Waves Book

Введение

Репозиторий, который у вас сейчас перед глазами, является попыткой помочь разработчикам, которые хотят делать децентрализованные приложения на базе блокчейна Waves, лучше понимать протокол и знать его особенности.

Данная книга может вам помочь максимально быстро сделать свое первое приложение на блокчейне Waves не допуская никаких фатальных ошибок. В этой книге акцент делается на то, что действительно важно для создания продуктов на базе Waves.

Эта книга акцентирована на примеры, разбор реальных кейсов, конкретные рецепты, которые вы можете применять в своих приложениях. Но, как и во многих аспектах жизни, переход к реальным задачам требует фундаментальных знаний о протоколе. Поэтому в первых трех главах рассказывается об особенностях работы блокчейна Waves, чем он отличается от других блокчейнов, наряду с разбором что и как может влиять на архитектуру вашего приложения и какие могут быть ограничения.

Книга сфокусирована на особенностях работы Waves, но не на базовых понятиях вроде "что такое блок?" и "как работает алгоритм консенсуса", так как материалов о том, как работает блокчейн, огромное количество, в то время как найти особенности отдельных протоколов, чтобы делать на них бизнес, достаточно проблематично.

Важно понимать, что книга не является "полным справочником" по Waves, так как делает акцент на прикладные аспекты протокола. Она не пытается конкурировать с документацией протокола, которая уже есть достаточно давно и покрывает многие аспекты работы с Waves.

1.1 История создания Waves

Что такое Waves?

Waves это Proof-of-Stake permissionless блокчейн платформа общего назначения, создаваемая с 2016 года и призванная помочь найти массовое применение технологии блокчейн. Блокчейн платформа Waves является одной из наиболее зрелых (как в виду возраста, так и в виду большого числа проектов) и легких для начинающих разработчиков, которые хотят использовать преимущества блокчейна с максимальной пользой. В рамках этой книги мы разберем основные технические особенности, поговорим о преимуществах и разберем много реального кода, но прежде чем заняться этим, немного поговорим про историю Waves, чтобы лучше понимать истоки тех или иных особенностей.

Начало проекта

Waves как блокчейн начался в 2016 году, когда основатель платформы Александр Иванов инициировал сбор средств в рамках ICO. Платформа с самого начала позиционировала себя как блокчейн общего назначения, без фокуса на отдельных специфических сферах применения. Основной задачей, которую собиралась решать (и во многом успешно решала) платформа является пропускная способность. На

начало 2016 года существовало очень мало блокчейнов, которые могли обрабатывать сотни транзакций в секунду. Фактически, на рынке полноценно работали только Bitcoin (и его форки вроде Litecoin) с 7 транзакциями в секунду и Ethereum с 15 транзакций/сек. Так что проблема пропускной способности блокчейна была крайне актуальной.

Основные вехи развития проекта

Успешный сбор средств на ICO был только началом проекта, дальше предстояло реализовать все обещанное максимально хорошо. Изначально была выбрана технологическая база фреймворка Scorex. От самого фреймворка в кодовой базе проекта сейчас почти ничего не осталось, но Scorex написан на языке Scala, что и определило надолго технологический стэк разработки протокола. До недавнего времени реализация ноды (то есть протокола) на Scala была единственной, только относительно недавно появилась так же реализация на Go. Стоит заметить, что на момент написания этих строк версия на Go по возможностям отставала от версии на Scala, об этом мы поговорим в следующих разделах. Запуск основной сети (в дальнейшем будем называть mainnet) Waves произошел в декабре 2016 года. Проект с самого начала имел особенности: легкий выпуск своих токенов/ассетов (с помощью отправки одной транзакции) и PoS с моделью лизинга (стейкинга). Данные особенности мы разберем в дальнейших разделах.

Другими примечательными вехами в истории протокола можно назвать следующие даты:

- Выпуск DEX (гибридной биржи) в 2017 году. Логичным продолжением нативной поддержки выпуска токенов с помощью транзакции (не контракта как в Ethereum), явился выпуск матчера, который обеспечивал работу децентрализованной биржи.
- В том же году был реализован протокол Waves NG, который позволил достичь хорошей пропускной способности. Waves NG был реализован на основе идей, изложенных в статье. Данные предложения были нацелены на реализацию в Bitcoin, но никогда не были там реализованы, зато были реализованы в Waves.
- Летом 2018 года был выпущен язык для смарт-контрактов Ride для базовой логики аккаунтов, а в января 2019 года появилась возможность писать контракты для токенов
- Летом 2019 года до мейннета добрался Ride для полноценных децентрализованных приложений (Ride4dApps)
- Осенью 2019 года появилась первая в своем роде (среди основных блокчейнов проектов) монетарная политика

Влияение истории Waves на протокол

История создания Waves достаточно сильно повлияла на технические детали проекта. Например, фреймворк Scorex корнями уходит в проект Nxt, другой Proof-of-Stake блокчейн с нативными токенами. Обе особенности были унаследованы Waves именно из Nxt. Легкость создания токенов и скорость работы протокола в дальнейшем сделали проект второй по популярности платформой для выпуска токенов и запуска ICO (сразу после Ethereum). Большое количество проектов на Waves использовали блокчейн именно для работы с токенами (выпуск, небольшая стоимость транзакций).

