Apla Blockchain Platform Guide

Apla

авг. 13, 2019

Обзор

Глава 1

О документации

Документация содержит описание Apla — блокчейн-платформы для построения цифровых экосистем.

Глава 2

О платформе

Блокчейн-платформа Apla это безопасная, простая и гибкая инфраструктура для растущего сегмента мировой экономики.

Платформа Apla предлагает малому и среднему бизнесу:

- Снижение эксплуатационных расходов и времени выхода на рынок.
- Интегрированную систему расчетов.
- Инфраструктуру, защищенную от схем отмывания денег и фальсификации данных.
- Возможность взаимодействия в системах с низким уровнем доверия участников.
- Широкие возможности масштабирования и глобальный охват аудитории для продуктов и сервисов.

Детальное описание платформы Apla на английском языке содержится в документе Apla White Paper.

Глава 3

Contents

3.1 Общая информация

3.1.1 Отличительные особенности

Блокчейн-платформа Apla разработанна для построения цифровых экосистем на базе интегрированной среды разработки приложений с многоуровневой системой управления правами доступа к данным, интерфейсам и смарт-контрактам. Платформа построена на базе программного продукта разработанного компанией EGAAS S.A.

Apla по своей структуре и функционированию принципиально отличается от большинства существующих блокчейн-сетей.

- Написание и использование блокчейн-приложений происходит в автономных программных средах с фиксированным членством, называемых *экоситемами*.
- Деятельность в экосистемах основана на создании *реестров* и поддержке их функционирования с помощью *смарт-контрактов*, а не на обмене транзакциями/сообщениями между аккаунтами.
- Управление правами доступа к реестрам и регулирование отношений между членами экосистем осуществляется с помощью правовой системы на базе *смарт-законов*.

3.1.2 Архитектура блокчейн-платформы Apla

Сеть

Apla построена на базе одноранговой сети. Полные узлы сети содержат актуальную версию блокчейна и базу данных, в которой фиксируется текущее состояние платформы. Пользователи сети получают данные обращаясь к базам данных полных узлов с помощью программного клиента (или команд REST AP). Новые данные отправляют в сеть в виде подписанных пользователем транзакций, которые по сути являются командами модификации базы данных. Из транзакций формируются блоки, которые присоединяются к блокчейнам на узлах сети, и одновременно с этим происходит отработка транзакции - изменение состояния базы данных.

Валидирующие узлы

Полные узлы сети, имеющие право формировать блоки, называются *валидирующими*. Число валидирующих узлов ограничено и задается параметром count_of_nodes в настройках платформы.

Список валидирующих узлов хранится в параметре full_nodes в формате

```
[["host1:port","-1222","nodepub1"], ["host2:ip","-1222", "nodepub2"]], где
```

- host1:port - это адрес хоста, на который пересылаются транзакции и новые блоки, также с этого адреса можно получить всю цепочку блоков начиная с 1-го;

- -1222 - номер виртуального аккаунта, на который нода получает комиссию; если кошелек не существует, то комиссия не снимается.
- nodepub1 - публичный ключ узла, необходимый для проверки подписей блоков, созданных им.

Транзакции

Транзакция формируется программным клиентом (или командой contract REST API) и содержит данные для выполнения специального программного контролера - контракта («смарт-контракта»), вызываемого пользователем. Транзакция имеет следующий формат:

- Type - ID вызываемого контракта,
- Data - параметры, передаваемые контракту,
- KeyID - идентификатор пользователя отправившего транзакцию,
- PublicKey - публичный ключ пользователя (опционно),
- BinSignatures - подпись транзакции,
- Time - время отправления транзакции,
- EcosystemID - номер экосистемы, из которой отправлена транзакция,
- TokenEcosystem - номер экосистемы, в токенах которой оплачивается транзакция,
- MaxSum - максимальная комиссия за транзакцию,
- PayOver - дополнительная плата за ускорение в очереди.

Транзакция подписывается приватным ключом владельца аккаунта. Ключ вместе с функцией подписания может храниться: в браузере, в программном клиенте, на сим-карте, на специальном физическом устройстве. В текущей реализации приватный ключ хранится в программном клиенте Molis в зашифрованном алгоритмом AES виде. Транзакции подписываются с помощью алгоритма ECDSA.

Сетевой протокол

Транзакция отправляется пользователем на один из валидирующих узлов, где она проходит базовую проверку формата и встраивается в очередь транзакций, а также рассылается по сети другим валидирующим узлам, где она также попадает в очередь транзакций.

Узел, имеющий в данный момент право генерировать блок (согласно параметру full_nodes), извлекает транзакции из очереди и посылает в блок-генератор. Параллельно с формированием блока происходит отработка транзакций: транзакция посылается в виртуальную машину, где происходит выполнение контракта с параметрами, переданными транзакцией, в результате чего происходит модификация состояния базы данных.

Новый блок проверяется на наличие ошибок, и если он признается валидным, то рассылается другим валидирующим узлам.

Валидирующие узлы добавляют полученный блок в очередь блоков. Очередной блок после валидации присоединяется к блокчейну, а содержащиеся в нем транзакции отрабатываются, обновляя состояние базы данных.

Проверка блока и транзакций

Проверка блока, проводимая валидирующим узлом после его формирования, а также на всех других валидирующих узлах после его получения, содержит следующие тесты:

- равен ли первый байт 0, если нет, то полученные данные не являются блоком,
- время генерации блока не больше текущего,
- имел ли право узел подписавший блок сделать это в указанное в блоке время,
- номер блока больше последнего блока в имеющейся цепочке,
- не превышен ли общий лимит на оплату транзакций блока,
- проверка правильности подписи блока ключом создавшего блок узла; подписываются BlockID, Hash предыдущего блока, Time, Position в full_nodes, MrklRoot от всех транзакций блока,
- проверка правильности всех транзакций блока:
  - уникальность хеша транзакции,
  - не превышен ли лимит транзакций подписанных одним ключом (max_block_user_tx),
  - не превышен размер транзакции (max_tx_size),
  - время посылки не больше времени формирования блока и не меньше времени формирования блока минус 86400 сек,
  - правильность подписи транзакции,
  - существуют ли токены, в которых происходит оплата ресурсов в списке sys_currencies,
  - достаточно ли токенов на виртуальном аккаунте пользователя для оплаты необходимых для выполнения транзакции ресурсов.

База данных платформы

Единая база данных платформы, копии которой поддерживаются на каждом полном узле сети, используется для хранения больших объемов данных (реестров) и быстрого получения значений контрактами и интерфейсами. При формировании очередного блока и присоединении его к блокчейну на всех полных узлах платформы происходит синхронное обновление таблиц базы данных. Таким образом, база данных хранит текущее (актуальное) состояние блокчейна, что обеспечивает идентичность данных на всех полных узлах и однозначность выполнения контрактов на любом из валидирующих узлов. При запуске нового полного узла сети актуальное состояние базы данных реализуется последовательным выполнением всех транзакций записанных в блоках блокчейна.

На данный момент на платформе используется СУБД PostgreSQL.

3.1.3 Экосистемы платформы

Пространство данных Apla разбито на множество относительно самостоятельных кластеров - *экосистем*, в которых реализуется деятельность пользователей сети. Экосистемы представляют собой автономные программные среды, включающие в себя множество приложений и пользователей, создающих приложения и работающих с ними. Открыть новую экосистему может любой владелец аккаунта.

Программно экосистема представляет собой совокупность приложений - систем интерфейсов, контрактов, таблиц базы данных. На принадлежность элементов приложений к конкретной экосистеме указывает префикс в их имени, например, @1name, в котором после знака «@» указывается ID экосистемы. При обращении к элементам приложений внутри одной экосистемы префикс можно опустить.

В каждой экосистеме через программный клиент Molis доступны инструменты управления таблицами базы данных, редактор контрактов, редактор интерфейсов и другой функционал, необходимый для проектирования приложений без привлечения каких-либо дополнительных программных модулей.

Пользователем платформы можно стать только получив приватный ключ для доступа в одну из экосистем (традиционно в экосистему №1). Пользователь может быть членом любого количества экосистем. Переход между экосистемами осуществляется при помощи специального меню программного клиента.

Интегрированная среда разработки

В программном клиенте платформы Molis для создания блокчейн-приложений реализована полнофункциональная интегрированная среда разработки (IDE), работа в которой не требует от программистов специальных знаний в области блокчейн-технологий. В состав IDE входят:

- таблица параметров экосистемы,
- редактор контрактов,
- инструменты для администрирования таблиц базы данных,
- редактор интерфейсов и визуальный конструктор интерфейсов,
- редактор языковых ресурсов,
- сервис экспорта/импорта приложений.

Приложения Apla

Приложение на платформе Apla - это система таблиц, контрактов, интерфейсов с настроенными правами доступа выполняющая некоторую функцию или реализующая отдельный сервис.

Каждая экосистема для создания приложений создает собственный набор таблиц, что, однако, не исключает возможность обращения к таблицам других экосистем, с указанием их префикса. Таблицы никак не связаны с конкретными контрактами и могут использоваться всеми приложениями. Возможность записи данных в таблицы контролируется настройками прав доступа. Для управления правами могут использоваться специальные контракты - смарт-законы.

Проектирование и создание приложений не требует от программистов знаний о структуре и протоколах сети, понимания алгоритма формирования блокчейна и синхронизации баз данных полных узлов. Работа в программном клиенте Molis - создание элементов приложений, чтение данных из таблиц, запуск контрактов, отображение результата - выглядит как оперирование модулями некой программной среды, развернутой на локальном компьютере.

Таблицы экосистемы

В каждой экосистеме возможно создание неограниченного числа таблиц в базе данных платформы. Как уже отмечалось, таблицы экосистемы идентифицируются по префиксу, содержащему номер экосистемы, который не отражается в программном клиенте при работе «внутри» экосистемы. Запись в таблицы других экосистем возможна, если позволяют настройки прав доступа.

Инструменты для администрирования таблиц

Инструменты управление таблицами экосистемы доступны в разделе Tables административной секции программного клиента Molis, где реализованы следующие функции:

- просмотр списка таблиц и их содержимого,
- создание новых таблиц,

- добавление в таблицы новых колонок с выбором типовых форматов данных: Text, Date/Time, Varchar, Character, JSON, Number, Money, Double, Binary,
- установление правами доступа на запись данных и изменение структуры таблиц.

Операции с данными таблиц

Для работы с базой данных язык контрактов Simvolio и язык шаблонизатора Protypo содержат функции DBFind, обеспечивающие получение из таблиц как отдельных значений, так и массивов. Язык контрактов содержит функции добавления строк в таблицы DBInsert и изменения значений в существующих записях DBUpdate (при изменении значения переписываются только данные в таблице базы данных, в блокчейн же добавляется новая транзакция с сохранением всех предыдущих транзакций). Данные в таблицах не удаляются.

С целью минимизации времени выполнения контрактов в функциях DBFind не реализовано обращение сразу к нескольким таблицам, то есть не поддерживаются запросы с JOIN. Поэтому целесообразно отказаться от нормализации таблиц приложений и записывать в строки таблиц полную информацию, дублирующую данные в других таблицах. Однако, это не просто вынужденная мера, а необходимое требование к блокчен-приложениям, в которых сохраняться (подписываться приватным ключом) должен некий полный, законченный, актуальный на определенный момент времени набор данных (документ), который не может быть модифицирован вследствие изменения значений в других таблицах (что неизбежно в реляционной схеме).

Параметры экосистем

В разделе Ecosystem parameters административной секции программного клиента Molis доступны для просмотра и редактирования параметры экосистемы, которые можно разделить на несколько групп:

- общие параметры: название экосистемы (ecosystem_name), описание (ecosystem_description), аккаунт основателя (founder_account) и некоторые другие,
- параметры доступа, которые определяют исключительные права доступа к элементам приложений (changing_tables, changing_contracts, changing_page, changing_menu, changing_signature, changing_language),
- технические параметры: например, пользовательские стили (stylesheet),
- пользовательские параметры экосистемы, в которых хранятся константы или списки (через запятую), необходимые для работы приложений.

Для каждого параметра экосистема указываются права на его изменения.

Для получения значений отдельных параметров экосистемы и в языке контрактов Simvolio, и в языке шаблонизатора Protypo имеется функция EcosysParam, в которой в качестве аргумента указывается имя параметра. Для возврата элемента списка (записанных в параметр экосистемы через запятую) необходимо вторым параметром функции указать его порядковый номер.

Параметры платформенной экосистемы

Все параметры платформы хранятся в таблице параметров платформенной экосистемы. Это такие параметры как:

- промежуток времени, отведенный на создание блока валидирующим узлом,
- код исходных страниц, контрактов, таблиц, меню новых экосистем,
- список валидирующих узлов,

- максимальные размеры транзакции, блока, максимальное число транзакций в блоке,
- максимальное количество транзакций от одного аккаунта в блоке,
- максимальное количество Fuel расходуемое на одну транзакцию, один блок,
- курс Fuel к APL и другие.

Управление параметрами платформенной экосистемы с программной точки зрения ничем не отличается от управления параметрами обычных экосистем. В отличие от обычных экосистем, при создании которых все права по управлению параметрами принадлежать основателю экосистемы, права по изменению параметров в платформенной экосистеме возможны только через контракт UpdSysContract, управление которым прописано в правовой системе платформы.

Список параметров платформенной экосистемы

- **default_ecosystem_page** - какой код по умолчанию вставить в первую страницу только что созданной экосистемы.
- **default_ecosystem_menu** - какой под по умолчанию вставить в первое меню только что созданной экосистемы.
- **gap_between_blocks** - множитель, на который умножается количество секунд, которые нода ждет, пока не получает право сгенерить следующий блок. *0 < gap_between_blocks < 86400*.
- **rb_blocks_1** - максимальное количество блоков, на которое можно откатиться. Влияет на количество блоков которое можно за раз скачать. *0 < rb_blocks1 < 10000*.
- **new_version_url** - хост для проверки доступности новых версий (апдейт сервера).
- **number_of_nodes** - максимальное количество полных нод. *0 < number_of_nodes < 1000*.
- **max_block_size** - максимальный размер блока в байтах. *max_block_size > 0*.
- **max_tx_size** - максимальный размер транзакции в байтах. *max_tx_size > 0*.
- **max_tx_count** - максимальное количество транзакции в блоке. *max_tx_count > 0*.
- **max_columns** - максимальное количество колонок в созданном пользователем реестре. *max_columns > 0*.
- **max_indexes** - максимальное количество индексов в созданном пользователем реестре. *max_indexes > 0*.
- **max_block_user_tx** - максимальное количество транзакции в блоке от одного пользователя. *max_block_user_tx > 0*.
- **max_block_generation_time** - максимальное время в миллисекундах на генерацию блока.
- **max_fuel_tx** - максимальный расход топлива на одну транзакцию. *max_fuel_tx > 0*.
- **max_fuel_block** - максимальный расход топлива на блок. *max_fuel_block > 0*.
- **size_fuel** - множитель, на который умножается плата за количество данных в смарт контракте.
- **commission_wallet** - адреса кошельков, на которые начисляется комиссия в зависимости от текущей экосистемы. Представляет из себя массив из пар (номер экосистемы, идентификатор кошелька). Например, *[[«1»,»-9436047199451325 08»]]*. При записи идет проверка на валидность номеров кошельков.
- **commission_size** - процент коммиссии, который будет начисляется с каждой операции на commission_wallet. *commission_size >= 0*.

- **fuel_rate** - курс конверсии APL к топливу. Представляет из себя массив из пар (номер экосистемы, множитель). Например, *[[«1»,»1000000000000000»]]*. Значение множителя должно быть больше 0.
- **full_nodes** - список нод, которые могут генерировать блоки. Представляет собой список списков вида [[хост, адрес аккаунта, публичный ключ],].
- **extend_cost_(funcname)** - стоимость вызова встроенной функции в fuel. *x_extend_cost >= 0*.
- **(table|column|page|menu|contract)_price** - стоимость создания какой либо сущности в fuel. *x_price >= 0*.

3.1.4 Механизм контроля прав доступа

Платформа обладает многоуровневой системой управления правами доступа. Условия доступа устанавливаются на операции создания и изменения всех элементов приложений: контрактов, таблиц базы данных, интерфейсов, параметров экосистемы. Также фиксируются и права на изменения прав.

По умолчанию все права на изменение всех элементов приложений экосистемы принадлежит ее основателю (что прописано в контракте MainCondition, который имеется в экосистеме по умолчанию). Однако после создания специальных смарт-законов, контроль прав может быть передан членам экосистемы или их группе.

Контролируемые операции

Права устанавливаются в поле Permissions в соответствующих разделах административной секции программного клиента Molis: в редакторах контрактов, таблиц, интерфейсов (страниц, меню, страничных блоков). Фиксируются права на следующие операции:

1. Table column permission - право на изменение значения в колонке таблицы,
2. Table Insert permission - право на запись в таблицу новой строки,
3. Table New Column permission - право на добавление новой колонки,
4. Conditions for changing of Table permissions - право на изменение прав, перечисленных в п.п. 1-3,
5. Conditions for change smart contract - право на изменение контракта,
6. Conditions for change page - право на изменение страницы интерфейса,
7. Conditions for change menu - право на изменение меню,
8. Conditions for change of ecosystem parameters - права на изменение определенного параметра настроечной таблицы экосистемы.

Способы управления правами

Правила, задающие права доступа, записываются в поля *Permissions* в виде произвольного выражения на языке Simvolio. Доступ предоставляется если на момент обращения выражение имеет значение true. Если поле *Permissions* остается пустым, то оно автоматом приобретает значение *false*, и выполнение соответствующих действий запрещается.

Простейшим способом предоставления прав является запись в поле *Permissions* логического выражения, например, $member == 2263109859890200332, в котором указан идентификационный номер конкретного члена экосистемы.

Универсальным и рекомендуемым методом фиксации прав является использование функции *ContractConditions*, которой в качестве аргумента передается имя контракта, содержащего условия, в которых могут использоваться данные таблиц (например, таблицы ролей) и параметры экосистемы.

Еще одним методом управления правами доступа является использование функции ContractAccess, которой в качестве параметров передается список контрактов, имеющих право реализовывать соответствующее действие. К примеру, если в таблице, содержащей аккаунты в токенах экосистемы, ввести в поле *Permissions* колонки amount функцию `ContractAccess("TokenTransfer")`, то изменение значения amount будет разрешено исключительно контракту *TokenTransfer* (все контракты, предусматривающие перевод токенов с аккаунта на аккаунт, смогут сделать это только через вызов контракта *TokenTransfer*). Условия получения доступа к самим контрактам контролируются в секции conditions и могут быть достаточно сложными, включающими множество других контрактов.

Исключительные права

Для разрешения конфликтных или опасных для деятельности экосистемы ситуаций в таблице Ecosistem parameters введены специальные параметры (*changing_smart_contracts, changing_tables, changing_pages*), в которых прописываются условия получения исключительных прав доступа к любым смарт-контрактам, таблицам или страницам. Эти права устанавливаются специальными смарт-законами, к примеру, предусматривающими голосование членов экосистемы или наличие нескольких подписей различных ролей.

3.1.5 Оффчейн-серверы

На платформе существует возможность создания оффчейн-серверов (OBS), обладающих полным функционалом обычных экосистем, но работающих вне блокчейна. В OBS можно создавать полноценные приложения с использованием языка контрактов и языка шаблонизатора, таблиц базы данных и другого функционала программного клиента. При этом через API возможно вызвать контракты из блокчейн-экосистем.

Обращение к web-ресурсам

Основным отличием OBS от обычных экосистем является возможность обращения из ее контрактов к любым web-ресурсам по HTTP/HTTPS. Для этого используется функция HTTPRequest, в которую передаются URL, метод запроса (GET или POST), заголовок и параметры запроса.

Права на чтение данных

Поскольку данные OBS не записываются в блокчейн (который доступен для чтения), то в них реализована возможность установления права на чтение таблиц. Права на чтение определяются как для отдельных колонок, так и для любых строк с помощью специального контракта.

Использование OBS

OBS можно использовать для создания регистрационных формы с отправкой пользователям на почту или телефон проверочной информации, хранения данных вне публичного доступа, для написания и тестирования работы приложений с последующим их экспортом и импортом в блокчейн экосистемы. Также в OBS есть возможность настроить запуск контрактов по таймеру, что позволяет создавать оракулы для получения web-данных и посылки их в блокчейн.

Создание OBS

OBS создается на любом из полных узлов сети. Администратор узла должен определить список экосистем, которым разрешено пользоваться функционалом оффчейн-серверов, а так же указать пользователя, который будет обладать правами основателя экосистемы: сможет устанавливать приложения, принимать в экосистему новых членов, настраивать права доступа к ресурсам экосистемы.

3.2 FAQ

1. Как бы вы кратко описали платформу?

- Блокчейн платформа, спроектированная для построения цифровых экосистем на базе интегрированной среды разработки приложений с многоуровневой системой управления правами доступа к данным, интерфейсам и смарт-контрактам.

2. Базируется платформа на блокчейне Bitcoin, Ethereum или каком-то другом?

- Платформа построена на основе оригинального блокчейна.

3. Чем отличается платформа от других публичных блокчейн платформ, у которых есть встроенный механизм исполнения смарт-контрактов, таких как Ethereum, Qtum и еще только проектируемых Tezos и EOS?

- платформа обладает функционалом, отсутствующим у перечисленных блокчейн-сетей, это:
- интегрированная среда разработки приложений реализованная на едином программном клиенте;
- специальный язык шаблонизатора для написания интерфейсов, согласованный с языком написания контрактов;
- многоуровневая система управления правами доступа к данным, контрактам, и интерфейсам с предоставлением прав персонам, ролям, контрактам;
- экосистемы - автономные программные среды для создания блокчейн-приложений и пользовательской работы с ними;
- правовая система - свод нормативов, зафиксированных в смарт-законах (специальных смарт-контрактах), которые регулируют отношения между пользователями платформы, устанавливают процедуры изменения параметров протоколов и используются для разрешения проблемных ситуаций.

4. Есть ли у платформы собственная криптовалюта?

- Да, Apla использует свои собственные токены, APL.

5. Что такое валидирующий узел?

- Валидирующим называется узел сети имеющий право проверять транзакции и создавать блоки.

6. Каждый ли узел сети может быть валидирующим?

- Нет в сети может быть фиксированное количество валидирующих узлов.

7. Кто может поддерживать валидирующий узел?

- любой узел сети с достаточной вычислительной мощностью и отказоустойчивостью может претендовать на роль валидирующего. За право узла быть валидирующими голосуют экосистемы, но не все, а только утвержденные инвесторами (владельцами токенов платформы) в качестве реально функционирующих. То есть на платформе реализован новый алгоритм консенсуса delegited Proof of Value of ecosystems (DPoV(E)). При такой схеме наиболее вероятно, что поддерживать

валидирующие узлы будут крупные экосистемы, как максимально заинтересованные в работоспособности сети.

8. Что такое экосистемы платформы?

- Экосистемы - это практически автономные программные среды для создания блокчейн-приложений и пользовательской работы с ними.

9. Кто может создать экосистему?

- Любой пользователь платформы может открыть новую экосистему.

10. Как можно стать членом экосистемы?

- Регистрация в сети происходит в одну из существующих экосистем; вариантов приема в членство может быть множество и они определяются политикой экосистем: от предоставления информации об экосистеме в специальном каталоге, до рассылки публичных ключей.

11. Возможно ли создать несколько экосистем одному пользователю?

- Да, каждый может создать любое количество экосистем, а также являться членом многих экосистем одновременно.

12. Что такое приложение платформы?

- Приложение - это целостный программный продукт, реализующий некоторую функцию или сервис. Структурно приложения состоят из таблиц базы данных, контрактов и интерфейсов.

13. Какой язык программирования используется для написания приложений?

- Контракты пишутся на языке Simvolio, написанном командой платформы (см. описание языка контрактов).
- Для написания интерфейсов используется оригинальный язык шаблонизатора Protypo (см. описание языка шаблонизатора).

14. Какое программное обеспечение необходимо для написания приложений и работы пользователей с ними?

- Приложения пишутся и выполняются в едином программном клиенте Molis; никакого другого программного обеспечения не требуется.

15. Могут ли контракты плафтформы обращаться к данным с использованием сторонних API-интерфейсов?

- Нет, контракты экосистем непосредственно не могут обращаться только к данным, размещенным в блокчейне. Для получения данных извне платформы используются специальные оффчейн-серверы.

16. Возможно ли изменить сохраненный в блокчейне контракт?

- Да, контракты редактируются. Права на редактирование контрактов определяются его создателем: имеется возможность как полностью запретить изменение контракта, так и предоставить право редактировать контракт конкретной персоне или задать сложный набор условий в специальном смарт-законе.
- Программный клиент Molis предоставляет доступ ко всем версиям контрактов.

17. Что такое смарт-закон?

- Смарт-закон - это контракт, созданный специально для контроля и ограничения работы обычных контрактов, а через них и действий членов экосистемы. Множество смарт-законов можно рассматривать как «правовую систему» экосистемы.

18. Может ли контракт вызвать другой контракт?

- Да, такая возможность есть, как путем прямого указания контракта с передачей ему необходимых параметров, так вызовом контракта по ссылке (имени) (см. описание языка контрактов).

19. Нужен ли для работы приложений мастер-контракт?

- Нет. Контракты представляют собой автономные программные модули, выполняющие некоторую фиксированную функцию. В каждом контракте определены данные, которые он должен получить, условия проверки этих данных, и выполняемое действие - транзакция в базу данных.

20. Можно ли писать приложения с локализацией интерфейсов?

- Да, программный клиент содержит механизм поддержки локализации, позволяющий создавать интерфейсы на любых языках.

21. Возможно ли создать собственные интерфейсы без применения языка шаблонизатора Protypo?

- Да, для этого можно воспользоваться REST API платформы.

22. Сохраняются ли интерфейсные страницы в блокчейне?

- Да, страницы, как и контракты, хранятся в блокчейне, что защищает их от фальсификации.

23. Какое хранилище данных используется для работы контрактов?

- Программный клиент Molis содержит инструменты для создания таблиц базы данных (сейчас используется PostgreSQL, но возможны изменения), а в языке программирования контрактов Simvolio есть все необходимые функции для записи/чтения данных, язык шаблонизатора Protypo содержит функции для чтения данных из таблиц.

24. Как контролируется доступ к данным в таблицах?

- Права на добавление колонок, на вставку строк, на редактирование данных в колонке могут быть предоставлены как членам экосистемы или ролям, так и конкретным контрактам (с запретом другим контрактам производить указанные операции).

25. Могут ли приложения одной экосистемы обмениваться данными с приложениями другой экосистемы?

- Да, обмен данными можно организовать через глобальные (доступные для всех экосистем) таблицы.

26. Нужно ли все приложения в новой экосистеме писать с нуля?

- Нет, в новой экосистеме доступен ряд приложений из коробки: механизм управления членами и ролями экосистемы, приложение для настройки и эмиссии токенов, система голосования, социальная новостная система с поощрением активности, мессенджер для общения членов экосистемы; приложения можно отредактировать и настроить под специфику экосистемы.

27. Нужно ли платить за работу приложений?

- Да, использование ресурсов валидирующих узлов оплачиваются в токенах плафтормы.

28. Кто оплачивает работу приложений?

- Аккаунт (привязанный аккаунт), с которого переводятся токены для оплаты ресурсов, определяется создателем контракта и может быть изменен в любой момент. Будут или нет члены экосистемы платить за работу с приложениями и если да, то какая будет форма этой оплаты (через взносы или иначе), задается с помощью смарт-законов экосистемы.

29. Как защищены приложения экосистем от неправомерного использования уязвимостей?

- Команда создателей платформы, понимая, что полностью избежать ошибок в программном коде приложений невозможно, тем более, когда приложения могут писаться любыми пользователями, приняла решение создать механизм устранения последствий ошибок. Платформа содержит правовую систему (ряд смарт-законов), позволяющих остановить работу атакуемого приложения и

произвести ряд транзакций восстанавливающих статус-кво. В смарт-законах правовой системы прописаны права на запуск таких контрактов и процедуры голосования для предоставления этих прав.

30. Какие новые функции будут реализованы на платформе в будущем?

- Визуальный редактор интерфейсов,
- Графический редактора смарт-контрактов,
- Поддержка гибридной (SQL и NoSQL) базы данных,
- Параллельная, во множество потоков обработка транзакций от разных экосистем.
- Хостинг экосистем и биржа вычислительных мощностей.
- Неполные узлы, хранящие на сервере только часть блоков.
- Семантический справочник (онтология) для унификации операций с данными в пределах всей платформы.

31. Есть ли подтверждение работоспособности платформы?

- За последние месяцы на платформе было реализовано несколько подтвержденных proof of concept: система опроса и голосования для одной из партий (Нидерланды), регистрация нового бизнеса (OAE), торговля финансовыми инструментами (Люксембург), земельный реестр (Индия), система управления документами и контрактами (OAE).

32. Есть ли явные минусы у платформы?

- Самым большим минусом платформы, скажем, по сравнению с Ethereum, является то, что она только запускается. Но время этот минус непременно превратит в большой плюс

33. Каким вам видится будущее платормы?

- Платформа проектировалась исходя из понимания, что полноценный эффект от использования блокчейн-технологии может быть достигнуть только при переносе всех видов деятельности, всех реестров, всех контрактов на один блокчейн. Как не может быть множество интернетов, так, в конечном итоге, не может сосуществовать и множество блокчейн-сетей. И платформа видится именно как таковая единая система, на которую в будущем должны перевести свою деятельность все государства мира.

3.3 Термины и определения

3.3.1 Общие термины блокчейн-технологии

- *блокчейн* - информационная система, обеспечивающая хранение данных с защитой их от фальсификации и потери, а также передачу и преобразование данных внутри системы с сохранением их достоверности; защита данных достигается (1) записью их в цепочку криптографически связанных блоков, (2) децентрализованным хранением копий цепочек блоков на узлах одноранговой сети и (3) синхронизацией цепочек блоков на всех полных узлах с помощью алгоритма консенсуса; сохранение достоверности данных при операциях с ними внутри сети обеспечивается хранением алгоритмов передачи и преобразования данных (контрактов) в блокчейне; блокчейном называется также и сама цепочка блоков.
- *одноранговая сеть* - компьютерная сеть с равноправными узлами (без центрального сервера).
- *блок* - группа транзакций, объединенных валидирующим узлом в специальную структуру после проверки их формата и подписей; блок содержит хеш предыдущего блока, что являчется одной

из мер криптографической защиты блокчейна; блок добавляется к блокчейну при достижении консенсуса с другими валидирующими узлами сети.

