of transactions in the block with additional information for each transaction:
    - *hash*

      Transaction hash.
    - *contract_name*

      Name of the contract.
    - *params*

      Array with contract parameters.
    - *key_id*

      For the first block, identifier of the key that signed the first block.

      For all other blocks, identifier of the key that signed the transaction.

Response example

```
200 (OK)
Content-Type: application/json
{"1":
    [{"hash":"PhHV1g7jUyDEwiETexBMLJPEwH4yEknCIIOAj43Dn4U=",
    "contract_name":"",
    "params":null,
    "key_id":-2157832554603111963}]
}
```

Errors

E_SERVER, E_NOTFOUND

detailed_blocks

GET/ Returns the list of blocks with detailed additional information about transactions in each block.

This request does not require authorization.

Request

```
GET
/api/v2/detailed_blocks
```

Response

- Block identifier
  - *header*

    Block header with the following fields:

    * *block_id*

      Block number.

    * *time*

      Block generation timestamp.

    * *key_id*

      Identifier of the key that signed the block.

    * *node_position*

      Position of the node that generated the block in the list of full nodes.

    * *version*

      Block structure version.

  - *hash*

    Hash of the block.

  - *node_position*

    Position of the node that generated the block in the list of full nodes.

  - *key_id*

    Identifier of the key that signed the block.

  - *time*

    Block generation timestamp.

  - *tx_count*

    Number of transactions in the block.

  - *rollback_hash*

    Hash of all transactions in the block.

  - *mrkl_root*

Merkle root of transactions in the block.

- *bin_data*

  Serialized block header, transactions in this block, previous block hash, and the private key of the node that generated this block.

- *sys_update*

  Block contains a transaction that updates system parameters.

- *transactions*

  List of transactions in the block with additional information for each transaction:

  * *hash*

    Transaction hash.

  * *contract_name*

    Name of the contract.

  * *params*

    Array with contract parameters.

  * *key_id*

    For all other blocks, key identifier of the key that signed the transaction.

  * *time*

    Transaction generation timestamp.

  * *type*

    Transaction type.

Response example

```
200 (OK)
Content-Type: application/json
{"1":
    {"header":
        {"block_id":1,
        "time":1545342081,
        "ecosystem_id":0,
        "key_id":3670289659738809576,
        "node_position":0,
        "sign":null,
        "hash":null,
        "version":1},
    "hash":"TjTSRXcyJNgCn8GHEu16S2CheO0IZglxKQa/4S/Xzw4=",
    "ecosystem_id":0,
    "node_position":0,
    "key_id":3670289659738809576,
    "time":1545342081,
    "tx_count":1,
    "rollbacks_hash":"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=",
    "mrkl_root":
↪"NWNhODNmZGRiYmZhNTk3OTc0MzI1ODY4YjFiNDM3NDU3NTliOGUyNThmODM0NjY1ZWExOTkwZGZjNTZjZjhlMg==
↪",
```

(continues on next page)

(continued from previous page)

```
    "bin_data":null,
    "sys_update":false,
    "gen_block":false,
    "stop_count":0,
    "transactions":[
        {"hash":"ZkGFY/WrvlsHXhZtpmodEoMX6MsBwF2Ji1G5Y7XgRjY=","contract_name":"",
↪"params":null,"key_id":0,"time":0,"type":0}]
    }
}
```

Errors

E_SERVER, E_NOTFOUND

/data/{table}/{id}/{column}/{hash}

GET/ If the specified hash matches the hash of the data in the specified table, column and record ID, then this request returns the data. Otherwise, returns an error.

This request does not require authorization.

Request

```
GET
/data/{table}/{id}/{column}/{hash}
```

- *table*

  Table name.

- *id*

  Record ID.

- *column*

  Column name.

- *hash*

  Hash of the requested data.

Response

Binary data.

keyinfo

GET/ Returns a list of ecosystems with roles where the specified key is registered.

This request does not require authorization.

Request

```
GET
/api/v2/keyinfo/{key_id}
```

- *key_id*

  Account identifier. Can be specified in any format: `int64, uint64, XXXX-...-XXXX`.

  The search is performed in all ecosystems.

Response

- *ecosystem*

  Ecosystem identifier.

- *name*

  Ecosystem name.

- *roles*

  List of roles in the ecosystem with *id* and *name* fields.

Response example

```
200 (OK)
Content-Type: application/json
[{
    "ecosystem":"1",
    "name":"platform ecosystem",
    "roles":[{"id":"1","name":"Admin"},{"id":"2","name":"Developer"}]
}]
```

Errors

E_SERVER, E_INVALIDWALLET

2.14.6 Getting metrics

keys

GET/ Returns the number of keys.

Request

```
GET
/api/v2/metrics/keys
```

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

blocks

GET/ Returns the number of blocks.

Request

```
GET
/api/v2/metrics/blocks
```

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

transactions

GET/ Returns the number of transactions.

Request

```
GET
/api/v2/metrics/transactions
```

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

ecosystems

GET/ Returns the number of ecosystems.

Request

```
GET
/api/v2/metrics/ecosystems
```

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

fullnodes

GET/ Returns the number of validating nodes.

```
GET
/api/v2/metrics/fullnodes
```

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

2.14.7 Working with ecosystems

ecosystemname

GET/ returns the name of an ecosystem by its identifer.

This request does not require authorization.

```
GET
/api/v2/ecosystemname?id=..
```

- *id*

  Ecosystem ID.

Response example

```
200 (OK)
Content-Type: application/json
{
    "ecosystem_name": "platform_ecosystem"
}
```

Errors

E_PARAMNOTFOUND

ecosystems

GET/ Returns a number of ecosystems.

```
GET
/api/v2/ecosystems/
```

Response

- *number*

  The number of installed ecosystems.

Response example

```
200 (OK)
Content-Type: application/json
{
    "number": 100,
}
```

appparams

GET/ Returns a list of application parameters in the current or specified ecosystem.

Request

- *[appid]*

  Application identifier.

- *[ecosystem]*

  Ecosystem ID; if not specified, the current ecosystem's parameters will be returned.

- *[names]*

  List of received parameters.

  A list of parameter names separated by commas can be specified, for example: `/api/v2/appparams/1?names=name,mypar`.

Response

- *list*

  An array where each element contains the following parameters:

  - *name*–parameter name
  - *value*–parameter value
  - *conditions*–permissions to change a parameter

Response example

```
200 (OK)
Content-Type: application/json
{
    "list": [{
        "name": "name",
        "value": "MyState",
        "conditions": "true",
    },
    {
        "name": "mypar",
        "value": "My value",
        "conditions": "true",
    },
    ]
}
```

Errors

E_ECOSYSTEM

appparam/{appid}/{name}

GET/ Returns information about the **{name}** parameter of the **{appid}** application in the current or specified ecosystem.

- *appid*

  Application ID.

- *name*

  Name of the requested parameter.

- *[ecosystem]*

  Ecosystem ID (optional parameter).

  By default, the current ecosystem will be returned.

Request

```
GET
/api/v2/{appid}/{appid}/{name}[?ecosystem=1]
```

Response

- *id*

  Parameter identifier.

- *name*

  Parameter name.

- *value*

  Parameter value.

- *conditions*

  Conditions to change a parameter.

Response example

```
200 (OK)
Content-Type: application/json
{
    "id": "10",
    "name": "par",
    "value": "My value",
    "conditions": "true"
}
```

Errors

E_ECOSYSTEM, E_PARAMNOTFOUND

ecosystemparams

GET/ Returns a list of ecosystem parameters.

Request

```
GET
/api/v2/ecosystemparams/[?ecosystem=...&names=...]
```

- *[ecosystem]*

  Ecosystem identifier. If not specified, current ecosystem's parameters will be returned.

- *[names]*

List of parameters to receive, separated by commas.