1.2 Подходы к разработке протокола Waves

Разработчики протокола Waves всегда руководствовались некоторыми базовыми принципами, которые сильно влияют на дальнейшее развитие протокола. Понимание данных принципов и мотивации за ними поможет легче следить за дальнейшим развитием проекта, поэтому перечислю данные особенности.

Блокчейн для людей

Мантрой Waves долгое время было "Blockchain for the people". Она полностью отражала и отражает то, что делает команда. Главное, чего хочет достичь платформа - **популяризировать технологию блокчейн для масс**. В данный момент блокчейн является технологией для очень небольшой группы людей, которые понимают, что это за технология и как ее правильно использовать. Waves хочет изменить такое положение вещей и сделать так, чтобы технология приносила максимальную пользу всем.

Многие люди думают, что технология блокчейн крайне сложная и наукоемкая (во многом так и есть), Waves же пытается скрывать всю сложность за неким слоем абсракции. Блокчейн - это не самая удобная база данных, у которой есть несколько важных свойств: децентрализация, неизменяемость и открытость. Данные особенности не являются ценностями сами по себе, а только в случае правильного применения разработчиками конкретных приложений. Цель Waves состоит в предоставлении таких инструментов разработчикам, которые позволят им быстрее,проще, без излишнего погружения в сложные технические давать ценность конечным пользователям.

Можно сказать, что принцип ориентации на реальное применение разбивается на несколько шагов:

1. Платформа предоставляет разработчикам приложений легкий инструмент для использования особенностей блокчейна
2. Разработчики приложений делают продукты, которые решают проблемы пользователей и правильно используют блокчейн
3. Пользователи получают получают преимущества блокчейна. При этом не обязательно, чтобы пользователи знали что-то про блокчейн. Главное, чтобы решалась их проблема.

Ориентация на практическую применимость

Всегда во время разработки нового функционала или продукта во главе угла ставится **практическая применимость** и какое количество людей **потенциально** смогут решить свои проблемы с помощью этого. Разработчики протокола пытаются не делать "космические корабли", решать "сферические проблемы в вакууме" или заниматься оверинжинирингом, выбирая применимость здесь и сейчас. Лучше ведь иметь работающее сейчас, чем идеальное через 10 лет? Конечно, данный принцип не должен вступать в противоречие с безопасностью сети.

Открытость разработки

Протокол Waves является полностью открытым и процесс разработки максимально децентрализован. Все исходные коды есть на github. Кроме непосредственно исходного кода там же обсуждаются пути развития протокола, различные проблемы и варианты их решения. Обновления протокола, связанные с изменением консенсуса всегда проходят процедуру обсуждения с помощью Waves Enhancement Proposals. Но обсуждения это только первый этап, ведь все обновления консенсуса должны еще проходить через процедуру активации с голосованием, о чем мы поговорим с следующем разделе. Теперь вы знаете, что делать, если захотите что-то изменить в протоколе.

1.3 Отличительные особенности блокчейна Waves

Если у вас уже есть опыт работы с другими блокчейнами, вам может быть интересно, чем же отличается Waves от условного Ethereum и почему он другой. Давайте быстро пройдемся по отличиям, которые будут детально рассматриваться в следующих разделах.

Работа с токенами/ассетами

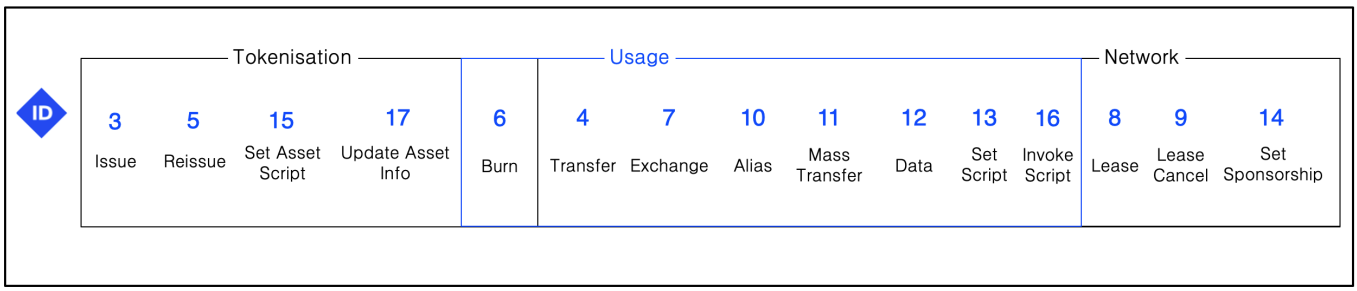
Одной из особенностей работы с Waves с первого дня была простота выпуска токенов. Для этого достаточно отправить транзакцию или заполнить форму из 5 полей) в любом UI клиенте. Выпущенный токен автоматически становится доступен для переводов, торговки на DEX, использования в dApp и сжигания.

В отличие от Ethereum, в Waves токены не являются смарт-контрактами, а являются "гражданами первого сорта", то есть являются отдельной полноценной сущностью. У этого есть как преимущества, так и недостатки, о которых мы поговорим в разделе 4 "Токены".

Транзакции

Другая отличительная особенность Waves заключается в наличии большого количества типов транзакций. Например, в Ethereum есть смарт-контракты, которые могут являться чем угодно, в зависимости от их реализации. ERC-20, описывающий токен, это просто описание интерфейса смарт контракта, какие методы он должен иметь. В Waves это не так, т.к. подразумеватся, что лучше иметь легковесные/узкие специфичные вещи, чем абстрактные вещи "обо всем и ни о чем". Специфичность примитивов упрощает во многих местах разработку, но это иногда является менее гибким решением.