- *хеш* - однозначно воспроизводимый криптографическое представление файла или иного набора цифровых данных; позволяет контролировать неизменность данных - любая модификация данных ведет к изменению хеша.
- *валидация блока* - проверка правильности структуры блока, времени его создания, совместимости его с предыдущим блоком, а так же подписей транзакций и соответствия транзакций данным блокчейна.
- *валидирующий узел* - узел сети имеющий право создавать и проверять блоки.
- *консенсус* - соглашение между валидирующими узлами сети о процедуре присоединения новых блоков к блокчейну или алгоритм этого соглашения.
- *транзакция* - единичная операция передачи данных в блокчейне или запись об этой операции в блокчейне.
- *токен* - учетная единица некоторого объема прав, фиксируемая в виде идентифицируемых числовых записей в реестре, содержащем механизм обмена долями прав между этими записями.
- *идентификация* - криптографическая процедура распознавания пользователя в системе.
- *уникальная идентификация* - процедура сопоставления пользователя с конкретной персоной; требует проведения правовых и организационных мероприятий для реализации биометрического или иного механизма сопоставления персоны и пользователя.
- *приватный (секретный) ключ* - строка символов, сохраняемая в тайне ее владельцем, с помощью которой осуществляется доступ к аккаунту (кошельку) в сети и подписывание транзакций.
- *публичный (открытый) ключ* - строка символов, с помощью которой можно проверить подлинность подписи, сделанной приватным ключом; публичный ключ однозначно рассчитывается из значения приватного, но значение публичного ключа не дает возможность получить приватный ключ.
- *цифровая подпись* - атрибут цифрового документа или сообщения, полученный с помощью криптографической обработки данных; цифровая подпись позволяет проверить целостность (отсутствие изменений) и подлинность документа (установить его авторство).
- *контракт* - программа, выполняющая операции с данными, записанными в блокчейне; контракты хранятся в блокчейне.
- *комиссия за транзакцию* - плата валидирующему узлу за выполнение транзакции.
- *двойное расходование* - атака на блокчейн сеть с целью использования одни и тех же токенов для произведения двух транзакций, осуществляется путем формирования и поддержки вилки (двух ветвей) блокчейна; возможна только при контролировании более 50% валидирующей мощности сети.
- *шифрование* - преобразование цифровых данных в форму исключающую их чтение без обладания ключом дешифрования.
- *приватный блокчейн* - блокчейн сеть, все узлы которой и доступ к данным контролируются централизовано (государством, корпорацией, частным лицом).
- *публичный блокчейн* - блокчейн сеть не контролируемая какой-либо организацией, все решения принимаются консенсусом участников сети, данные находятся в свободном доступе.
- *delegated proof of stake (DPoS)* - алгоритм достижения консенсуса в блокчейн сети, при котором валидирующие узлы выбираются делегатами, обычно, владельцами токенов сети, которые голосут своими долями.

3.3.2 Термины платформы

- *testnet* - версия сети, используемая для тестирования ПО.
- *mainnet* - основная версия сети.
- *токен платформы* - токены платформы, в которых производится плата за использование ресурсов сети (комиссии).
- *транзакция платформы* - команды вызова контракта, содержащая передаваемые ему параметры; результатом выполения транзакции узлом является обносление базы данных платформы.
- *fuel* - условная единица, в которой зафиксирован размер комиссии за выполнение определенных операций; курс fuel относительно токенов платформы, определятся голосованием валидирующих узлов.
- *аккаунт* - условное хранилище токенов, доступ к которому контролируется парой ключей - приватным и публичным.
- *адрес* - символьное обозначение идентификатора пользователя сети, трактуемое как имя его аккаунта.
- *привязанный аккаунт* - аккаунт, с которого производится оплата за выполнение контракта, привязка выполняется при создании контракта и может быть изменена в любой момент; по умолчанию (до привязки) оплата идет с аккаунта пользователя, запустившего контракт.
- *Molis* - программный клиент, используемый для подключения к сети; Molis обеспечивает функционирование виртуального аккаунта, построение экосистем и создание приложений в интегрированной среде разработки (создание, редактирование таблиц, интерфейсных страниц и контрактов).
- *web-Molis* - полнофункциональный программный клиент реализованный в виде веб-приложения;
- *экосистемы платформы* - относительно замкнутые программные среды, включающие в себя множество приложений и пользователей, создающих их и работающих с ними; в экосистеме может быть инициирована эмиссия собственного токена, установлены правила взаимоотношений членов и условия их доступа к элементам экосистемы с помощью системы смарт-законов.
- *параметры экосистемы* - настраевыемые атрибуты экосистемы (название, описание, логотип, название токена и параметры его эмиссии и др.); хранятся и редактируются в специальной таблице настроек.
- *члены экосистемы* - пользователи, имеющие доступа к приложениям определенной экосистеме.
- *оффчейн-сервер* (OBS) - узел, обладающий полным функционалом, но работающий вне блокчейна (не сохраняющих данные в блокчейне платформы); на оффчейн-серверах контракты могут обращаться к любым web-ресурсам по HTTP/HTTPS, а также могут настраиваться права на чтение данных.
- *delegited Proof of Value of Ecosystem (DPoV(E))* - алгоритм консенсуса в сети по умолчанию, при котором валидирующие узлы утверждаются голосованием значимых, эффективно работающих на платформе экосистем (значимые экосистемы), как наиболее заинтересованных в поддержке работоспособности сети; утверждение экосистем достигших фиксированных показателей (число транзакций, число членов) как значимых производится голосованием держателями токенов (для предотвращения допуска к утверждению валидаторов злонамеренно сгенеренных экосистем).
- *Simvolio* - скриптовый язык написания контрактов; Simvolio содержит функции для обработки данных получаемых от страниц интерфейса и функции оперирования значениями таблиц базы данных; контракты создаются и редактируются в редакторе программного клиента Molis.

- *Protypo* - языка шаблонизатора, содержит функции необходимые для получения значений из таблиц базы данных, операторы для формирования страниц интерфейса и отправки пользовательских данных в контракты.
- *интегрированная среда разработки* - комплекс программных средств для создания приложений; интегрированная среда разработки программного клиента Molis содержит редактор контрактов, редактор страниц, инструменты работы с таблицами базы данных, редактор языковых ресурсов, функционал экспорта и импорта приложений; развитие среды идет в сторону создания визуальных редакторов с использованием семантических инструментов.
- *конструктор интерфейсов* - инструмент программного клиента Molis для создания интерфейса страниц приложений путем манипулирования элементами (html-контейнерами, полями форм, кнопками и пр.) непосредственно на экране.
- *визуальный редактор интерфейсов* - инструмент программного клиента Molis для создания станиц приложений, включает в себя конструктор интерфейсов и функционал для генерации кода страницы на языке Protypo.
- *визуальный редактор контрактов* - инструмент программного клиента Molis для создания контрактов в графическом интерфейсе.
- *языковые ресурсы* - модуль программного клиента Molis выполняющий локализацию интерфейса приложений - связывает лейбл, встроенный в страницу приложения, с текстовым значением на выбранном языке.
- *экспорт приложения* - сохранение кода приложений (множества таблиц, страниц и контрактов) в виде отдельного файла.
- *импорт приложения* - загрузка приложения из экспортированного файла в экосистему из файла экспорта.
- *смарт-закон* - запись в блокчейне, содержащая нормативную информацию, используемую для контроля работы контрактов и управления правами доступа к реестрам; выполняются смарт-законы, специальными контрактами.
- *правовая система* - свод нормативов, зафиксированных в смарт-законах; правовая система регулирует отношения между пользователями платформы, устанавливает процедуры изменения параметров протоколов и содержит механизмы разрешения проблемных ситуаций.
- *приложение платформы* - функционально целостный программный продукт написанный в интегрированной среде разработки клиента Molis, приложение состоит из таблиц базы данных, контрактов и интерфейсных страниц.
- *страница интерфейса приложения* - программный код, написанный на языке шаблонизатора Protypo, формирующий экранный интерфейс.
- *страничный блок* - программный код, написанный на языке шаблонизатора Protypo, встраиваемый в страницы приложения.
- *привязка контракта* - связывание контракта с виртуальным аккаунтом, с которого будет сниматься комиссия за выполнение операций контракта.
- *права доступа* - условия получения доступа к созданию и редактированию таблиц, контрактов и страниц; права доступа к таблицам устанавливаются на чтение и редактирование строк, столбцов, а также на запись новых строк;
- *полный узел* - узел сети платформы, содержащий полную актуальную версию блокчейна.
- *неполный узел* - узел сети платформы, содержащий только блоки с данными одной экосистемы.
- *параллельная обработка транзакций* - метод повышения скорости обработки транзакций, основанный на одновременной обработке данных от разных экосистем.

3.4 Смарт-контракты

Смарт-контракт (далее просто «контракт») - это базовый элемент приложений, с помощью которого реализуется выполнение единичного действия (обычно записи в таблицу базы данных), инициированного в интерфейсе пользователем или другим контрактом. Все операции с данными в приложениях оформляются в виде системы контрактов, взаимодействующих через таблицы базы данных или путем вызова друг с друга в теле контракта.

Контракты пишутся на оригинальном (написаном разработчиками платформы) тьюринг-полном скриптовом языке Simvolio с компиляцией в байт-код. Язык содержит необходимый набор функций, операторов и конструкций для реализации алгоритмов обработки данных и операций со значениями.

Контракты могут редактироваться (если при его создании эта возможность не была запрещена указанием `false` в правах доступа к редактированию контракта). Операции с данными блокчейна выполняет актуальная (последняя по времени) версия контракта. Вся история изменений контрактов сохраняется в блокчейне и доступна в программном клиенте.

3.4.1 Структура контракта

Контракт определяется ключевым словом contract, после которого указывается имя контракта. Тело контракта заключается в фигурные скобки. Контракт состоит из трех секций:

1. **data** - используется для описания входящих данных (имена переменных и их типы),
2. **conditions** - реализует проверку входных данных на корректность,
3. **action** - содержит описание действия контракта.

Структура контракта:

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

Секция data

Входные данные контракта, а так же параметры формы для приема этих данных описываются в секции `data`. Данные перечисляются построчно: сначала указывается имя переменной (передаются только переменные, а не массивы), затем тип и опционально через пробел в двойных кавычках параметры для построения формы интерфейса:

- *optional* - элемент формы без обязательного заполнения.

```
contract my {
  data {
      Name string
      RequestId int
      Photo file "optional"
      Amount money
      Private bytes
  }
  ...
}
```

Секция conditions

в секции реализуется проверка входных данных с выводом сообщений об ошибках с помощью команд: `error`, `warning`, `info`. Все эти команды генерируют ошибку, останавливающую работу контракта, но

выводят в интерфейсе различные сообщения: *критическая ошибка*, *предупреждение*, и *информативная ошибка*. Например,

```
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 been already registered"
}
```

Секция action

Секция action содержит основной программный код контракта выполняющий получение дополнительных данных и запись результата в таблицы базы данных. Например,

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

Кроме секции action контракт может содержать функцию **price**. Эта функция нужна для установления дополнительной стоимости в единицах топлива при выполнении контракта. Она может возвращать как число типа *int*, так и типа *money*. Возвращенное значение из функции price будет добавлено к стоимости выполнения контракта и умножено на коэффициент **fuel_rate**.

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

Переменные в контракте

Входные данные контракта, описанные в секции data, передаются в другие секции через переменные с именами данных и с символом $ перед ними. Возможно определить и дополнительные переменные со знаком $, которые будут глобальными в рамках выполнения контракта, включая вложенные контракты.

В контракте доступны и предопределенные переменные, содержащие данные о транзакции, из которой был вызван данный контракт.

- `$time` - время транзакции int,
- `$ecosystem_id` - идентификатор экосистемы int,
- `$block` - номер блока, в который запечатана транзакция int,
- `$key_id` - адрес кошелька подписавшего транзакцию, если контракт вне экосистемы с `ecosystem_id == 0`,

- `$block_key_id` - адрес ноды, сформировавшей блок, в который входит транзакция,
- `$block_time` - время формирования блока, который содержит транзакцию с текущим контрактом.
- `$original_contract` - имя контракта, который был изначально вызван при обработке транзакции. Если эта переменная равна пустой строке, то значит контракт вызван при проверке какого-то условия. Чтобы проверить, вызвался ли данный контракт другим контрактом или напрямую из транзакции, следует сравнить **$original_contract** и **$this_contract**. Если они равны, то значит контракт был вызван из транзакции.
- `$this_contract` - имя текущего выполняемого контракта.
- `$guest_key` - идентификатор гостевого кошелька.
- `$stack` - стэк вызовов контрактов. Имеет тип *array* и содержит строки с именами вызванных контрактов. Нулевой элемент массива - это текущий выполняемый контракт, последний элемент массива - это имя оригинального контракта вызванного при обработке транзакции.

Предопределенные переменные доступны не только в контрактах, но и в полях Permissions, в которых указываются условия доступа к элементам приложений (таблицам, контрактам, страницам и пр.) для составления логических выражений. При этом переменные имеющие отношения к формированию блока (`$time`, `$block` и др.) имеют нулевое значение.

Предопределенная переменная `$result` используется при необходимости возвратить значение из вложенного контракта.

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

3.4.2 Вложенные контракты

В секциях *conditions* и *action* контракта может быть вызван другой контракт с передачей ему данных из текущего контракта. Вызов вложенного контракта возможен как непосредственно, с указанием параметров в скобках после его имени (`NameContract(Params)`), так и с помощью функции *CallContract*, для которой имя контракта передается через строковую переменную.

3.4.3 Загрузка файлов

Для загрузки файлов из `multipart/form-data` форм, требуется использовать поля контрактов с типом `file`. Пример:

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

Для загрузки и хранения файлов предусмотрен системный контрак *UploadBinary*. Для получения ссылки на скачивание файла из шаблонизатора, предусмотрена функция шаблонизатора *Binary*.

3.4.4 Редактор контрактов

Контракты создаются и редактируются в специальном редакторе программного клиента Molis. При создании нового контракта в него уже вписана типовая структура с тремя секциями `data, conditions, action`. Редактор контрактов обеспечивает:

- написание кода контрактов (с подсветкой ключевых слов языка Simvolio),
- форматирование кода контракта,
- привязку контракта к виртуальному аккаунту, с которого будет происходить оплата его работы,
- задание прав на редактирование контракта,
- просмотр истории изменения контрактов с возможностью восстановления прежних версий.

3.4.5 Язык написания контрактов Simvolio

Язык написания контрактов Simvolio обеспечивает:

- объявление переменных с различными типами значений, а также простых и ассоциативных массивов: `var, array, map`,
- использование условной конструкции `if` и конструкции цикла `while`,
- получение значений из базы данных и запись значений в базу данных `DBFind, DBInsert, DBUpdate`,
- работу с контрактами,
- преобразование значений переменных,
- операции со строковыми значениями.

Базовые элементы и конструкции языка

Типы значений и переменные

Переменные языка объявляются с указанием типа значения. В очевидных случаях применяется автоматическое преобразование типов. Используются следующие типы значений:

- `bool` - булевый, принимает значения true или false;
- `bytes` - последовательность байтов;
- `int` - 64-разрядное целое число;
- `array` - массив значений с произвольными типами;

- `map` - ассоциативный массив значений с произвольными типами со строковыми ключами;
- `money` - целое число типа big integer; значения хранятся в базе данных без десятичных точек, которые вставляется при выводе в интерфейсе в соответствии с настройками валюты;
- `float` - 64-разрядное число с плавающей точкой;
- `string` - строка; указываются в двойных или обратных кавычках - «This is a line» или *This is a line*;
- `file` - ассоциативный массив с определенным набором ключей и значений:
  - `Name` - название файла, тип string
  - `MimeType` - mime-тип файла, тип string
  - `Body` - содержимое файла, тип bytes

Все идентификаторы - имена переменных, функций, контрактов и пр. - регистрозависимы (MyFunc и myFunc - это разные имена).

Переменные объявляются с помощью ключевого слова **var**, после которого указывается имя или имена переменных и их тип. Переменные определяются и действуют внутри фигурных скобок. При описании переменных им автоматически присваивается значение по умолчанию: для типа *bool* это *false*, для всех числовых типов - нулевые значения, для строк - пустая строка. Примеры объявления переменных:

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

Массивы

Язык поддерживает два типа массивов:

- `array` - простой массив с числовым индексом, начинающимся с 0;
- `map` - ассоциативный массив со строковыми ключами.

Присваивание и получение элементов осуществляется указанием индекса в квадратных скобках. Следует заметить, что мульти-индексы не поддерживаются. То есть, вы не можете обратиться к элементу массива массивов как *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
```

Кроме этого, вы можете определять массивы array и map перечислением элементов в [] (для array) и {} (для map).

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

Вы можете подставлять такую инициализацию прямо в выражения - например в параметрах вызова функций.

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

Для ассоциативных массивов обязательно указание ключа - он должен быть в виде строки в двойных кавычках. Если имя ключа содержит только буквы, цифры и подчеркивание, то двойные кавычки можно опускать.

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

В качестве значений массивов можно указывать строки, числа, имена переменных любого типа и имена переменных со знаком доллара. Так как значением может быть другой map или array, то можно указывать инициализацию вложенных массивов. Нельзя указывать в качестве значений любые выражения. В таких случаях следует заводить промежуточную переменную.

```
[1+2, myfunc(), name["param"]] // нельзя
[1, 3.4, mystr, "string", $ext, myarr, mymap, {"ids": [1,2, i], company: {"Name": "MyCompany"}} ] /
↪/ можно

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

Конструкции if и while

Язык описания контрактов содержит стандартные условную конструкцию **if** и конструкцию цикла **while**, которые используются внутри функций, и контрактов. Эти конструкции могут вкладывать друг в друга.

После ключевого слова должно идти условное выражение. Если условное выражение возвращает число, то оно считается *ложь* при значении 0. Например, *val == 0* эквивалентно *!val*, а *val != 0* тоже самое, что просто *val*. Конструкция **if** может иметь блоки **elif** и блок **else**, который выполняется если условное выражение **if** ложно. Блоки **elif** должны содержать очередное проверяемое условие. В условном выражении можно использовать операции сравнения: `<`, `>`, `>=`, `<=`, `==`, `!=`, а также `||` (ИЛИ) и `&&` (И).

```
if val > 10 || id != $citizen {
  ...
} elif val == 5 {
   ...
} elif val < 0 {
   ...
} else {
  ...
}
```

Конструкция **while** предназначена для реализации циклов. Блок **while** выполняется до тех пор, пока его условие истинно. Для прекращения цикла внутри блока используется оператор **break**. Для исполнения блока цикла сначала используется оператор **continue**.

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

Кроме условных выражений, язык поддерживает стандартные арифметические действия: `+,-,*,/` Если в качестве условия вы укажете переменную типа **string** или **bytes**, то условие будет истино, если длина строки (bytes) больше нуля. На пустой строке условие будет ложь.

Функции

Функции языка написания контрактов выполняют операции с данными, полученными в секции `data` контракта: чтение значений из базы данных и запись значений в базу данных, преобразование типов значений и установление связи между контрактами.

Функция определяется с помощью ключевого слова **func**, после которого указывается имя функции, в круглых скобках через запятую передаваемые параметры с указанием типа, после закрывающей скобки - тип возвращаемого значения. Тело функции заключается в фигурные скобки. Если функция не имеет параметров, то круглые скобки можно опустить. Для возврата значения из функции используется ключевое слово `return`.

```
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..."
}
```

Функции не возвращают ошибок, так как все проверки на ошибки происходят автоматически. При генерации ошибки в любой из функции, контракт прекращает свою работу и выводит описание ошибки в специальном окне. Ошибки при выполнении любой функции обрабатываются автоматически, вызывая остановку выполнения контракта и вывод соответствующего сообщения.

Имеется возможность передавать функции неопределенное количество параметров. Для этого у последнего параметра необходимо вместо типа указать `....` В этом случае, последний параметр будет иметь тип *array* и содержать все, начиная с данного параметра, указанные при вызове переменные. Можно передавать переменные любых типов, но вы должны самостоятельно предотвращать конфликты выполнения из-за несовпадений типов.

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

    while i < Len(values) {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        res = res + values[i]
        i = i + 1
    }
    Println(out, res)
}

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

Рассмотрим ситуацию, когда функция может иметь много параметров, но часто при вызове необходимо указывать только некоторые из них. В этом случае,опциональные параметры можно описывать следующим образом `func myfunc(name string).Param1(param string).Param2(param2 int) {...}`. При вызове вы можете в любом порядке указывать только некоторые из дополнительных параметров `myfunc("name").Param2(100)`. В теле функции вы как обычно можете обращаться к этим переменным. Если при вызове расширенный параметр не указан, то он принимает значение по умолчанию, например, пустая строка для строки и ноль для числа. Также, следует заметить, что можно указывать несколько расширенных параметров и использовать `...` - `func DBFind(table string).Where(params map)` и вызов `DBFind("mytable").Where({id: {"$gt": myid}, type: 2)`

```
func DBFind(table string).Columns(columns string).Where(params map)
         .Limit(limit int).Offset(offset int) string  {
   ...
}
```

Некоторые предопределенные функции позволяют передавать неопределенное количество параметров. Имеется возможноть динамически формировать и передавать список параметров. Для этого необходимо записать их в переменную типа *array* и передать её с троеточием.

```
var names, values array
...
MyFunc("mytable", Join(names, ","), values...)
```

Предопределенные переменные

При выполнении контракта доступны следующие переменные.

- `$key_id` - числовой идентификатор (int64) аккаунта, от которого подписана транзакциюя,
- `$ecosystem_id` - идентификатор экосистемы, в которой была создана транзакция,
- `$type` - идентификатор вызываемого контракта. Если, например, контракт вызвал другой контракт, то здесь будет хранится идентификатор оригинального контракта,
- `$time` - время указанное в транзакции в формате Unix,
- `$block` - номер блока, в котором запечаталась данная транзакция,
- `$block_time` - время указанное в блоке,
- `$block_key_id` - числовой идентифкатор (int64) ноды, которая подписала блок,
- `$auth_token` - токен авторизации, который можно использовать в OBS контрактах, например, при вызове контрактов через api с помощью функции `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)
```

Предопределенные переменные доступны не только в контрактах, но и в полях Permissions, в которых указываются условия доступа к элементам приложений (таблицам, контрактам, страницам и пр.) для составления логических выражений. При этом переменные имеющие отношения к формированию блока (*$time*, *$block* и др.) имеют нулевое значение.

Предопределенная переменной `$result` используется при необходимости возвратить значение из вложенного контракта.

Получение значений из таблиц базы данных

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

Функция возвращает значение указанного параметра из параметров приложения (таблица *app_params*).

- *app* - идентификатор приложения,
- *name* - имя получаемого параметра,
- *ecosystemid* - идентифкатор экосистемы.

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

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

Функция возвращает массив *array* из таблицы базы данных *table* в соответствии с указанным запросом. Массив *array* состоит из ассоциативных массивов *map*, содержащих данные из записей таблицы. Для получения массива *map* первого элемента (первой записи запроса) используется функция `.Row()`. Единичное значение колонки с именем *column* из первого элемента массива возвращается добавлением функции `.One(column string)`.

- *table* - имя таблицы,
- *columns* - список возвращаемых колонок, можно указать в виде массива array или в виде строки с перечислением через запятую. Если не указано, то возвратятся все колонки,
- *Where* - условие поиска. Например, `.Where({name: "John"})` или `.Where({"id": {"$gte": 4}})`,

В параметр **where** должен передаваться ассоциативный массив, в котором описаны условия поиска. Массив может состоять из элементов любой вложенности. Имеются следующие управляющие конструкции:

- **{«field1»: «value1», «field2» : «value2} → field1 = «value1» AND field2 = «value2**
- **{«field1»: {«$eq»:»value»}} → field = «value»**
- **{«field1»: {«$neq»: «value»}} → field != «value»**
- **{«field1: {«$in»: [1,2,3]} → field IN (1,2,3)**
- **{«field1»: {«$nin» : [1,2,3]} → field NOT IN (1,2,3)**

- **{«field»: {«$lt»: 12}} → field < 12**
- **{«field»: {«$lte»: 12}} → field <= 12**
- **{«field»: {«$gt»: 12}} → field > 12**
- **{«field»: {«$gte»: 12}} → field >= 12**
- **{«$and»: [<expr1>, <expr2>, <expr3>]} → expr1 AND expr2 AND expr3**
- **{«$or»: [<expr1>, <expr2>, <expr3>]} → expr1 OR expr2 OR expr3**
- **{field: {«$like»: «value»}} → field like „%value%“ (поиск подстроки)**
- **{field: {«$begin»: «value»}} → field like „value%“ (начинается с value)**
- **{field: {«$end»: «value»}} → field like „%value“ (заканчивается value)**
- **{field: {«$ilike»: «value»}} → field ilike „%value%“ (регистронезависимый поиск подстроки)**
- **{field: {«$ibegin»: «value»}} → field ilike „value%“ (регистронезависимый поиск - начинается с value)**
- **{field: {«$iend»: «value»}} → field ilike „%value“ (регистронезависимый поиск - заканчивается value)**
- **{field: «$isnull»} → field is null**

При перечислении элементов массивов **$or** или **$and** можно не указывать фигурные скобки у элементов. Например

```
m = DBFind("contracts").Where({id: 10, name: "EditColumn", $or: [id: 10, id: {$neq: 20}]})
```

Имеется ещё один момент. Предположим есть запрос *id>2 and id<5*. Написать так *{id:{«$gt»: 2}, id:{«$lt»: 5}}* нельзя, так как у нас в массиве второе присваивание ключа перекроет первое и останется только *id<5*. В этом случае можно применять один из двух способов.

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

второй способ более короткий - мы все варианты для колонки указываем в виде массива.

- *id* - поиск по идентификатору. Достаточно указать значение идентификатора. Например, `.WhereId(1)`,
- *order* - поле, по которому нужно отсортировать. По умолчанию, сортируется по *id*. Если сортируется только по одному полю, то его можно указать в качестве строки. В противном случае, необходимо передавать массив строк и объектов *{«field»: «-1»}* или *{«field»: «1»}*. *{«field»: «-1»} = field desc*, *{«field»: «1»} = field asc*. Например, **.Order({name: «-1»}, {amount: «1»})**
- *limit* - количество возвращаемых записей. По умолчанию, 25. Максимально возможное количество - 250,
- *offset* - смещение возвращаемых записей,
- *ecosystemid* - идентификатор экосистемы. По умолчанию, берутся данные из таблицы в текущей экосистеме.

```
var i int
ret = DBFind("contracts").Columns(["id","value"]).Where({id: [{"$gt": 3}, {"$lt":8}]}).Order("id")
while i < Len(ret) {
    var vals map
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Функция возвращает ассоциативный массив *map*, с данными полученными из таблицы *table* в соответствии с указанным запросом.

- *table* - имя таблицы,
- *columns* - список возвращаемых колонок, можно указать в виде массива array или в виде строки с перечислением через запятую. Если не указано, то возвратятся все колонки,
- *Where* - условие поиска, подробнее описано в функции **DBFind**; например, `.Where({name: "John"})` или `.Where({"id": {"$gte": 4}})`,
- *id* - идентификатор возвращаемой строки; например, `.WhereId(1)`,
- *order* - поле по которому производится сортировка; по умолчанию, сортируется по *id*. Более подробно описано в функции **DBFind**.
- *ecosystemid* - идентификатор экосистемы; по умолчанию, id текущей экосистемы.

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

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

Функция возвращает массив *array* с агрегированными данными для метрики *metric* за указанный интервал времени *timeInterval*, агрегация осуществляется через функцию *aggregateFunc*. Массив *array* состоит из ассоциативных массивов *map*, содержащих данные *key* - ключ, *value* - значение.

Названия метрик:

- *ecosystem_ pages* - кол-во страниц экосистемы, *key* - номер экосистемы, *value* - значение,
- *ecosystem_ members* - кол-во участников экосистемы, *key* - номер экосистемы, *value* - значение,
- *ecosystem_ tx* - кол-во транзакций экосистемы, *key* - номер экосистемы, *value* - значение.

Метрики обновляются через каждые 100 блоков и хранятся в разрезе за каждый день.