Example: `/api/v2/ecosystemparams/?names=name,currency,logo*`.

Response

- *list*

  An array where each element stores the following parameters:

  - *name*–parameter name
  - *value*–parameter value
  - *conditions*–conditions to change the parameter

Response example

```
200 (OK)
Content-Type: application/json
{
    "list": [{
        "name": "name",
        "value": "MyState",
        "conditions": "true",
    },
    {
        "name": "currency",
        "value": "MY",
        "conditions": "true",
    },
    ]
}
```

Errors

E_ECOSYSTEM

ecosystemparam/{name}

GET/ Returns information about the **{name}** parameter in the current or specified ecosystem.

Request

```
GET
/api/v2/ecosystemparam/{name}[?ecosystem=1]
```

- *name*–name of the requested parameter,
- *[ecosystem]*–ecosystem ID can be specified. The current ecosystem value will be returned by default,

Response

- *name*–parameter name
- *value*–parameter value
- *conditions*–condition for parameter change

Response example

```
200 (OK)
Content-Type: application/json
{
    "name": "currency",
    "value": "MYCUR",
    "conditions": "true"
}
```

Errors

E_ECOSYSTEM

tables/[?limit=...&offset=...]

GET/ Returns a list of tables in the current ecosystem. You can add set an offset and specify a number of requested tables.

Request

- *[limit]*–number of entries (25 by default),
- *[offset]*–entries start offset (0 by default),

```
GET
/api/v2/tables
```

Response

- *count*–total number of entries in the table
- *list*–an array where each element stores the following parameters:
  - *name*–table name (returned without prefix)
  - *count*–number of entries in the table

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": "100"
    "list": [{
        "name": "accounts",
        "count": "10",
    },
    {
        "name": "citizens",
        "count": "5",
   },
    ]
}
```

table/{name}

GET/ Returns information about the requested table in the current ecosystem.

The next fields are returned:

- *name*–table name
- *insert*–rights to insert the elements
- *new_column*–rights to insert the column
- *update*–rights to change the rights
- *columns*–array of the columns with fields `name, type, perm` (name, type, rights for change).

Request

```
GET
/api/v2/table/mytable
```

- *name*–table name (without ecosystem ID prefix),

Response

- *name*–table name (without ecosystem ID prefix)
- *insert*–right for adding an entry
- *new_column*–right for adding a column
- *update*–right for changing entries
- *conditions*–right for changing table configuration
- *columns*–an array of information about columns:
  - *name*–column name
  - *type*–column type. Possible values include: `varchar,bytea,number,money,text,double, character`
  - *perm*–right for changing an entry in a column.

Response example

```
200 (OK)
Content-Type: application/json
{
    "name": "mytable",
    "insert": "ContractConditions(`MainCondition`)",
    "new_column": "ContractConditions(`MainCondition`)",
    "update": "ContractConditions(`MainCondition`)",
    "conditions": "ContractConditions(`MainCondition`)",
    "columns": [{"name": "mynum", "type": "number", "perm":
↪"ContractConditions(`MainCondition`)" },
        {"name": "mytext", "type": "text", "perm":"ContractConditions(`MainCondition`)
↪" }
    ]
}
```

Errors

E_TABLENOTFOUND

list/{name}[?limit=...&offset=...&columns=]

GET/ Returns a list of entries of the specified table in the current ecosystem. An offset and the number of requested table entries can be specified.

Request

- *name*–table name
- *[limit]*–number of entries (25 by default)
- *[offset]*–entries start offset (0 by default)
- *[columns]*–list of requested columns, separated by commas, if not specified, all columns will be returned. The id column will be returned in all cases.

```
GET
/api/v2/list/mytable?columns=name
```

Response

- *count* - total number of entries in the table,
- *list* - an array where each element stores the following parameters:
  - *id* - entry ID,
  - sequence of requested columns.

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": "10"
    "list": [{
        "id": "1",
        "name": "John",
    },
    {
        "id": "2",
        "name": "Mark",
   },
    ]
}
```

sections[?limit=...&offset=...&lang=]

GET/ Returns a list of records from the *sections* table of the current ecosystem. An offset and a record limit can be specified.

If the *role_access* field contains a list of roles and the current role is not present in this field, then the record will not be returned. The *title* column data is replaced with language resources.

Request

- *[limit]* - maximum number of returned records (25 by default)
- *[offset]* - offset for records (0 by default)
- *[lang]* - language code or lcid to enable language resources in this language. Examples: *en, ru, fr, en-US, en-GB*. If the specified language resource is not found (for example, *en-US*), it is searched in the language group (*en*).

```
GET
/api/v2/sections
```

Response

- *count* - total number of records in the *sections* table
- *list* - an array where each element contains all columns from the *sections* table.

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": "2"
    "list": [{
        "id": "1",
        "title": "Development",
```

(continues on next page)

(continued from previous page)

```
        "urlpage": "develop",
        ...
    },
    ]
}
```

Errors

E_TABLENOTFOUND

row/{tablename}/{id}[?columns=]

GET/ Returns a table entry with specified id in the current ecosystem. Columns to be returned can be specified.

Request

- *tablename*–table name
- *id*–entry ID
- *[columns]*–a list of requested columns, separated by commas. If not specified, all columns will be returned. The id column will be returned in all cases.

```
GET
/api/v2/row/mytable/10?columns=name
```

Response

- *value* - an array of received column values:
  - *id* - entry ID,
  - order of requested columns.

Response example

```
200 (OK)
Content-Type: application/json
{
    "values": {
    "id": "10",
    "name": "John",
    }
}
```

systemparams

GET/ Returns a list of system parameters.

Request

```
GET
/api/v2/systemparams/[?names=...]
```

- *[names]* - list of requested parameters, a list of parameters to receive can be specified separated by commas. For instance, `/api/v2/systemparams/?names=max_columns,max_indexes`.

Reply

- *list*–array, each element of which contains the following parameters:
- *name*–parameter name
- *value*–parameter value
- *conditions*–conditions for parameter change.

Response example

```
200 (OK)
Content-Type: application/json
{
    "list": [{
        "name": "max_columns",
        "value": "100",
        "conditions": "ContractAccess("@0UpdSysParam")",
    },
    {
        "name": "max_indexes",
        "value": "1",
        "conditions": "ContractAccess("@0UpdSysParam")",
    },
    ]
}
```

history/{name}/{id}

GET/ Returns the changelog of an entry in the specified table in the current ecosystem.

Request

- *name*–table name,
- *id*–entry identifier.

Reply

- *list* an array, the elements of which contain modified parameters of the requested entry

Reply Example

```
200 (OK)
Content-Type: application/json
{
    "list": [
        {
            "name": "default_page",
            "value": "P(class, Default Ecosystem Page)"
        },
        {
            "menu": "default_menu"
        }
    ]
}
```

interface/{page|menu|block}/{name}

GET/ Searches the current ecosystem and returns a record from the selected table (page, menu or block) with the specified name.