Ниже представлен список актуальных транзакций на момент написания этих строк:



Fair PoS

В Waves используется алгоритм Proof-of-Stake для определения права на генерацию блока. Блоки генерируются в среднем каждую минуту, а вероятность генерации блока нодой зависит от 3 параметров:

- Генерирующего баланса ноды, то есть баланса самой ноды + количества токенов, которые сдали ей в лизинг.
- Текущего времени и рандома (великий рандом есть всегда в жизни!)
- Генерящего баланса сети, ведь не все токены в сети участвуют в генерации блоков, они могут быть в ордерах на биржах или лежать на холодных кошельках.

Чтобы начать генерировать блоки достаточно иметь 1000 Waves генерирующего баланса (свои + полученные в лизинг). Зачем вам генерировать блоки? За каждый блок нода получает на свой баланс комиссии из транзакций в этом блоке и вознаграждение "из воздуха". Оба этих момента не такие простые, так что рассмотрим их чуть позже.

Очень частый вопрос, который встречается - сколько блоков я буду генерировать в месяц с балансом N. Точное числе предсказать невозможно, так как зависит от случайностей и изменений в сети, но примерно предсказать можн. Чтобы сделать это надо знать текущие параметры сети:

1. *Сколько токенов Waves участвуют в генерации блоков?* Это так называемый генерирующий баланс сети в целом. Для легкости рассчета скажем, что 50 млн токенов участвуют в генерации блоков.
2. *Какой баланс нашей ноды?* То есть сколько у нас своих токенов на аккаунте ноды и сколько нам сдали в лизинг. В нашем случае возьмем генерирующий баланс равный 10 000 Waves.
3. Среднее время блока составляет 1 минуту, то есть в сети генерируется примерно 1440 блоков в день или 43200 блоков в месяц.

Для вычисления примерного количества блоков, которые мы сгенерируем, делим количество блоков за период на генерирующий баланс сети и умножаем на наш баланс:

$ForgedBlocks = BlocksCountInPeriod / NetworkGenBalance * NodeGenBalance$

Сделав нехитрые вычисления получаем:

```
43200 / 50000000 * 10000 = 8.64
```

То есть, в среднем нода будет генерировать 8-9 блоков в месяц, если будет работать стабильно и параметры сети не изменятся, но, такого, конечно, не бывает, ведь в сети постоянно делается большое количество транзакций.

Leasing

В Waves есть механизм стейкинга, который называется leasing. Любой владелец токена Waves может отправить токены в лизинг любой ноде Waves, чтобы та производила блоки "от имени этих токенов". Лизингодатель передает право на генерацию блоков от имени его токенов лизингополучателю (владельцу ноды). Обычно это делается, когда пользователь не хочет или не может заниматься разворачиваем своей ноды и ее поддержкой. Обычно владельцы лизнговых пулов выплачивают бОльшую часть заработанного за счет лизинга средств лизингодателям. Отправить в лизинг средства можно моментально, но в генерируюющем балансе ноды они начнут учитываться только через 1000 блоков.

Забрать токены из лизинга можно моментально.

Community driven monetary policy

Первое время в Waves была ограниченная эмиссия в 100 млн токенов, которые были выпущены сразу на момент запуска мейннета, но с осени 2019 года в комьюнити решили, что для дальнейшего роста экосистемы лучше будет при наличии эмиссии токенов. То есть в каждом новом блоке появляются новые токены Waves. Какое именно количество токенов определяется сообществом, которое голосует за размер вознаграждения каждые 100 тысяч блоков. На момент написания этих строк вознаграждение

за блок составляло 6 Waves. При этом гарантируется, что каждый период голосования размер вознаграждения не может изменяться больше, чем на 0.5 Waves.

Sponsorship

Функция спонсирования токена является способом снижения входных барьеров для пользователей. Суть в том, что аккаунт, выпустивший токен, может спонсировать транзакции с этим токеном. Представим, есть токен A, его выпустил Cooper и включил спонсирование. Например, Alice заработала 100 токенов A и хочет 10 из них отпраивть Bob. Мы то с вами знаем, что для каждой транзакции в блокчейне необхоодимо платить комиссию, в сети Waves майнеры принимают только Waves в виде комиссии, а у Alice нет Waves. Придется идти покупать Waves как-то?

Нет. Спонсорство токена позволяет владельцу токена сказать, что он готов взять на себя комиссии за операции с этим токеном (токеном A в нашем случае). Владельцы токена A при отправке транзакции будут платить этот же токен в виде комиссии. В нашем примере, Alice сможет указать в своей транзакции, что получатель Bob, количество для отправления - 10 токенов A, комиссия - 5 токенов A. В итоге с ее аккаунта спишется 15 токенов, 10 получит Bob, 5 получит Cooper, как выпустивший и спонсирующий токен, а майнер получит Waves с аккаунта Cooper'a.