- *metric* - название метрики,
- *timeInterval* - интервал вермени, за который требуется получить значения метрик. Например, `1 day` или `30 days`,
- *aggregateFunc* - функция агрегации. Например, `max`, `min` или `avg`,

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

var i int
while(i < Len(rows)) {
   var row map
   row = rows[i] // row содержит map, с ключами key и value, где key - номер экосистемы, value -␣
↪среднее кол-во транзакций за 30 дней
   i = i + 1
}
```

EcosysParam(name string) string

Функция возвращает значение указанного параметра из настроек экосистемы (таблица *parameters*).

- *name* - имя получаемого параметра,
- *num* - порядковый номер параметра.

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

GetHistory(table string, id int) array

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

Функция возвращает массив ассоциативных массивов типа *map* с историей изменений записи в указанной таблице с именем **table**. Каждый ассоциативный массив содержит поля записи перед очередным изменением. Результирующий список отсортирован от последних изменений к более ранним. В результирующей таблице поле *id* указывает на id в таблице *rollback_tx*. Также возвращаются поля *block_id* - номер блока, *block_time* - время блока. Функция **GetHistoryRow** возвращает только одну запись с указанным идентификатором в таблице *rollback_tx* в виде ассоциативного массива map.

- *table* - имя таблицы.
- *id* - идентификатор записи.
- *RollbackId* - (для функции **GetHistoryRow**) идентификатор записи id в таблице *rollback_tx*.

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

GetColumnType(table, column string) string

Функция возвращает тип указанной колонки в указанной таблице. Возвращается наименование внутреннего типа -например, *text,varchar,number,money,double,bytea,json,datetime,double*.

- *table* - имя таблицы,
- *column* - имя колонки.

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

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

Функция возвращает данные в виде массива массивов ячеек из таблицы XLSX.

- *binId* - идентификатор загруженной XLSX таблицы из таблицы *binary*,
- *line* - строка с которой необходимо получить данные, счёт с нуля,
- *count* - количество возвращаемых строк,
- *sheet* - номер листа в XLSX файле, счёт с 1.

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

GetRowsCountXLSX(binId int, sheet int) int

Функция возвращает количество строк на указанном листе в XLSX файле.

- *binId* - идентификатор загруженной XLSX таблицы из таблицы *binary*,
- *sheet* - номер листа в XLSX файле, счёт с 1.

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

LangRes(label string, lang string) string

Функция возвращает языковой ресурс с именем label для языка lang, заданного двухсимвольным кодом, например, *en,fr,ru*. Если для указанного языка нет ресурса, то возвращается значение на английском языке. Используется для перевода текста в всплывающих окнах, инициируемых контрактами.

- *label* - имя языкового ресурса.
- *lang* - двухсимвольный код языка.

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

GetBlock(blockID int64) map

Функция возвращает информацию о блоке *blockID*. Информация возвращается в виде ассоциативного массива *map*, содержащего данные:

- *id* - номер блока,
- *time* - время генерации блока в Unix,
- *key_ id* - ключ ноды, которая сгенерировала блок.

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

Изменение значений в таблицах базы данных

DBInsert(table string, params map) int

Функция добавляет запись в таблицу *table* и возвращает **id** вставленной записи.

- *tblname* - имя таблицы в базе данных,
- *params* - ассоциативный массив *map*, в котором в качестве ключей передаются имена полей и соответствующие им значения.

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

DBUpdate(tblname string, id int, params map)

Функция изменяет значения столбцов в таблице в записи с указанным **id**. Если записи с таким идентификатором не существует, то будет выдаваться ошибка.

- *tblname* - имя таблицы в базе данных,
- *id* - идентификатор **id** изменяемой записи,
- *params* - ассоциативный массив *map*, в котором в качестве ключей передаются имена полей и соответствующие им значения.

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

DBUpdateExt(tblname string, where map, params map)

Функция обновляет столбцы в записи, которая удовлетворяет параметрам поиска.

- *tblname* - имя таблицы в базе данных,
- *where* - условие поиска. Например, `{name: "John"}`.“{«id»: {«$gte»: 4}}“, `{id: $key_id, ecosystem: $ecosystem_id}`. Полное описание возможностей по созданию условий поиска имеется в описании функции **DBFind**,
- *params* - ассоциативный массив *map*, в котором в качестве ключей передаются имена полей и соответствующие им значения.

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

DelColumn(tblname string, column string)

Функция удаляет столбец в указанной таблице. Таблица не должна содержать записей.

- *tblname* - имя таблицы в базе данных,
- *column* - имя удаляемой колонки.

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

DelTable(tblname string)

Функция удаляет указанную таблицу. Таблица не должна содержать записей.

- *tblname* - имя таблицы в базе данных.

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

Операции с массивами

Append(src array, val someType) array

Функция вставляетв src значение val любого типа и возвращает результирующий массив

- *src* - исходный массив
- *val* - значение, которое необходимо добавить в массив

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

Join(in array, sep string) string

Функция объединяет элементы массива *in* в строку с указанным разделителем *sep*.

- *in* - имя массива типа *array*, элементы которого необходимо объединить,
- *sep* - строка-разделитель.

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

Split(in string, sep string) array

Функция возвращает массив, полученный из элементов строки *in*, при ее разбивании в соответствии с разделителем *sep*.

- *in* - исходная строка,
- *sep* - строка-разделитель.

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

Len(val array) int

Функция возвращает количество элементов в указанном массиве.

- *val* - массив типа *array*.

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

Row(list array) map

Функция возвращает первый ассоциативный массив *map* из массива *list*. Если список *list* пустой, то результат вернет пустой *map*. Используется преимущественно с функцией DBFind, в этом случае параметр *list* не указывается.

- *list* - массив map, возвращаемый функцией **DBFind**.

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

One(list array, column string) string

Функция возвращает значение ключа *column* из первого ассациативного массива в массиве *list*. Если список *list* пустой, то возвращается nil. Используется преимущественно с функцией DBFind, в этом случае параметр *list* не указывается.

- *list* - массив map, возвращаемый функцией **DBFind**,
- *column* - имя возвращаемого ключа.

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

GetMapKeys(val map) array

Функция возвращает массив ключей из ассациативного массива *val*.

- *val* - массив map.

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

SortedKeys(val map) array

Функция возвращает отсортированный массив ключей из ассациативного массива *val*.

- *val* - массив map.

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

Операции с контрактами и условиями

CallContract(name string, params map)

Функция вызывает контракт по его имени. В передаваемом массиве должны быть перечислены все параметры, указанные в section *data* контракта. Функция возвращает значение, которое было присвоено переменной **$result** в контракте.

- *name* - имя вызываемого контракта,
- *params* - ассоциативный массив с входными данными для контракта.

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

ContractAccess(name string, [name string]) bool

Функция проверяет, совпадает ли имя выполняемого контракта с одним из имен, перечисленных в параметрах. Используется для контроля доступа контрактов к таблицам. Функция прописывается в полях *Permissions* колонок таблицы или в полях *Insert* и *New Column* в разделе *Table permission*.

- *name* - имя контракта.

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

ContractConditions(name string, [name string]) bool

Функция вызывает секцию **conditions** контрактов с указанными именами (у контрактов секция *data* должен быть пустой). Если секция *conditions* выполнилась без ошибок, то возвращается *true*, в противном случае «false*. Функция используется в полях *Permissions* для задания прав доступа к соответствующим операциям с элементами приложений (страницами, таблицами, контрактами и пр), а так же в секции *conditions* контрактов - если в процессе выполнения перечисленный в парараметрах контрактов сгенерировалась ошибка, то родительский контракт также завершится с данной ошибкой.

- *name* - имя контракта.

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

EvalCondition(table string, name string, condfield string)

Функция берет из таблицы *table* значение поля *condfield* из записи с полем *„name“*, которое равно параметру *name*, и проверяет выполнено ли условие полученное из поля *condfield* или нет. Если условие не выполнено, то генерируется ошибка, с которой и завершается вызывающий контракт.

- *table* - имя таблица,
- *name* - значение для поиска по полю „name“,
- *condfield* - имя поля где хранится условие, которое необходимо будет проверить.

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

CheckCondition(condition string) bool

Функция проверяет выполнено ли условие, которое указано в параметре *condition*. Если условие выполнено, то возвращается *true*, в противном случае возвращается *false*.

- *condition* - условие, которое необходимо проверить.

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

GetContractById(id int) string

Функция возвращает имя контракта по его идентификатору. Если контракт не найден, то возвращается пустая строка.

- *id* - идентификатор контракта в таблице *contracts*.

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

GetContractByName(name string) int

Функция возвращает идентификатор контракта в таблице *contracts* по его имени. Если контракт не найден, то возвращается ноль.

- *name* - идентификатор контракта в таблице *contracts*.

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

RoleAccess(id int, [id int]) bool

Функция проверяет, совпадает ли идентификатор роли того, кто вызвал контракт, с одним из идентификаторов, перечисленных в параметрах. Используется для контроля доступа контрактов к таблицам и прочим данным.

- *id* - идентификатор роли.

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

TransactionInfo(hash: string)

Функция ищет транзакцию по указанному хэшу и возвращает информацию о вызванном контракте и его параметрах. Функция возвращает строку в формате json *{«contract»:»ContractName», «params»:{«key»: «val»}, «block»: «N»}*, где в поле *contract* возвращается имя контракта, *params* - переданные параметры, *block* - номер блока в котором была обработана данная транзакция.

- *hash* - хэш транзакции в виде шестндцатеричной строки.

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

Throw(ErrorId: string, ErrDescription: string)

Функция генерирует ошибку выполнения типа *exception*, но добавляет туда дополнительное поле *id*. Результат выполнения такой транзакции будет иметь вид *{«type»:»exception»,»error»:»Error description»,»id»:»Error ID»}*

- *ErrorId* - идентификатор ошибки.
- *ErrDescription* - описание ошибки.

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

ValidateCondition(condition string, ecosystemid int)

Функция пытается скомпилировать условие, указанное в параметре *condition*. Если в процессе компиляции условия возникнет ошибка, то будет сгенерирована ошибка и вызывающий контракт закончит свою работу. Данная функция предназначена для проверки правильности условий при их изменении.

- *condition* - проверяемое условие,
- *ecosystemid* - идентифкатор экосистемы.

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

Операции с адресами аккаунтов

AddressToId(address string) int

Функция возвращает числовой идентификатор владельца аккаунта по строковому значению адреса аккаунта. Если указан несуществующий адрес, то возвращается 0.

- *address* - адрес аккаунта в формате XXXX-...-XXXX или в виде числа.

```
account = AddressToId($Recipient)
```

HexToPub(hexpub string) bytes

Функция конвертирует шестнадцатеричную строку с публичным ключом в переменную типа *bytes*. При этом входящая стркоа может содержать в начале префикс „04“.

- *hexpub* - публичный ключ в шестнадцатеричном виде

```
pub = HexToPub(hexkey)
```

IdToAddress(id int) string

Функция возвращает строковый адрес аккаунта по числовому идентификатору его владельца. Если указан несуществующий id, то возвращается „invalid“.

- *id* - числовой идентификатор.

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

PubToHex(pub string | bytes) string

Функция конвертирует публичный ключ в шестнадцатеричную строку. При этом добавляется в начало строка „04“.

- *pub* - публичный ключ в двоичном виде

```
pub = DBFind("@1keys").Columns("pub").Where({id:"2367267345348734"}).One("pub")
var hex string
hex = PubToHex(pub)
```

PubToID(hexkey string) int

Функция возвращает числовой идентификатор владельца публичного ключа. При ошибке возвращает ноль.

- *hexkey* - публичный ключ в виде шестнадцатиричной строки

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

Операции со значениями переменных

DecodeBase64(input string) string

Функция раскодирует строку в кодировке base64.

- *input* - входящая строка в кодировке base64.

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

EncodeBase64(input string) string

Функция кодирует строку в кодировку base64 и возвращает строку в закодированном виде.

- *input* - входящая строка.

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

Float(val int|string) float

Функция преобразует целое число *int* или *string* в число с плавающей точкой.

- *val* - целое число или строка.

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

HexToBytes(hexdata string) bytes

Функция преобразует строку с шестнадцатеричной кодировкой в значение типа *bytes* (последовательность байт).

- *hexdata* - строка, содержащая шестнадцатеричную запись.

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

FormatMoney(exp string, digit int)

Функция возвращает строковое значение exp/10^digit. Если параметр digit не указан, то он будет браться из параметра **money_digit** экосистемы.

- *exp* - Числововое значение в виде строки,
- *digit* - степень 10 в выражении exp/10^digit. Может быть как положительным, так и отрицательным. В случае положительного значения определяет количество цифр после запятой.

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

Random(min int, max int) int

Функция возвращает случайное число в диапазоне между min и max (min <= result < max). min и max должны быть положительными числами.

- *min* - минимальное значение случайного числа,
- *max* - Случайное значение будет меньше этого числа.

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

Int(val string) int

Функция преобразует строковое значение в целое число.

- *val* - строка содержащая число.

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

Hash(val interface{}) string, error

Функция принимает массив байт или строку и возвращает Hash, полученный с помощью системного криптопровайдера.

- *val* - входящая строка или массив байт

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

Sha256(val string) string

Функция возвращает хэш **SHA256** от указанной строки.

- *val* - входящая строка, для которой нужно вычислить хэш **Sha256**.

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

Str(val int|float) string

Функция преобразует числовое значение типа *int* или *float* в строку.

- *val* - целое или число с плавающей точкой.

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

UpdateLang(appID int, name string, trans string)

Функция обновляет языковой ресурс в памяти. Используется в транзакциях, которые меняют языковые ресурсы.

- *appID* - id приложения.
- *name* - имя языкового ресурса.
- *trans* - ресурс с переводами.

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

Операции с JSON

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

Функция конвертирует число, строку или массив *src* в строку в формате JSON.

- *src* - Данные которые требуется конвертировать в JSON.

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

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

Функция конвертирует число, строку или массив *src* в строку в формате JSON с указанными отступами.

- *src* - Данные которые требуется конвертировать в JSON,
- *indent* - Строка, которая будет использоваться в качестве отступов.

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

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

Функция конвертирует строку *src* с данными в формате JSON в число, строку или массив.

- *src* - Строка с данными в JSON формате.

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

Операции со строковыми значениями

HasPrefix(s string, prefix string) bool

Функция возвращает true, если строка начинается с указанной подстроки *prefix*.

- *s* - проверяема строка,
- *prefix* - проверяемый префикс у данной строки.

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

Contains(s string, substr string) bool

Функция возвращает true, если строка *s* содержит подстроку *substr*.

- *s* - проверяема строка,
- *substr* - подстрока, которая ищется в указанной строке.

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

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

Функция заменять в строке *s* все вхождения строки *old* на строку *new* и возвращает полученный результат.

- *s* - исходная строка,
- *old* - заменяемая строка,
- *new* - новая строка.

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

Size(val string) int

Функция возвращает количество символов в указанной строке.

- *val* - входящая строка.

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

Sprintf(pattern string, val ...) string

Функция формирует строку на основе указанного шаблона и параметров, можно использовать `%d` (число), `%s` (строка), `%f` (float), `%v` (для любых типов).

- *pattern* - шаблон для формирования строки.

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

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

Функция возвращает подстроку от указанной строки начиная со смещения *offset* (считается с 0) и длиной *length*. В случае некорректных смещений или длины возвращается пустая строка. Если сумма смещения и *length* больше размера строки, то возвратится подстрока от смещения до конца строки.

- *val* - входящая строка,
- *offset* - начальное смещение подстроки,
- *length* - размер подстроки.

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

ToLower(val string) string

Функция возвращает указанную строку в нижнем регистре .

- *val* - входящая строка.

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

ToUpper(val string) string

Функция возвращает указанную строку в верхнем регистре .

- *val* - входящая строка.

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

TrimSpace(val string) string

Функция возвращает указанную строку с удаленными начальными и конечными пробелами, переводами строки и знаками табуляции.

- *val* - входящая строка.

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

Математические функции

Floor(x float|int|string) int

Функция возвращает ближайшее целое, которое меньше или равно данному числу.

- x - число.

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

Log(x float|int|string) float

Функция возвращает натуральный логарифм.

- x - число для вычисления логарифма.

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

Log10(x float|int|string) float

Функция возвращает десятичный логарифм.

- x - число для вычисления логарифма.

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

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

Функция возводит число x в степень y.

- x - основание.
- y - степень.

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

Round(x float|int|string) int

Функция округляет число до ближайшего целого.

- x - число.

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

Sqrt(x float|int|string) float

Функция возвращает квадратный корень.

- x - число для вычисления квадратного корня.

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

Операции с байтами

StringToBytes(src string) bytes

Функция преобразует строку в байты.

- *src* - строка.

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

BytesToString(src bytes) string

Функция преобразует байты в строку.

- *src* - байты.

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

Операции с системными параметрами

SysParamString(name string) string

Функция возвращает значение указанного системного параметра.

- *name* - имя параметра.

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

SysParamInt(name string) int

Функция возвращает значение указанного системного параметра в виде числа.

- *name* - имя параметра.

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

DBUpdateSysParam(name, value, conditions string)

Функция обновляет значение и условие системного параметра. Если значение или условие менять не нужно, то в соответствующем параметре следует указать пустую строку.

- *name* - имя параметра,
- *value* - новое значение параметра,
- *conditions* - новое условие изменения параметра.

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

UpdateNotifications(ecosystemID int, keys int . . .)

Функция получает список уведомления для указанных ключей из базы данных и рассылает по этим ключам уведомления в центрифугу.

- *ecosystemID* - идентификатор экосистемы,
- *key* - ключи через запятую, по которым проходит проверка. Можно отправить один массив array со списком ключей.

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

UpdateRolesNotifications(ecosystemID int, roles int . . .)

Функция получает список уведомления для всех ключей из указанных ролей из базы данных и рассылает по этим ключам уведомления в центрифугу.

- *ecosystemID* - идентификатор экосистемы,
- *roles* - идентификаторы ролей через запятую, по которым проходит проверка. Можно отправить один массив array со списком ролей.

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

Работа с JSON в запросах к PostgreSQL

В качестве типа колонок вы можете указывать тип **JSON**. В этом случае, если вы хотите обращаться к полям записи, вам следует использовать запись вида **имяколонки->имяполя**. Полученное значение будет записано в колонку с именем **имяколонки.имяполя**. Обращение **имяколонки->имяполя** можно использовать в параметрах *Columns,One,Where* при запросах **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")
```

Операции датой/временем в запросах к PostgreSQL

Функции не дают возможности напрямую отправлять запросы с select, update и т.д., но они позволяют использовать возможности и функции PostgreSQL при получении значений и описания условий where в выборках. Это относится в том числе и к функциям работающим с датами и временем. Например, необходимо сравнить колонку *date_ column* и текущее время. Если *date_ column* имеет тип timestamp, то выражение будет следующим `date_column > now()`, а если *date_ column* хранит время в Unix формате в виде числа, то тогда выражение будет `to_timestamp(date_column) > now()`.

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

BlockTime()

Функция возвращает время генерации блока в SQL формате. Данная функция должна использоваться вместо функции получения текущего времени NOW().

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

DateTime(unixtime int) string

Функция конвертирует unixtime в формат времени *YYYY-MM-DD HH:MI:SS*.

```
DateTime(1532325250)
```

UnixDateTime(datetime string) int

Функция конвертирует строку с форматом времени *YYYY-MM-DD HH:MI:SS* в unixtime.

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

DateTimeLocation(unixtime int, loc string) string

Функция конвертирует unixtime с учетом временной зоны loc в формат времени *YYYY-MM-DD HH:MI:SS*.

```
DateTimeLocation(1532325250, "Europe/Luxembourg")
```

UnixDateTimeLocation(datetime string, loc string) int

Функция конвертирует строку с форматом времени *YYYY-MM-DD HH:MI:SS* в unixtime с учетом временной зоны loc.

```
UnixDateTimeLocation("2018-07-20 14:23:10", "Europe/Luxembourg")
```

Список поддерживаемых временных зон содержится в системной таблице @1time_zones.

Функции для OBS

Данные функции можно использовать только в контрактах Off-Blockchain Server (OBS).

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

Функция отправляет HTTP запрос на указанный адрес.

- *url* - адрес, на который будет отправлен запрос,
- *method* - метод запроса - GET или POST,
- *heads* - массив данных для формирования заголовка,
- *pars* - параметры.

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

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

Функция подобна функции *HTTPRequest*, но отправляет *POST* запрос и параметры передаются одной строкой.

- *url* - адрес, куда будет отправлен запрос,
- *heads* - массив данных для формирования заголовка,
- *pars* - параметр в виде json строки.

```
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)
```

Функции для мастера OBS

Данные функции можно использовать только в режиме OBSMaster

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

Функция создает дочернюю OBS

- *OBSName* - имя OBS, может содержать только латиницу и цифры, без пробелов
- *DBUser* - имя роли для базы данных
- *DBPassword* - пароль для новой роли
- *OBSAPIPort* - порт для http запросов

ListOBS()

Возвращает ассоциативный массив дочерних OBS, где в качестве ключа используется имя OBS, а в качестве значение статус процесса

RunOBS(OBSName string)

Запускает процесс для OBS с именем OBSName

- *OBSName* - имя OBS, может содержать только латиницу и цифры, без пробелов

StopOBS(OBSName string)

Останавливает процесс для OBS с именем OBSName

- *OBSName* - имя OBS, может содержать только латиницу и цифры, без пробелов

RemoveOBS(OBSName string)

Удаляет процесс для OBS с именем OBSName, останавливает и удаляет связанный процесс

- *OBSName* - имя OBS, может содержать только латиницу и цифры, без пробелов

3.4.6 Системные контракты

Системные контракты создаются по умолчанию при установке платформы в экосистеме №1. Поэтому при вызове их из других экосистем необходимо указывать полное имя, например, `@1NewContract`. Также, следует заметить, что страницы могут подгружаться из других экосистем. Если на странице имеются кнопки или ссылки для вызова контрактов, то они не будут работать при подгрузке с других экосистем. В этом случае, нужно также явно указывать экосистему вместе с именем контракта.

Список системных контрактов

NewEcosystem

Контракт создает создает новую экосистему. Для получения идентификатора созданной экосистемы необходимо обратиться к полю *result*, которое возвращается в txstatus. Параметры:

- *Name string* - имя экосистемы (можно изменить в дальнейшем).

EditEcosystemName

Контракт позволяет изменить имя экосистемы в таблице 1_ecosystems, таблица присутствует только в первой экосистеме * *SystemID* - код экосистемы, имя которой требуется изменить * *NewName* - новое имя экосистемы

MoneyTransfer

Контракт переводит токены платформы с аккаунта текущего пользователя на указанный аккаунт в первой экосистеме. Для перевода необходимо иметь на счету дополнительно 0.1 APL из которых будет оплачена комиссия за перевод. Параметры:

- *Recipient string* - акканут получателя в любом формате - число или `XXXX-....-XXXX`,
- *Amount string* - сумма переводимых токенов,
- *Comment string «optional»* - комментарий.

NewContract

Контракт создает новый контракт в текущей экосистеме. Параметры:

- *Value string* - текст контракта. На верхнем уровне должен быть только один контракт.
- *Conditions string* - условие изменения контракта.
- *Wallet string «optional»* - идентификатор аккаунта пльзователя, к которому планируется привязать контракт (по умолчанию основателя экосистемы).
- *TokenEcosystem int «optional»* - идентификатор экосистемы в токенах которой будет происходить оплата, если контракт будет активрован.

EditContract

Изменение контракта в текущей экосистеме. Параметры:

- *Id int* - идентификатор изменяемого контракта,
- *Value string «optional»* - текст контракта или контрактов,
- *Conditions string «optional»* - права доступа на изменение контракта.

BindWallet

Привязка контракта к аккаунту в текущей экосистеме. Привязка возможна с к аккаунту, который был указан при создании контракта. После привязки, с указанного аккаунта будет оплачивать выполнение данного контракта. Параметры:

- *Id int* - идентификатор привязываемого контракта.

UnbindWallet

Отвязка контракта от аккаунта в текущей экосистеме. Отвязка возможна с того аккаунта, к которому был привязан контракт. После отвязки контракта, его выполнение будут оплачивать вызывающие его пользователи. Параметры:

- *Id int* - идентификатор отвязываемого контракта.

NewParameter

Контракт добавляет новый параметр к текущей экосистеме. Параметры:

- *Name string* - имя параметра,
- *Value string* - значение параметра,
- *Conditions string* - права на изменение параметра.

EditParameter

Контракт изменяет существующий параметр в текущей экосистеме. Параметры:

- *Name string* - имя изменяемого параметра,
- *Value string* - новое значение параметра,
- *Conditions string* - новые права на изменение параметра.

NewMenu

Контракт добавляет новое меню к текущей экосистеме. Параметры:

- *Name string* - имя меню,
- *Value string* - текст меню,
- *Title string «optional»* - заголовок меню,
- *Conditions string* - права на изменение меню.

EditMenu

Контракт изменяет существующее меню в текущей экосистеме. Параметры:

- *Id int* - идентификатор изменяемого меню,
- *Value string «optional»* - новай текст меню,

- *Title string «optional»* - заголовок меню,
- *Conditions string «optional»* - новое права на изменение меню.

AppendMenu

Контракт добавляет текст к существующему меню в текущей экосистеме. Параметры:

- *Id int* - идентификатор дополняемого меню,
- *Value string* - добавляемый текст.

NewPage

Контракт добавляет новую страницу в текущей экосистеме. Параметры:

- *Name string* - имя страницы,
- *Value string* - текст страницы,
- *Menu string* - имя меню, привязанного к данной странице,
- *Conditions string* - права на изменение страницы,
- *ValidateCount int «optional»* - кол-во нод для проверки валидности страницы, если параметр не задан, то используется значение из параметра экосистемы *min_page_validate_count*. Значение не может быть меньше *min_page_validate_count* и больше *max_page_validate_count*,
- *ValidateMode int «optional»* - количество проверок страниц. 0 - только при загрузке, 1 - при загрузке и при уходе со страницы.

EditPage

Контракт изменяет существующую страницу в текущей экосистеме. Параметры:

- *Id int* - идентификатор изменяемой страницы,
- *Value string «optional»* - новый текст страницы,
- *Menu string «optional»* - имя нового меню страницы,
- *Conditions string «optional»* - новые права на изменение страницы,
- *ValidateCount int «optional»* - кол-во нод для проверки валидности страницы, если параметр не задан, то используется значение из параметра экосистемы *min_page_validate_count*. Значение не может быть меньше *min_page_validate_count* и больше *max_page_validate_count*.
- *ValidateMode string «optional»* - количество проверок страниц. 0 - только при загрузке, 1 - при загрузке и при уходе со страницы.

AppendPage

Контракт добавляет текст к существующей странице текущей экосистеме. Параметры:

- *Id int* - идентификатор изменяемой страницы,
- *Value string* - добавляемый текст к странице.

NewBlock

Контракт добавляет новый страничный блок в текущей экосистеме. Параметры:

- *Name string* - имя блока,
- *Value string* - текст блока,
- *Conditions string* - права на изменение блока.

EditBlock

Контракт изменяет существующий блок в текущей экосистеме. Параметры:

- *Id int* - идентификатор изменяемого блока,
- *Value string «optional»* - новый текст блока,
- *Conditions string «optional»* - новые права на изменение блока.

NewTable

Контракт добавляет новую таблицу в текущей экосистеме. Параметры:

- *Name string* - имя таблицы (только латинские символы),
- *Columns string* - массив колонок в JSON формате `[{"name":"...", "type":"...","index": "0", "conditions":"..."},...]`, где
  - *name* - наименование колонки - латинские символы,
  - *type* - тип `varchar, bytea, number, datetime, money, text, double, character`,
  - *index* - неиндексируемое поле - «0», создать индекс - «1».
  - *conditions* - права на изменение данных в столбце; если необходимо указать права доступа на чтение, то нужно использовать JSON формат. Например, `{"update":"ContractConditions(`MainCondition`)", "read":"ContractConditions(`MainCondition`)"}`
- *Permissions string* - права на доступ в JSON формате `{"insert": "...", "new_column": "...", "update": "..."}`, где
  - *insert* - права на вставку записей,
  - *new_ column* - права на добавление колонки,
  - *update* - права на изменение прав.

EditTable

Контракт изменяет права на доступ к таблице в текущей экосистеме. Параметры:

- *Name string* - имя таблицы,
- *Permissions string* - Разрешения на доступ в JSON формате `{"insert": "...", "new_column": "...", "update": "..."}`, где
  - *insert* - права на вставку записей,
  - *new_ column* - права на добавление колонки,

– *update* - права на изменение прав.

NewColumn

Контракт добавляет новую колонку к таблице в текущей экосистеме. Параметры:

- *TableName string* - имя таблицы,
- *Name* - наименование колонки (тольколатинские символы),
- *type* - тип `varchar, bytea, number, datetime, money, text, double, character`,
- *Index* - неиндексируемое поле - «0», создать индекс - «1»,
- *Permissions* - права на изменение данных в столбце; если необходимо указать права доступа на чтение, то нужно использовать JSON формат, например, `{"update":"ContractConditions(`MainCondition`)", "read":"ContractConditions(`MainCondition`)"}`.

EditColumn

Контракт меняет права на изменение колонки в таблице в текущей экосистеме. Параметры:

- *TableName string* - имя таблицы,
- *Name* - имя колонки,
- *Permissions* - права на изменение значений в колонке, если необходимо указать права доступа на чтение, то нужно использовать JSON формат, например, `{"update":"ContractConditions(`MainCondition`)", "read":"ContractConditions(`MainCondition`)"}`.

NewLang

Контракт добавляет языковые ресурсы в текущей экосистеме. Права на добавление определяются в параметре *changing_language* в настройках экосистемы. Параметры:

- *Name string* - имя языкового ресурса (только латинские символы).
- *Trans* - языковые ресурсы в виде строки в JSON формате, где ключ - двухсимвольный код языков, значение - перевод, например: `{"en": "English text", "ru": "Английский текст"}`.
- *[Lang string]* - опциональный параметр. Указывает язык для сообщений об ошибках во время выполнения контракта.

EditLang

Контракт обновляет языковой ресурс в текущей экосистеме. Права на обновление определяются в параметре *changing_language* в настройках экосистемы. Параметры

- *Id int* - ID языкового ресурса.
- *Name string* - имя языкового ресурса.
- *Trans* - языковые ресурсы в виде строки в JSON формате, где ключ - двухсимвольный код языков, значение - перевод, например: `{"en": "English text", "ru": "Английский текст"}`.

- *[Lang string]* - опциональный параметр. Указывает язык для сообщений об ошибках во время выполнения контракта.

NewSign

Контракт создает данные для для контрактов с подписью в текущей экосистеме. Параметры:

- *Name string* - имя контракта, который будет использовать дополнительную подпись.
- *Value string* - описание параметров в виде JSON строки, где
  - *title* - текст сообщения,
  - *params* - массив параметров, которые показываются пользователю, где **name** - имя поля, **text** - описание параметра.
- *Conditions string* - права на изменение записи.

Пример значения *Value*

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

EditSign

Контракт обновляет данные для для контрактов с подписью в текущей экосистеме. Параметры:

- *Id int* - идентификатор изменяемой подписи,
- *Value string* - новое значение параметров,
- *Conditions string* - новые права на изменение параметров подписи.

Import

Контракт импортирует данные из файла *.sim в экосистему. Параметры:

- *Data string* - импортируемые данные, полученные при экспорте приложений в *.sim файл.

NewCron

Контракт добавляет новую задачу в cron для запуска по таймеру. Контракт присутствует только в OBS системах. Параметры:

- *Cron string* - строка, определяющая запуск контракта по таймеру в формате *cron*,
- *Contract string* - имя запускаемого в OBS контракта, контракт не должен содержать параметров в секции `data`,
- *Limit int* - необязательное поле, в котором можно указать количество запусков (пока не исполняется),
- *Till string* - необязательно поле с временем окончания задачи (пока не учитывается),
- *Conditions string* - права на изменение задачи.