```
GET
/api/v2/interface/page/default_page
```

Request

- *name*–name of the record in the specified table.

Response

- *id* – identifier of the record,
- *name* – name of the record,
- other columns of the table.

Response example

```
200 (OK)
Content-Type: application/json
{
    "id": "1",
    "name": "default_page",
"value": "P(Page content)",
"default_menu": "default_menu",
"validate_count": 1
}
```

Errors

E_QUERY, *E_NOTFOUND*

2.14.8 Functions for working with contracts

contracts[?limit=...&offset=...]

GET/ Returns a list of contracts in the current ecosystem. An offset and a number of requested contracts can be specified.

Request

- *[limit]*–number of entries (25 by default)
- *[offset]*–entries start offset (0 by default)

```
GET
/api/v2/contracts
```

Response

- *count*–total number of entries in the table
- *list*–an array where each element stores the following parameters:
  - *id*–entry ID
  - *name*–contract name
  - *value*–initial text of the contract
  - *active*–equals "1" if the contract is bound to the account or "0" otherwise
  - *key_id*–account bound to the contract
  - *address*–address of the account bound to the contract in the `XXXX-...-XXXX` format
  - *conditions*–conditions for change
  - *token_id*–identifier of the ecosystem, which currency will be used to pay for the contract

Response example

```
200 (OK)
Content-Type: application/json
{
    "count": "10"
    "list": [{
        "id": "1",
        "name": "MainCondition",
        "token_id":"1",
        "key_id":"2061870654370469385",
        "active":"0",
```

(continues on next page)

(continued from previous page)

```
          "value":"contract MainCondition {
conditions {
    if(StateVal(`founder_account`)!=$citizen)
    {
        warning `Sorry, you dont have access to this action.`
      }
    }
  }",
  "address":"0206-1870-6543-7046-9385",
  "conditions":"ContractConditions(`MainCondition`)"
   },
  ...
    ]
  }
```

contract/{name}

GET/ Returns information about the {name} smart contract. By default, the smart contract will be searched for in the current ecosystem.

Request

- *name*–contract name.

```
GET
/api/v2/contract/mycontract
```

Response

- *id*–identifier of the contract in VM.
- *name*–name of the smart contract with ecosystem ID. Example: `@{idecosystem}name`.
- *state*–ID of the ecosystem where the contract was created.
- *walletid* -ID the contract owner wallet.
- *tokenid* - tokens that are accepted as contract payment.
- *address*–address of the account bound to the contract in the `XXXX-...-XXXX` format.
- *tableid*–entry ID in the contracts table, where the source code of the contract is stored.
- *fields*–an array that contains information about every parameter in the **data** section of the contract and contains the following fields:
  - *name*–field name,
  - *type*–parameter type,
  - *optional*–parameter optionality flag, this value is `true` if a parameter is optional and `false` if it is mandatory.

Response example

```
200 (OK)
Content-Type: application/json
{
    "fields" : [
        {"name":"amount", "type":"int", "optional": false},
        {"name":"name", "type":"string", "optional": true}
    ],
    "id": 150,
    "name": "@1mycontract",
    "tableid" : 10,
}
```

sendTX

POST/ Accepts transactions passed in the parameters and adds them to the transaction queue. If the execution is successful, transaction hashes are returned. A hash of the transaction can be used to get the block number for the transaction, or an error message in case of an error.

Request

- *tx_key* - transaction contents. You can specify any name for this parameter. To specify several transactions, use different names.

```
POST
/api/v2/sendTx

Headers:
Content-Type: multipart/form-data

Parameters:
tx1 - contents of transaction 1
txN - contents of transaction N
```

Response

- ***hashes* - dictionary with transaction hashes**
  - *tx1* - hex hash of transaction 1
  - *txN* - hex hash of transaction N

Response example

```
200 (OK)
Content-Type: application/json
{
    "hashes": {
        "tx1": "67afbc435634.....",
        "txN": "89ce4498eaf7.....",
}
```

Errors

E_LIMITTXSIZE

txstatus

POST/ Returns a block number or an error for a transaction with the specified hash. If the returned values of *blockid* and *errmsg* are empty, then the transaction hasn't yet been included into a block.

Request

- *data*–json with a list of transaction hashes.

```
{"hashes":["contract1hash", "contract2hash", "contract3hash"]}
```

```
POST
/api/v2/txstatus/
```

Response

- *results*–dictionary with transaction hashes as keys and transaction execution details as values.

  hash–transaction hash

  - *blockid*–number of the block if the transaction was processed successfully.
  - *result*–result of the transaction operation returned through the **$result** variable.
  - *errmsg*–error message if the transaction was refused.

Response example

```
200 (OK)
Content-Type: application/json
{"results":
  {
    "hash1": {
         "blockid": "3123",
         "result": "",
     },
     "hash2": {
          "blockid": "3124",
          "result": "",
     }
   }
 }
```

Errors

E_HASHWRONG, E_HASHNOTFOUND

txinfo/{hash}

GET/ Returns information about a transaction with a specified hash. Response contains the block number and amount of confirmations. As an option the corresponding contract name and its parameters can be returned.

Request

- *hash* - hash of a transaction.
- *[contractinfo]* - contract information flag. To get information about the contract and parameters for this transaction, specify `1`.

```
GET
/api/v2/txinfo/2353467abcd7436ef47438
```

Response

- *blockid* - number of the block where this transaction was included. If this value is `0`, then a transaction with this hash was not found.
- *confirm* - number of confirmations for this block.
- *data* - if *contentinfo* is `1`, then a json with contract information is returned in this parameter.

Response example

```
200 (OK)
Content-Type: application/json
{
    "blockid": "4235237",
    "confirm": "10"
}
```

Errors

E_HASHWRONG

txinfoMultiple/

GET/ Returns information about transactions with specified hashes.

Request

- *data* - json with a list of transaction hashes in hexadecimal string format.
- *[contractinfo]* - contract information flag. To get information about the contract and parameters for this transaction, specify `1`.

```
{"hashes":["contract1hash", "contract2hash", "contract3hash"]}
```

```
GET
/api/v2/txinfoMultiple/
```

Response

- *results* - dictionary that has transaction hashes as keys and transaction information as values.

  hash - hash of a transaction

  - *blockid* - number of the block where this transaction was included. If this value is `0`, then a transaction with this hash was not found.
  - *confirm* - number of confirmations for this block.
  - *data* - if *contentinfo* is `1`, then a json with contract information is returned in this parameter.

Response example

```
200 (OK)
Content-Type: application/json
{"results":
  {
    "hash1": {
         "blockid": "3123",
         "confirm": "5",
     },
     "hash2": {
          "blockid": "3124",
          "confirm": "3",
     }
   }
 }
```

Errors

E_HASHWRONG

/page/validators_count/{name}

GET/ Returns the number of nodes required for the validation of the specified page.

Request

- *name* - page name with ecosystem index prefix. The format is `@ecosystem_id%%page_name%`. For example, `@1main_page`.

```
GET
/api/v2/page/validators_count/@1page_name
```

Response

- *validate_count* - number of nodes required to validate the specified page.

Response example

```
200 (OK)
Content-Type: application/json
{"validate_count":1}
```

Errors

E_NOTFOUND, E_SERVER

content/{menu|page}/{name}

POST/ Returns a JSON representation of the code of the specified page or menu named **{name}**, which is the result of processing by the template engine. The request can have additional parameters, which can be used in the template engine. If the page or menu can't be found, the 404 error is returned.

Request

- *menu|page–page* or *menu* to recieve the page or menu
- *name*–the name or menu of the page
- *[lang]*–either lcid or a two-letter language code can be specified to address the corresponding language resources. For example, *en,ru,fr,en-US,en-GB*. If, for example, the *en-US* resource will not be found, the *en* resources will be used instead of the missing *en-US* ones,
- *[app_id]*–application ID. Passed together with lang, because the functions that work with the language in the template engine don't automatically recognize the AppID. Should be passed as a number,

```
POST
/api/v2/content/page/default
```

Response

- *menu*–the menu name for the page when calling *content/page/...*
- *menutree*–JSON menu tree for the page when calling *content/page/...*
- *title*–head for the menu *content/menu/...*
- *tree*–JSON tree of objects

Response example

```
200 (OK)
Content-Type: application/json
{
    "tree": {"type":"......",
          "children": [
               {...},
               {...}
          ]
    },
}
```

Errors

E_NOTFOUND

content/source/{name}

POST/ Returns a JSON-representation of the **{name}** page code without executing any functions or receiving any data. The returned tree corresponds to the page template and can be used in the visual designer. If the page or the menu are not found, a 404 error is returned.

Request

- *name*–name of the requested page,

Response