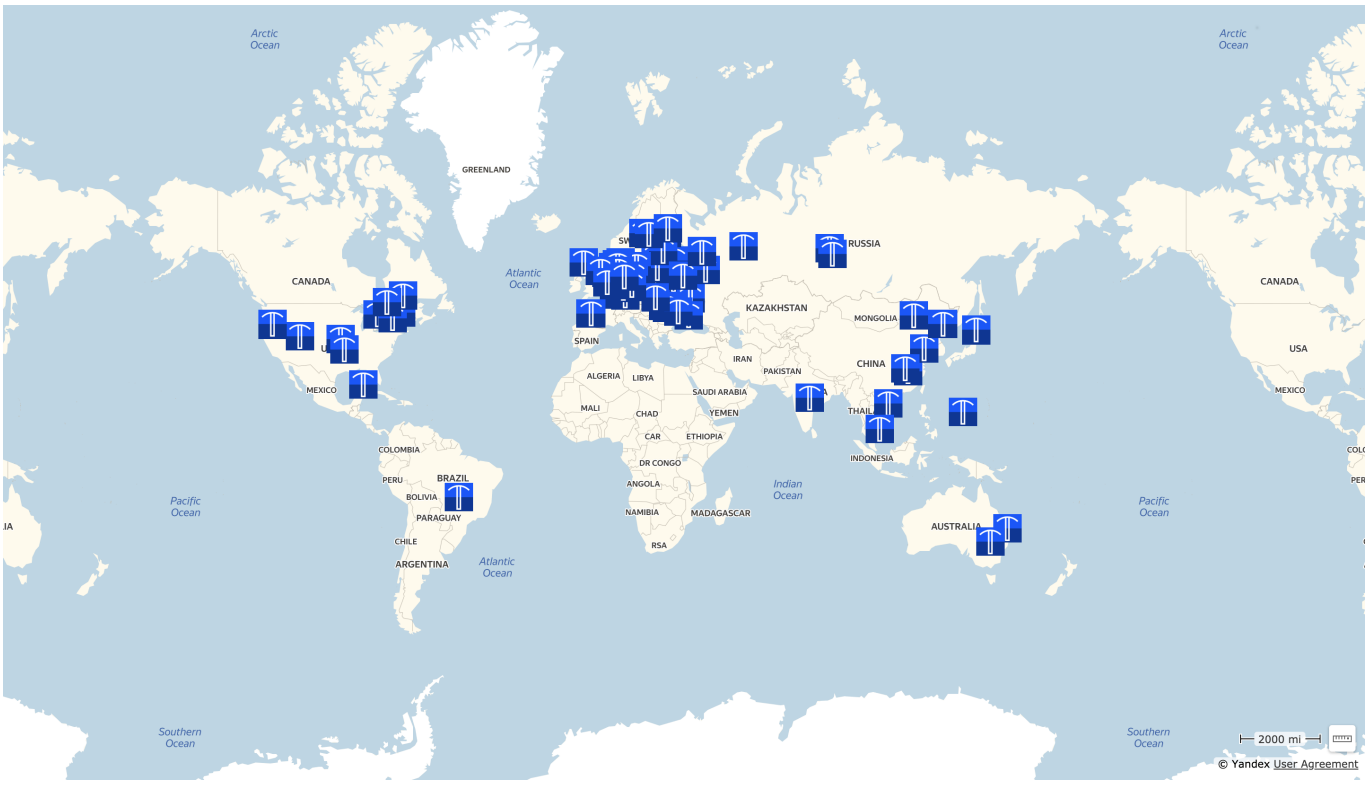
Почему Alice заплатит 5 токенов и сколько получит майнер? Поговорим об этом в следующих главах. Главное, что сейчас необходимо запомнить - **в Waves существуют способы сделать транзакции не имея токенов Waves у себя на балансе.**

Ride и смарт контракты

Waves является блокчейном общего назначения, не специализирующемся на чем-то одном, поэтому появление смарт-контрактов стало логичным продолжением развития платформы. Про смарт-контракты в Waves мы поговорим в разделе 6 "Ride". Сейчас стоит отметить, что контракты пишутся на языке Ride, который был придуман специально для смарт-контрактов и является не Тьюринг-полным. В языке нет циклов, но зато нет газа, не бывает failed транзакций в блокчейн, стоимость транзакции всегда известна заранее. Заинтриговал? Это ведь только начало, мы поговорим про модель исполнения Ride и синтаксис языка позже.