EditCron

Контракт изменяет настройки задачи в cron для запуска по таймеру. Контракт присутствует только в OBS системах. Параметры:

- *Id int* - идентификатор задачи,
- *Cron string* - строка, определяющая запуск контракта по таймеру в формате *cron*; чтобы отключить задачу, нужно не указывать этот параметр или указать пустую строку,
- *Contract string* - имя запускаемого OBS контракта, контракт не должен содержать параметров в секции `data`,
- *Limit int* - необязательное поле, в котором можно указать количество запусков (пока не исполняется),
- *Till string* - необязательно поле с временем окончания задачи (пока не учитывается),
- *Conditions string* - новые права на изменение задачи.

NewAppParam

Контракт добавляет новый параметр приложения в текущей экосистеме. Параметры:

- *App int* - идентификатор приложения,
- *Name string* - имя параметра,
- *Value string* - значение параметра,
- *Conditions string* - права на изменение параметра.

EditAppParam

Контракт изменяет существующий параметр приложения в текущей экосистеме. Параметры:

- *Id int* - идентификатор параметра,
- *Value string* - новое значение параметра,
- *Conditions string* - новые права на изменение параметра.

NewDelayedContract

Контракт добавляет новое задание в планировщик запуска отложенных контрактов. Планировщик запуска отложенных контрактов запускает необходимые контракты для текущего генерируемого блока. Параметры:

- *Contract string* - название контракта, который требуется запустить,
- *EveryBlock int* - шаг в блоках, через который требуется запускать контракт,
- *Conditions string* - права на изменение задания,
- *BlockID int «optional»* - номер блока в котором требуется запустить контракт, если не указан, то рассчитывается автоматически «текущий номер блока» + $EveryBlock,
- *Limit int «optional»* - лимит кол-ва запусков задания, если лимит не указан, то задание с запуском контракта будет выполняться неограниченное кол-во раз.

EditDelayedContract

Контракт изменяет задание в планировщике запуска отложенных контрактов. Параметры:

- *Id int* - идентификатор задания,
- *Contract string* - название контракта, который требуется запустить,
- *EveryBlock int* - шаг в блоках, через который требуется запускать контракт,
- *Conditions string* - права на изменение задания,
- *BlockID int «optional»* - номер блока в котором требуется запустить контракт, если не указан, то рассчитывается автоматически «текущий номер блока» + $EveryBlock,
- *Limit int «optional»* - лимит кол-ва запусков задания, если лимит не указан, то задание с запуском контракта будет выполняться неограниченное кол-во раз,
- *Deleted int «optional»* - отключение задания, *1* - отключает, *0* - включает.

UploadBinary

Контракт добавляет/перезаписывает статичный файл в X_binaries. При вызове контракта через HTTP API, требуется использовать `multipart/form-data`, параметр `DataMimeType` будет использован из данных формы.

Параметры:

- *Name string* - название статичного файла,
- *Data bytes «file»* - содержимое статичного файла,
- *DataMimeType string «optional»* - mime тип статичного файла,
- *AppID int* - идентификатор приложения,
- *MemberID int «optional»* - идентификатор пользователя, по умолчанию 0.

Если DataMimeType не передан, то по умолчанию используется `application/octet-stream`. Если MemberID не передан, то статика является системной.

3.5 Контракты с подписью

Поскольку язык написания контрактов позволяет выполнять вложенные контракты, то существует возможность выполнения такого вложенного контракта без ведома пользователя запустившего внешний контракт, что может привести к подписи пользователем несанкционированных им транзакций, скажем перевода денег со своего счета.

К примеру, пусть имеется контракт перевода денег *MoneyTransfer*:

```
contract MoneyTransfer {
    data {
      Recipient int
      Amount    money
    }
    ...
}
```

Если в некотором контракте, запущенном пользователем, будет вписана строка MoneyTransfer(«Recipient,Amount», 12345, 100), то будет осуществлен перевод 100 монет на кошелек 12345. При этом пользователь, подписывающий внешний контракт, останется не в курсе осуществленной транзакции. Исключить такую ситуацию возможно, если контракт MoneyTransfer будет требовать получения дополнительной подписи пользователя при вызове его из других контрактов. Для этого необходимо:

1. Добавить в секцию *data* контракта *MoneyTransfer* поле с именем **Signature** с параметрами *optional* и *hidden*, которые позволяют не требовать дополнительной подписи при прямом вызове контракта, поскольку в поле **Signature** уже будет подпись.

```
contract MoneyTransfer {
    data {
      Recipient int
      Amount    money
      Signature string "optional hidden"
    }
    ...
}
```

2. Добавить в таблицу *Signatures* запись содержащую:

- имя контракта *MoneyTransfer*,
- имена полей, значения которых будут показываться пользователю, и их текстовое описание,
- текст, который будет выводиться при подтверждении.

В текущем примере достаточно указать два поля **Receipient** и **Amount**:

- **Title**: Are you agree to send money this recipient?
- **Parameter**: Receipient Text: Wallet ID
- **Parameter**: Amount Text: Amount (qEGS)

Теперь если вставить вызов контракта *MoneyTransfer(«Recipient, Amount», 12345, 100)*, то будет получена системная ошибка *«Signature is not defined»*. Если же контракт будет вызван следующим образом *MoneyTransfer(«Recipient, Amount, Signature», 12345, 100, «xxx...xxxx»)*, то возникнет ошибка при проверке подписи. При вызове контракта проверяется подпись следующих данных: ««время оригинальной транзакции, id пользователя, значение полей указанных в таблице signatures»«, и подделать эту подпись невозможно.

Для того, чтобы пользователь при вызове контракта *MoneyTransfer* увидел подтверждение на перевод денег, во внешний контракт необходимо добавить поле с произвольным названием и типом *string* и дополнительным параметром *signature:contractname*. При вызове вложенного контракта *MoneyTransfer* необходимо просто передать этот параметр. Также следует иметь в виду, что параметры для вызова защищенного контракта должны также быть описаны в секции *data* внешнего контракта (они могут быть скрытыми, но они все равно будут отображаться при подтверждении). Например,

```
contract MyTest {
  data {
      Recipient int "hidden"
      Amount    money
      Signature string "signature:send_money"
  }
  func action {
      MoneyTransfer("Recipient,Amount,Signature",$Recipient,$Amount,$Signature)
  }
}
```

При отправке контракта *MyTest*, у пользователя будет запрошено дополнительное подтверждение для перевода суммы на указанный кошелек. Если во вложенном контракте будут указаны другие значения, например *MoneyTransfer(«Recipient,Amount,Signature»,$Recipient, $Amount+10, $Signature)*, то будет получена ошибку, что подпись неверна.

3.6 Интерфейсы пользователя

3.6.1 Построение интерфейсов

Программный клиент Molis, реализованый на *библиотеке JavaScript React*, содержит редактор и визуальный конструктор интерфейсов. Страницы интерфейса являются неотъемлемой частью приложений и обеспечивают получение и отображение данных из таблиц базы данных, создание форм для получения данных от пользователя, передачу данных в контракт, а также навигацию между страницами приложений. Страницы, как и контракты, хранятся в блокчейне, что обеспечивает возможность контроля их нефальсифицированности при загрузке в программный клиент.

Шаблонизатор интерфейсов

Элементы интерфейса (страницы и меню) формируются на валидирующих узлах в так называемом *шаблонизаторе* из шаблонов, написанных программистами в редакторе интерфейсов программного клиента Molis. Используется специально написанный разработчиками платформы функциональный язык Protypo. Интерфейсы запрашиваются у узлов сети командой *content* API. Шблонизатор посылает не готовую HTML страницу, а JSON код, который представляет собой HTML тэги, расположенные в виде дерева в соответствии со структурой шаблона. Для тестирования можно отправить POST запрос в `api/v2/content` с параметром *template*, который содержит шаблон для обработки.

Создание шаблонов интерфейсов

Интерфейсы создаются и редактируются в специальном редакторе секции **Interface** административного раздела Molis. Редактор обеспечивает:

- написание кода интерфейсных страниц с подсветкой ключевых слов языка шаблонизатора Protypo,
- выбор меню, которое будет отображаться на странице,
- переход к редактированию привязанного меню,
- задание прав на редактирование страницы (традиционно через указание имени контракта с определёнными в нем правами в специальной функции *ContractConditions* или прямым указанием условий доступа в поле *Change conditions*),
- вызов визуального редактора,
- переход к просмотру страницы.

Визуальный конструктор интерфейсов

Визуальный конструктор интерфейсов позволяет создавать дизайн страниц без обращения к коду на языке Protypo. В конструкторе методом drag-and-drop настраиваются положение элементов форм и текста, размеры и стилевое оформление блоков страницы. В конструкторе имеется набор готовых блоков для отображения типовых моделей данных: панели с заголовками, формы, информационные панели. Программная логика (получение данных, условные конструкции) вводится в стандартном редакторе страниц после создания дизайна (в дальнейшем планируется реализовать полнофункциональный визуальный редактор интерфейсов).

Использование стилей

По умолчанию страницы интерфейса отображаются с использование классов *Angular Bootstrap Angle*. При необходимости можно создавать собственные стили, для хранения которых используется специальный параметр настроечной таблицы экосистемы stylesheet.

Страничные блоки

Для использования на множестве страниц типового фрагмента кода имеется возможность создавать страничные блоки и встраивать их в код интерфейса с помощью команды *Insert*. Создаются и редактируются блоки на странице *Interface* административной секции Molis. Для блоков, как и для страниц, задаются права на редактирование.

Редактор языковых ресурсов

Программный клиент Molis содержит механизм локализации интерфейса: специальная функция языка шаблонизатора LangRes подставляет в текст страницы вместо языкового ресурса (лейбла) его перевод на язык установленный в программном клиенте (или в браузере для web-версии клиента). Вместо функции LangRes также возможно использование сокращенного синтаксиса $lable$. Перевод сообщений во всплывающих окнах, инициируемых контрактами, выполняется функцией LangRes, имеющейся в языке Simvolio.

Ввод и редактирование языковых ресурсов осуществляется в специальном разделе Language resources административной секции программного клиента Molis. При создании языкового ресурса задается его лейбл (имя) и необходимое число переводов с указанием двухсимвольного идентификатора языка (en, fr и др.).

Права на добавление и изменение языковых ресурсов настраиваются стандартными для таблиц методами в таблице languages (раздел Tables административной секции Molis).

3.6.2 Язык шаблонизатора Protypo

Функции Protypo обеспечивают выполнение следующих операций:

- получение значений из базы данных: DBFind;
- представление данных, полученных и базы данных в виде таблиц и диаграмм;
- присваивание и вывод значений переменных, оперирование с данными;
- вывод и сравнение значений времени и даты;
- построение форм с необходимым набором полей для ввода данных пользователя;
- валидация данных в полях формы с выводом сообщений об ошибках;
- вывод элементов навигации;
- вызов контрактов;
- создание элементов HTML разметки страницы – различных контейнеров с возможностью указания css классов;
- встраивание изображения в страницу и загрузку изображения;
- условный вывод фрагментов шаблонов страниц: `If, ElseIf, Else`;
- создание многоуровневого меню;
- локализация интерфейсов.

Общее описание языка Protypo

Язык построения шаблонов страниц является функциональным языком, в котором вызываются функции вида `FuncName(parameters)`. Функции могут вкладываться друг в друга. Параметры можно не заключать в кавычки. Если параметр не передаётся при текущем вызове функции, его можно опустить.

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

Если параметр содержит запятую, то его нужно заключить в обратные или двойные кавычки. При этом, если параметр у функции возможен только один, то в нём можно использовать запятые не обрамляя его в кавычки. Также кавычки нужно использовать если в параметре имеется непарная закрывающая скобка.

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

Если вы заключили в кавычки параметр, в содержимом которого также используются кавычки, можно использовать разные кавычки или дублировать их в тексте.

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

При описании функций каждый параметр имеет определенное имя. Вы можете вызывать функции и указывать параметры в том порядке, в котором они описаны, а можете явно указывать только нужные параметры по их именам в любом порядке как `Имя_параметра: Значение_параметра`. Такой подход позволяет безболезненно добавлять новые параметры в функции без нарушения совместимости с текущими шаблонами. Например, если у вас есть функция, описанная как `FuncName(Class,Value, Body)`, то все эти вызовы будут корректными с точки зрения языка:

```
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)
```

Некоторые функции возвращают просто текст, некоторые создают HTML элемент (например, `Input`), а некоторые функцию создают HTML элемент с вложенными HTML элементами (`Div, P, Span`). В последнем случае для определения вложенных элементов используется параметр с предопределённым именем **Body**. Например, два элемента *div*, вложенные в другой *div*, могут выглядеть так

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

Для указания вложенных элементов, которые описываются в параметре *Body*, можно использовать слежующее представление: `FuncName(...){...}`, где в фигурных скобках указываются вложенные элементы.

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

Если подряд используется несколько одинаковых функций, то вместо имён второй и последующих можно ставить только точку. Например, следующие две строчки эквивалентны

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

В языке можно присваивать переменные с помощью функции **SetVar**. Для подстановки значений переменных используется запись `#varname#`.

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

Для подстановки языковых ресурсов экосистемы можно использовать запись `$langres$`, где *langres* имя языкового ресурса.

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

Существуют следующие предопределенные переменные:

- `#key_id#` - идентификатор-аккаунта текущего пользователя.
- `#ecosystem_id#` - идентификатор текущей экосистемы.
- `#guest_key#` - идентификатор гостевого кошелька.
- `#isMobile` - переменная равна 1, если клиент запущен на мобильном устройстве.

Передача параметров странице через PageParams

Есть ряд функций, которые принимают параметр **PageParams**. Он служит для передачи параметров при переходе на новую страницу. Например, `PageParams: "param1=value1,param2=value2"`. Параметры могут содержать как обычные строки так и строки с подстановкой значений переменных. При передаче странице параметров создаются переменные с именем параметра, например, `#param1#` и `#param2#`.

- `PageParams: "hello=world"` - страница получит параметр hello со значением world
- `PageParams: "hello=#world#"` - страница получит параметр hello со значением переменной world

Кроме этого, существует функция **Val**, которая позволяет получать данные из форм, в которые были внесены данные на момент перехода. В этом случае,

- `PageParams: "hello=Val(world)"` - страница получит параметр hello с содержимым элемента формы с именем world

Вызов контрактов

Вызов контрактов Protypo происходит при клике на кнопке формы (функция *Button*). При этом производится передача в контракт данных, введённых пользователем в поля формы (если имена полей формы и имена переменных в секции data вызываемого контракта совпадают, то данные передаются автоматически). В функции Button возможен вызов модального окна для подтверждения пользователем запуска контракта (Alert), а также инициация перехода на указанную страницу после успешного выполнения контракта с передачей странице перечисленных параметров.

3.6.3 Функции Protypo

Операции с переменными

GetVar(Name)

Функция возвращает значение указанной переменной, если она существует, и возвращает пустую строку, если переменная с данным именем не определена. Элемент с именем **getvar** создается только при запросе дерева для редактирования. Отличие `GetVar(varname)` от использования `#varname#` состоит в том, что если *varname* не существует, то *GetVar* возвратит пустую строку, а *#varname#* выведется как текст.

- *Name* - имя переменной.

```
If(GetVar(name)){#name#}.Else{Name is unknown}
```

SetVar(Name, Value)

Присваивает переменной с именем *Name* значение *Value*.

- *Name* - имя переменной,
- *Value* - значение переменной, может содержать ссылку на другие переменные.

```
SetVar(name, John Smith).(out, I am #name#)
Span(#out#)
```

VarAsIs(Name, Value)

Присваивает переменной с именем *Name* значение *Value*, но, в отличии от функции *SetVar*, не происходит подстановка переменных в значении *Value*.

- *Name* - имя переменной,
- *Value* - значение переменной, может содержать ссылку на другие переменные, но они не подставляются.

```
VarAsIs(name, I am #name#)
```

Навигация

AddToolButton(Title, Icon, Page, PageParams) [.Popup(Width, Header)]

Добавляет кнопку в панель инструментов. Создает элемент **addtoolbutton**.

- *Title* - заголовок кнопки,
- *Icon* - иконка для кнопки,
- *Page* - имя страницы для перехода,
- *PageParams* - параметры, передаваемые странице.

Popup - используется для вывода модального окна.

- *Header* - заголовок окна,
- *Width* - ширина окна в процентах, принимает значения от 1 до 100.

```
AddToolButton(Help, help, help_page)
```

Button(Body, Page, Class, Contract, Params, PageParams) [.CompositeContract(Contract, Data)] [.Alert(Text, ConfirmButton, CancelButton, Icon)] [.Popup(Width, Header)] [.Style(Style)] [.ErrorRedirect((ErrorID,PageName,PageParams)][.Action(Name, Params)]

Создает элемент HTML-формы **button**, по клику на котором инициируется выполнение контракта или переход на другую страницу.

- *Body* - дочерний текст или элементы, используется для ввода имени кнопки,
- *Page* - имя страницы для перехода,
- *Class* - классы для данной кнопки,
- *Contract* - имя вызываемого контракта,
- *Params* - список передаваемых контракту значений; по умолчанию, значения параметров контракта (секция `data`) берутся из HTML элементов (например, полей формы) с одноименными идентификаторами (`id`); если имена идентификаторов элементов, значения которых требуется передать в контракт, отличаются от имен параметров контракта, то используется присваивание парамтеров в формате `contractField1=idname1, contractField2=idname2`,
- *PageParams* - параметры для перехода на страницу в формате `contractField1=idname1, contractField2=idname2`, при этом на странице перехода создаются переменные с именами параметров `#contractField1#` и `#contractField2#` с присвоением им указанных значений (особенности передачи параметров см. в разделе выше «*Передача параметров странице через PageParams*»).

CompositeContract - используется для навешивании на кнопку дополнительных контрактов. Можно для одной кнопки указывать несколько *CompositeContract*.

- *Name* - имя контракта,
- *Data* - параметры для контракта в виде JSON массива с параметрами.

Action - используется для указании действий по наатию на кнопку. Можно для одной кнопки указывать несколько *Action*. Действия возвращаются в виде массива.

- *Name* - имя действия,
- *Params* - параметры вида *param1=val1,param2=val2*.

Alert - используется для вывода сообщений.

- *Text* - текст сообщения,
- *ConfirmButton* - текст кнопки подтверждения,
- *CancelButton* - текст кнопки отмены,
- *Icon* - иконка.

Popup - используется для вывода модального окна.

- *Header* - заголовок окна,
- *Width* - ширина окна в процентах, принимает значения от 1 до 100.

Style - служит для указания css стилей

- *Style* - css стили.

ErrorRedirect - служит для указания редиректа в случае получения ошибки, сгенерированной функцией *Throw* во время выполнения контракта. Может быть несколько вызовов *ErrorRedirect*. В результате возвращается атрибут *errredir* со списком ключей *ErrorID* и параметрами в качестве значения.

- *ErrorID* - идентификатор ошибки,
- *PageName* - имя страницы,
- *PageParams* - передаваемые параметры.

```
Button(Submit, default_page).CompisiteContract(NewPage, [{"Name":"Name of Page"},{"Value":
↪"Span(Test)"}])
Button(Submit, default_page, mybtn_class).Alert(Alert message)
Button(Submit, default_page, mybtn_class).Popup(Header: message, Width: 50)
Button(Contract: MyContract, Body:My Contract, Class: myclass, Params:"Name=myid,Id=i10,Value")
```

LinkPage(Body, Page, Class, PageParams) [.Style(Style)]

Создает элемент **linkpage** для ссылки на страницу.

- *Body* - дочерние текст или элементы,
- *Page* - имя страницы перехода,
- *Class* - классы элемента,
- *PageParams* - параметры для перехода на страницу в формате `contractField1=idname1, contractField2=idname2` (особенности передачи параметров см. в разделе выше «*Передача параметров странице через PageParams*»),

Style - служит для указания css стилей,

- *Style* - css стили.

```
LinkPage(My Page, default_page, mybtn_class)
```

Операции с данными

And(parameters)

Функция возвращает результат выполнения логической операции **И** со всеми перечисленными в скобках через запятую параметрами. Значение параметра принимается как `false`, если он равен пустой строке (`""`), 0 или `false`. Во всех остальных случаях значение параметра считается `true`. Соответственно функция возвращает 1 в случае истины и в противном случае 0. Элемент с именем `and` создается только при запросе дерева для редактирования.

```
If(And(#myval1#,#myval2#), Span(OK))
```

Calculate(Exp, Type, Prec)

Функция возвращает результат арифметического выражения, переданного в параметре **Exp**. Можно использовать операции +, -, *, / и круглые скобки ().

- **Exp** - арифметическое выражение. Может содержать числа и переменные *#name#*.

- **Type** - тип результата: **int, float, money**. Если не указан, то если есть числа с десятичной точкой, то берется тип *float*, в противном случае *int*.
- **Prec** - для типа *float* и *money* можно указать количество значащих цифр после точки.

```
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)
```

CmpTime(Time1, Time2)

Функция сравнивает два значения времени в одинаковом формате (поддерживается формат unixtime, строковый - `YYYY-MM-DD HH:MM:SS`, а также можно и в произвольном при условии соблюдения последовательности от годов к секундам, например, `YYYYMMDD`). Возвращает:

- **-1** - Time1 < Time2,
- **0** - Time1 = Time2,
- **1** - Time1 > Time2.

```
If(CmpTime(#time1#, #time2#)<0){...}
```

DateTime(DateTime, Format, Location)

Функция выводит на экран значение даты и времени в заданном формате.

- *DateTime* - время в unixtime или стандартном формате `2006-01-02T15:04:05`.
- *Format* - шаблон формата : `YY` короткий год, `YYYY` полный год, `MM` - месяц, `DD` - день, `HH` - часы, `MM` - минуты, `SS` – секунды, например, `YY/MM/DD HH:MM`. Если формат не указан, то будет использовано значение параметра *timeformat* определенное в таблице *languages*, если его нет, то `YYYY-MM-DD HH:MI:SS`.
- *Location* - временная зона, список поддерживаемых временных зон содержится в системной таблице @1time_zones.

```
DateTime(2017-11-07T17:51:08)
DateTime(#mytime#,HH:MI DD.MM.YYYY)
DateTime(DateTime: 1560938400, Location: "Europe/Moscow")
```

Money(Exp, Digit)

Функция возвращает значение Exp/10^Digit. Если параметр Digit не указан, то он будет браться из параметра **money_digit** экосистемы.

- *Exp* - Числововое значение,
- *Digit* - степень 10 в выражении Exp/10^Digit. Может быть как положительным, так и отрицательным. В случае положительного значения определяет количество цифр после запятой.

```
Money(#val#, #digit#)
Money(123456723722323332)
```

Or(parameters)

Функция возвращает результат выполнения логической операции **ИЛИ** со всеми перечисленными в скобках через запятую параметрами. Значение параметра принимается как `false`, если он равен пустой строке (`""`), 0 или `false`. Во всех остальных случаях значение параметра считается `true`. Соответственно функция возвращает 1 в случае истины и в противном случае 0. Элемент с именем **or** создается только при запросе дерева для редактирования.

```
If(Or(#myval1#,#myval2#), Span(OK))
```

Отображение данных

Code(Text)

Создает элемент **code** для вывода указанного кода. При этом переменные *#name#* будут заменены на их значения.

- *Text* - исходный код, который необходимо вывести.

```
Code( P(This is the first line.
    Span(This is the second line.))
)
```

CodeAsIs(Text)

Создает элемент **code** для вывода указанного кода, но при этом, в отличии от функции Code, не происходит замена переменных вида *#name#*.

- *Text* - исходный код, который необходимо вывести.

```
CodeAsIs( P(This is the #test1#.
    Span(This is the #test2#.))
)
```

Chart(Type, Source, FieldLabel, FieldValue, Colors)

Создает HTML диаграмму.

- *Type* - тип диаграммы,
- *Source* - имя источника данных, например, из команды *DBFind*,
- *FieldLabel* - название поля, используемого для заголовков,
- *FieldValue* - название поля, используемого для значений,
- *Colors* - список используемых цветов

```
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")
```

ForList(Source, Index){Body}

Выводит список элементов из источника данных *Source* в формате шаблона, заданного в *Body*. Создает элемент **forlist**.

- *Source* - источник данных из функций *DBFind* или *Data*,
- *Index* - можно указать имя переменной для счетчика итераций, счет ведется с 1. Если параметр *Index* не указан, то счетчик будет записываться в переменную *[Source]_ index*,
- *Body* - шаблон, задающий формат вывода элементов списка.

```
ForList(mysrc){Span(#mysrc_index#. #name#)}
```

Hint(Icon,Title,Text)

Создает элемент **hint** для вывода подсказки.

- *Icon* - имя иконки,
- *Title* - заголовок подсказки,
- *Text* - текст подсказки.

```
Hint(myicon, My Header, This is a hint text)
```

Image(Src,Alt,Class) [.Style(Style)]

Создает HTML элемент **image**.

- *Src* - источник изображения, файл или `data:...`,
- *Alt* - альтернативный текст для изображения,
- *Class* - список классов.

```
Image(\images\myphoto.jpg)
```

MenuGroup(Title, Body, Icon)

Функция формирует в меню вложенное подменю и возвращает элемент **menugroup**. В параметре *name* также будет возвращено значение *Title* до подстановки языковых ресурсов.

- *Title* - имя пункта меню,
- *Body* - дочерние элементы подменю,
- *Icon* - иконка.

```
MenuGroup(My Menu){
    MenuItem(Interface, sys-interface)
    MenuItem(Dahsboard, dashboard_default)
}
```

MenuItem(Title, Page, Params, Icon, Obs)

Служит для создания пункта меню и возвращает элемент **menuitem**.

- *Title* - имя пункта меню,
- *Page* - имя страницы перехода,
- *Params* - параметры, передаваемые странице в формате *var:value* через запятую,
- *Icon* - иконка,
- *Obs* - параметр, определяющий переход на оффчейн-сервер; если `Obs: true`, то ссылка ведёт в OBS, если `Obs: false`, то в блокчейн, если параметр не указан, то решается в зависимости от того, где было загружено меню.

```
MenuItem(Interface, interface)
```

QRcode(Text)

Возвращает элемент *qrcode* с указанным текстом для генерации QR кода.

- *Text* - текст для генерации QR кода.

```
QRcode(#name#)
```

Table(Source, Columns) [.Style(Style)]

Создает HTML элемент **table**.

- *Source* - имя источника данных, например, из команды *DBFind*,
- *Columns* - заголовки и соответствующие имена колонок в виде `Title1=column1,Title2=column2`.

Style - служит для указания css стилей,

- *Style* - css стили.

```
DBFind(mytable, mysrc)
Table(mysrc,"ID=id,Name=name")
```

Получение данных

Address(account)

Функция возвращает адрес аккаунта в формате `1234-5678-...-7990` по числовому значению адреса; если адрес не указан, то в качестве аргумента принимается значение адреса текущего владельца аккаунта.

```
Span(Your wallet: Address(#account#))
```

AddressToId(Wallet)

Функция возвращает числовой идентификатор владельца аккаунта по строковому значению адреса аккаунта.

- *Wallet* - адрес аккаунта в формате XXXX-...-XXXX или в виде числа.

```
AddressToId(#wallet#)
```

AppParam(App, Name, Index, Source)

Функция выводит на экран значение параметра приложения из таблицы app_param текущей экосистемы. Если есть языковый ресурс с полученным именем, то автоматически подставится его значение.

- *App* - идентификатор приложения,
- *Name* - имя параметра,
- *Index* - порядковый номер элемента параметра (начиная с 1) в случае, если значение параметра представлено списоком через запятую, например, `type = full,light`, тогда `AppParam(1, type, 2)` возвратит *light*. Этот параметр не совместим с параметром *Source*,
- *Source* - создается объекта *data* со элементами значения параметра, представленого списоком через запятую; объект указывается как источник данных в функциях *Table* и *Select* (в этом случае функция не будет возвращать значение). Этот параметр не совместим с парарметром *Index*.

```
AppParam(1, type, Source: mytype)
```

Data(Source,Columns,Data) [.Custom(Column){Body}]

Создает элемент **data**, заполняет его перечисленными в параметрах данными и помещает в конструкцию *Source*, которая потом указывается в *Table* и других командах, получающих *Source* в качестве входных данных. Последовательность записей в *data* соотвествует последовательности имен колонок.

- *Source* - произвольное имя источника данных,
- *Columns* - список имен колонок через запятую,
- *Data* - данные по одной записи на строку с разделением на колонки через запятую; при наличии запятых, значение заключается в двойные кавычки, при наличии кавычек в значении, оно заключается в удвоеные двойные кавычки,
- **Custom** - создает в источнике дополнительные колонки, для вывода данных, вычисляемых из значений основных колонок, например, кнопки, ссылки; допускается определять несколько столбцов *Custom*; используется для вывода в *Table* и других командах, получающих *Source* в качестве входных данных
  - *Column* - произвольное имя колонки,
  - *Body* - шаблон, можно использовать значения из других колонок текущей записи с помощью переменных `#columnname#`.

```
Data(mysrc,"id,name"){
    "1",John Silver
    2,"Mark, Smith"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   3,"Unknown ""Person"""
}.Custom(link){Button(Body: View, Class: btn btn-link, Page: user, PageParams: "id=#id#"}
```

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)]

Создает элемент **dbfind**, заполняет его данными, полученными из таблицы *table*, и помещает его в конструкцию *Source*, которая потом указывается в *Table* и других командах, получающих *Source* в качестве входных данных. Последовательность записей в *data* должна соотвествовать последовательности имен колонок.