```
POST
/api/v2/content/source/default
```

- *tree*–JSON object tree.

Response example

```
200 (OK)
Content-Type: application/json
{
    "tree": {"type":"......",
          "children": [
               {...},
               {...}
          ]
    },
}
```

Errors

E_NOTFOUND, E_SERVER

content/hash/{name}

POST/ Returns a SHA256 hash of the **{name}** page. If the page or menu is not found, a 404 error is returned.

This request does not require authorization. Because of this, to receive a correct hash when making a request to other nodes, *ecosystem*, *keyID*, *roleID*, and *isMobile* parameters must be also passed. To receive pages from other systems, a `@ecosystem_id` prefix must be added to the page name. For example: `@2mypage`.

Request

- *name*–page name
- *ecosystem*–ecosystem identifier
- *keyID*–user identifier
- *roleID*–user role identifier
- *isMobile*–mobile platform execution flag

```
POST
/api/v2/content/hash/default
```

Response

- *hex*–resulting hash in the hex string format

Response example

```
200 (OK)
Content-Type: application/json
{
    "hash": "01fa34b589...."
}
```

Errors

E_NOTFOUND, E_SERVER, E_HEAVYPAGE

content

POST/ Returns a JSON-representation of the page source code from the **template** parameter. If the additional parameter **source** is specified as true or 1, the JSON-representation will be returned without execution of functions and without receiving data. The returned tree corresponds to the sent template and can be used in the visual designer.

This request does not require authorization.

Request

- *template*–page template source code to be processed
- *[source]*–if set to true or 1, the tree will be returned without execution of functions and without receiving data.

```
POST
/api/v2/content
```

Response

- *tree*–JSON object tree.

Response example

```
200 (OK)
Content-Type: application/json
{
    "tree": {"type":"......",
          "children": [
               {...},
               {...}
          ]
    },
}
```

Errors

E_NOTFOUND, E_SERVER

node/{name}

POST Calls the **{name}** smart contract on behalf of a node. Used for calling smart contracts from OBS contracts though the **HTTPRequest** function. Since in this case the contract can't be signed with an account key, it will be signed with the node's private key. All other parameters are similar to those when sending a contract. The called contract should be bound to an account, because the node's private key account does not have enough funds to execute the contract. If the contract is called from a OBS contract, then the authorization token **$auth_token** should be passed to **HTTPRequest**.

```
var pars, heads map
heads["Authorization"] = "Bearer " + $auth_token
pars["obs"] = "false"
ret = HTTPRequest("http://localhost:7079/api/v2/node/mycontract", "POST", heads, pars)
```

Request

```
POST
/api/v2/node/mycontract
```

Response

- *hash*–hex hash of the sent transaction

Reply example

```
200 (OK)
Content-Type: application/json
{
    "hash" : "67afbc435634.....",
}
```

maxblockid

GET/ Returns the highest block ID on the current node.

This request does not require authorization.

Request

```
GET
/api/v2/maxblockid
```

Reply

- *max_block_id*–highest block id on the current node

Reply Example

```
200 (OK)
Content-Type: application/json
{
    "max_block_id" : 341,
}
```

Errors

E_NOTFOUND

block/{id}

GET/ Returns information on the block with the specified ID.

This request does not require authorization.

Request

- *id*–id of the requested block.

```
POST
/api/v2/block/32
```

Reply

- *hash*–hash of the block
- *ecosystem_id*–ecosystem id
- *key_id*–key which signed the block
- *time*–block generation timestamp
- *tx_count*–number of transactions in the block
- *rollbacks_hash*–hash of rollbacks, created by transactions in the block

Reply example

```
200 (OK)
Content-Type: application/json
{
    "hash": "\x1214451d1144a51",
    "ecosystem_id": 1,
    "key_id": -13646477,
    "time": 134415251,
    "tx_count": 3,
    "rollbacks_hash": "\xa1234b1234"
}
```

Errors

E_NOTFOUND

avatar/{ecosystem}/{member}

GET/ Returns user's avatar (available without login).

Request

- *ecosystem*–user's ecosystem ID
- *member*–user ID

```
GET
/api/v2/avatar/1/7136200061669836581
```

Response

Header Content-Type with the image type. Image data is returned in the in the response body.

Response example

```
200 (OK)
Content-Type: image/png
```

Errors

E_NOTFOUND E_SERVER

config/centrifugo

GET/ Returns centrifugo's host and port.

This request does not require authorization.

Request

```
GET
/api/v2/config/centrifugo
```

Response

String in the `http://address:port` format that is returned in the response body. Example: `http://127.0.0.1:8000`

Errors

E_SERVER

updnotificator

POST/ Sends all messages that haven't been sent yet to the centrifugo notification service. Only sends messages for the specified ecosystems and members.

This request does not require authorization.

Request

A list in the format:

- *id*

  Member ID.

- *ecosystem*

  Ecosystem ID.

```
POST
/updnotificator
```

Response example

```
200 (OK)
Content-Type: application/json
{
    "result": true
}
```

2.15 Platform parameters

2.15.1 About platform parameters

Platform parameters are configuration parameters for the blockchain platform. These parameters apply to the blockchain network and to all ecosystems in the network.

Where platform parameters are stored

Platform parameters are stored in the `system_parameters` table.

This table is available in the first (default) ecosystem that is created on the blockchain network.

Changing platform parameters

Changes to platform parameters must be introduced as a result of voting.

2.15.2 Platform parameters by purpose

Blockchain network

Nodes:

- *full_nodes*
- *number_of_nodes*

Node ban:

- *incorrect_blocks_per_day*
- *node_ban_time*
- *node_ban_time_local*

New ecosystems

Default pages and menus:

- *default_ecosystem_page*
- *default_ecosystem_menu*

Default contract:

- *default_ecosystem_contract*

Database

Table limits:

- *max_columns*
- *max_indexes*

Block generation

Time limits:

- *gap_between_blocks*
- *max_block_generation_time*

Transaction number limits:

- *max_tx_block*
- *max_tx_block_per_user*

Size limits:

- *max_tx_size*
- *max_block_size*
- *max_forsign_size*

Fuel limits:

- *max_fuel_block*
- *max_fuel_tx*

Block rollback:

- *rollback_blocks*

Fuel and currencies

Rewards and commission:

- *block_reward*
- *commission_wallet*
- *commission_size*

Fuel units exchange:

- *fuel_rate*

Prices for data:

- *price_tx_data*

Prices for new elements:

- *price_create_contract*
- *price_create_menu*
- *price_create_page*

Prices for operations:

- *price_exec_bind_wallet*
- *price_exec_address_to_id*
- *price_exec_column_condition*
- *price_exec_compile_contract*
- *price_exec_contains*
- *price_exec_contracts_list*
- *price_exec_contract_by_name*
- *price_exec_contract_by_id*
- *price_exec_create_column*
- *price_exec_create_ecosystem*
- *price_exec_create_table*
- *price_exec_unbind_wallet*
- *price_exec_ecosys_param*
- *price_exec_eval*
- *price_exec_eval_condition*
- *price_exec_flush_contract*
- *price_exec_has_prefix*
- *price_exec_id_to_address*
- *price_exec_is_object*
- *price_exec_join*
- *price_exec_json_to_map*
- *price_exec_len*
- *price_exec_perm_column*
- *price_exec_perm_table*
- *price_exec_pub_to_id*
- *price_exec_replace*
- *price_exec_sha256*
- *price_exec_size*
- *price_exec_substr*
- *price_exec_sys_fuel*
- *price_exec_sys_param_int*
- *price_exec_sys_param_string*
- *price_exec_table_conditions*
- *price_exec_update_lang*
- *price_exec_validate_condition*

Deprecated

Deprecated parameters:

- *blockchain_url*

2.15.3 Platform parameters

block_reward

Amount of APL tokens that is awarded to the node that generated a block.

An account that receives the reward is specified in the *full_nodes* parameter.

This parameter is measured in APL tokens.

blockchain_url

This parameter is deprecated.

commission_size

Commission percent.

This amount of commission is collected from the total contract cost. Commission is applied to the total contract cost in APL tokens.

Tokens are transferred to the account specified in the *commission_wallet* parameter.

commission_wallet

Account that collects commission for operations.

Size of the commission is specified in the *commission_size* parameter.

default_ecosystem_contract

Source code of the default contract for a new ecosystem.

This contract provides access rights to the ecosystem founder.

default_ecosystem_menu

Source code of the default menu for a new ecosystem.

default_ecosystem_page

Source code of the default page for a new ecosystem.

fuel_rate

Exchange rate for tokens of different ecosystems to fuel units.

Format for this parameter is:

```
[["ecosystem_id", "token_to_fuel_rate"], ["ecosystem_id2",
"token_to_fuel_rate2"], ...]
```

- `ecosystem_id`

Ecosystem identifier.

- `token_to_fuel_rate`

  Exchange rate of tokens to fuel units.

Example:

```
[["1","1000000000000000"], ["2", "1000"]]
```

One token from ecosystem 1 is exchanged to 1000000000000000 fuel units. One token from ecosystem 2 is exchanged to 1000 fuel units.

full_nodes

List of validating nodes of the blockchain network.

Format for this parameter is:

```
[["host:port","wallet_id","node_pub"], ["host2:port2",
"wallet_id2","node_pub2"]]
```

- `host:port`

  Address and port of the node host.

  Transactions and new blocks are sent to this host. This address can also be used to obtain the full blockchain starting from the first block.

- `wallet_id`

  Wallet (account identifier) that receives rewards for generating new blocks and processing transactions.

- `node_pub`

  Public key of the node. This key is used to check block signatures.

gap_between_blocks

Amount of time, in seconds, that a node can use to create a new block.

This parameter is a network parameter. All nodes in the network use it to determine when to generate new blocks. If a node did not create a block in this time period, the turn passes to the next node in a list of validating nodes.

Minimum value for this parameter is `1` (one second).

incorrect_blocks_per_day

Amount of incorrect blocks per day that a node may generate before it is banned from the network.

When more than half of nodes in a network have received this amount of incorrect blocks from a certain node, this node is banned from the network for *node_ban_time* amount of time.

max_block_generation_time

Maximum amount of time that a node may spend to generate a block, in ms.

max_block_size

Maximum block size, in bytes.

max_columns

Maximum number of columns in tables.

The predefined `id` column is not included in this maximum.

max_forsign_size

Maximum size, in bytes, of a forsign (string to be signed) generated for a transaction.

max_fuel_block

Maximum total fuel cost of a single block.

max_fuel_tx

Maximum total fuel cost of a single transaction.

max_indexes

Maximum number of index fields in a table.

max_tx_block

Maimum number of transactions in a single block.

max_tx_block_per_user

Maximum number of transactions in one block that belong to one account.

max_tx_size

Maximum transaction size, in bytes.

node_ban_time

Global ban period for nodes, in ms.

When more than half of nodes in a network have received *incorrect_blocks_per_day* amount of blocks from a certain node, this node is banned from the network for the specified amount of time.

node_ban_time_local

Local ban period for nodes, in ms.

When a node receives an incorrect block from another node, it bans the sender node locally for this amount of time.

number_of_nodes

Maximum number of validating nodes in the *full_nodes* parameter.

price_create_contract

Fuel cost for creating a new contract.

This parameter defines additional fuel cost of the `@1NewContract` contract. When this contract is executed, fuel costs for executing functions in this contract are also counted and added to the total cost.

This parameter is measured in fuel units. Fuel units are exchanged to APL tokens using *fuel_rate*.

price_create_menu

Fuel cost for creating a new menu.

This parameter defines additional fuel cost of the `@1NewMenu` contract. When this contract is executed, fuel costs for executing functions in this contract are also counted and added to the total cost.

This parameter is measured in fuel units. Fuel units are exchanged to APL tokens using *fuel_rate*.

price_create_page

Fuel cost for creating a new page.

This parameter defines additional fuel cost of the `@1NewPage` contract. When this contract is executed, fuel costs for functions in this contract are also counted and added to the total cost.

This parameter is measured in fuel units. Fuel units are exchanged to APL tokens using *fuel_rate*.

price_exec_address_to_id

Fuel cost of `AddressToId` function call.

price_exec_bind_wallet

Fuel cost of `Activate` function call.

price_exec_column_condition

Fuel cost of `ColumnCondition` function call.

price_exec_compile_contract

Fuel cost of `CompileContract` function call.

price_exec_contains

Fuel cost of `Contains` function call.

price_exec_contract_by_id

Fuel cost of `GetContractById` function call.

price_exec_contract_by_name

Fuel cost of `GetContractByName` function call.

price_exec_contracts_list

Fuel cost of `ContractsList` function call.

price_exec_create_column

Fuel cost of `CreateColumn` function call.

price_exec_create_ecosystem

Fuel cost of `CreateEcosystem` function call.

price_exec_create_table

Fuel cost of `CreateTable` function call.

price_exec_ecosys_param

Fuel cost of `EcosysParam` function call.

price_exec_eval

Fuel cost of `Eval` function call.

price_exec_eval_condition

Fuel cost of `EvalCondition` function call.

price_exec_flush_contract

Fuel cost of `FlushContract` function call.

price_exec_has_prefix

Fuel cost of `HasPrefix` function call.

price_exec_id_to_address

Fuel cost of `IdToAddress` function call.

price_exec_is_object

Fuel cost of `IsObject` function call.

price_exec_join

Fuel cost of `Join` function call.

price_exec_json_to_map

Fuel cost of `JSONToMap` function call.

price_exec_len

Fuel cost of `Len` function call.

price_exec_perm_column

Fuel cost of `PermColumn` function call.

price_exec_perm_table

Fuel cost of `PermTable` function call.

price_exec_pub_to_id

Fuel cost of `PubToID` function call.

price_exec_replace

Fuel cost of `Replace` function call.

price_exec_sha256

Fuel cost of `Sha256` function call.

price_exec_size

Fuel cost of `Size` function call.

price_exec_substr

Fuel cost of `Substr` function call.

price_exec_sys_fuel

Fuel cost of `SysFuel` function call.

price_exec_sys_param_int

Fuel cost of `SysParamInt` function call.

price_exec_sys_param_string

Fuel cost of `SysParamString` function call.

price_exec_table_conditions

Fuel cost of `TableConditions` function call.

price_exec_unbind_wallet

Fuel cost of `Deactivate` function call.

price_exec_update_lang

Fuel cost of `UpdateLang` function call.

price_exec_validate_condition

Fuel cost of `ValidateCondition` function call.

price_tx_data

Fuel cost taken per 1024 bytes of data passed to a transaction.

This parameter is measured in fuel units.

rollback_blocks

Number of blocks that can be rolled back in case when a fork is detected in the blockchain.

2.16 Backend configuration file

This section describes parameters in the backend configuration file.

2.16.1 About the backend configuration file

Backend configuration file defines configuration for a Apla node.

2.16.2 Location

This file is located in the directory with the backend binary and is named `config.toml`.

2.16.3 Sections

The configuration file has several sections:

default section
: This section defines general parameters.

[TCPServer]
: This section defines parameters for TCPServer.

  TCPServer supports the network interaction between nodes.

[HTTP]
: This section defines parameters for HTTPServer.

  HTTPServer provides REST API.

[DB]
: This section defines parameters for the node's database engine, PostgreSQL.

[StatsD]
: This section defines parameters for the node operation metrics collector, StatsD.

[Centrifugo]
: This section defines parameters for the notifications service, Centrifugo.

2.16.4 Configuration file example