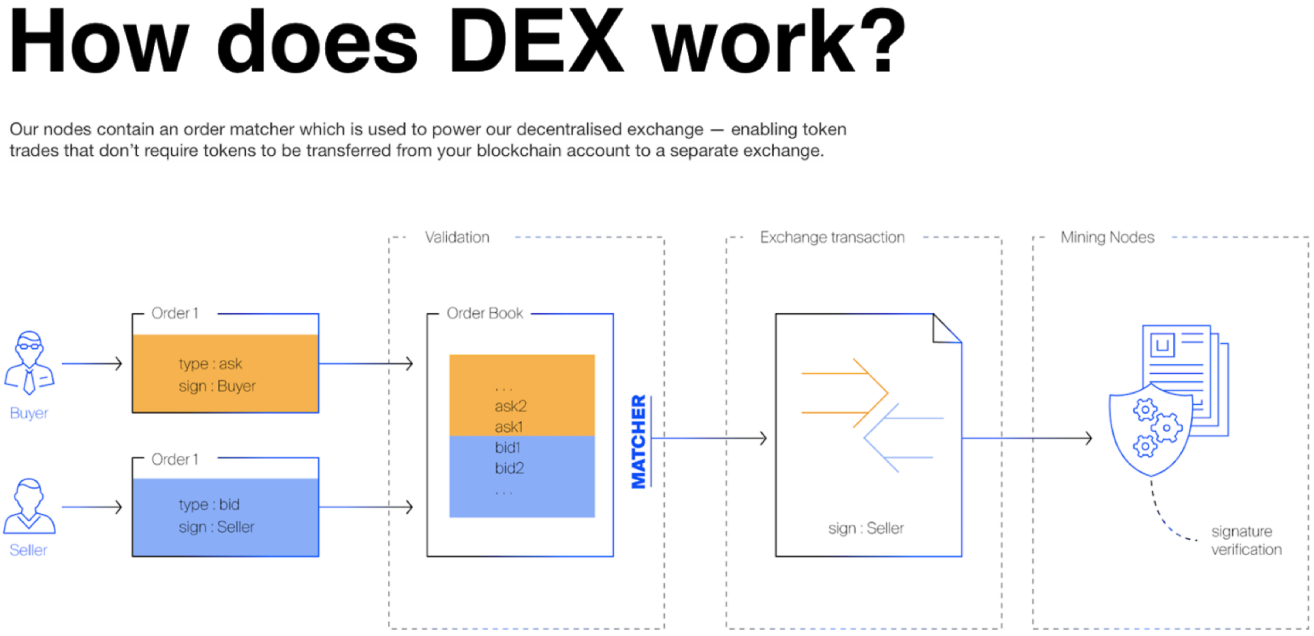
Waves NG

Чуть раньше я уже затрагивал тему, связанную с Waves NG и упоминал, что она позволяет транзакциям быстрее попадать в блоки и работать блокчейну так быстро, что платформа в состоянии обрабатывать сотни транзакций в секунду в основной сети. А там, на минуточку, больше 400 нод, распределенных по всему миру, на совершенно разном железе и пропускной способностью.



DEX

Легкий и быстрый выпуск токенов наряду с "классовым равенством" (помните, что токены являются гражданами первого сорта?) позволяет сделать торговлю токенами простой. Нода Waves (речь про версию на Scala) поддерживает возможность создания расширений, одним из таких расширений является матчер. Матчер принимает ордера на покупку и продажу токенов и хранит их (централизованно). Например, Alice хочет продать токен wBTC и купить Waves, а Bob наоборот. Они формируют ордера (криптографически подписанные примитивы) и отправляют их в матчер, который определяет, что эти ордера выставлены в одной паре и их можно сматчить по определенной цене. В результате матчер формирует Exchange транзакцию, которая содержит 2 ордера (один от Alice, другой от Bob) и отправляет в блокчейн. При этом, матчер забирает себе комиссию, нода, которая смайнит блок с Exchange транзаций, тоже получает комиссию.



2.1 Нода Waves и как она работает, ее конфигурация

Непосредственное техническое знакомство с платформой Waves я бы рекомендовал начать с установки и настройки ноды. Не обязательно для основной сети, можно для testnet (полная копия по техническим возможностям) или stagenet (экспериментальная сеть). Почему я рекомендую начать с установки ноды? Во-первых, я сам свое знакомство с блокчейном Waves начинал именно с этого, а во-вторых, установка и настройка заставляют разобраться в том, какие есть настройки у ноды, какие параметры есть у сети.

Из чего состоит нода Waves

Как и почти во всех блокчейнах, нода - программный продукт, который отвечает за прием транзакций, создание новых блоков, синхронизацию данных между разными узлами и достижение консенсуса между ними. Каждый участник сети запускает свою копию ноды и синхронизируется с остальными. Про правила консенсуса мы поговорим чуть позже, сейчас давайте разберемся, что из себя представляет нода с точки зрения ПО.

По большому счету, нода это исполняемый файл (`jar` файл для Scala версии и бинарный для Go), который в момент запуска читает конфигурационный файл, чтобы на основе этих параметров начать общаться с другими узлами в сети по протоколу поверх TCP. Получаемые и генерируемые данные нода складывает в LevelDB (key-value хранилище). По большому то счету все, но дьявол кроется в деталях. По мере углубления в особенности работы вы поймете, что это далеко не так просто, как может показаться. А пока, давайте поговорим о том, с чего нода начинает свою работу в момент запуска - конфигурационного файла.

Установка ноды

В данной книге мы не будем разбирать процесс установки ноды, так как это много раз описано уже в разных источниках (документация, видео на youtube, посты на форуме). Нам никакой книги не хватит, если мы захотим окунуться в эту тему, потому что существует много способов запуска ноды:

1. Запуск `jar` файла
2. Запуск Docker контейнера
3. Установка и запуск из `.deb` файла
4. Установка из `apt-` репозитория

Лично я предпочитаю запуск Docker контейнера, так как это упрощает и поддержку, и настройку, и конфигурирование. Другой причиной моей любви к Docker может быть то, что я делал Docker образ, размещенный в Docker Hub и отлично знаю как там что работает. Хотя вряд ли, это просто удобнее! 😀

Конфигурация ноды

Конфигурационный файл ноды Waves описан в формате HOCON (это как JSON, только с комментариями, возможностью композиции нескольких файлов и, что не менее важно, с меньшим количеством кавычек). Полный файл с конфигурацией выглядит громоздко, но я все-таки приведу его здесь, версия на момент написания этих строк (файл постоянно меняется, но актульную версию можно найти в репозитории Waves на Github).