- *table* - имя таблицы,
- *Source* - произвольное имя источника данных,
- **Columns** - список возвращаемых колонок; если не указано, то возвратятся все колонки. Если имеются колонки типа JSON, то вы можете использовать обращение к полям записи с помощью записи вида **имяколонки->имяполя**. В этом случае, имя результирующей колонки будет **имяколонки.имяполя**. Также, формат запросов для поля Columns описан в разделе Смарт-контракты - Язык написания контрактов Symvolio - Получение значений из таблиц базы данных - DBFind.
- **Where** - условие поиска данных. Формат запросов описан в разделе Смарт-контракты - Язык написания контрактов Symvolio - Получение значений из таблиц базы данных - DBFind. Если имеются колонки типа JSON, то вы можете использовать обращение к полям записи с помощью записи вида **имяколонки->имяполя**.
- **WhereId** - условие поиска по идентификатору, например, `.WhereId(1)`,
- **Order** - поле, по которому происходит отсортировать. Формат запросов для поля Order описан в разделе Смарт-контракты - Язык написания контрактов Symvolio - Получение значений из таблиц базы данных - DBFind,
- **Limit** - количество возвращаемых записей - по умолчанию - 25, максимально возможное - 250,
- **Offset** - смещение первой возвращаемой записи,
- **Count** - имеется возможность вместе с данными получить общее кол-во записей для данного условия *where*. Для этого следует добавить этот вызов и в параметре указать имя переменной, куда будет записано общее количество записей по указанной выборке. Также, это значение возвратится в параметре *count* у элемента *dbfind*. Если не были определены *Where* и *WhereId*, то будет возвращаться общее кол-во записей в таблице,
- **Ecosystem** - идентификатор экосистемы; по умолчанию, берутся данные из таблицы в текущей экосистеме,
- **Cutoff** - служит для обрезания и представления в виде ссылок больших текстовых данных; в параметре необходимо в виде строки через запятую указать список колонок, которые будут обрабатываться таким образом. В значении колонки возвращается json объект в двумя полями - *link* и *title*. Если значение колонки больше 32 символов, то возвращается ссылка на полный текст и первые 32 символа. В противном случае, ссылка пустая и возвращается всё значение.
- **Custom** - определяет дополнительные колонки, для вывода данных, вычисляемых их значений основных колонок, например, кнопки, ссылки; допускается определять несколько столбцов *Custom*; используется для вывода в *Table* и других командах, получающих *Source* в качестве входных данных

  - *Column* - произвольное имя колонки.
  - *Body* - шаблон, можно использовать значения из других колонок текущей записи с помощью переменных `#columnname#`.

- **Vars** - функция формирует множество переменных со значениями из первой записи, полученной по данному запросу (параметр *Limit* автоматически становится равным 1),
  - *Prefix* - префикс имен формируемых переменных, то есть переменные имеют вид `#prefix_id#, #prefix_name#`, где после знака подчеркивания указывается имя колонки таблицы. Если есть колонки из JSON полей, то тогда результирующая переменная будет иметь вид #**prefix_columnname_field**#.

```
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#)
}
```

EcosysParam(Name, Index, Source)

Функция выводит на экран значение параметра из таблицы parameters текущей экосистемы. Если есть языковый ресурс с полученным именем, то автоматически подставится его значение.

- *Name* - имя параметра,
- *Index* - порядковый номер элемента параметра (начиная с 1) в случае, если значение параметра представлено списоком через запятую, например, `gender = male,female`, тогда `EcosysParam(gender, 2)` возвратит *female*. Этот параметр не совместим с парарметром *Source*,
- *Source* - создается объекта *data* со элементами значения параметра, представленого списоком через запятую; объект указывается как источник данных в функциях *Table* и *Select* (в этом случае функция не будет возвращать значение). Этот параметр не совместим с парарметром *Index*.

```
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)
```

GetHistory(Source, Name, Id, RollbackId)

Создает элемент **gethistory**, заполняет его данными с историей изменений указанной записи в таблице с именем **Name**, и помещает его в конструкцию *Source*, которая потом указывается в *Table* и других командах, получающих *Source* в качестве входных данных. Результирующий список отсортирован от последних изменений к более ранним. В результирующей таблице поле *id* указывает на id в таблице *rollback_tx*. Также возвращаются поля *block_id* - номер блока, *block_time* - время блока.

- *Source* - произвольное имя источника данных,
- *Name* - имя таблицы,
- *Id* - идентификатор записи.
- *RollbackId* - необязательный параметр. Если указан, то возвратится только одна запись с данным id в таблице *rollback_tx*.

```
GetHistory(blocks, BlockHistory, 1)
```

GetColumnType(Table, Column)

Функция возвращает тип указанной колонки в указанной таблице. Возвращается наименование внутреннего типа -например, *text,varchar,number,money,double,bytea,json,datetime,double*.

- *Table* - имя таблицы,
- *Column* - имя колонки.

```
SetVar(coltype,GetColumnType(members, member_name))Div(){#coltype#}
```

JsonToSource(Source, Data)

Создает элемент **jsontosource**, заполняет его парами *key* и *value*, которые переданы в json объекте в *Data* и помещает в конструкцию *Source*, которая потом указывается в *Table* и других командах, получающих *Source* в качестве входных данных. Записи в результирующих данных отсортированы в алфавитном порядке по ключам JSON объекта.

- *Source* - произвольное имя источника данных,
- *Data* - может указываться как JSON объект так и переменная *#name#*, которпая содержит JSON объект.

```
JsonToSource(src, #myjson#)
JsonToSource(dat, {"param":"value", "param2": "value 2"})
```

ArrayToSource(Source, Data)

Создает элемент **arraytosource**, заполняет его парами *key* и *value*, которые переданы в json массиве в *Data* и помещает в конструкцию *Source*, которая потом указывается в *Table* и других командах, получающих *Source* в качестве входных данных.

- *Source* - произвольное имя источника данных,
- *Data* - может указываться как JSON массив так и переменная *#name#*, которпая содержит JSON массив.

```
ArrayToSource(src, #myjsonarr#)
ArrayToSource(dat, [1, 2, 3])
```

LangRes(Name, Lang)

Возвращает указанный языковой ресурс. В случае запроса дерева для редактирования возвращается элемент **langres**. Возможно использование сокращенной записи вида `$langres$`.

- *Name* - имя языкового ресурса,
- *Lang* - двухсивольный идентификатор языка; по умолчанию, возвращается язык который определен в запросе в *Accept-Language*. Также можно указывать lcid идентификаторы, например, *en-US,en-GB*. В этом случае, если не будет найдено соответствие, например, для *en-US*, то ресурс будет искаться для *en*.

```
LangRes(name)
LangRes(myres, fr)
```

Range(Source,From,To,Step)

Создает элемент **range**, заполняет его целочисленными значениями от *From* до *To* (не включая *To*) с шагом *Step* и помещает в конструкцию *Source*, которая потом указывается в *Table* и других командах, получающих *Source* в качестве входных данных. Значения записываются в колонку с именем *id*. Если будут указаны неверные параметры, то будет возвращен пустой источник.

- *Source* - произвольное имя источника данных,
- *From* - начальное значение (i = From),
- *To* - конечное значение (i < To),
- *Step* - шаг изменения значения. Если не указан, то берется 1.

```
Range(my,0,5)
SetVar(from, 5).(to, -4).(step,-2)
Range(Source: neg, From: #from#, To: #to#, Step: #step#)
```

SysParam(Name)

Функция выводит значение системного параметра из таблицы system_parameters.

- *Name* - имя значения.

```
Address(SysParam(founder_account))
```

Binary(Name, AppID, MemberID)[.ById(ID)][.Ecosystem(ecosystem)]

Функция возвращает ссылку на статичный файл, который хранится в таблице binaries.

- *Name* - имя файла,
- *AppID* - идентификатор приложения,
- *MemberID* - идентификатор пользователя, по умолчанию 0,
- *ID* - идентификатор статичного файла.
- *ecosystem* - идентификатор экосистемы, с которой запрашиваеются двоичные данные. Если не указан, то запрашиваются из текущей экосистемы.

```
Image(Src: Binary("my_image", 1))
Image(Src: Binary().ById(2))
Image(Src: Binary().ById(#id#).Ecosystem(#eco#))
```

TransactionInfo(Hash)

Функция ищет транзакцию по указанному хэшу и возвращает информацию о вызванном контракте и его параметрах. Функция возвращает строку в формате json *{«contract»:»ContractName»,*

«params»:{«key»: «val»}, «block»: «N»}, где в поле *contract* возвращается имя контракта, *params* - переданные параметры, *block* - номер блока в котором была обработана данная транзакция.

- *hash* - хэш транзакции в виде шестнадцатеричной строки.

```
P(TransactionInfo(#hash#))
```

Элементы форматирования данных

Div(Class, Body) [.Style(Style)] [.Show(Condition)] [.Hide(Condition)]

Создает HTML элемент **div**.

- *Class* - классы для данного *div*,
- *Body* - дочерние элементы.

Style - служит для указания css стилей

- *Style* - css стили.

Show - Определяет условия, при каких следует показывать данный блок. **Hide** - Определяет условия, при каких следует скрывать данный блок.

- *Condition* - можно через запятую перечислить условия в виде пар InputName=Value. Условие будет выполнено, когда для каждой из пар текст соответствующего input равен указанному значению. Если указано несколько вызовов *Show* или *Hide*, то в этом случае действие будет применимо, если выполнено условие хотя бы одного вызова.

```
Div(class1 class2, This is a paragraph.).Show(inp1=test,inp2=none)
```

Em(Body, Class)

Создает HTML элемент **em**.

- *Body* - дочерний текст или элементы,
- *Class* - классы для элемента.

```
This is an Em(important news).
```

P(Body, Class) [.Style(Style)]

Создает HTML элемент **p**.

- *Body* - дочерние текст или элементы,
- *Class* - классы для элемента,

Style - служит для указания css стиле,

- *Style* - css стили.

```
P(This is the first line.
  This is the second line.)
```

SetTitle(Title)

Устанавливает заголовок страницы. Создается элемент с именем **settitle**.

- *Title* - заголовок страницы.

```
SetTitle(My page)
```

Label(Body, Class, For) [.Style(Style)]

Создает HTML элемент **label**.

- *Body* - дочерниuу текст или элементы,
- *Class* - классы элемента,
- *For* - значение *for* для данного *label*,

Style - служит для указания css стилей,

- *Style* - css стили.

```
Label(The first item).
```

Span(Body, Class) [.Style(Style)]

Создает HTML элемент **span**.

- *Body* - дочерние текст или элементы,
- *Class* - классы для элемента,

Style - служит для указания css стилей,

- *Style* - css стили.

```
This is Span(the first item, myclass1).
```

Strong(Body, Class)

Создает HTML элемент **strong**.

- *Body* - дочерние текст или элементы,
- *Class* - классы для элемента.

```
This is Strong(the first item, myclass1).
```

Элементы форм

Form(Class, Body) [.Style(Style)]

Создает HTML элемент **form**.

- *Class* - классы для данного *form*,

- *Body* - дочерние элементы.

Style - служит для указания css стилей

- *Style* - css стили.

```
Form(class1 class2, Input(myid))
```

ImageInput(Name, Width, Ratio, Format)

Создает элемент **imageinput** для загрузки картинок. По желанию в третьем параметре можно указать либо высоту картинки, либо отношение сторон в виде *1/2*, *2/1*, *3/4* и т.п. По умолчанию берется ширина в 100 пикселей и отношение сторон *1/1*.

- *Name* - имя элемента,
- *Width* - ширина вырезаемого изображения,
- *Ratio* - отношение сторон (ширины к высоте) или высота картинки,
- *Format* - формат загружаемой картинки.

```
ImageInput(avatar, 100, 2/1)
```

Input(Name,Class,Placeholder,Type,Value,Disabled) [.Validate(validation parameters)] [.Style(Style)]

Создает HTML элемент **input**.

- *Name* - имя элемента,
- *Class* - классы элемента,
- *Placeholder* - *placeholder* для элемента,
- *Type* - типа элемента,
- *Value* - значение элемента.
- *Disabled* - задизейблен ли элемент.

Validate - параметры валидации.

Style - служит для указания css стилей

- *Style* - css стили.

```
Input(Name: name, Type: text, Placeholder: Enter your name)
Input(Name: num, Type: text).Validate(minLength: 6, maxLength: 20)
```

InputErr(Name,validation errors)]

Создает элемент **inputerr** с текстами для ошибок валидации.

- *Name* - имя соответствующего элемента **Input**.

```
InputErr(Name: name,
    minLength: Value is too short,
    maxLength: The length of the value must be less than 20 characters)
```

RadioGroup(Name, Source, NameColumn, ValueColumn, Value, Class) [.Validate(validation parameters)] [.Style(Style)]

Создает элемент **radiogroup**.

- *Name* - имя элемента,
- *Source* - имя источника данных из функций *DBFind* или *Data*,
- *NameColumn* - имя колонки, из которой получаются имена элементов,
- *ValueColumn* - имя колонки, из которой получаются значения элементов; в этом параметре нельзя указывать имена колонок созданных через Custom,
- *Value* - значение по умолчанию,
- *Class* - классы для элемента.

Validate - параметры валидации.

Style - служит для указания css стилей

- *Style* - css стили.

```
DBFind(mytable, mysrc)
RadioGroup(mysrc, name)
```

Select(Name, Source, NameColumn, ValueColumn, Value, Class) [.Validate(validation parameters)] [.Style(Style)]

Создает HTML элемент **select**.

- *Name* - имя элемента,
- *Source* - имя источника данных, например, из команды *DBFind* или *Data*,
- *NameColumn* - имя колонки, из которой будет браться текст для элементов,
- *ValueColumn* - имя колонки, из которой будут браться значения для элементов; в этом параметре нельзя указывать имена колонок созданных через Custom,
- *Value* - значение по умолчанию,
- *Class* - классы для элемента,

Validate - параметры валидации,

Style - служит для указания css стилей,

- *Style* - css стили.

```
DBFind(mytable, mysrc)
Select(mysrc, name)
```

InputMap(Name, Type, MapType, Value)

Создаёт текстовое поле ввода адреса с возможностью визуального выбора координат на карте.

- *Name* - имя элемента

- *Value* - значение по умолчанию, объект в виде строки, например (*{«coords»:[{«lat»:number,»lng»:number},]}* или *{«zoom»:int, «center»:{«lat»:number,»lng»:number}}*. Также поддерживается поле address для сохранения значения адреса(поскольку при отрисовке InputMap с предустановленным Value - поле для ввода адреса не должно быть пустым)
- *Type* - «polygon»
- *MapType* - тип карты. Одно из значений: hybrid, roadmap, satellite, terrain

```
InputMap(Name: Coords,Type: polygon, MapType: hybrid, Value: `{"zoom":8, "center":{"lat":55.
↪749942860682545,"lng":37.6207172870636}}`)
```

Map(Hmap, MapType, Value)

Создаёт визуальное отображение карты для отображения координат в произвольном формате.

- *Hmap* - высота HTML-элемента на странице, по умолчанию 100.
- *Value* - значение, объект в виде строки, например (*{«coords»:[{«lat»:number,»lng»:number},]}* или *{«zoom»:int, «center»:{«lat»:number,»lng»:number}}*. Если center не указан явно, то окно отображения карты будет автоматически подстроено для того, чтобы выбранные координаты «вписались» в него.
- *MapType* - тип карты. Одно из значений: hybrid, roadmap, satellite, terrain

```
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, Московская обл., Россия, 143041"})
```

Операции с кодом

If(Condition){ Body } [.ElseIf(Condition){ Body }] [.Else{ Body }]

Условный оператор. Возвращаются дочерние элементы первого `If` или `ElseIf` у которого выполнено условие `Condition`. В противном случае, возвращаются дочерние элементы `Else`, если он присутствует.

- *Condition* - условие; считается не выполненным если равно *пустой строке* (`""`), `0` или `false`, в остальных случаях считается истинным,
- *Body* - дочерние элементы.

```
If(#value#){
   Span(Value)
}.ElseIf(#value2#){Span(Value 2)
}.ElseIf(#value3#){Span(Value 3)}.Else{
   Span(Nothing)
}
```

Include(Name)

Команда вставляет в код страницы шаблон блока с именем *Name*.

- *Name* - имя блока.

```
Div(myclass, Include(mywidget))
```

3.6.4 Стили для мобильного приложения

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>`

3.7 Приложения Apla

Приложение платформы - это совокупность контрактов, таблиц и интерфейсов совместно реализующих определенный сервис. Приложение не представляет собой автономный программный модуль - элементы приложения объединены только выполнением общего функционала и обменом данными. Границы приложения не всегда строго определены поскольку его элементы могут одновременно использоваться в нескольких приложениях.

Ниже описан типичный функциональный фрагмент приложения:

1. Переход на страницу, на которой отражается

- полученная из таблиц базы данных информация (функция *DBFind*),
- поля формы для ввода новых данных,
- кнопка (функция *Button*), по которой вызывается контракт.

2. При вызове контракта:

- в секцию `data` передаются данные из полей формы,
- в секции `conditions` проверяются права пользователя на запуск данного контракта и валидность полученных со страницы данных, при недопустимости выполнения контракта по каким-либо причинам выдается сообщение (`error`, `warning` или `info`), пользователь остается на исходной странице,
- в секции `action` получаются дополнительные данные из таблиц (функция DBFind) и выполняется одна или несколько записей в базу данных (*DBUpdate*, *DBInsert*).

3. После успешного выполнения контракта происходит переход на страницу, имя которой было указано в параметре *Page* функции функция *Button*, запустившей контракт, с передачей странице параметров, перечисленных в *PageParams*.

Страница может иметь несколько кнопок для вызова разных контрактов. Кнопки, вызывающие контракты и реализующие переход на страницы, могут встраиваться в строки таблиц, отображающих в интерфейсе данные об объекта. В этом случае в параметрах кнопок используются идентификаторы объектов, что, например, можно использовать для перехода на страницу редактирования объекта.

3.7.1 Таблицы и особенности хранения данных

Таблицы базы данных, используемые в приложении, условно можно разбить на два типа:

1. таблицы реестры, в которых хранятся большие массивы структурированных данных об объектах (персонах, организациях, объектах недвижимости и пр.),
2. таблицы документов, в которых фиксируются состояния реализуемых приложением бизнес или иных процессов (тип процесса, стадия, подписи и пр.) или данные о текущих операциях (оповещениях, сообщениях, записях о голосованиях, сделках).

Таблицы реестров обычно содержат актуальную информацию, то есть записи об объектах редактируются при поступлении новых данных. При работе с документами реализуется принципиально иной подход. Поскольку функции *DBFind* языков Simvolio и Protypo могут обращаться только к одной таблице (то есть не поддерживают *JOIN*), то при создании таблиц, хранящих документы, следует записывать в них исчерпывающую информацию (названия, имена, изображения). К примеру, обращаясь к таблице, хранящей сообщения, мы должны получить не `id` пославшего сообщение пользователя, а полную информацию, необходимую для отображения сообщения на странице, включая имя и аватарку пользователя. Данные в таблицах документов не должны изменяться. Следует особо отметить, что отказ от нормализации данных в таблицах, связан не только с техническими ограничениями на использование *JOIN*, а в большей степени, предписывается самой идеологией блокчейна как темпоральной базой данных, призванной хранить полную историю событий. Это означает, что документ (скажем, сообщение), подписанный и сохраненный в таблице не должен ни при каких условиях быть изменен в будущем, скажем, при смене имени его автора в реестре пользователей (что неизбежно при реляционной схеме данных).

3.7.2 Навигация

Переходы между отдельными приложениями осуществляется либо с помощью разделов, отображаемых в программном клиенте вкладками, либо внутри одного раздела с помощью вложенных меню. При клике на вкладке раздела осуществляется переход на главную страницу раздела, которая задается в административной зоне.

Навигация внутри приложения прежде всего осуществляется с помощью меню (к каждой странице привязывается одно меню). Возможен переход между страницам по ссылкам (*LinkPage*) или кликами на кнопках (*Button*). В обоих случаях доступна передача параметров *PageParams* целевой странице *Page*. Вызов новой страницы также возможен и после успешного выполнения контракта. Для этого в параметрах функции *Button*, с помощью которой вызывается контракт, необходимо указать имя страницы перехода *Page* и передаваемые параметры *PageParams*.

В многопользовательских приложениях удобно организовывать навигацию с помощью установленных по умолчанию в экосистемах приложений *Notification* и *Roles*. Сообщения приложени *Notification* для конкретного пользователя или представителей заданной роли отображаются в программном клиенте Molis. Сообщение содержит заголовок и ссылку на заданную страницу. Также возможно передать адресной странице параметры *PageParams* в формате, принимаемом функциями *LinkPage* и *Button*. Нередки ситуации, когда на адресную страницу, например, на которой пользователю надо принять некоторое решение, возможно попасть только через оповещение - никакие ссылки из меню или с других страниц на нее не ведут. Оповещение формируется контрактами *Notifications_Single_Send* или *Notifications_Roles_Send*, которые запускаются из контракта, завершающего некоторый функциональный этап приложения. После получения пользователем оповещения, перехода на адресную страницу и выполнения им необходимого действия, то есть запуска требуемого контракта, оповещение гасится вызываемыми в этом контракте специальными контрактами *Notifications_Single_Close* или *Notifications_Roles_Finishing*. Список и статус оповещений отображается на специальной странице приложения *Notification*.

3.7.3 Интерфейсные страницы

Структура страницы формируется с помощью условного оператора `If(Condition){Body } . ElseIf(Condition){Body} .Else{Body}`, в котором в условиях используются параметры PageParam передаваемые странице при ее вызове функциями LinkPage и Button. При необходимости использовать на многих страницах идентичные фрагменты кода, их необходимо записывать в страничные блоки. Вызываются блоки функцией *Include*.

3.7.4 Переменные, названия колонок и языковые ресурсы

Значительно ускоряет программирование приложений и упрощает чтение кода унификация имен переменных (на страницах и в контрактах), идентификаторов полей страничных форм, имен колонок таблиц и лейблов языковых ресурсов. Если имя поля формы username совпадает с именем переменной username в секции data контракта, в которую передается значение из данного поля, то эту пару (`username=username`) не обязательно указывать в параметрах *Params* в функции *Button*. Совпадение имен переменных и имен колонок упрощает написание функций DBInsert и DBUpdate, например, `DBUpdate("member", $id, {username: $username})`. Совпадение имен переменных и лейбла языкового ресурса удобно при выводе названий колонок интерфейсных таблиц `Table(mysrc,"ID=id, $username$=username")`.

3.7.5 Права доступа

Важнейшей составляющей приложения является система управления правами доступа к его ресурсам. Права устанавливаются на нескольких уровнях:

1. Разрешение на вызов конкретного контракта текущим пользователем. Разрешение определяется в секции `conditions` контракта логическим выражением в конструкции `If` или вложенными контрактами, например, *MainConditions*, *RoleConditions*, в которых определяются типовые права или права представителей ролей.

2. Разрешение текущему пользователю изменять с помощью контрактов значения в колонках таблицы, добавлять в таблицы строки и колонки. Разрешение устанавливается функцией *ContractConditions* в полях *Permissions* колонок таблиц и в полях *Write permissions* / *Insert* / *Update* / *New column* на странице редактирования таблицы. Условия прописанные в поле *Update* задают права на изменение в целом всех колонок таблицы, условия в полях *Permissions* накладывают дополнительные ограничения для каждой колонки в отдельности.

3. Разрешение на изменение значений в колонках таблицы или добавление в таблицы строк только для конкретных контрактов. Имена контрактов указывается в параметрах функции *ContractAccess*, которая вписывается в поля *Permissions* колонок таблиц и в поле *Permissions* / *Insert* на странице редактирования таблицы.

4. Разрешение на редактирование элементов приложения (контрактов, страниц, меню, страничных блоков). Разрешение задается в полях *Change conditions* в редакторах элементов. Делается это с помощью функции *ContractConditions*, которой в качестве параметра передается имя контракта, проверяющего права текущего пользователя.

3.7.6 Пример приложения SendTokens

Приложение реализует пересылку токенов с одного пользовательского аккаунта на другой. Суммы токенов на аккаунтах фиксируются в таблицах *keys* (колонка *amount*), устанавливаемых в экосистемах по умолчанию. В примере подразумевается, что токены уже распределены по аккаунтам.

Системный контракт

Основным для этого приложения является контракт *TokenTransfer*, которому предоставляется исключительное право изменять значения в колонке *amount* таблицы *keys*. Для реализации этого права в поля *Permissions* колонки записывается функция `ContractAccess("TokenTransfer")`. Теперь все операции с токенами возможны только через вызов TokenTransfer.

Чтобы избежать вызов контракта TokenTransfer внутри другого контракта незаметно от владельца аккаунта, TokenTransfer должен быть оформлен как контракт с подтверждением, то есть в секции

`data` у него должна быть строка `Signature string "optional hidden"`, а на странице *Контракты с подтверждением* административного раздела Molis должны быть введены параметры подтверждения: текст, выводимый в сплывающем окне, и отображаемые в окне параметры (подробнее см. *Контракты с подтверждением*).

```
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})
}
}
```

В секции conditions контракта TokenTransfer проверяется наличие аккаунтов, неравенство нулю переводимого количества токенов и баланс аккаунта, с которого производится перевод. В секции action производится изменение значений в колонке amount аккаунтов отправителя и получателя.

Форма отправки токенов

Форма для отправки токенов содержит поля для ввода суммы токенов и адреса аккаунта получателя.

```
Div(Class: panel panel-default){
  Form(){
    Div(Class: list-group-item text-center){
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      Span(Class: h3, Body: LangRes(SendTokens))
    }
    Div(Class: list-group-item){
      Div(Class: row df f-valign){
        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)
      }
    }
  }
}
```

В функции Button возможно было бы сразу вызвать контракт TokenTransfer с передачей ему адреса аккаунта текущего пользователя, который переводит токены, но для демонстрации работы контрактов с подтверждением создадим промежуточный пользовательский контракт SendTokens. Отметим, что поскольку названия данных в секции data контракта и имена полей формы совпадают, то в функции Button не указаны передаваемые параметры Params.

Форма может быть размещена на любой странице в программного клиента. После выполнения контракта пользователь останется на текущей странице (в Button не указана адресная страница Page).

Пользовательский контракт

Поскольку TokenTransfer определен как контракт с подтверждением, то для его вызова из другого контракта необходимо в секции data иметь строку Signature string «signature:TokenTransfer». В секции conditions контракта SendTokens проверяется наличие аккаунта, а в action вызывается контракт TokenTransfer с передачей ему параметров.

```
contract SendTokens {
    data {
        Amount money
        Recipient_Account string
        Signature string "signature:TokenTransfer"
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    conditions {
        $recipient = AddressToId($Recipient_Account)
        if $recipient == 0 {
            error Sprintf("Recipient %s is invalid", $Recipient_Account)
        }
    }

    action {
        TokenTransfer("Amount,Sender_AccountId,Recipient_AccountId,Signature", $Amount, $key_id,
↪$recipient, $Signature)
    }
}
```

3.8 Компилятор и виртуальная машина

В данном разделе рассматривается код организация и работа кода в директории packages/script, который относится к компиляции и работе виртуальной машины языка Simvolio. Документация предназначена в первую очередь для разработчиков.

Работа с контрактами организована следующим образом: контракты, функции пишутся на языке Simvolio и хранятся в таблицах contracts в экосистемах. При запуске программы происходит чтение исходного кода из базы данных и компиляция его в байт-код. При добавлении или изменении контрактов и записи их в блокчейн, обновляемые данные компилируются и добавляется/обновляется соответствующий байт-код в виртуальной машине. Физически байт код никуда не сохраняется, при выходе из программы и новом запуске компиляция происходит заново. Виртуальная машина представляет из себя набор объектов - контракты, функции, типы и т.п. Весь исходный код, описанный в таблице contracts у всех экосистем, компилируется в строгой последовательности в одну виртуальную машину и на всех нодах состояние виртуальной машины одно и тоже. При вызове контракта сама виртуальная машина никак не изменяет своего состояния. Любое выполнение контракта или вызов функции происходят на отдельном runtime стеке, который создается при каждом внешнем вызове. Каждая экосистема может иметь так называемую виртуальную экосистему, которая работает со своими таблицами вне блокчейна, в рамках одной ноды, и напрямую не может оказывать влияния на блокчейн или другие виртуальные экосистемы. В этом случае нода, которая хостит такую виртуальную экосистему, компилирует её контракты и создает для неё свою виртуальную машину.

3.8.1 Виртуальная машина

Рассмотрим, как организована виртуальная машина в памяти.

```
type VM struct {
   Block
   ExtCost func(string) int64
   FuncCallsDB map[string]struct{}
   Extern bool
}
```

- **Block** - это самая главная структура, которая содержит всю информацию.
- **ExtCost** - функция, которая возвращает стоимость выполнения внешних golang функций.
- **FuncCallsDB** - map имен golang функций, которые возвращают стоимость выполнения первым параметром. Это функции работы с БД, которые вычисляют стоимость с помощью EXPLAIN.

- **Extern** - при создании ВМ устанавливается в true и при компиляции кода не требует наличия вызываемого контракта. То есть дает вызывать контракт, который будет определен в дальнейшем.

Виртуальная машина представляет собой дерево из объектов типа **Block**. По сути блок - это самостоятельная единица, содержащая какой-то байт-код. Простыми словами - всё, что в языке заключается в фигурные скобки, является блоком. Например,

```
func my() {
     if true {
          while false {
               ...
           }
     }
}
```

создает блок с функцией, в котором блок с if и в котором, в свою очередь, блок с while.

```
type Block struct {
    Objects map[string]*ObjInfo
    Type int
    Owner *OwnerInfo
    Info interface{}
    Parent *Block
    Vars []reflect.Type
    Code ByteCodes
    Children Blocks
}
```

- **Objects** - map внутренних объектов типа указателей на **ObjInfo**. Если, к примеру, в блоке имеется переменная, то мы можем быстро получить информацию о ней по имени.
- **Type** - тип блока, функций и контрактов равен **ObjFunc** и **ObjContract**.
- **Owner** - ссылка на структуру **OwnerInfo**, которая содержит информацию о владельце компилируемого контракта. Указывается при компиляции контрактов или получается при загрузке из таблице **contracts**.
- **Info** - содержит непосредственно информацию об объекте и зависит от типа блока.
- **Parent** - указатель на родительский блок.
- **Vars** - массив с типами переменных данного блока.
- **Code** - непосредственно байт-код, который начнет исполнятся при передаче управления данном блоку. Например, в случае вызова функции или тела цикла.
- **Children** - массив дочерних блоков. Например, вложенные функции, циклы, условные операторы.