```
LogLevel = "ERROR"
LogFileName = ""
InstallType = "PRIVATE_NET"
NodeStateID = "*"
TestMode = false
StartDaemons = ""
KeyID = -3785392309674179665
EcosystemID = 0
BadBlocks = ""
FirstLoadBlockchainURL = ""
FirstLoadBlockchain = ""
MaxPageGenerationTime = 0
WorkDir = "files"
PrivateDir = "files"
RunningMode = "privateBlockchain"

[TCPServer]
  Host = "127.0.0.1"
  Port = 7078
```

(continues on next page)

(continued from previous page)

```
[HTTP]
  Host = "127.0.0.1"
  Port = 7079

[DB]
  Name = "egaas"
  Host = "localhost"
  Port = 5432
  User = "egaas"
  Password = "egaas"

[StatsD]
  Name = "apla"
  Host = "127.0.0.1"
  Port = 8125

[Centrifugo]
  Secret = ""
  URL = ""

[Autoupdate]
  ServerAddress = "http://127.0.0.1:12345"
  PublicKeyPath = "update.pub"
```

2.17 Update Management Client

Update_client is a REST client that allows to communicate with the update server using command prompt.

Update_client can be used for:

- Uploading binaries to the update server
- Downloading binaries from the update server
- Removing binaries from the update server
- Requesting a list of versions of binaries available on the update server
- Generating keys
- Updating the go-apla binary

2.17.1 Location

tools/update_client/

2.17.2 Commands and Flags

add-binary

Adds a binary to the update server.

- **–server**–address of the update server.
- **–login**–your login on the update server.

- **–password**–your password on the update server.
- **–binary-path**–path to the binary.
- **–start-block**–the block number from which this binary can be used.
- **–version**–version name of the binary.
- **–key-path**–path to the private key for signature of the binary.

get-binary

Download a binary from the update server.

- **–server**–address of the update server.
- **–version**–binary version to download.
- **–binary-path**–path to the directory to download the binary to.
- **–publ-key-path**–path to the public key.

remove-binary

Remove a binary version from the update server.

- **–server**–address of the update server.
- **–login**–your login on the update server.
- **–password**–your password on the update server.
- **–version** –binary version to remove.

generate-keys

Generate a private-public key pair.

- **–publ-key-path**–path to the public key. By default – resources/key.pub.
- **–key-path**–path to the private key. By default – resources/key.

versions

Request versions of binaries available for downloading.

- **–server**–address of the update server.
- **–version**–can be used to check the availability of a specific version of the binary.

2.18 Synchronization Monitoring

Desync_monitor is a special tool that allows to verify that the databases on specified nodes are synchronized.

The tool can work as a daemon process or can be launched to perform a one-time check.

The tool's operation principle is based on the following:

1. Each block contains the hash of all changes made by all transactions in the block. The specified nodes are requested to provide the ID of their last common block.
2. A block with this ID is then requested from all nodes, and the aforementioned hash is compared.
3. If the hash differs, a synchronization error message is sent to the email address specified in the command.

2.18.1 Location

The tool is located in `tools/desync_monitor/`.

2.18.2 Command-prompt flags

The following flags can be used from the command prompt:

- **confPath**–path to the configuration file. By default – config.toml.
- **nodesList**–a list of nodes to request blocks from, separated by commas. By default – 127.0.0.1:7079.
- **daemonMode**–launch as a daemon process; should be used in case the verification is required every N seconds. This flag is set to false by default.
- **queryingPeriod**–if the tool is launched as a daemon process, this parameter sets the time interval between checks (in seconds). By default – 1 second.
- **alertMessageTo**–an email address to which the synchronization error alerts will be sent. By default – alert@apla.io.
- **alertMessageSubj**–message subject to put in the alert message. By default – problem with nodes synchronization.
- **alertMessageFrom**–address from which the message will be sent. By default – monitor@apla.io.
- **smtpHost**–SMTP server, which will be used to send the email message. By default – “”.
- **smtpPort**–SMTP server port, which will be used to send the email message. By default – 25.
- **smtpUsername**–SMTP server username. By default – “”.
- **smtpPassword**–SMTP server password. By default – “”.

2.18.3 Configuration

The tool uses a configuration file in the toml format.

By default, it looks for the config.toml file in the folder from which the binary was launched.

The file path can be changed using the configPath flag.

Listing 2: Configuration file example

```
nodes_list = ["http://127.0.0.1:7079", "http://127.0.0.1:7002"]

[daemon]
daemon = false
querying_period = 1

[alert_message]
to = "genesis@genesis.space"
```

(continues on next page)

(continued from previous page)

```
subject = "problem with genesis nodes"
from = "monitor@genesis.space"

[smtp]
host = ""
port = 25
username = ""
password = ""
```

nodes_list

- **nodes_list**–list of nodes (hosts) to request information from.

[daemon]

Daemon mode configuration.

- **daemon_mode**–tells the tool to work as a daemon process and perform synchronization checks.
- **querying_period**–time interval between synchronization checks.

[alert_message]

Alert message parameters.

- **to**–address to which the synchronization error alert messages will be sent.
- **subject**–message subject.
- **from**–sender.

[smtp]

Parameters of the SMTP server, which will be used to send synchronization error messages.

- **host**–SMTP server, which will be used to send the email messages.
- **port**–SMTP server port, which will be used to send the email messages.
- **username**–SMTP server username.
- **password**–SMTP server password.

2.19 Blockchain network deployment

This guide demonstrates how to deploy your own blockchain network.

Note:

- If you want to try the Apla blockchain network instead, check out Apla Testnet.
- If you want to deploy a local testing environment quickly, consider using Apla quickstart which is based on Docker images.

2.19.1 Example deployment

This guide deploys the example blockchain network of three nodes.

The network has three nodes:

- Node 1 is the first node in the blockchain network. It can generate new blocks and send transactions from the clients connected to it.
- Node 2 is an extra validating node. It can generate new blocks and send transactions from the clients connected to it.
- Node 3 is an extra node. It cannot generate new blocks but can send transactions from the clients connected to it.

Nodes are deployed in this configuration:

- Each node will use its own instance of PostgreSQL database system.
- Each node will use its own instance of Centrifugo service.
- The go-apla services will be deployed on the same host with other backend components.

Addresses and ports used by nodes are described in the following table:

Node	Component	IP and Port
1	PostgreSQL	127.0.0.1:5432
1	Centrifugo	10.10.99.1:8000
1	go-apla (TCP-server)	10.10.99.1:7078
1	go-apla (API-server)	10.10.99.1:7079
2	PostgreSQL	127.0.0.1:5432
2	Centrifugo	10.10.99.2:8000
2	go-apla (TCP-server)	10.10.99.2:7078
2	go-apla (API-server)	10.10.99.2:7079
3	PostgreSQL	127.0.0.1:5432
3	Centrifugo	10.10.99.3:8000
3	go-apla (TCP-server)	10.10.99.3:7078
3	go-apla (API-server)	10.10.99.3:7079

2.19.2 Deployment stages

Your own blockchain network must be deployed in several stages:

1. *Backend deployment*
   (a) *Deploying the first node*
   (b) *Deploying additional nodes*
2. *Frontend deployment*
3. *Blockchain network configuration*
   (a) *Creating the founder's account*
   (b) *Importing apps, roles, and templates*
   (c) *Adding first node to the list of nodes*
4. *Adding extra validating nodes*
   (a) *Adding a member to Consensus role*

(b) *Creating the node owner's account*

(c) *Adding node owner to Validators role*

(d) *Adding the validating node via voting*

2.19.3 Backend deployment

Deploying the first node

First node is a special node because it must be used to start the blockchain network. First block of the blockchain is generated by the first node and all other nodes download the blockchain from it. The owner of the first node becomes the platform *founder*.

Dependencies and environment setup

sudo

All commands for Debian 9 must be run as a non-root user. But some system commands need superuser privileges to be executed. By default, sudo is not installed on Debian 9, and you must install it first.

1. Become the root superuser.