Файл содержит большое количество комментариев, поясняющих каждый параметр, поэтому подробно разбирать все параметры мы не будем. Поговорим только об основных моментах. В конфигурации содержатся следующие разделы:

- waves
- kamon
- metrics
- akka

Последние 3 раздела являются полностью служебными, которые отвечают за параметры логгирования, отправку метрик и фреймворка akka. Нас интересует только первый раздел, который касается непосредственно протокола и содержит на первом уровне следующие подразделы:

- **db** - параметры для работы с LevelDB и настройки какие данные сохранять. Например, результаты некоторых транзакций можно сжимать (экономить до 10 ГБ), но это вредит удобству работу с API. Поэтому будьте осторожны с тем, что включать, а что выключать и экономить место на диске.
- **network** - параметры общения с другими нодами в сети. В этом разделе много важных параметров, которые мы разберем чуть ниже.
- **wallet** - параметры файла для сохранения ключей. Каждая нода, которая хочет генерировать блоки (чтобы получать за них вознаграждение), должна подписывать свои блоки. Для этого у ноды должен быть доступ к ключу аккаунта. В этой секции задается приватный ключ (а если быть точнее, то seed фраза в кодировке base58), пароль для того, чтобы эту фразу зашифровать, и путь, по которому хранить файл с этим зашифрованным ключом.
- **blockchain** - параметры блокчейна, в котором будет работать нода. В данном разделе есть настройка `type`, которая позволяет задать одну из заранее определенных типов блокчейнов (stagenet, testnet или mainnet). При указании значения `custom` можно менять все параметры

блокчейна, в том числе первоначальное количество токенов, их распределение, байт сети (уникальный идентификатор каждой сети), поддерживаемые фичи и т.д.

- **miner** - параметры генерации новых блоков. По умолчанию генерация включена (надо понимать, что это будет работать только при наличии генерирующего баланса больше 1000 Waves на аккаунте), но ее можно отключить с помощью параметра `enable`. Может пригодиться, если, например, нужна нода, которая будет только валидировать блоки. Другой полезный параметр - `quorum`, который определяет сколько соединений с другими нодами необходимо, чтобы нода пыталась генерировать блоки. Если задать данный параметр, равный 0, то можно запустить блокчейн, состоящий из 1 узла. Зачем вам такой блокчейн, если этот один узел может переписывать всю историю и делать все, что захочет? Для тестирования! Идеальная песочница для игр с блокчейном.
- **rest-api** - в ноде есть встроенный REST API, который по умолчанию отключен, но после включения (за это отвечает параметр `enable`) позволяет делать HTTP запросы для получения данных из блокчейна или записи в нее новых транзакций. В данном API есть как открытые методы, так и те, которые требуют API ключ, который задается в этой же секции настроек. Параметр `port` задает на каком порту будет слушать нода входящие HTTP запросы, на этом же порту будет доступен Swagger UI с описанием всех методов API. `api-key-hash` позволяет указать хэш от API-ключа, с которым будут доступны приватные методы. То есть в конфигурационном файле мы указываем не сам ключ, а хэш от него. А какой хэш надо взять? SHA-1? SHA-512? Или, прости господи, MD5?

- **synchronization** - параметры синхронизации в сети, в том числе максимальная длина форка и параметр `max-rollback` задает сколько блоков может быть откачено). Фактически можно сказать, что время финализации транзакции, после которого точно можно быть уверенным, что транзакция не пропадет из сети, составляет 100 минут (среднее время блока составляет 1 минуту).
- **utx** задает параметры пула неподтвержденных транзакций. Каждая нода настраивает эти параметры в зависимости от объема оперативной памяти, мощности и количества CPU. Параметр `max-size`, задающий максимальное количество транзакций в списке ожидания составляет 100 000 по умолчанию, а `max-bytes-size` имеет значение `52428800`. При достижении любого из этих лимитов, нода перестанет принимать транзакции в свой список ожидания. Про UTX мы поговорим отдельно в разделе транзакций.
- **features** - каждый новый функционал и изменения консенсуса (именно правил консенсуса, не изменения API или внутренностей ноды!) должны проходить через процедуру голосования. Принцип работы голосования мы разберем позже в этом разделе. Сейчас просто скажем, что каждая нода голосует используя массив `supported` в этой части конфигурации. Так же нода может автоматически выключиться, если вся сеть приняла какое-то обновление, которое не поддерживается этой версией, используя флаг `auto-shutdown-on-unsupported-feature`.
- **rewards** позволяет установить размер вознаграждения для майнера блока. Как и в случае с обновлениями проотокола проводится голосование, но голосование за размер вознаграждения работает по другому принципу.
- **extensions** описывает какие расширения вместе с этой нодой необходимо запускать.

В разделе `waves` на первом уровне так же лежат некоторые параметры:

- `directory` - путь к директории, в которой нода будет сохранять все файлы, которые относятся к ней, к том числе файлы LevelDB с данными

- `ntp-server` - сервер синхронизации времени
- `extensions-shutdown-timeout` - это время, которое дается расширенниям, подключенным к ноде, корректно завершиться при выключении самой ноды

В следующим разделах мы будем подробнее разбирать какие параметры влияют на поведение ноды и каким образом.