Рассмотрим еще одну важную структуру **ObjInfo**.

```
type ObjInfo struct {
   Type int
   Value interface{}
}
```

Type - тип объекта может принимать одно и следующих значений:

- **ObjContract** - контракт,
- **ObjFunc** - функция,

- **ObjExtFunc** - внешняя golang функция,
- **ObjVar** - переменная,
- **ObjExtend** - переменная $name.

Value - содержит соответствующую структуру для каждого типа.

```
type ContractInfo struct {
    ID uint32
    Name string
    Owner *OwnerInfo
    Used map[string]bool
    Tx *[]*FieldInfo
    Settings map[string]interface{}
}
```

- **ID** - идентификатор контракта. Это значение указывается в блокчейне для вызове контракта.
- **Name** - имя контракта.
- **Owner** - дополнительная информация о контракте.
- **Used** - map имен контрактов, которые вызываются внутри.
- **Tx** - массив данных, которые описаны в разделе data у контракта.

```
type FieldInfo struct {
       Name string
      Type reflect.Type
      Tags string
}
```

где **Name** - имя поля, **Type** - тип, **Tags** - дополнительные теги для поля.

Settings - map значений, которые описываются в разделе settings у контракта.

Как видно, информация во многом дублируется со структурой блок. Это можно считать архитектурным недостатком, от которого желательно избавиться.

Для типа **ObjFunc** поле **Value** содержит структуру **FuncInfo**

```
type FuncInfo struct {
     Params []reflect.Type
     Results []reflect.Type
    Names *map[string]FuncName
    Variadic bool
    ID uint32
}
```

- **Params** - массив типов параметров,
- **Results** - массив возвращаемых типов,
- **Names** - map данных для tail функций. Например, `DBFind().Columns()`.

```
type FuncName struct {
   Params []reflect.Type
   Offset []int
   Variadic bool
}
```

- **Params** - массив типов параметров

- **Offset** - массив смещений для этих переменных. По сути, все параметры, которые передаются в функциях через точку, являются переменными, которым могут быть присвоены инициализирующие значения.
- **Variadic** - true, если tail описание может иметь переменной количество параметров.
- **Variadic** - true, если у функции может быть переменной число параметров.
- **ID** - идентификатор функции.

Для типа **ObjExtFunc** поле **Value** содержит структуру **ExtFuncInfo**. Она описывает функции на golang.

```
type ExtFuncInfo struct {
   Name string
   Params []reflect.Type
   Results []reflect.Type
   Auto []string
   Variadic bool
   Func interface{}
}
```

Совпадающие параметры как у структуры **FuncInfo**. **Auto** - массив переменных, которые дополнительно передаются в golang функций, если они есть. Например, переменные *sc* типа *SmartContract*, **Func** - golang функция.

Для типа **ObjVar** поле **Value** содержит структуру **VarInfo**

```
type VarInfo struct {
   Obj *ObjInfo
   Owner *Block
}
```

- **ObjInfo** - информация о типе и значении переменной.
- **Owner** - указатель на блок-хозяина.

Для объектов типа **ObjExtend** поле **Value** содержит строку с именем переменной или функции.

Команды виртуальной машины

Идентификаторы команд виртуальной машины описаны в файле *cmds_list.go*. Байт-код представляет из себя последовательность структур типа **ByteCode**.

```
type ByteCode struct {
   Cmd uint16
   Value interface{}
}
```

В поле **Cmd** хранится идентификатор команды, а в поле **Value** сопутствующее значение. Как правило, команды осуществляют операции над конечными элементами стека, и если необходимо, то записывают туда результирующее значение.

- **cmdPush** - поместить значение из поля *Value* в стек. Например, используется для помещения в стек чисел, строк.
- **cmdVar** - поместить значение переменной в стек. *Value* содержит указатель на структуру *VarInfo* с информацией о переменной.

- **cmdExtend** - поместить в стек значение внешней переменной, они начинаются с **$**. *Value* содержит строку с именем переменной.
- **cmdCallExtend** - вызвать внешнюю функцию, их имена начинаются с **$**. Из стека будут взяты параметры функции, а результат(ы) функции будут помещены в стек. *Value* содержит имя функции.
- **cmdPushStr** - поместить строку из *Value* в стек
- **cmdCall** - вызвать функцию виртуальной машины. *Value* содержит указать на структуру **ObjInfo**. Эта команда применима как для *ObjExtFunc* golang функций, так и для *ObjFunc* Simvolio функций. При вызове функции передаваемые параметры берутся из стека, а результирующие значения возвращаются в стек.
- **cmdCallVari** - аналогично команде **cmdCall** вызывает функцию виртуальной машины, но эта команда применяется для вызова функций с переменным числом параметров.
- **cmdReturn** - служит для выхода из функции. При этом возвращаемые значения помещаются в стек. *Value* не используется.
- **cmdIf** - передает управление байткоду в структуре **Block**, указатель на который передан в поле *Value*. Управление передается только, если вызов функции *valueToBool* с крайним элементом стека возвращает *true*. В противном случае, управление передается следующей команде.
- **cmdElse** - команда работает аналогично команде **cmdIf**, но управление указанному блоку передается только, если *valueToBool* с крайним элементом стека возвращает false.
- **cmdAssignVar** - получаем из *Value* список переменных типа **VarInfo**, которым будет присваиваться значение с помощью команды **cmdAssign**.
- **cmdAssign** - присвоить переменным полученным командой **cmdAssignVar** значения из стека.
- **cmdLabel** - определяет метку, куда будет возвращаться управление в цикле while.
- **cmdContinue** - команда передает управление на метку **cmdLabel**. Осуществляет новую итерацию цикла. *Value* не используется.
- **cmdWhile** - проверяет крайний элемент стека с помощью *valueToBool* и вызывает **Block** передаваемые в поле *Value*, если значение true.
- **cmdBreak** - осуществляет выход из цикла.
- **cmdIndex** - получение в стек значения *map* или *array* по индексу. *Value* не используется. *(map|array) (index value) => (map|array[index value])*
- **cmdSetIndex** - присвоить элементу map или array крайнее значение стека. *Value* не используется. *(map|array) (index value) (value) => (map|array)*
- **cmdFuncName** - добавляет параметры , которые передаются с помощью последовательных описаний через точку *func name Func(...).Name(...)*.
- **cmdError** - команда создается прекращает работу контракта или функции с ошибкой, которая была указана в *error, warning* или *info*.

Ниже идет команды непосредственно для работы со стеком. Поле *Value* в них не используется. Следует заметить, что сейчас нет полностью автоматического приведения типов. Например, *string + float|int|decimal => float|int|decimal, float + int|str => float*, но *int + string => runtime error*.

- **cmdNot** - логическое отрицание *(val) => (!valueToBool(val))*
- **cmdSign** - смена знака. *(val) => (-val)*
- **cmdAdd** - сложение. *(val1)(val2) => (val1+val2)*
- **cmdSub** - вычитание. *(val1)(val2) => (val1-val2)*

- **cmdMul** - умножение. *(val1)(val2) => (val1*val2)*
- **cmdDiv** - деление. *(val1)(val2) => (val1/val2)*
- **cmdAnd** - логическое И. *(val1)(val2) => (valueToBool(val1) && valueToBool(val2))*
- **cmdOr** - логическое ИЛИ. *(val1)(val2) => (valueToBool(val1) || valueToBool(val2))*
- **cmdEqual** - сравнение на равенство, возвращается bool. *(val1)(val2) => (val1 == val2)*
- **cmdNotEq** - сравнение на неравенство, возвращается bool. *(val1)(val2) => (val1 != val2)*
- **cmdLess** - сравнение на меньше, возвращается bool. *(val1)(val2) => (val1 < val2)*
- **cmdNotLess** - сравнение на больше или равно, возвращается bool. *(val1)(val2) => (val1 >= val2)*
- **cmdGreat** - сравнение на больше, возвращается bool. *(val1)(val2) => (val1 > val2)*
- **cmdNotGreat** - сравнение на меньше или равно, возвращается bool. *(val1)(val2) => (val1 <= val2)*

Как уже было замечено ранее, выполнение байт-кода не влияет на виртуальную машину. Это, например, позволяет одновременно запускать различные функции и контракты в рамках одной виртуальной машины. Для запуска функций и контрактов, а также любых выражений и байт-кода используется структура **Runtime**.

```
type RunTime struct {
   stack []interface{}
   blocks []*blockStack
   vars []interface{}
   extend *map[string]interface{}
   vm *VM
   cost int64
   err error
}
```

- **stack** - стек, на которым происходит выполнение байт-кода,
- **blocks** - стек вызовов блоков.

```
type blockStack struct {
     Block *Block
     Offset int
}
```

- **Block** - указатель на выполняемый блок.
- **Offset** - смещение последней выполняемой команды в байт-коде указанного блока.
- **vars** - стек значений переменных. При вызове байт-кода в блоке, его переменные добавляются в этот стек переменных.

После выхода из блока, размер стека переменных возвращается к предыдущему значению.

- **extend** - указатель на map со значениями внешних переменных ($name).
- **vm** - указатель на виртуальную машину.
- **cost** - результирующая стоимость выполнения.
- **err** - ошибка выполнения, если она была.

Выполнение байт-кода происходит в функции *RunCode*. Она содержит цикл, который выполняет соответствующие действия для каждой команды бай-кода. Перед началом обработки байт-кода,мы должны инициализировать необходимые данные. Здесь мы добавляем наш блок в

```
rt.blocks = append(rt.blocks, &blockStack{block, len(rt.vars)})
```

Далее мы получаем информацию о параметрах «хвостовых» функциях, которые должны находится в последнем элементе стека.

```
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]
}
```

Далее мы должны инициализировать начальными значениями все переменные, которые определены в данном блоке.

```
start := len(rt.stack)
varoff := len(rt.vars)
for vkey, vpar := range block.Vars {
   rt.cost--
   var value interface{}
```

Так как у нас переменные функции тоже являются переменными, то мы должны взять их из последних элементов стека в том же порядке, в каком они описаны в самой функции.

```
if block.Type == ObjFunc && vkey < len(block.Info.(*FuncInfo).Params) {
    value = rt.stack[start-len(block.Info.(*FuncInfo).Params)+vkey]
} else {
```

Здесь мы инициализируем локальные переменные начальными значениями.

```
     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)
}
```

Далее нам необходимо обновить значения у параметров-переменных, которые были переданы в «хвостовых» функциях.

```
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 {
```

Если у нас может передаваться переменное количество параметров, то мы объединяем их в одну переменную массив.

```
                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
            }
        }
    }
}
```

После этого нам остается только сдвинуть стек убрав из вершины значения, которые были переданы как параметры функции. Их значения мы уже скопировали выше в массив переменных.

```
if block.Type == ObjFunc {
     start -= len(block.Info.(*FuncInfo).Params)
}
```

После окончания работы цикла по выполнению команд байт-кода мы должны корректно очистить стек.

```
last := rt.blocks[len(rt.blocks)-1]
```

убираем из стека блоков текущий блок

```
rt.blocks = rt.blocks[:len(rt.blocks)-1]
if status == statusReturn {
```

В случае успешного выхода из выполняемой функции мы добавляем к предыдущему концу стека возвращаемые значения.

```
  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++
    }
  status = statusNormal
} else {
```

Как видно, если у нас выполняется не функция, то мы не восстанавливаем состояние стека, а выходим из функции как есть. Дело в том, что блоком с байт-кодом также являются циклы и условные конструкции, которые уже выполняются внутри какой-то функции.

```
    return
  }
}
rt.stack = rt.stack[:start]
```

Рассмотрим другие функции для работы с виртуальной машиной. Любая виртуальная машина создается с помощью функции NewVM. В каждую виртуальную машину сразу добавляются три функции **ExecContract**, **CallContract** и **Settings**. Добавление происхходит с помощью функции **Extend**.

```
for key, item := range ext.Objects {
    fobj := reflect.ValueOf(item).Type()
```

Мы перебираем все передаваемые объекты и смотрим только функции.

```
switch fobj.Kind() {
case reflect.Func:
```

По информации, полученной о функции, мы заполняем структуру **ExtFuncInfo** и добавляем её в map **Objects** верхнего уровня по ее имени.

```
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++ {
```

У нас есть так называемые **Auto** параметры. Как правило, это первый параметр, например sc *SmartContract* или rt *Runtime*. Мы не можем передавать их из языка Simvolio, но они нам необходимы при выполнении некоторых golang функций. Поэтому мы указываем какие переменные будут автоматически подставляться в момент вызова функции. В данном случае, функции **ExecContract**, **CallContract** имеют такой параметр rt *Runtime*.

```
if isauto, ok := ext.AutoPars[fobj.In(i).String()]; ok {
  data.Auto[i] = isauto
}
```

Заполняем информацию о параметрах

```
  data.Params[i] = fobj.In(i)
}
```

и о типах возвращаемых значений

```
for i := 0; i < fobj.NumOut(); i++ {
   data.Results[i] = fobj.Out(i)
}
```

Добавление функции в корневой Objects позволят компилятору в дальнейшем находить их при использовании из контрактов.

```
         vm.Objects[key] = &ObjInfo{ObjExtFunc, data}
    }
}
```

3.8.2 Компиляция

За компиляцию массива лексем, полученных от лексического анализатора, отвечают функции, расположенные в файле *compile.go*. Компиляцию условно можно разделить на два уровня. На верхнем уровне мы обрабатываем функции, контракты, блоки кода, условные операторы и операторы цикла, определение переменных и т.д. На нижнем уровне, мы компилируем выражения, которые находятся внутри блоков кода или условий в цикле и условном операторе. В начале рассмотрим более простой нижний уровень.

Перевод выражений в байт код осуществляется в функции **compileEval**. Так как у нас виртуальная машина работает со стеком, то необходимо переводить обычную инфиксную запись выражений в постфиксную нотацию или обратную польскую запись. Например, 1+2 должно быть преобразовано в 12+, тоо есть вы помещаем 1 и 2 в стек, а затем применяем операцию сложения для двух последних элементов в стеке и записываем результат в стек. Сам алгоритм перевода можно найти в Интернете - например https://master.virmandy.net/perevod-iz-infiksnoy-notatsii-v-postfiksnuyu-obratnaya-polskaya-zapis/. В глобальной переменной *opers = map[uint32]operPrior* содержатся приоритеты операций, которые необходимы при переводе в обратную польскую нотацию. В начале функции определяются следующие переменные:

- **buffer** - временный буфер для команд байт-кода,

- **bytecode** - итоговый буфер команд байт-кода,
- **parcount** - временный буфер для подсчета параметров при вызове функций,
- **setIndex** - переменная в процессе работы устанавливается в *true*, когда у нас происходит присваивание элементу *map* или *array*. Например, *a[«my»] = 10*. В этом случае, нам нужно будет использовать специальную команду **cmdSetIndex**.

Далее имеется цикл в котором мы получаем очередную лексему и обрабатываем её соответствующим образом. Например, при обнаружении фигурных скобок

```
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
```

мы прекращаем разбор выражения, а при переносе строки мы смотрим не является ли предыдущий оператор операцией и не находимся ли мы внутри круглых скобок, в противном случае также происходит выход и разбора выражения. В целом сам алгоритм, соответствует алгоритму перевода в обратную польскую запись, с учетом того, что приходится учитывать вызовы функций, контрактов, обращения по индексу и прочие вещи, которые не встретишь в случае разбора, например, для калькулятора. Рассмотрим вариант разбора лексемы с типом *lexIdent*. Мы ищем переменную, функцию или контракт с данным именем. Если у нас ничего не найдено и это не является вызовом функции или контракта, то мы выдаем ошибку.

```
objInfo, tobj := vm.findObj(lexem.Value.(string), block)
if objInfo == nil && (!vm.Extern || i > *ind || i >= len(*lexems)-2 || (*lexems)[i+1].Type !=␣
↪isLPar) {
      return fmt.Errorf(`unknown identifier %s`, lexem.Value.(string))
}
```

У нас может быть ситуация, когда вызывается контракт, который будет описан в дальнейшем. В этом случае, если не найдена функция и переменная с таким именем, то мы считаем, что у нас будет вызов контракта. В языке вызовы контрактов или функции ничем не отличаются. Но вызов контракта мы должны осуществлять через функцию **ExecContract**, которую мы и подставляем в байт-код.

```
if objInfo.Type == ObjContract {
    objInfo, tobj = vm.findObj(`ExecContract`, block)
    isContract = true
}
```

В *count* мы пока запишем количество переменных и это значение также пойдет в стек, с количеством параметров функций. При каждом последующем обнаружении параметра мы просто увеличиваем это количество на единицу в последнем элементе стека.

```
count := 0
if (*lexems)[i+2].Type != isRPar {
    count++
}
```

Так как для контрактов у нас имеется список вызываемых им *Used*, то в случае вызова контракта мы должны сделать такие отметки, и в случае, когда вызов контракта без параметров *MyContract()*, мы должны добавить два пустых параметра для вызова **ExecContract**, который должен получить минимум два параметра.

```
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++

}
```

Если мы видим что далее идет квадратная скобка, то мы добавляем команду **cmdIndex** для получения значения по индексу.

```
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})
}
```

Если функция **compileEval** непосредственно формирует байт-код выражений в блоках, то функция **CompileBlock** формирует как дерево объектов, так и байт-код не относящийся к выражениям. Компиляция также основана на работе конечного автомата, подобно тому как это было сделано для лексического анализа, но со следующими отличиями. Во-первых, мы оперируем уже не символами, а лексемами, а во-вторых, мы все состояния и переходы сразу описываем в переменной *states*. Она представляет собой массив *map* с индексами по типам лексем и для каждой лексемы указывается структура *compileState* с новым состоянием в поле *NewState* и, если понятно какую конструкцию мы разобрали, то указывается функция обработчик в поле *Func*.

Рассмотрим, например, главное состояние

```
{ // stateRoot
   lexNewLine: {stateRoot, 0},
   lexKeyword | (keyContract << 8): {stateContract | statePush, 0},
   lexKeyword | (keyFunc << 8): {stateFunc | statePush, 0},
   lexComment: {stateRoot, 0},
   0: {errUnknownCmd, cfError},
},
```

Если мы встречаем перевод строки или комментарии, то остаемся на этом же состоянии. Если встречаем ключевое слово **contract**, то переходим в состояние *stateContract* и начинаем разбор этой конструкции. Если встречаем ключевое слово **func**, то переходим в состояние *stateFunc*. В случае получения

других лексем будет вызвана функция генерации ошибки. Предположим, что у нас встретилось ключевое слово *func* и мы перешли в состояние *stateFunc*.

```
{ // stateFunc
    lexNewLine: {stateFunc, 0},
    lexIdent: {stateFParams, cfNameBlock},
    0: {errMustName, cfError},
},
```

Так как после ключевого слова **func** должно идти имя функции, то при переводе строки мы остаемся в этом же состоянии, а при всех других лексемах мы генерируем соответствующую ошибку. Если мы получили имя функции в лексеме-идентификаторе, то мы переходим в состояние *stateFParams* в котором мы получим параметры функции. При этом мы вызываем функцию **fNameBlock**. Следует заметить, что структура типа *Block* была создана по флагу *statePush* и здесь мы берем её из буфера и заполняем нужными нам данными.

```
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
}
```

Функция **fNameBlock** используется для контрактов и функции (в том числе вложенных в другие функции и контракты). Она заполняет поле *Info* соответствующей структурой и заносит себя в map *Objects* родительского блока. Это чтобы затем мы могли вызывать данную функцию или контракт по данному имени. Подобным образом мы создаем функции для всех состояний и вариантов, эти функции как правило очень небольшие и выполняют определенную работу по формированию дерева виртуальной машины. Что касается функции **CompileBlock**, то она просто проходит по всем лексемам и переключает состояния в соответствии с состояниями описанными в *states*. Почти весь дополнительный код обработки дополнительных флагов.

- **statePush** - происходит добавление объекта *Block* в дерево объектов.
- **statePop** - используется при окончании блока на закрывающих фигурных скобках.
- **stateStay** - указывает на то, что при переходе в новое состояние нужно остаться на текущей лексеме.
- **stateToBlock** - указывает на переход в состояние *stateBlock*. Используется для обработки while и if, когда необходимо после обработки выражения перейти в обработку блока внутри фигурных скобок.
- **stateToBody** - указывает на переход в состояние *stateBody*.

- **stateFork** - сохраняет позицию лексемы. Используется когда выражение начинается в идентификаторе или имени с **$**. У нас может быть или вызов функции или присваивание.
- **stateToFork** - используется для получения лексемы сохраненной по флагу *stateFork*. Эта лексема будет передаваться в функцию обработчик.
- **stateLabel** - служит для вставки команды **cmdLabel**. Этот флаг нужен для конструкции while.
- **stateMustEval** - проверяет наличие условного выражения в начале конструкций if и while.

Кроме функции **CompileBlock** следует упомянуть ещё функцию **FlushBlock**. Дело в том, что при компиляции строится дерево блоков независимо от существующей виртуальной машины. Точнее, мы берем информацию о существующих функциях и контрактах в виртуальной машине, но откомпилированные блоки собираем в отдельное дерево. В противном случае, в случае возникновения ошибки при компиляции, мы обязаны будем откатить состояние виртуальной машины к предыдущему состоянию. Поэтому мы собираем дерево отдельно, но должны вызвать функцию **FlushContract** после успешного окончания компиляции. Эта функция добавляет наше готовое дерево блоков в текущую виртуальную машину. На этом этап компиляции считается законченным.

3.8.3 Лексический анализ

Лексический анализатор обрабатывает входящую строку и формирует последовательность лексем следующих типов:

- **sys** - системная лексема. например: {}[](),.
- **oper** - оператор +-/*
- **number** - число,
- **ident** - идентификатор,
- **newline** - перевод строки,
- **string** - строка,
- **comment** - комментарий.

В данной версии предварительно с помощью *script/lextable/lextable.go* строится таблица переходов (конечный автомат) для разбора лексем, которая записывается в файл *lex_ table.go*. В принципе, можно избавиться от предварительной генерации этого файла и создавать таблицу переходов сразу в памяти при запуске (в init()). Сам лексический анализ происходит в функции *lexParser* в *lex.go*.

lextable/lextable.go

Здесь мы определяем алфавит, с которым будет работать наш язык, и описываем конечный автомат, который переходит из одного состояния в другое в зависимости от очередного полученного символа.

states содержит JSON объект содержащий список состояний.

Кроме конкретных символов, за d обозначены все символы, которые не указаны в состоянии

n - 0x0a, s - пробел, q - обратные кавычки ‘, Q - двойные кавычки, r - символы >= 128 a - A-Z и a-z, 1 - 1-9

В качестве ключей выступают имена состояний, а в объекте-значении перечислены возможные наборы символов, и затем для каждого такого набора идет новое состояние, куда следует сделать переход, далее имя лексемы, если нам нужно вернуться в начальное состояние и третьим параметром идут служебные флаги, которые указывают, что делать с текущим символом.

Например, у нас состояние main и входящий символ /. `"/": ["solidus", "", "push next"],`

push даёт команду запомнить его в отдельном стеке, а **next** - перейти к следующему символу, при этом мы меняем состояние на **solidus**. После этого, берем следующий символ и смотрим на состояние **solidus**.

Если у нас / или * - то мы переходим в состояние комментарий, так они начинаются с // или /*. При этом видно, что для каждого комментария разные последующие состояния, так как заканчиваются они разными символами.

А если у нас следующий символ не / и не *, то мы все что у нас положено в стек (/) записываем как лексему с типом oper, очищаем стэк и возвращаемся в состояние main.

Данный модуль переводит данное дерево состояний в числовой массив и записывает его в файл *lex_ table.go*.

В первом цикле

```
for ind, ch := range alphabet {
i := byte(ind)
```

мы формируем алфавит допустимых символов. Далее в *state2int* мы каждому состоянию даем свой порядковый идентификатор.

```
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))
```

Далее проходимся по всем состояниям и для каждого множества в состоянии и для каждого символа в этом множестве мы записываем в двумерный массив *table* трех-байтное число [id нового состояния (0=main)] + [тип лексемы (0-нет лексемы)] + [флаги]. Двухмерность массива *table* заключена в том, что разбит на состояния и 33 входящих символа из массива *alphabet* расположенных в таком же порядке. То есть, в дальнейшем мы будем работать с этой таблице примерно следующим образом.

Находимся в состоянии *main* на нулевой строке таблицы *table*. Берем первый символ, смотрим его индекс в массиве *alphabet* и берем значение из столбца с данным индексом. Далее из полученного значения в младшем байте получаем флаги, во втором байте - тип полученной лексемы, если её разбор закончен, и в третьем байте получаем индекс нового состояния, куда нам следует перейти. Всё это подробнее будет рассмотрено в функции **lexParser** в файле *lex.go*.

Если нужно добавить какие-то новые символы, то нужно добавить их в массив *alphabet* и увеличить константу *AlphaSize*. Если нужно добавить новую комбинацию символов, то их следует описать в *states*, аналогично существующим вариантам. После этого следует, запустить *lextable.go*, чтобы обновился файл *lex_ table.go*.

lex.go

Функция **lexParser** непосредственно производит лексический анализ и на основе входящей строки возвращает массив полученных лексем. Рассмотрим структуру лексемы

```
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** - тип лексемы. Может быть одним из следующих значений: *lexSys, lexOper, lexNumber, lexIdent, lexString, lexComment, lexKeyword, lexType, lexExtend*,

- **Value** - значение лексемы. Тип значения зависит о типа. Рассмотрим подробнее,
- **lexSys** - сюда относятся скобки, запятые и т.п. В этом случае, *Type = ch<<8 | lexSys* - смотрите константы *isLPar ... isRBrack*, а само *Value* равно *uint32(ch)*,
- **lexOper** - значения представляют из себя эквивалентную последовательность символов в виде *uint32*. Например, смотрите константы *isNot... isOr*,
- **lexNumber** - числа хранятся в виде *int64* или *float64*. Если у числа указана десятичная точка, то это *float64*.
- **lexIdent** - идентификаторы хранятся в виде строк,
- **lexNewLine** - символ перевода строки. Также служит для подсчета строки и позиции лексемы.
- **lexString** - строки хранятся в виде строки *string*,
- **lexComment** - комментарии также хранятся в виде строк *string*,
- **lexKeyword** - ключевые слова хранят только соответствующий индекс - константы от *keyContract... keyTail*. В этом случае, *Type = KeyID << 8 | lexKeyword*. Также, следует заметить, что ключевые слова *true,false,nil* сразу преобразуются в лексемы типа *lexNumber*, с соответствующими типами *bool* и *intreface{}*,
- **lexType** - в этом случае, значение содержит соответствующее значение типа *reflect.Type*,
- **lexExtend** - это идентификаторы, начинающиеся со знака доллара **$**. Эти переменные и функции передаются извне и поэтому выделяются в специальный тип лексем. Значение содержит имя в виде строки без начального знака доллара.
- **Line** - строка, где обнаружена лексема.
- **Column** - Позиция лексемы в строке.

Рассмотрим подробнее функцию **lexParser**. Функция *todo* - в ней на основе текущего состояния и переданного символа находит индекс символа в нашем алфавите и из таблицы переходов получает новое состояние, идентификатор лексемы, если он есть, и дополнительные флаги. Сам разбор заключается в последовательном вызове этой функции для каждого очередного символа и переключении на новое состояние. Как только мы видим, что у нас получена лексема, мы создаем соответствующую лексему в выходном максиме и продолжаем разбор. Следует заметить, что в процессе разбора мы не не накапливаем символы лексемы в отдельном стеке или массиве, мы только сохраняем смещение, откуда начинается наша лексема. После того как лексема получена, мы сдвигаем смещение для следующей лексемы на текущую позицию разбора.

Осталось рассмотреть флаги, которые используются при разборе:

- **push** - этот флаг означает, что начинаем накапливать символы в новую лексему,
- **next** - символ необходимо добавить к текущей лексеме,
- **pop** - получение лексемы закончено. Как правило, с этим флагом у нас выдается идентификатор-тип разобранной лексемы,
- **skip** - этот флаг используется для исключения символа из разбора. Например, управляющие слэш символы в строке - *\n \r ''*. Они автоматически заменяются на этапе этого лексического анализа.

3.8.4 Язык Simvolio

Лексемы

Исходный текст программы должен быть в кодировке UTF-8. Обрабатываются следущие типы лексем:

- **Ключевые слова** - action break conditions continue contract data else error false func if info nil return settings true var warning while
- **Числа** - принимаются только числа в десятичной системе исчисления. Других видов записи чисел на данный момент нет. Имеется два базовых типа - **int** и **float**. При наличии десятичной точки число становится типом *float*. *int* соотвествует типу **int64** (в golang), а тип *float* типу **float64** (в golang).
- **Строки** - строка может заключаться в двойные («строка») или обратные кавычки (*строка*). Строки обоих типов могут включать в себя переносы строк. Кроме этого, строка в двойных кавычках может содержать в себе двойные кавычки, перенос строки и возврат каретки, определенные с помощью слэша (\). Например, *«Это "первая строка".\r\nЭто вторая строка.»*.
- **Комментарии** - имеется два вида комментариев. Комментарий до конца строки нчинается с двойного слэша. Например, // это комментарирй до конца строки. Комментарий между слешами со звездочками может быть на несоклько строк. Например, /* Этот комментарий может быть на несколько строк */.
- **Идентификаторы** - имена переменных и функций, которые состоят из латинских букв, UTF-8 символов, цифр и знака подчеркивания. Они могут начинаться с буквы, подчёркивания, @ и $. Со знака доллара начинается обращение к переменным-параметрам, которые определены в секции *data*. Также с помщью доллара можно определять глобальные переменные, в общей области видимости функций *conditions* и *action*. Со знака амперсенд, можно вызывать контракт с указанием экосистемы. Например, *@1NewTable(...)*.

Типы

Рядом с типами указаны соответствующие типы из golang.

- **bool** - bool, значение по умолчанию **false**,
- **bytes** - []byte{}, по умолчанию присваивается пустой массив байт.
- **int** - int64, значение по умолчанию **0**,
- **address** - uint64, значение по умолчанию **0**,
- **array** - []interface{}, по умолчанию создается пустой массив,
- **map** - map[string]interface{}, по умолчанию создается пустой ассоциативный массив,
- **money** - decimal.Decimal, значение по умолчанию **0**,
- **float** - float64, значение по умолчанию **0**,
- **string** - string, по умолчанию создается пустая строка.

В можете описать переменные данных типов с помощью ключевого слова **var**. Например, *var var1, var2 int*. При этом, переменным сразу присвоится значение по умолчанию, которое определено для данного типа.

Выражения

Выражения могут состоять из арифметических, логических операций и вызовов функций. Все выражения вычисляются слева направо в соответствии с приоритетом операции. При равентстве приоритетов выполнение также идет слева направо. Вот список операций от большего приоритета к меньшему.

- **Самый большой приоритет операция у вызовов функций и круглых скобок**. При вызове функций, вначале слева направо вычисляются передаваемые параметры.
- **Унарные операции** - логическое отрицание **!** и арифметическая смена знака **-**.
- **Умножение и деление** - арифметические умножение и деление ***** и /.
- **Сложение и вычитание** - арифметические сложение и вычитание **-** и +.

- **Логические сравнения** - >= > < <=.
- **Логические равенства и неравенства** - == !=.
- **Логическое И** - &&.
- **Логическое ИЛИ** - ||.

Следует заметить, что при логическом **&&** и || в любом случае вычисляются левая и права сторона выражения.

В языке сейчас отсутствует проверка типов на этапе компиляции, но при вычислении операндов с разными типами делается попытка привести тип операндов к более сложному типу. Типы в порядке возрастная сложности можно расположить следующим образом - *string, int, float, money*. К сожалению, сейчас реализованы не все комбинации операндов разных типов. Для строк существует операция сложения, которая означает конкатенацию строк.

Например, *string + string = string, money - int = money, int * float = float*

Что касается вызовов функций, то там проверка типов осуществляется только в момент выполнения и проверяются только типы *string* и *int*.

Для типов **array** и **map** существует обращение по индексу []. В случае *array* в качестве индекса должно указываться значение тип *int*, а при обращении к элементу *map* необходимо указывать переменную или значение типа *string*. Если происходит присваивание элементу *array* с индексом больше чем размер массива, то в данный массив автоматически будут добавлены недостающие элементы, которые будут инициализированы значениями *nil*. Например,

```
var my array
my[5] = 0
var mymap map
mymap["index"] = my[3]
```

В выражениях, где необходимо логическое значение, таких как **if, while, &&, ||, !** существует автоматическое преобразование типов к логическому значению.

- **bytes** - true, если размер больше 0,
- **int** - true, если не 0,
- **array** - true, если не nil и размер больше 0,
- **map** - true, если не nil и размер больше 0,
- **money** - true, если не 0,
- **float** - true, если не 0,
- **string** - true, если размер больше 0,

```
var mymap map
var val string
if mymap && val {
...
}
```

Область видимости

Фигурные скобки определяют блок, который может иметь свои переменные. По умолчанию, область видимости переменной распространяется на текущий блок и все вложенные блоки. Внутри блока можно определять переменные с уже существующими именами переменных. В этом случае, внешняя переменная с таким же именем становится недоступна.