```
su -
```

2. Upgrade your system.

```
apt update -y && apt upgrade -y && apt dist-upgrade -y
```

3. Install sudo.

```
apt install sudo -y
```

4. Add your user to the sudo group.

```
usermod -a -G sudo user
```

5. After the reboot, the changes take effect.

Go language

Install Go as described in the official documentation.

1. Download the latest stable version of Go (> 1.10.x) from the Golang official site or via the command line:

```
wget https://dl.google.com/go/go1.11.2.linux-amd64.tar.gz
```

2. Extract the package to `/usr/local`.

```
tar -C /usr/local -xzf go1.11.2.linux-amd64.tar.gz
```

3. Add `/usr/local/go/bin` to the PATH environment variable (either to `/etc/profile` or `$HOME/.profile`).

```
export PATH=$PATH:/usr/local/go/bin
```

4. For changes to take effect, `source` this file. For example:

```
source $HOME/.profile
```

5. Remove the temporary file:

```
rm go1.11.2.linux-amd64.tar.gz
```

PostgreSQL

1. Install PostgreSQL (> v.10) and psql:

```
sudo apt install -y postgresql
```

Centrifugo

1. Download Centrifugo version 1.8.0 from GitHub or via command line:

```
wget https://github.com/centrifugal/centrifugo/releases/download/v1.8.0/centrifugo-1.
↪8.0-linux-amd64.zip \
&& unzip centrifugo-1.8.0-linux-amd64.zip \
&& mkdir centrifugo \
&& mv centrifugo-1.8.0-linux-amd64/* centrifugo/
```

2. Remove temporary files:

```
rm -R centrifugo-1.8.0-linux-amd64 \
&& rm centrifugo-1.8.0-linux-amd64.zip
```

Directories

For Debian 9 OS, it is recommended to store all software used by the blockchain platform in a separate directory.

This guide uses the `/opt/apla` directory, but you can use any directory. In this case, change all commands and configuration files accordingly.

1. Make a directory for the blockchain platform:

```
sudo mkdir /opt/apla
```

2. Make your user the owner of this directory:

```
sudo chown user /opt/apla/
```

3. Make subdirectories for Centrifugo, go-apla, and node data. In this guide, all node data is stored in the directories with `nodeX` name, where `X` is the node number. Depending on which node you are deploying, this will be `node1` for node 1, `node2` for node 2, and so on.

```
mkdir /opt/apla/go-apla \
mkdir /opt/apla/go-apla/node1 \
mkdir /opt/apla/centrifugo \
```

Creating the database

1. Change user's password postgres to Apla's default password. You can set your own password, but then you must change it in the node configuration file config.toml.

```
sudo -u postgres psql -c "ALTER USER postgres WITH PASSWORD 'apla'"
```

2. Create a node current state database, for example 'apladb':

```
sudo -u postgres psql -c "CREATE DATABASE apladb"
```

Configuring Centrifugo

1. Create Centrifugo configuration file:

```
echo '{"secret":"CENT_SECRET"}' > /opt/apla/centrifugo/config.json
```

You can set your own "secret", but then you also must change it in the node configuration file config.toml.

Installing go-apla

1. Download and build the latest release of go-apla from GitHub:

```
go get -v github.com/AplaProject/go-apla
```

2. Copy the go-apla binary to the `/opt/apla/go-apla` directory. If you use the default Go workspace then the binary is located in the `$HOME/go/bin` directory:

```
cp $HOME/go/bin/go-apla /opt/apla/go-apla
```

Configuring the first node

1. Create the node 1 configuration file:

```
/opt/apla/go-apla/go-apla config \
    --dataDir=/opt/apla/go-apla/node1 \
    --dbName=apladb \
    --centSecret="CENT_SECRET" --centUrl=http://10.10.99.1:8000 \
    --httpHost=10.10.99.1 \
    --httpPort=7079 \
    --tcpHost=10.10.99.1 \
    --tcpPort=7078
```

4. Generate node 1 keys:

```
/opt/apla/go-apla/go-apla generateKeys \
    --config=/opt/apla/go-apla/node1/config.toml
```

5. Generate the first block:

Note: If you are creating your own blockchain network. you must use the `--test=true` option. Otherwise you will not be able to create new accounts.

```
/opt/apla/go-apla/go-apla generateFirstBlock \
    --config=/opt/apla/go-apla/node1/config.toml \
    --test=true
```

6. Initialize the database:

```
/opt/apla/go-apla/go-apla initDatabase \
    --config=/opt/apla/go-apla/node1/config.toml
```

Starting the first node backend

To start the first node backend, you must start two services:

- centrifugo
- go-apla

If you did not create these as services, you can just execute binary files from their directories in different consoles.

1. Run centrifugo:

```
/opt/apla/centrifugo/centrifugo \
    -a 10.10.99.1 -p 8000 \
    --config /opt/apla/centrifugo/config.json
```

2. Run go-apla:

```
/opt/apla/go-apla/go-apla start \
    --config=/opt/apla/go-apla/node1/config.toml
```

Deploying additional nodes

All other nodes (Node 2 and Node 3) are deployed like the first node with three differences:

- You do not need to generate the first block. Instead, it must be copied to the node data directory from node 1.
- The node must be configured to download blocks from node 1 via `--nodesAddr` option.
- The node must be configured to use its own addresses and ports.

Node 2

Follow this sequence of actions:

1. *Dependencies and environment setup*
2. *Creating the database*
3. *Configuring Centrifugo*
4. *Installing go-apla*
5. Create the node 2 configuration file:

```
/opt/apla/go-apla/go-apla config \
    --dataDir=/opt/apla/go-apla/node2 \
    --dbName=apladb \
    --centSecret="CENT_SECRET" --centUrl=http://10.10.99.2:8000 \
    --httpHost=10.10.99.2 \
    --httpPort=7079 \
    --tcpHost=10.10.99.2 \
    --tcpPort=7078 \
    --nodesAddr=10.10.99.1
```

6. Copy the first block file to Node 2. For example, you can do it via `scp` on Node 2:

```
scp user@10.10.99.1:/opt/apla/go-apla/node1/1block /opt/apla/go-apla/
↪node2/
```

7. Generate node 2 keys:

```
/opt/apla/go-apla/go-apla generateKeys \
    --config=/opt/apla/go-apla/node2/config.toml
```

8. Initialize the database:

```
./go-apla initDatabase --config=node2/config.toml
```

9. Run centrifugo:

```
/opt/apla/centrifugo/centrifugo \
    -a 10.10.99.2 -p 8000 \
    --config /opt/apla/centrifugo/config.json
```

10. Run go-apla:

```
/opt/apla/go-apla/go-apla start \
    --config=/opt/apla/go-apla/node2/config.toml
```

As a result, the node will download the blocks from the first node. This node is not the validating node, so it cannot generate new blocks. Node 2 will be added to the list of validating nodes later in this guide.

Node 3

Follow this sequence of actions:

1. *Dependencies and environment setup*
2. *Creating the database*
3. *Configuring Centrifugo*
4. *Installing go-apla*
5. Create the node 3 configuration file:

```
/opt/apla/go-apla/go-apla config \
    --dataDir=/opt/apla/go-apla/node3 \
    --dbName=apladb \
    --centSecret="CENT_SECRET" --centUrl=http://10.10.99.3:8000 \
    --httpHost=10.10.99.3 \
```

(continues on next page)

(continued from previous page)