Особенности конфигурации

Вы уже увидели, что конфигурация ноды осуществляется с помощью HOCON файлов, которые поддерживают возможность композиции. Другими словами, в файле конфигурации можно использовать инструкции `include filename.conf`, который может загружать разные разделы конфигурации из другого файла. Некоторые разделы могут повторяться в разных файлах, поэтому там так же есть механизм разрешения конфликтов (чем ниже подключен файл, тем больший приоритет он имеет). Если у вас есть опыт работы с CSS, то принцип такой же. В некоторых местах документации Waves приводится нотация вроде `waves.network.port`, как нетрудно догадаться, это обозначает параметр в конфигурации вместе с путем.

Безопасность

При конфигурировании ноды имеет значение уделить особое внимание тем параметрам, которые напрямую влияют на безопасность - разделам `wallet` и `rest-api`. Хорошей практикой считается указать в конфигурационном файле base58 представление seed фразы и пароль, запустить ноду, а затем удалить из файла (не перезапуская ноду). Таким образом, запущенная нода будет знать приватный ключ и пароль (это есть в оперативной памяти), но в файле конфигурации ничего не осталось. Если вдруг кто-то получит доступ к вашей конфигурации или даже `wallet` файлу, он не сможет расшифровать ключ.

API ключ ноды не менее важен, потому что он позволяет отправить с ноды транзакции, подписанные ключами, хранящимися в ноде. В отличие от данных аккаунта, в конфигурации хранится только хэш, поэтому удалять оттуда после запуска смысла нет, но есть смысл использовать максимально сложный ключ и никак не тот, который стоит по умолчанию (в старых версиях был ключ по-умолчанию, в последних такого уже нет).

Оптимальные настройки блокчейна

Очень часто задаваемый вопрос - какие же настройки оптимальные? В первую очередь зависит от того, о какой сети мы говорим - `stagenet`, `testnet`, `mainnet`, `custom`? Например, для `custom` не существует оптимальных, существуют требования к сети. Но надо понимать, что нода Waves не всемогущая, эмпирически показано, что при времени блока меньше 12 секунд (`waves.blockchain.genesis.average-block-delay`), времени микроблока меньше 3 секунд (`waves.miner.micro-block-interval`) и относительно большом количестве узлов в сети (10+), сеть может быстро форкаться.

Важным параметром, который надо настраивать с учетом особенностей окружения, является максимальное количество транзакций в UTX. 100 000 транзаций является оптимальным для ноды, удовлетворяющей минимальным системным требованиям.