```
var a int
a = 3
{
   var a int
   a = 4
   Println(a) // 4
}
Println(a) // 3
```

Выполнение контрактов

При вызове контракта ему должны передаваться параметры описанные в секции **data**. Перед выполнением контракта виртуальная машина получает эти парамтры и присваивает их соответствующим переменным ($Param). После этого, вызывается предопределенная функция **conditions** и после неё функция **action**. Если в контракте определена функция **rollback**, то она будет вызываться при откате данного контракта.

Ошибки, которые возникают в момент выполнения контракта можно разделить на два типа - генерируемые ошибки и ошибки среды выполнения. Ошибки первого типа генерируются с помощью специальных команд **error, warning, info**, а также при выполнении встроенных функций, когда они возвращает *err* не равный *nil*. Ошибки среды выполнения могут возникать в результате некорректных данных, которые преварительно не были проверены. Например, выход за границы массива, несовпадение типов при вызове функций и т.д. Следует заметить, что все значения переменных изначально имеют тип *interface{}* и затем им присваивается значение соответствующего golang типа. Так, например, типы *array* и *map* это соответственно golang типы *[]interface{}* и *map[string]interface{}*. Оба типа массивов могут содержать элементы любых типов. Ещё, если мы присваиваем сразу myarr[2] = 0, то у нас автоматически вставляется 0 и 1 элемент со значением *nil*. В этом случае при использовании элемента myarr[0] в каком-нибудь арифметическом выражении мы получим ошибку виртуальной машины. Обработка исключений в языке отсутствует, то есть любая ошибка завершает выполнение контракта. Так как при выполнении каждого контракта создается свой стэк и структуры для хранения значений переменных, то после завершения работы контракта эти данные будут автоматически удалены сборщиком мусора golang.

БНФ

<десятичная цифра> ::= „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“

<десятичное число> ::= <десятичная цифра> {<десятичная цифра>}

<код символа> ::= „“„<любой символ>“„“

<действительное число> ::= [„-„] <десятичное число>“.“[<десятичное число>]

<целое число> ::= [„-„] <десятичное число> | <код символа>

<число> := <целое число> | <действительное число>

<буква> ::= „A“ | „B“ | . . . | „Z“ | „a“ | „b“ | . . . | „z“ | 0x80 | 0x81 | . . . | 0xFF

<пробел> ::= 0x20

<табуляция> ::= 0x09

<конец строки> := 0x0D 0x0A

<спецсимвол> ::= „!“ | „»“ | „$“ | „““ | „(„ | „)“ | „*“ | „+“ | „,“ | „-„ | „.“ | „/“ | „<“ | „=“ | „>“ | „[„ | „“ | „]“ | „_“ | „|“ | „}“ | „{„ | <табуляция> | <пробел> | <конец строки>

<символ> ::= <десятичная цифра> | <буква> | <спецсимвол>

<имя> ::= (<буква> | „_“) {<буква> | „_“ | <десятичная цифра>}

<имя функции> ::= <имя>

<имя переменной> ::= <имя>

<имя типа> ::= <имя>

<стр символ> ::= <табуляция> | <пробел> | „!“ | „#“ | . . . | „[„ | „]“ | . . .

<элемент строки> ::= {<стр символ> | „»“ | „n“ | „r“ }

<строка> ::= „»“ { <элемент строки> } „»“ | „‘“ { <элемент строки> } „‘“

<оператор присваивания> ::= „=“

<оператор унарный> ::= „-„

<оператор бинарный> ::= „==“ | „!=“ | „>“ | „<“ | „<=“ | „>=“ | „&&“ | „||“ | „*“ | „/“ | „+“ | „-„

<оператор> ::= <оператор присваивания> | <оператор унарный> | <оператор бинарный>

<параметры> ::= <выражение> {„,“<выражение>}

<вызов контракта> ::= <имя контракта> „(„ [<параметры>] „)“

<вызов функции> ::= <вызов контракта> [{„.“ <имя> „(„ [<параметры>] „)“}]

<содержимое блока> ::= <команда блока> {<конец строки><команда блока>}

<блок> ::= „{„<содержимое блока>“}“

<команда блока> ::= (<блок> | <выражение> | <определение переменных> | <if> | <while> | break | continue | return)

<if> ::= if <выражение><блок> [else <блок>]

<while> ::= while <выражение><блок>

<contract> ::= contract <имя> „{„[<секция data>] {<функция>} [<conditions>] [<action>]“}“

<секция data> ::= data „{„ {<параметр data><конец строки>} „}“

<параметр data> ::= <имя переменной> <имя типа> „»“{<tag>}“»“

<tag> ::= optional | image | file | hidden | text | polymap | map | address | signature:<имя>

<conditions> ::= conditions <блок>

<action> ::= action <блок>

<функция> ::= func <имя функции>“(„[<описание переменной>{„,“<описание переменной>}]“)“[{<tail>}] [<имя типа>] <блок>

<описание переменной> ::= <имя переменной> {„,“ <имя переменной>} <имя типа>

<tail> ::= „.“<имя функции>“(„[<описание переменной>{„,“<описание переменной>}]“)“

<определение переменных> ::= var <описание переменной>{„,“<описание переменной>}

3.9 REST API v2

Все функции, доступные в программном клиенте Molis, включая аутентификацию, получение данных об экосистемах, обработку ошибок, оперирование с таблицами базы данных, страницами интерфейсов, запуск контрактов (транзакций сети) доступны через REST API платформы. Таким образом, благодаря REST API разработчики могут получить доступ ко всему функционалу платформы без использования программного клиента Molis.

Вызов команды происходит при обращении к `/api/v2/command/[param]`, где **command** - имя команды, **param** - дополнительный параметр, например, имя изменяемого или получаемого ресурса. Отправлять параметры запросов следует с `Content-Type: x-www-form-urlencoded`. Ответ сервера представлен в JSON формате.

3.9.1 Обработка ошибок

В случае успешного выполнения запроса возвращается статус 200. В случае ошибки, кроме ошибочного статуса возвращается объект JSON с полями:

- **error** - идентификатор ошибки,
- **msg** - текст ошибки,
- **params** - массив дополнительных параметров ошибки, которые могут быть подставлены в сообщение об ошибке.

Пример ответа

```
400 (Bad Request)
Content-Type: application/json
{
    "err": "E_INVALIDWALLET",
    "msg": "Wallet 1111-2222-3333 is not valid",
    "params": ["1111-2222-3333"]
}
```

Список ошибок

- **E_CONTRACT** - There is not %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 tx is too big (%d),
- **E_NOTFOUND** - Content page or menu has not been found,
- **E_NOTINSTALLED** - platform is not installed. В этом случае нужно запустить установку командно *install*,
- **E_PARAMNOTFOUND** - parameter has not been found,
- **E_QUERY** - DB query is wrong,
- **E_RECOVERED** - API recovered, возвращается в случае panic error,
- **E_SERVER** - Server error. Возвращается в случае ошибки в библиотечных функциях golang; поле *msg* содержит текст ошибки,

- **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,
- **E_UNDEFINEVAL** - Value %s is undefined,
- **E_UNKNOWNUID** - Unknown uid,
- **E_OBSCREATED** - Off-Blockchain Server is already created.

Если, неважно где, возвращается ошибка **E_RECOVERED**, то это баг, который требует обнаружения и исправления. Ошибка **E_NOTINSTALLED** должна возвращаться любой командой кроме *install*, в случае, если система еще не установлена. Ошибка **E_SERVER** теоретически может возвратится в любой команде. Если она возникает на неверных входных параметрах, то её можно заменить на соответствующие ошибки. В противном случае, это ошибка сообщает о неверном функционировании или настройке системы, то есть требует более детального изучения. Ошибка **E_UNAUTHORIZED** может возвращаться на любой команде кроме *install, getuid, login* в случае, если не был осуществлен login или сессия закончилась.

3.9.2 Маршруты недоступные в OBS

```
txstatus
txinfo
txinfoMultiple
appcontent
appparam
appparams
history
balance
block
maxblockid
ecosystemparams
ecosystemparam
systemparams
```

3.9.3 Аутентификация

Для аутентификации используется **JWT токен** http://www.jwt.org. После получения JWT токена необхоимо передавать его при каждом запросе в заголовке: `Authorization: Bearer TOKEN_HERE`.

getuid

GET/ Возвращает уникальное значение, которое нужно подписать своим приватным ключом и отправить обратно серверу с помощью команды **login**. На данный момент создается временный JWT токен, который нужно передать в **Authorization** при вызове **login**.

```
GET
/api/v2/getuid
```

Ответ

- *uid* - строка для подписи.
- *token* - временный токен для передачи в login. Время жизни временного токена сейчас составляет 5 секунд.
- *network_id* - идентификатор NetworkID сервера.

В случае, когда авторизация не требуется, то возвращаются

- *expire* - количество секунд до истечения срока,
- *ecosystem* - идентификатор экосистемы,
- *key_id* - идентификатор кошелька,
- *address* - адрес кошелька в формате `XXXX-XXXX-.....-XXXX`.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "uid": "28726874268427424",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6I........AU3yPRp64SLO4aJqhN-kMoU5HNYTDplQXbVuOY",
    "network_id": "5324976953280995276"
}
```

Ошибки: *E_SERVER*

login

POST/ Аутентификация пользователя. Предварительно необходимо вызвать команду **getuid** для получения уникального значения и подписать его. В заголовке нужно передать временный JWT токен полученный вместе с getuid. В случае успешного завершения нужно полученный в ответе токен передавать во всех запросах в заголовке *Authorization*.

Запрос

```
POST
/api/v2/login
```

- *[ecosystem]* - идентификатор экосистемы. Если не указан, то берется первая экосистема,
- *[expire]* - время длительности JWT токена в секундах, по умолчанию - 36000,
- *[pubkey]* - публичный ключ hex; если в блокчейне уже есть хранится ключ, то нужно передать идентификатор аккаунта в любом виде в параметре *key_id*,
- *[key_id]* - идентификатор аккаунта в виде числа или `XXXX-...-XXXX`; использовать, если публичный ключ уже хранится в блокчейне; нельзя указывать совместно с *pubkey*,
- *signature* - подпись uid полученного через getuid hex.

Ответ

- *token* - JWT токен,
- *ecosystem* - идентификатор экосистемы,
- *key_id* - идентификатор аккаунта,
- *address* - адрес аккаунта в формате `XXXX-XXXX-.....-XXXX`,

- *notify_key* - ключ для получения уведомлений,
- *isnode* - true или false - является ли владельцем данной ноды,
- *isowner* - true или false - является ли владельцем данной экосистемы,
- *obs* - true или false - есть ли у экосистемы virtual dedicated ecosystem.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6I........AU3yPRp64SLO4aJqhN-kMoU5HNYT8fNGODpOY"
    "ecosystem":"1",
    "key_id":"12345",
    "address": "1234-....-3424"
}
```

Ошибки: *E_SERVER, E_UNKNOWNUID, E_SIGNATURE, E_STATELOGIN, E_EMPTYPUBLIC*

3.9.4 Служебные команды

network

GET/ Возвращает информацию о текущей сети, включая список нод. Не требует авторизации.

```
GET
/api/v2/network
```

Ответ

- *network_id* - идентификатор сети,
- *centrifugo_url* - адрес сентрифуги,
- *test* - равно *true*, если сеть работает в тестовом режиме,
- *private* - равно *true*, если это частная сеть,
- *full_nodes* - массив нод.

Каждая нода описывает следующими параметрами

- *tcp_address* - адрес TCP сервера,
- *api_address* - адрес API сервера,
- *public_key* - публичный ключ ноды,
- *public_key* - публичный ключ ноды,
- *unban_time* - Если нода забанена, то это время, до которого действует запрет,
- *stopped* - равно *true*, если нода остановлена.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
     "network_id":"1",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "centrifugo_url":"http://127.0.0.1:8000",
    "test":true,
    "private":false,
    "full_nodes": [
        {"tcp_address":"http://mysite.com:7080",
         "api_address":"http://mysite.com:7079",
         "public_key":"04c0b50....3ba20e",
         "unban_time":"",
         "stopped":false
        }
     ]
}
```

Ошибки: *E_SERVER*

version

GET/ Возвращает текущую версию сервера. Запрос доступен без авторизации.

Запрос

```
GET
/api/v2/version
```

Вариант ответа

```
200 (OK)
Content-Type: application/json
"0.1.6"
```

3.9.5 Функции получения данных

balance

GET/ Возвращает баланс указанного аккаунта в текущей экосистеме.

Запрос

```
GET
/api/v2/balance/{key_id}
```

- *key_id* - идентификатор аккаунта, может быть представлен в любом формате - `int64`, `uint64`, `XXXX-...-XXXX`; поиск указанного аккаунта осуществляется в экосистеме, в которую вошел пользователь.

Ответ

- *amount* - сумма на аккаунте в минимальных единицах,
- *money* - сумма на аккаунте в единицах.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "amount": "123450000000000000000",
    "money": "123.45"
}
```

Ошибки: *E_SERVER, E_INVALIDWALLET*

blocks

GET/ Возвращает список блоков и их краткое содержимое. Запрос не требует авторизации.

Запрос

```
GET
/api/v2/blocks
```

Ответ

Номер блока

Для каждой транзакции в блоке: * *hash* - Хеш транзакции * *contract_name* - имя контракта * *params* - массив, содержащий параметры контракта * *key_id* - для первого блока - ID ключа, которым подписан первый блок, иначе - ID ключа, подписавшего транзакцию.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{"1":
    [{"hash":"PhHV1g7jUyDEwiETexBMLJPEwH4yEknCIIOAj43Dn4U=",
    "contract_name":"",
    "params":null,
    "key_id":-215783255460311196 3}]
}
```

Ошибки: *E_SERVER, E_NOTFOUND*

detailed_blocks

GET/ Возвращает список блоков и расширенную информацию об их содержимом. Запрос не требует авторизации.

Запрос

```
GET
/api/v2/detailed_blocks
```

Ответ

Номер блока

- ***header* - содержимое заголовка блока, содержит следующие поля:**
  - *block_id* - номер блока
  - *time* - время генерации блока

  - *key_id* - ID ключа подписавшего блок
  - *node_position* - номер ноды, сгенерировавшей блок
  - *version* - версия структуры блока
- *hash* - хеш блока
- *node_position* - номер ноды, сгенерировавшей блок
- *key_id* - ID ключа, подписавшего блок
- *time* - время генерации блока
- *tx_count* - количество транзакций в блоке
- *rollback_hash* - хеш транзакций в блоке
- *mrkl_root* - хеш листа дерева Меркла, соответствующего данному блоку
- *bin_data* - сериализованные заголовок блока; транзакции, вошедшие в блок; хеш предыдущего блока и приватный ключ ноды, сгенерировавшей блок
- *sys_update* - в блоке есть транзакция, обновившая системные параметры убрать из апи
- ***transactions* - массив транзакций блока, каждая запись содержит следующие поля:**
  - *hash* - хеш транзакции
  - *contract_name* - название конракта
  - *params* - массив параметров, переданных в контракт
  - *key_id* - ID ключа, подписавшего транзакцию
  - *time* - время генерации транзакции
  - *type* - тип транзакции

Вариант ответа

```
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":"TjTSRXcyJNgCn8GHEu16S2CheOOIZglxKQa/4S/Xzw4=",
    "ecosystem_id":0,
    "node_position":0,
    "key_id":3670289659738809576,
    "time":1545342081,
    "tx_count":1,
    "rollbacks_hash":"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=",
    "mrkl_root":
↪"NWNhODNmZGRiYmZhNTk3OTc0MzI1ODY4YjFiNDM3NDU3NTliOGUyNThmODM0NjY1ZWExOTkwZGZjNTZjZjhlMg==",
    "bin_data":null,
    "sys_update":false,
    "gen_block":false,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "stop_count":0,
    "transactions":[
        {"hash":"ZkGFY/WrvlsHXhZtpmodEoMX6MsBwF2Ji1G5Y7XgRjY=","contract_name":"","params":null,
↪"key_id":0,"time":0,"type":0}]
    }
}
```

Ошибки: *E_SERVER, E_NOTFOUND*

/data/{table}/{id}/{column}/{hash}

GET/ Возвращает данные из ячейки, заданной таблицей, колонкой и id записи, в случае, если хеш данных совпадает с отправленным хешем, иначе возвращает ошибку. Запрос не требует авторизации.

Запрос

```
GET
/data/{table}/{id}/{column}/{hash}
```

- *table* - название таблицы
- *id* - id записи
- *column* - имя колонки
- *hash* - хеш запрашиваемых данных

Ответ

Бинарные данные

keyinfo

GET/ Возвращает список экосистем с ролями, где зарегистрирован данный ключ. Запрос не требует авторизации.

Запрос

```
GET
/api/v2/keyinfo/{key_id}
```

- *key_id* - идентификатор аккаунта, может быть представлен в любом формате - `int64, uint64, XXXX-...-XXXX`; поиск указанного аккаунта осуществляется во всех экосистемах.

Ответ

- *ecosystem* - идентификатор экосистемы,
- *name* - наименование экосистемы,
- *roles* - список ролей пользователя в этой экосистеме с полями *id* и «name».

Вариант ответа

```
200 (OK)
Content-Type: application/json
[{
     "ecosystem":"1",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "name":"platform ecosystem",
    "roles":[{"id":"1","name":"Admin"},{"id":"2","name":"Developer"}]
}]
```

Ошибки: *E_SERVER, E_INVALIDWALLET*

3.9.6 Получение метрик

keys

GET/ Возвращает количество ключей

```
GET
/api/v2/metrics/keys
```

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

blocks

GET/ Возвращает количество блоков

```
GET
/api/v2/metrics/blocks
```

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

mem

GET/ Возвращает информацию об используемой памяти. Данный вызов доступен без авторизации.

```
GET
/api/v2/metrics/mem
```

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "alloc": 9608184,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "sys": 72349944
}
```

transactions

GET/ Возвращает количество транзакций

```
GET
/api/v2/metrics/transactions
```

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

ecosystems

GET/ Возвращает количество экосистем

```
GET
/api/v2/metrics/ecosystems
```

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

fullnodes

GET/ Возвращает количество валидирующих нод

```
GET
/api/v2/metrics/fullnodes
```

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "count": 28
}
```

3.9.7 Работа с экосистемами

ecosystemname

GET/ Возвращает имя экосистемы по коду. Данный метод api не требует авторизации.

```
GET
/api/v2/ecosystemname?id=..
```

- *id* - код экосистемы

Варианты ответа

```
200 (OK)
Content-Type: application/json
{
    "ecosystem_name": "platform_ecosystem"
}
```

Ошибки: *E_PARAMNOTFOUND*

appparams

GET/ Возвращает список параметров приложения в текущей или указанной экосистеме.

Запрос

```
GET
/api/v2/appparams/{appid}[?ecosystem=...&names=...]
```

- *[appid]* - идентификатор приложения,
- *[ecosystem]* - идентификатор экосистемы; если не указан, то будут возвращены параметры текущей экосистемы,
- *[names]* - список получаемых параметров; при желании можно указать через запятую список имен получаемых параметров, например, `/api/v2/appparams/1?names=name,mypar`.

Ответ

- *list* - массив, каждый элемент которого содержит следующие параметры.
  - *name* - наименование параметра,
  - *value* - значение параметра,
  - *conditions* - права на изменение параметра.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "list": [{
        "name": "name",
        "value": "MyState",
        "conditions": "true",
    },
    {
        "name": "mypar",
        "value": "My value",
        "conditions": "true",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    },
    ]
}
```

Ошибки: *E_ECOSYSTEM*

appcontent

GET/ Возвращает списки (id , название) для страниц, интерфейсных блоков и контрактов для заданного приложения.

Запрос

```
GET
/api/v2/appcontent/{appid}[?ecosystem=...]
```

- *[appid]* - идентификатор приложения,
- *[ecosystem]* - идентификатор экосистемы; если не указан, то будут возвращены параметры текущей экосистемы

Ответ

- *[list], [list], [list]* - массив с описанием блоков интерфейса, массив с описанием страниц, массив с описанием контрактов. Каждый массив содержит элементы с полями:
  - *id* - наименование параметра,
  - *name* - значение параметра.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "blocks": [
        { "id": 1, "name": "admin_link" },
        { "id": 2, "name": "export_info" }
    ],
    "pages": [
        { "id": 1, "name": "admin_index" },
        { "id": 2, "name": "developer_index" }
    ],
    "contracts": [
        { "id": 1, "name": "AdminCondition" },
        { "id": 2, "name": "DeveloperCondition" }
    ]
}
```

Ошибки: *E_ECOSYSTEM*

appparam/{appid}/{name}

GET/ Возвращает информацию о параметре приложения с идентификатор **{appid}** и с именем **{name}** в текущей или указанной экосистеме.

Запрос

```
GET
/api/v2/{appid}/{appid}/{name}[?ecosystem=1]
```

- *appid* - идентификатор приложения,
- *name* - имя запрашиваемого параметра,
- *[ecosystem]* - можно указать идентификатор экосистемы. По умолчанию, возвратится значение текущей экосистемы.

Ответ

- *id* - идентификатор параметра,
- *name* - наименование параметра,
- *value* - значение параметра,
- *conditions* - условие изменения параметра.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "id": "10",
    "name": "par",
    "value": "My value",
    "conditions": "true"
}
```

Ошибки: *E_ECOSYSTEM,E_PARAMNOTFOUND*

ecosystemparams

GET/ Возвращает список параметров экосистемы.

Запрос

```
GET
/api/v2/ecosystemparams/[?ecosystem=...&names=...]
```

- *[ecosystem]* - идентификатор экосистемы; если не указан, то будут возвращены параметры текущей экосистемы,
- *[names]* - список получаемых параметров; при желании можно указать через запятую список имен получаемых параметров, например, `/api/v2/ecosystemparams/?names=name,currency,logo`.

Ответ

- *list* - массив, каждый элемент которого содержит следующие параметры.
  - *name* - наименование параметра,
  - *value* - значение параметра,
  - *conditions* - права на изменение параметра.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "list": [{
        "name": "name",
        "value": "MyState",
        "conditions": "true",
    },
    {
        "name": "currency",
        "value": "MY",
        "conditions": "true",
    },
    ]
}
```

Ошибки: *E_ECOSYSTEM*

ecosystemparam/{name}

GET/ Возвращает информацию о параметре с именем **{name}** в текущей или указанной экосистеме.

Запрос

```
GET
/api/v2/ecosystemparam/{name}[?ecosystem=1]
```

- *name* - имя запрашиваемого параметра,
- *[ecosystem]* - можно указать идентификатор экосистемы. По умолчанию, возвратится значение текущей экосистемы.

Ответ

- *name* - наименование параметра,
- *value* - значение параметра,
- *conditions* - условие изменения параметра.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "name": "currency",
    "value": "MYCUR",
    "conditions": "true"
}
```

Ошибки: *E_ECOSYSTEM,E_PARAMNOTFOUND*

tables/[?limit=...&offset=...]

GET/ Возвращает список таблиц в текущей экосистеме. Можно указать смещение и количество запрашиваемых таблицы.

Запрос

- *[limit]* - количество записей, по умолчанию - 25,
- *[offset]* - смещение начала записей, по умолчанию - 0,

```
GET
/api/v2/tables
```

Ответ

- *count* - общее количество записей в таблице,
- *list* - массив, каждый элемент которого содержит следующие параметры:
  - *name* - имя таблицы, возвращается без префикса,
  - *count* - количество записей в таблице.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "count": "100"
    "list": [{
        "name": "accounts",
        "count": "10",
    },
    {
        "name": "citizens",
        "count": "5",
   },
    ]
}
```

table/{name}

GET/ Возвращает информацию о таблице с указанным именем в текущей экосистеме.

Возвращаются следующие поля: * *name* - имя таблицы, * *insert* - права на вставку элементов, * *new_column* - права на добавление клонки, * *update* - права на изменене прав, * *columns* - массив колонок с полями `name, type, perm` - имя, тип, права на изменение.

Запрос

```
GET
/api/v2/table/mytable
```

- *name* - имя таблицы (без префикса-идентифкатора экосистемы)

Ответ

- *name* - имя таблицы (без префикса-идентифкатора экосистемы),
- *insert* - право на добавление записей,
- *new_column* - право на добавление колонки,
- *update* - право на изменение записей,
- *conditions* - право на изменение настроек таблицы,
- *columns* - массив информации о колонках:

  - *name* - имя столбца,
  - *type* - тип колонки; dозможны следующие значения: `varchar, bytea, number, money, text, double, character`,
  - *perm* - права на изменение записе в столбце.

Вариант ответа

```
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`)" }
    ]
}
```

Ошибки: *E_ TABLENOTFOUND*

list/{name}[?limit=. . . &offset=. . . &columns=]

GET/ Возвращает список записей указанной таблицы в текущей экосистеме. Можно указать смещение и количество запрашиваемых элементов таблицы.

Запрос

- *name* - имя таблицы,
- *[limit]* - количество записей, по умолчанию - 25,
- *[offset]* - смещение начала записей, по умолчанию - 0,
- *[columns]* - список запрашиваемых колонок через запятую; если не указано, то будут возвращены все колонки; колонка id возвращается в любом случае

```
GET
/api/v2/list/mytable?columns=name
```

Ответ

- *count* - общее количество записей в таблице,
- *list* - массив, каждый элемент которого содержит следующие параметры:
  - *id* - идентификатор записи,
  - *columns* - последовательность запрошенных колонок.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "count": "10"
    "list": [{
        "id": "1",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "name": "John",
    },
    {
        "id": "2",
        "name": "Mark",
   },
    ]
}
```

Ошибки: *E_ TABLENOTFOUND*

sections[?limit=. . . &offset=. . . &lang=]

GET/ Возвращает список записей таблицы *sections* в текущей экосистеме. Можно указать смещение и количество запрашиваемых элементов таблицы. При этом, если поле *roles_ access* содержит список ролей и текущей роли там нет, то эта запись не будет возвращаться. Также, вданных столбца *title* происходит замена языковых ресурсов.

Запрос

- *[limit]* - количество записей, по умолчанию - 25,
- *[offset]* - смещение начала записей, по умолчанию - 0,
- *[lang]* - можно указать двухбуквенный код языка или lcid, для подключения соответствующих языковых ресурсов. Например, *en,ru,fr,en-US,en-GB*. Если, например, не будет найден ресурс для *en-US*, то он будет искаться для *en*.

```
GET
/api/v2/sections
```

Ответ

- *count* - общее количество записей в таблице,
- *list* - массив, каждый элемент которого содержит все столбцы таблицы *sections*.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "count": "2"
    "list": [{
        "id": "1",
        "title": "Development",
        "urlpage": "develop",
        ...
    },
    ]
}
```

Ошибки: *E_ TABLENOTFOUND*

row/{tablename}/{id}[?columns=]

GET/ Возвращает запись таблицы с указанным id в текущей экосистеме. Можно указать возвращаемые колонки.