```
    --httpPort=7079 \
    --tcpHost=10.10.99.3 \
    --tcpPort=7078 \
    --nodesAddr=10.10.99.1
```

6. Copy the first block file to Node 3. For example, you can do it via `scp` on Node 3:

```
scp user@10.10.99.1:/opt/apla/go-apla/node1/1block /opt/apla/go-apla/
↪node3/
```

7. Generate node 3 keys:

```
/opt/apla/go-apla/go-apla generateKeys \
    --config=/opt/apla/go-apla/node3/config.toml
```

8. Initialize the database:

```
./go-apla initDatabase --config=node3/config.toml
```

9. Run centrifugo:

```
/opt/apla/centrifugo/centrifugo \
    -a 10.10.99.3 -p 8000 \
    --config /opt/apla/centrifugo/config.json
```

10. Run go-apla:

```
/opt/apla/go-apla/go-apla start \
    --config=/opt/apla/go-apla/node3/config.toml
```

As a result, the node will download the blocks from the first node. This node is not the validating node, so it cannot generate new blocks. lients can connect to this node and it can send transactions to the network.

2.19.4 Frontend deployment

Molis client can be build by the yarn package manager only on Debian 9 (Stretch) 64-bit official distributive with **installed GNOME GUI**.

Software prerequisites

Node.js

1. Download Node.js LTS version 8.11 from the Node.js official site or via the command line:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash
```

2. Install Node.js:

```
sudo apt install -y nodejs
```

Yarn

1. Download Yarn version 1.7.0 from yarn GitHub repository or via command line:

```
cd /opt/apla \
&& wget https://github.com/yarnpkg/yarn/releases/download/v1.7.0/yarn_1.7.0_all.deb
```

2. Install Yarn:

```
sudo dpkg -i yarn_1.7.0_all.deb && rm yarn_1.7.0_all.deb
```

Building Molis App

1. Download latest release of Molis from Molis GitHub repository via git:

```
cd /opt/apla \
&& git clone https://github.com/AplaProject/apla-front.git
```

2. Install Molis dependencies via Yarn:

```
cd /opt/apla/apla-front/ \
&& yarn install
```

Adding the blockchain network configuration

1. Create settings.json file that contains connections information about full nodes:

```
cp /opt/apla/apla-front/public/settings.json.dist \
    /opt/apla/apla-front/public/public/settings.json
```

2. Edit settings.json file in any text editor and add required settings in this format:

```
http://Node_IP-address:Node_HTTP-Port
```

Example settings.json file for three nodes:

```
{
    "fullNodes": [
        "http://10.10.99.1:7079",
        "http://10.10.99.2:7079",
        "http://10.10.99.3:7079"
    ]
}
```

Building as Molis Desktop App

1. Build the desktop app with Yarn:

```
cd /opt/apla/apla-front \
&& yarn build-desktop
```

2. The desktop app must be packed to the AppImage:

```
yarn release --publish never -l
```

After that, your application will be ready to use, but its *connection settings* cannot be changed in the future. If these settings will change, you must build a new version of the application.

Building as Molis Web App

1. Build the web app:

```
cd /opt/apla/apla-front/ \
&& yarn build
```

After building, redistributable files will be placed to the '/build' directory. You can serve it with any web-server of your choice. Settings.json file must be also placed there. Note that you do not need to build your application again if your connection settings will change. Instead, edit the settings.json file and restart web-server.

2. For development or testing purposes, you can build Yarn's web-server:

```
sudo yarn global add serve \
&& serve -s build
```

After this, your Molis Web App will be available at: `http://localhost:5000`

2.19.5 Blockchain network configuration

Creating the founder's account

Create an account for the first node owner. This account is the founder of the new blockchain platform and will have administrator access rights.

1. Run Molis (frontend).
2. Import an existing account using the following data:
   - Backup payload is the node owner's private key located in the `/opt/apla/go-apla/node1/PrivateKey` file.

     Note: There are two private key files in this directory. `PrivateKey` file is for node owner's account. It is used to create the node owner's account. `NodePrivateKey` is the private key of the node itself and must be kept secret.

3. Login under this new account. Because roles haven't been created at this moment, use the *Without role* login option.

Importing apps, roles, and templates

At this moment, the blockchain platform is in the blank state. You can configure it by adding the framework of roles, templates and apps that support the basic ecosystem functions.

1. Clone the applications repository.

```
cd /opt/apla \
&& git clone https://github.com/AplaProject/apps.git
```

2. In Molis, navigate to *Developer* > *Import*.
3. Import apps in this order:
   (a) /opt/apla/apps/system.json
   (b) /opt/apla/apps/lang_res.json
   (c) /opt/apla/apps/basic.json
   (d) /opt/apla/apps/conditions.json
4. Navigate to *Admin* > *Roles* and click *Install default roles*.
5. Sign out of the system via the profile menu.
6. Log into the system under the *Admin* role.
7. Navigate to *Home* > *Votings* > *Templates list* and click *Install default templates*.

Adding first node to the list of nodes

1. Navigate to *Admin* > *Platform parameters* and click the cogwheel icon for the *full_nodes* parameter.
2. Specify the parameters for the first blockchain network node. Node's public key is located in the `/opt/apla/go-apla/node1/NodePublicKey` file. Node owner's key identifier is located in the `/opt/apla/go-apla/node1/KeyID` file.

```
{"api_address":"http://10.10.99.1:7079","key_id":"%node_owner_key_id%","public_key":"
↪%node_public_key%","tcp_address":"10.10.99.1:7078"}
```

2.19.6 Adding extra validating nodes

Adding a member to Consensus role

By default, only members of the Consensus role (Apla Consensus asbl) can participate in votings required for adding extra validating nodes. It means that a member of the ecosystem must be appointed to this role before new validating nodes can be added.

In this guide, the founder's account will be appointed as a sole member of the Consensus. In a production environment, this role must be assigned to members that perform platform governance.

1. As an ecosystem founder, navigate to *Home* > *Roles* and click the Consensus role (Apla Consensus asbl).
2. Click *Assign* and assign founder's account to this role.

Creating the node owner's account

1. Run Molis (frontend)
2. Import an existing account using the following data:
   - Backup payload is the node owner's private key located in the `/opt/apla/go-apla/node2/PrivateKey` file.
3. Login under this new account. Because a role hasn't been assigned to this account, use the *Without role* login option.
4. Navigate to *Home* > *Profile* and click on the new profile name.

5. Add account details (profile name, description, etc.)

Adding node owner to Validator role

1. As a new node owner:
   (a) Navigate to *Home* > *Candidates for the validators*.
   (b) Click *Create request* and fill the Candidates for the validators request form.
   (c) Click *Send request*.
2. As a founder:
   (a) Login under the Consensus role (Apla Consensus asbl).
   (b) Navigate to *Home* > *Candidates for the validators*.
   (c) Start the voting for by clicking the "play" icon by the candidate's request.
   (d) Navigate to *Home* > *Votings* and click *Update votings statuses*.
   (e) Click the voting name and vote for the node owner (click *Vote*).

As a result, node owner's account will be assigned the Validator role.

Adding the validating node via voting

1. As a founder, login under the Consensus role (Apla Consensus asbl).
2. Navigate to *Admin* > *Platform parameters* and click the cogwheel icon for the *full_nodes* parameter.
3. Specify the *node 2 parameters*:
   - TCP address is `10.10.99.2:7078`.
   - API address is `http://10.10.99.2:7079`.
   - Key ID of the node founder is located in the `/opt/apla/go-apla/node2/KeyID` file.
   - Node's public key is located in the `/opt/apla/go-apla/node2/NodePublicKey` file.
4. Click *Vote*.
5. Navigate to *Home* > *Votings* and click *Update votings statuses*.
6. Click the "Voting for System param value" voting and vote for the system parameter change (Click *Accept*).

As a result, a new node will be added to the list of validating nodes.