Запрос

- *tablename* - имя таблицы,
- *id* - идентификатор записи,
- *[columns]* - список запрашиваемых колонок через запятую, если не указано, то будут возвращены все колонки; колонка id возвращается в любом случае.

```
GET
/api/v2/row/mytable/10?columns=name
```

Ответ

- *value* - массив полученных значений колонок:
  - *id* - идентификатор записи,
  - последовательность запрошенных колонок.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "values": {
    "id": "10",
    "name": "John",
    }
}
```

Ошибки: *E_QUERY*

row/{tablename}/{column}/{value}[?columns=]

GET/ Возвращает запись таблицы с указанным значением колонки в текущей экосистеме. Можно указать возвращаемые колонки.

Запрос

- *tablename* - имя таблицы,
- *column* - имя колонки,
- *value* - значение колонки,
- *[columns]* - список запрашиваемых колонок через запятую, если не указано, то будут возвращены все колонки; колонка id возвращается в любом случае.

```
GET
/api/v2/row/mytable/name/john?columns=name
```

Ответ

- *value* - массив полученных значений колонок:
  - *id* - идентификатор записи,

– последовательность запрошенных колонок.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "values": {
    "id": "10",
    "name": "John",
    }
}
```

Ошибки: *E_QUERY*

systemparams

GET/ Возвращает список системных параметров.

Запрос

```
GET
/api/v2/systemparams/[?names=...]
```

- *[names]* - список получаемых параметров; при желании можно указать через запятую список имен получаемых параметров, например, `/api/v2/systemparams/?names=max_columns,max_indexes`.

Ответ

- *list* - массив, каждый элемент которого содержит следующие параметры:
  - *name* - наименование параметра,
  - *value* - значение параметра,
  - *conditions* - права на изменение параметра.

Вариант ответа

```
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/ Возвращает историю изменения записи указанной таблицы в текущей экосистеме.

Запрос

- *name* - имя таблицы,
- *id* - идентификатор записи.

```
GET
/api/v2/history/pages/1
```

Ответ

- *list* - массив, каждый элемент которого содержит измененные параметры для запрашиваемой записи

Вариант ответа

```
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/ Возвращает запись таблицы page, menu или block с указанным name в текущей экосистеме.

Запрос

- *name* - название записи в указанной таблице.

```
GET
/api/v2/interface/page/default_page
```

Ответ

- *id* - идентификатор записи,
- *name* - название записи,
- другие колонки таблицы.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "id": "1",
    "name": "default_page",
    "value": "P(Page content)",
    "default_menu": "default_menu",
    "validate_count": 1
}
```

Ошибки: *E_ QUERY*, *E_ NOTFOUND*

3.9.8 Функции работы с контрактами

contracts[?limit=. . . &offset=. . .]

GET/ Возвращает список контрактов в текущей экосистеме. Можно указать смещение и количество запрашиваемых контрактов.

Запрос

- *[limit]* - количество записей, по умолчанию - 25,
- *[offset]* - смещение начала записей, по умолчанию - 06.

```
GET
/api/v2/contracts
```

Ответ

- *count* - общее количество записей в таблице,
- *list* - массив, каждый элемент которого содержит следующие параметры:
  - *id* - идентификатор записи,
  - *name* - имя контракта,
  - *value* - исходный текст контракта,
  - *active* - равно «1», если контракт привязан к аккаунту, и «0» в противном случае,
  - *key_ id* - аккаунт привязанный к контракту,
  - *address* - адрес аккаунта привязанного к контракту в формате `XXXX-...-XXXX`.
  - *conditions* - права на изменение контракта,
  - *token_ id* - экосистема, в токенах которой оплачивается контракт.

Вариант ответа

```
  200 (OK)
  Content-Type: application/json
  {
      "count": "10"
      "list": [{
          "id": "1",
          "name": "MainCondition",
          "token_id":"1",
          "key_id":"2061870654370469385",
          "active":"0",
          "value":"contract MainCondition {
conditions {
    if(StateVal(`founder_account`)!=$citizen)
    {
        warning `Sorry, you dont have access to this action.`
      }
    }
  }",
  "address":"0206-1870-6543-7046-9385",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  "conditions":"ContractConditions(`MainCondition`)"
   },
  ...
    ]
  }
```

contract/{name}

GET/ Возвращает информацию о смарт конракте с именем **{name}**. По умолчанию, смарт контракт ищется в текущей экосистеме.

Запрос

- *name* - имя смарт контракта.

```
GET
/api/v2/contract/mycontract
```

Ответ

- *id* - идентификатор контракта в Виртуальной машине
- *name* - имя смарт контракта с идентификатором экосистемы, например, `@{idecosystem}name`.
- *state* - ID экосистемы, в которой создан контракт,
- *walletid* - ID кошелька владельца контракта,
- *tokenid* - токены, в которых производится оплата за контракт,
- *address* - адрес аккаунта привязанного к контракту в формате `XXXX-...-XXXX`,
- *tableid* - идентификатор записи в таблице contracts, где хранится исходный код контракта,
- *fields* - массив, содержащий информацию о каждом параметре в разделе **data** контракта и содержит поля:
  - *name* - имя поля,
  - *type* - тип парметра,
  - *optional* - true если параметр опциональный и false в противном случае.

Вариант ответа

```
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/ Принимает транзакции, переданные в параметрах и складывает в очередь на обработку. В случае успешного выполнения возвращается хэш транзакции, с помощью которого можно получить номер блока в случае успешного выполнения или текст ошибки.

Запрос

- *any_ key* - содержимое транзакции, в качестве названия параметра может быть произвольным.

Метод поддерживает прием нескольких транзакций.

```
POST
/api/v2/sendTx

Заголовки:
Content-Type: multipart/form-data

Параметры:
tx1 - содержимое первой транзакции
txN - содержимое N-ой транзакции
```

Ответ

- ***hashes* - словарь с хешами отправленных транзакций**
  - *tx1* - hex хэш 1 транзакции;
  - *txN* - hex хэш N-ой транзакции.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "hashes": {
        "tx1": "67afbc435634.....",
        "txN": "89ce4498eaf7.....",
}
```

Ошибки: *E_ LIMITTXSIZE*

txstatus/

POST/ Возвращает номер блока или ошибку отправленных транзакции с данными хэшами. Если возвращаемые значения *blockid* и *errmsg* пустые, значит транзакция еще не была запечатана в блок.

Запрос

- *data* - json содержащий список хэшей проверяемых транзакций.

```
{"hashes":["contract1hash", "contract2hash", "contract3hash"]}
```

```
POST
/api/v2/txstatus/
```

Ответ

- *results* - словарь содержащий в качестве ключа хэш транзакции а в качестве значения результат выполнения.

hash - хэш транзакции

- *blockid* - номер блока, в случае успешной обработки транзакции,
- *result* - результат работы транзакции, возвращаемый через переменную **$result**,
- *errmsg* - текст ошибки, в случае отклонения транзакции.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{"results":
  {
    "hash1": {
         "blockid": "3123",
         "result": "",
     },
     "hash2": {
          "blockid": "3124",
          "result": "",
     }
   }
 }
```

Ошибки: *E_HASHWRONG, E_HASHNOTFOUND*

txinfo/{hash}

GET/ Возвращает данные о транзакции с данным хэшем. Возвращается номер блока и количество поджтверждений, кроме этого, можно получить имя соответствующего контракта и параметры, с которыми он был вызван.

Запрос

- *hash* - хэш проверяемой транзакции,
- *[contractinfo]* - для получения информации о контракте и параметрах, укажите этот параметр со значением 1.

```
GET
/api/v2/txinfo/2353467abcd7436ef47438
```

Ответ

- *blockid* - номер блока, в который попала транзакции. Если равен 0, то транзакция не найдена,
- *confirm* - количество подтверждений данного блока,
- *data* - если был указан параметр *contentinfo*, то здесь вернется json информация о контракте и параметрах.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "blockid": "4235237",
    "confirm": "10"
}
```

Ошибки: *E_HASHWRONG*

txinfoMultiple/

GET/ Возвращает инфоромацию о транзакциях с данными хэшами.

Запрос

- *data* - json содержащий список хэшей проверяемых транзакций в виде шестнадцатеричных строк.
- *[contractinfo]* - для получения информации о контракте и параметрах, укажите этот параметр со значением 1.

```
{"hashes":["contract1hash", "contract2hash", "contract3hash"]}
```

```
GET
/api/v2/txinfoMultiple/
```

Ответ

- *results* - словарь содержащий в качестве ключа хэш транзакции а в качестве значения результат выполнения.

  hash - хэш транзакции

  - *blockid* - номер блока, вкоторый попала транзакции,
  - *confirm* - количество подтверждения данного блока,
  - *data* - если был указан параметр *contentinfo*, то здесь вернется json информация о контракте и параметрах.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{"results":
  {
    "hash1": {
         "blockid": "3123",
         "confirm": "5",
     },
     "hash2": {
          "blockid": "3124",
          "confirm": "3",
     }
   }
 }
```

Ошибки: *E_HASHWRONG*

/page/validators_count/{name}

GET/ Возвращает количество нод валидации для страницы **{name}**

Запрос

- *name* - имя страницы с префиксом экосистеме, в формате: @1page_name, где @1 указывает на индекс экосистемы

```
GET
/api/v2/page/validators_count/@1page_name
```

Ответ

- *validate_ count* - количество нод для валидации

Вариант ответа

```
200 (OK)
Content-Type: application/json
{"validate_count":1}
```

Ошибки: *E_ NOTFOUND, E_ SERVER*

content/{menu|page}/{name}

POST/ Возвращает JSON представление кода указанной страницы или меню с именем **{name}**, которое получается после обработки шаблонизатором. При запросе можно передавать дополнительные параметры, которые можно использовать в шаблонизаторе. Если страница или меню не найдены, то возвращается ошибка 404.

Запрос

- *menu|page* - *page* или *menu* для получения страницы или меню соответственно,
- *name* - имя получаемой страницы или меню,
- *[lang]* - можно указать двухбуквенный код языка или lcid, для подключения соответствующих языковых ресурсов. Например, *en,ru,fr,en-US,en-GB*. Если, например, не будет найден ресурс для *en-US*, то он будет искаться для *en*.
- *[app_ id]* - ID приложения. Передается вместе с lang,т.к функции работающие с языком в шаблонизаторе не знают AppID. Передавать как число.

```
POST
/api/v2/content/page/default
```

Ответ

- *menu* - имя меню для страницы при вызове *content/page/...*,
- *menutree* - JSON дерево меню для страницы при вызове *content/page/...*,
- *title* - заголовок для меню *content/menu/...*,
- *tree* - JSON дерево объектов.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "tree": {"type":"......",
          "children": [
               {...},
               {...}
          ]
    },
}
```

Ошибки: *E_NOTFOUND, E_SERVER, E_HEAVYPAGE*

content/source/{name}

POST/ Возвращает JSON представление кода указанной страницы с именем **{name}** без выполнения функций и получения данных. Возвращаемое дерево соответствует шаблону страницы и может быть использовано в визуальном конструкторе. Если страница или меню не найдены, то возвращается ошибка 404.

Запрос

- *name* - имя получаемой страницы.

```
POST
/api/v2/content/source/default
```

Ответ

- *tree* - JSON дерево объектов.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "tree": {"type":"......",
          "children": [
               {...},
               {...}
          ]
    },
}
```

Ошибки: *E_NOTFOUND, E_SERVER*

content/hash/{name}

POST/ Возвращает SHA256 хэш-значение страницы с именем **{name}**. Если страница или меню не найдены, то возвращается ошибка 404. Данный метод api не требует авторизации. Так как метод не требует авторизации, то для того, чтобы получить правильный хэш при обращении к другим нодам, необходимо также передавать параметры, которые перечислены после *name*. Для получения страниц из других экосистем необходимо добавить префикс @(ecosystemId) к имени страницы. Например, *@2mypage*.

Запрос

- *name* - имя получаемой страницы,
- *ecosystem* - идентификатор экосистемы,
- *keyID* - идентифкатор пользлзователя,
- *roleID* - идентификаторо роли пользователя,
- *isMobile* - признак запуска на мобильной платформе.

```
POST
/api/v2/content/hash/default
```

Ответ

- *hex* - результирующий хэш в виде шестнадцатеричной строки,

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "hash": "01fa34b589...."
}
```

Ошибки: *E_NOTFOUND, E_SERVER, E_HEAVYPAGE*

content

POST/ Возвращает JSON представление кода указанного в параметре **template**. Если указан дополнительный параметр **source** равный *true* или *1*, то возвратится JSON представление без выполнения функций и получения данных. Возвращаемое дерево соответствует переданному шаблону и может быть использовано в визуальном конструкторе. Запрос доступен без авторизации.

Запрос

- *template* - текст шаблона страницы для разбора,
- *[source]* - если равен *true* или *1*, то дерево возвратится без выполнения функций и получения данных.

```
POST
/api/v2/content
```

Ответ

- *tree* - JSON дерево объектов.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "tree": {"type":"......",
          "children": [
               {...},
               {...}
          ]
    },
}
```

Ошибки: *E_NOTFOUND, E_SERVER*

node/{name}

POST/ Вызвает смарт-контракт с указанным именем **{name}** от имени ноды. Используется для вызова смарт контрактов из OBS контрактов через функцию **HTTPRequest**. Так как в этом случае мы не можем подписать контракт, то контракт будет подписан приватным ключом ноды. Все остальные параметры, такие же как при отправке контракта. Также нужно учитывать, чтобы вызываемый контракт

был привязан к аккаунту. В противном случае, на счету у приватного ключа ноды нет средств на выполнение контракта. Если вызов происходит из obs контракта, то необходимо передать в **HTTPRequest** токен авторизации **$auth_token**.

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

Запрос

```
POST
/api/v2/node/mycontract
```

Ответ

- *hash* - hex хэш отправленной транзакции.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "hash" : "67afbc435634.....",
}
```

Ошибки: *E_CONTRACT, E_EMPTYPUBLIC, E_EMPTYSIGN*

maxblockid

GET/ Возвращает максимальный id блока на текущей ноде. Данный метод api не требует авторизации.

Запрос

```
GET
/api/v2/maxblockid
```

Ответ

- *max_block_id* - максимальный id блока на текущей ноде.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "max_block_id" : 341,
}
```

Ошибки: *E_NOTFOUND*

block/{id}

GET/ Возвращает информацию о блоке с указанным ID. Данный метод api не требует авторизации.

Запрос

- *id* - id запрашиваемого блока.

```
POST
/api/v2/block/32
```

Ответ

- *hash* - хэш блока.
- *ecosystem_id* - id экосистемы.
- *key_id* - каким ключом был подписан блок.
- *time* - timestamp генерации блока.
- *tx_count* - количество транзакции в блоке.
- *rollbacks_hash* - хэш роллбеков, созданных транзакциями блока.

Вариант ответа

```
200 (OK)
Content-Type: application/json
{
    "hash": "\x1214451d1144a51",
    "ecosystem_id": 1,
    "key_id": -13646477,
    "time": 134415251,
    "tx_count": 3,
    "rollbacks_hash": "\xa1234b1234"
}
```

Ошибки: *E_NOTFOUND*

avatar/{ecosystem}/{member}

GET/ Возвращает аватар пользователя (доступно без авторизации)

Запрос

- *ecosystem* - id экосистемы пользователя
- *member* - id пользователя

```
GET
/api/v2/avatar/1/7136200061669836581
```

Ответ

Заголовок Content-Type с типом изображения Изображение в теле

Вариант ответа

```
200 (OK)
Content-Type: image/png
```

Ошибки: *E_NOTFOUND E_SERVER*

config/centrifugo

GET/ Возвращает хост и порт для подключения к centrifugo. Запрос доступен без авторизации.

Запрос

```
GET
/api/v2/config/centrifugo
```

Ответ

Строка http://127.0.0.1:8000 в теле ответа

Ошибки: *E_SERVER*

updnotificator

POST/ инициирует отправку неотправленных сообщений в центрифугу для заданных экосистем и пользователей. Запрос доступен без авторизации.

Запрос

Список вида:

- *id* - ID пользователя
- *ecoysystem* - ID экосистемы

```
POST
/updnotificator
```

Ответ

```
200 (OK)
Content-Type: application/json
{
    "result": true
}
```

3.10 Клиент для управления обновлениями

REST клиент, позволяющий обращаться к update-серверу из командной строки. Позволяет:

- Залить бинарник на update сервер.
- Стянуть бинарник с апдейт сервера
- Удалить бинарник с апдейт сервера
- Получить список версий с апдейт сервера
- Сгенерировать ключи
- Обновить бинарник go-apla

3.10.1 Где находится

tools/update_client/

3.10.2 Команды и флаги

add-binary

Добавить бинарник на апдейт сервер.

- –**server** - адрес сервера обновлений.
- –**login** - ваш логин на сервере обновлений.
- –**password** - ваш пароль на сервере обновлений.
- –**binary-path** - путь к загружаемому бинарнику.
- –**start-block** - с какого блока данный бинарь можно использовать.
- –**version** - название версии загружамого бинарника.
- –**key-path** - путь к приватному ключу для подпись бинарника.

get-binary

Скачать бинарник с апдейт сервера.

- –**server** - адрес сервера обновлений.
- –**version** - какую версию скачать.
- –**binary-path** - куда сохранить бинарник.
- –**publ-key-path** - путь к публичному ключу.

remove-binary

Удалить бинарник определенной версии с апдейт сервера.

- –**server** - ардес сервера обновлений.
- –**login** - ваш логин на сервере обновлений.
- –**password** - ваш пароль на сервере обновлений.
- –**version** - версия которая будет удалена.

generate-keys

Сгенерировать пару ключей - публичный и приватный.

- –**publ-key-path** - путь к публичному ключу. По умолчанию - resources/key.pub.
- –**key-path** - путь к приватному ключу. По умолчанию - resources/key.

versions

Получить список версий доступных для скачивания.

- –**server** - ардес сервера обновлений.
- –**version** - используется в случае если необходимо узнать о наличии конкретной версии.

3.11 Утилита мониторинга рассинхронизации

Специальная утилита, которая позволяет узнать, не произошло ли рассинхронизации БД на указанных нодах. Утилита может работать в качестве демона или быть вызвана однократно. Принцип работы основан на следующем: В каждый блок записывается хэш от всех изменений, сделанными всеми транзакциями блока.Затем у указанных нод берется максимальный общий блок, блок с этим ID запрашивается у всех нод, сравнивается хэш изменений, если он различается, то на заданный email посылается сообщение о рассинхронизации.

3.11.1 Где находится

tools/desync_monitor/

3.11.2 Флаги

Утилита имеет следующие флаги командной строки:

- **confPath** - путь до файла конфигурации. По умолчанию config.toml.
- **nodesList** - список хостов опрашиваемых нод, через запятую. По умолчанию 127.0.0.1:7079.
- **daemonMode** - запуск в виде демона, применяется если нужен опрос каждые N секунд. По умолчанию false.
- **queryingPeriod** - если осуществлен запуск в виде демона, раз в сколько секунд опрашивать ноды. По умолчанию 1 секунда.
- **alertMessageTo** - на какой email отправлять сообщение с ошибкой. По умолчанию alert@apla.io.
- **alertMessageSubj** - тема сообщения об рассинхронизации. По умолчанию problem with nodes synchronization.
- **alertMessageFrom** - от кого будет получено сообщение. По умолчанию monitor@apla.io.
- **smtpHost** - хост SMTP сервера через который будет отправлен email. По умолчанию «».
- **smtpPort** - порт SMTP сервера через который будет оправлен email. По умолчанию 25.
- **smtpUsername** - имя пользователя, для подключения к SMTP. По умолчанию «».
- **smtpPassword** - пароль для подключения к SMTP. По умолчанию «».

3.11.3 Конфигурация

Утилита также имеет конфигурационный файл в формате toml. По умолчанию ищется файл config.toml в папке в которой запущен бинарник. Переопределить путь к файлу можно с помощью флага configPath

Пример конфигурационного файла

```
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)

(продолжение с предыдущей страницы)

```
subject = "problem with genesis nodes"
from = "monitor@genesis.space"

[smtp]
host = ""
port = 25
username = ""
password = ""
```

- **nodes_list** - список хостов опрашиваемых нод

[daemon]

Настройки режима демона.

- **daemon_mode** - работать в виде демона, периодически опрашивая указанный список хостов нод.
- **querying_period** - с каким периодом проводить опрос в секундах.

[alert_message]

Настройки отправляемого сообщения.

- **to** - на какой адрес слать сообщение об рассинхронизации блокчейна.
- **subject** - тема сообщения об рассинхронизации.
- **from** - от кого будет получено сообщение.

[smtp]

Настройки SMTP сервера, через который сообщение об рассинхронизации блокчейна будет отправлено.

- **host** - хост SMTP серевера, через который будет отправлен email.
- **port** - порт SMTP сервера, через который будет отправлен email.
- **username** - имя пользователя, для подключения к SMTP.
- **password** - пароль для подключения к SMTP.

3.12 Программа для сбора и шифрования блокчейна

Репозитарий: https://github.com/AplaProject/apla-pgp

Программа периодически обращается к базе данных к таблице block_chain и смотрит наличие новых блоков. Если имеются новые блоки, то она шифрует их публичным PGP ключом и сохраняет в определенную директорию в виде отдельных файлов с именем {номерблока}.block.

Программа хранит хэши сохраненных блоков, если она видит, что последний хэш не совпадает, то она ищет последний блок с одинаковым хешем и начинает скачивать блоки с того момента. Если старые файлы еще находятся в директории, то они будут перезаписаны.

Предварительно необходимо сгенерировать PGP ключи командой

$ gpg –gen-key

В процессе необходимо будет указать парольную фразу для приватного ключа - её нужно будет запомнить.

Для работы программы необходим файл настроек. Вот пример

```
LogFile = "apla.log"
StoreFile = "./store/apla-pgp.store"
OutPath = "./backup"

[Settings]
  Timeout = 2
  Compression = 1
  NodePrivateKey = "/home/ak/apla/apla-data/NodePrivateKey"

[PGP]
  Path = "/home/ak/.gnupg"
  Phrase = "1234"

[TCP]
  Host = "127.0.0.1:7078"
```

- **LogFile** - имя (можно с путем) лог-файла.
- **StoreFile** - имя файла, где будут хранится хэши обработанных блоков.
- **OutPath** - директория куда будут складываться зашифрованные файлы.
- **Timeout** - периодичность опроса в секундах.
- **Compression** - 0 - не сжимать блоки, если указать 1, то данные блоков будут предварительно сжиматься по формату gzip, если они больше 1024 байт. В файле .block ставится соответствующая метка.
- **NodePrivateKey** - приватный ключ , с помощью которого подписываются зашифрованные данные.
- **Path** - директория с PGP ключом .pubring.gpg

Если в этой директории также находится файл secring.gpg, то программа будет работать в тестовом режиме. В этом случае должна быть указана секретная фраза для приватного ключа Phrase. В тестовом режиме программа после сохранения .block файла, читает его, расшифровывает приватным PGP ключом, в случае необходимости, распаковывает и затем сравнивает с оригинальным блоком.

В секции TCP необходимо указать host для подключения по tcp.

3.13 Установка, настройка и запуск

В этом разделе рассматриваются вопросы установки, настройки и запуска .

3.13.1 Конфигурация

При установке в директории, в которой был запущен бинарник(если только не указан флаг workDir), будет создан файл config.toml, который используется для управление конфигурацией ноды.

Пример:

```
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

[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"

[BanKey]
  BanTime = 15
  BadTx = 3
  BadTime = 5
```

- **LogLevel** - Используемый уровень логгирования. Значения - одно из «DEBUG», «INFO», «WARN», «ERROR». В случае некорректного значения будет выставлен логлевел INFO.
- **LogFileName** - Имя файла, куда будет писаться лог. В случае пустого значения - лог пишеться в stderr.
- **InstallType** - тип установки. Сейчас есть только одно значение этого параметра - «PRIVATE_NET».
- **NodeStateID** - список id экосистем, относящихся к данной ноде. Допустимые значения - «*»,

«Список айдишников через запятую».

- **TestMode** - если выставлен - вместо параметра sysparam.GetRemoteHosts() в демоне Confirmations используется localhost.
- **StartDaemons** - какие демоны запускать. Допустимые значения «» - запускать все демоны, «null» - не запускать ни одного демона, «Список демонов через запятую».Имена демонов - BlocksCollection, BlockGenerator, Disseminator, QueueParserTx, QueueParserBlocks, Confirmations, Notificator.
- **KeyID** - адрес основного кошелька ноды.
- **MaxPageGenerationTime** - Максимальное время генерации страницы в миллисекундах.
- **EcosystemID** - ID основной экосистемы ноды.
- **BadBlocks** - список ID некорректных блоков.
- **FirstLoadBlockchain** - тип первоначальной загрузки блокчейна.
- **FirstLoadBlockchainURL** - с какого URL первоначально скачивать блокчейн.
- **WorkDir** - рабочая директория, где искать файлы первого блока, конфига, pidfile, роллбека.
- **PrivateDir** - директория, где хранятся файлы ключей. NodePrivateKey, NodePublicKey, PrivateKey, PublicKey, и KeyID.
- **RunningMode** - режим запуска ноды, допустимые значения: OBS, OBSMaster, PrivateBlockchain, PublicBlockchain.

TCPServer

Настройки TCPServer - компонента отвечающего за сетевое взаимодействие между нодами.

- **Host** - хост, на котором работает TCPServer.
- **Port** - порт, на котором работает TCPServer.

HTTPServer

Настройки HTTPServer, компонента, который предоставляет REST API.

- **Host** - хост, на котором работает HTTPServer.
- **Port** - порт, на котором работает HTTPServer.

DB

Настройки БД, в данном случае PostgreSQL.

- **Name** - имя базы данных , к которой производится подключение.
- **Host** - хост базы данных, к которой производится подключение.
- **Port** - порт базы данных, к которой производится подключение.
- **User** - имя пользователя БД.
- **Password** - пароль пользователя БД.

StatsD

Настройки StatsD - сборщика метрик работы ноды.

- **Name** - имя, с которого будет начинаться иерархия счетчиков метрик.
- **Host** - хост для подключения к StatsD.
- **Port** - порт для подключения к StatsD.

Centrifugo

Настройки Centrifugo - компонента, отвечающего за доставку уведомлений.

- **URL** - URL запущенного сервера centrifugo(https://github.com/centrifugal/centrifugo).
- **Secret** - Секретная строка centrifugo.

BanKey

Настройки бана ключей, которые отправляют плохие транзакции. Ключ заносится в бан на время **BanTime** (в минутах), если от ключа в течении времени **BadTime** (в минутах) пришло **BadTx** плохих транзакций.

- **BanTime** - время бана ключа в минутах. По умолчанию, 15 минут.
- **BadTx** - количество плохих транзакций за время **BadTime**, после которых ключ попадает в бан. По умолчанию, равно 3.
- **BadTime** - период в минутах, за который отслеживаются плохие транзакции. По умолчанию, 5 минут.

3.13.2 Флаги

- **workDir** - указать рабочую директорию.
- **centrifugoSecret** - секретный ключ для centrifugo.
- **centrifugoUrl** - url для centrifugo, в формате host:port.
- **checkReadAccess** - Проверять ли поля доступа к колонкам таблиц на чтение. Используется только для OBS.
- **configPath** - путь к .toml конфигу ноды.
- **dbHost** - хост БД. По умолчанию 127.0.0.1
- **dbName** - имя БД. По умолчанию apla.
- **dbPassword** - пароль к БД.
- **dbPort** - порт БД. По умолчанию 5432.
- **dbUser** - под каким бользователем соединятся с БД.
- **endBlockId** - На каком блоке blockCollection прекращает работу.
- **firstBlockHost** - Хост, который будет прописан в первом блоке при его генерации. По умолчанию 127.0.0.1.
- **firstBlockNodePublicKey** - Какой публичный ключ ноды будет прописан в первом блоке.

- **firstBlockPath** - Где брать первый блок.
- **firstBlockPublicKey** - Какой публичный ключ будет прописан в первом блоке когда он будет сгенерен.
- **httpHost** - хост, на котором запущен http - сервер. По умолчанию 127.0.0.1.
- **httpPort** - порт, на котором запущен http сервер. По умолчанию 7079.
- **generateFirstBlock** - сгенерить связку ключей(приватный/публичный ключ пользователя/ноды) и затем первый блок с этими ключами, если это не переопределено во флагах firstBlock*.
- **initConfig** - сохранить конфиг из флагов.
- **initDatabase** - инициализировать подключение к БД.
- **keyID** - указать свой KeyID для работы.
- **logFile** - фаил для логгирования.
- **logLevel** - уровень детализации логирования. Один из:ERROR,WARN,INFO,DEBUG. По умолчанию ERROR.
- **logSQL** - логировать SQL запросы, которые генерит ORM. Используется для отладки.
- **logStackTrace** - отображать путь, по которому была вызвана данная строка в виде N названии функции. Используется для отладки.
- **noStart** - не стартовать демон, но выполнить все что указано в флагах. Используется когда демон запускается как тулза.
- **privateBlockchain** - приватный блокчейн или нет. Влияет на списание комиссии, в случае приватного блокчейна она не списывается.
- **privateDir** - директория для публичных/приватных ключей.
- **rollbackToBlockId** - указать Id блока к которому хотелось бы откатиться. Используется для ручного отката блокчейна к нужному блоку.
- **startBlockId** - с какого блока BlockCollection собирает блоки.
- **tcpHost** - хост TcpServer. По умолчанию 127.0.0.1.
- **tcpPort** - порт TcpServer. По умолчанию 7078.
- **testRollBack** - запустить специальный набор демонов(BlockCollection, Confirmations) для тестирования роллбеков.
- **tls** - Принимать HTTP запросы только по HTTPS. Указывает директорию где лежит .well-known и ключи.
- **updateInterval** - интервал, с которым проверяется наличие обновлениий. По умолчанию - 1 час.
- **updatePublicKeyPath** - публичный ключ для сервера автообновлений. По умолчанию «update.pub».
- **updateServer** - ардес сервера для автоапдейтов. По умолчанию http://127.0.0.1:12345.
- **runMode** - режим запуска ноды: OBS, OBSMaster, PrivateBlockchain, PublicBlockchain
- **banTime** - время бана ключа в минутах. По умолчанию, 15 минут.
- **badTx** - количество плохих транзакций за время **BadTime**, после которых ключ попадает в бан. По умолчанию, равно 3.
- **badTime** - период в минутах, за который отслеживаются плохие транзакции. По умолчанию, 5 минут.