Я описал выше только самые основные параметры, многие другие мы будем рассматривать в следующих разделах, по мере погружения в протокол и его особенности. Сейчас же приведу полный файл конфигурации с комментариями:

```
# Waves node settings in HOCON
# HOCON specification:
https://github.com/lightbend/config/blob/master/HOCON.md
waves {
  # Node base directory
  directory = ""

  db {
    directory = ${waves.directory}"/data"
    store-transactions-by-address = true
    store-invoke-script-results = true
    # Limits the size of caches which are used during block validation.
Lower values slightly decrease memory consummption,
    # while higher values might increase node performance. Setting ghis
value to 0 disables caching alltogether.
    max-cache-size = 100000

    max-rollback-depth = 2000
    remember-blocks = 3h
  }

  # NTP server
  ntp-server = "pool.ntp.org"

  # P2P Network settings
  network {
    # Peers and blacklist storage file
    file = ${waves.directory}"/peers.dat"

    # String with IP address and port to send as external address during
handshake. Could be set automatically if UPnP
    # is enabled.
    #
    # If `declared-address` is set, which is the common scenario for nodes
running in the cloud, the node will just
    # listen to incoming connections on `bind-address:port` and broadcast
its `declared-address` to its peers. UPnP
    # is supposed to be disabled in this scenario.
    #
    # If declared address is not set and UPnP is not enabled, the node
will not listen to incoming connections at all.
    #
    # If declared address is not set and UPnP is enabled, the node will
attempt to connect to an IGD, retrieve its
    # external IP address and configure the gateway to allow traffic
through. If the node succeeds, the IGD's external
    # IP address becomes the node's declared address.
    #
```

```
    # In some cases, you may both set `decalred-address` and enable UPnP
(e.g. when IGD can't reliably determine its
    # external IP address). In such cases the node will attempt to
configure an IGD to pass traffic from external port
    # to `bind-address:port`. Please note, however, that this setup is not
recommended.
    # declared-address = "1.2.3.4:6863"

    # Network address
    bind-address = "0.0.0.0"

    # Port number
    port = 6863

    # Node name to send during handshake. Comment this string out to set
random node name.
    # node-name = "default-node-name"

    # Node nonce to send during handshake. Should be different if few
nodes runs on the same external IP address. Comment this out to set random
nonce.
    # nonce = 0

    # List of IP addresses of well known nodes.
    known-peers = ["52.30.47.67:6863", "52.28.66.217:6863",
"52.77.111.219:6863", "52.51.92.182:6863"]

    # How long the information about peer stays in database after the last
communication with it
    peers-data-residence-time = 1d

    # How long peer stays in blacklist after getting in it
    black-list-residence-time = 15m

    # Breaks a connection if there is no message from the peer during this
timeout
    break-idle-connections-timeout = 5m

    # How many network inbound network connections can be made
    max-inbound-connections = 30

    # Number of outbound network connections
    max-outbound-connections = 30

    # Number of connections from single host
    max-single-host-connections = 3

    # Timeout on network communication with other peers
    connection-timeout = 30s

    # Size of circular buffer to store unverified (not properly
handshaked) peers
    max-unverified-peers = 100
```

```
    # If yes the node requests peers and sends known peers
    enable-peers-exchange = yes

    # If yes the node can blacklist others
    enable-blacklisting = yes

    # How often connected peers list should be broadcasted
    peers-broadcast-interval = 2m

    # When accepting connection from remote peer, this node will wait for
handshake for no longer than this value. If
    # remote peer fails to send handshake within this interval, it gets
blacklisted. Likewise, when connecting to a
    # remote peer, this node will wait for handshake response for no
longer than this value. If remote peer does not
    # respond in a timely manner, it gets blacklisted.
    handshake-timeout = 30s

    suspension-residence-time = 1m

    # When a new treansaction comes from the network, we cache it and
doesn't push this transaction again when it comes
    # from another peer.
    # This setting setups a timeout to remove an expired transaction in
the elimination cache.
    received-txs-cache-timeout = 3m

    upnp {
      # Enable UPnP tunnel creation only if you router/gateway supports
it. Useful if your node is runnin in home
      # network. Completely useless if you node is in cloud.
      enable = no

      # UPnP timeouts
      gateway-timeout = 7s
      discover-timeout = 3s
    }

    # Logs incoming and outgoing messages
    traffic-logger {
      # Codes of transmitted messages to ignore. See
MessageSpec.messageCode
      ignore-tx-messages = [23, 25] # BlockMessageSpec,
TransactionMessageSpec

      # Codes of received messages to ignore. See MessageSpec.messageCode
      ignore-rx-messages = [25] # TransactionMessageSpec
    }
  }

  # Wallet settings
  wallet {
    # Path to wallet file
    file = ${waves.directory}"/wallet/wallet.dat"
```

```
    # Password to protect wallet file
    # password = "some string as password"

    # The base seed, not an account one!
    # By default, the node will attempt to generate a new seed. To use a
specific seed, uncomment the following line and
    # specify your base58-encoded seed.
    # seed = "BASE58SEED"
  }

  # Blockchain settings
  blockchain {
    # Blockchain type. Could be TESTNET | MAINNET | CUSTOM. Default value
is TESTNET.
    type = TESTNET

    # 'custom' section present only if CUSTOM blockchain type is set. It's
impossible to overwrite predefined 'testnet' and 'mainnet' configurations.
    #    custom {
    #      # Address feature character. Used to prevent mixing up
addresses from different networks.
    #      address-scheme-character = "C"
    #
    #      # Timestamps/heights of activation/deactivation of different
functions.
    #      functionality {
    #
    #        # Blocks period for feature checking and activation
    #        feature-check-blocks-period = 10000
    #
    #        # Blocks required to accept feature
    #        blocks-for-feature-activation = 9000
    #
    #        reset-effective-balances-at-height = 0
    #        generation-balance-depth-from-50-to-1000-after-height = 0
    #        block-version-3-after-height = 0
    #        max-transaction-time-back-offset = 120m
    #        max-transaction-time-forward-offset = 90m
    #        pre-activated-features {
    #          1 = 100
    #          2 = 200
    #        }
    #        lease-expiration = 1000000
    #      }
    #      # Block rewards settings
    #      rewards {
    #        term = 100000
    #        initial = 600000000
    #        min-increment = 50000000
    #        voting-interval = 10000
    #      }
    #      # List of genesis transactions
    #      genesis {
```

```
    #        # Timestamp of genesis block and transactions in it
    #        timestamp = 1460678400000
    #
    #        # Genesis block signature
    #        signature = "BASE58BLOCKSIGNATURE"
    #
    #        # Initial balance in smallest units
    #        initial-balance = 100000000000000
    #
    #        # Initial base target
    #        initial-base-target =153722867
    #
    #        # Average delay between blocks
    #        average-block-delay = 60s
    #
    #        # List of genesis transactions
    #        transactions = [
    #          {recipient = "BASE58ADDRESS1", amount = 50000000000000},
    #          {recipient = "BASE58ADDRESS2", amount = 50000000000000}
    #        ]
    #      }
    #    }
  }

  # New blocks generator settings
  miner {
    # Enable/disable block generation
    enable = yes

    # Required number of connections (both incoming and outgoing) to
attempt block generation. Setting this value to 0
    # enables "off-line generation".
    quorum = 1

    # Enable block generation only in the last block is not older the
given period of time
    interval-after-last-block-then-generation-is-allowed = 1d

    # Mining attempts delay, if no quorum available
    no-quorum-mining-delay = 5s

    # Interval between microblocks
    micro-block-interval = 5s

    # Max amount of transactions in key block
    max-transactions-in-key-block = 0

    # Max amount of transactions in micro block
    max-transactions-in-micro-block = 255

    # Miner references the best microblock which is at least this age
    min-micro-block-age = 4s

    # Minimal block generation offset
```

```
    minimal-block-generation-offset = 0

    # Max packUnconfirmed time
    max-pack-time = ${waves.miner.micro-block-interval}
  }

  # Node's REST API settings
  rest-api {
    # Enable/disable REST API
    enable = yes

    # Network address to bind to
    bind-address = "127.0.0.1"

    # Port to listen to REST API requests
    port = 6869

    # Hash of API key string
    api-key-hash = ""

    # Enable/disable CORS support
    cors = yes

    # Enable/disable X-API-Key from different host
    api-key-different-host = no

    # Max number of transactions
    # returned by /transactions/address/{address}/limit/{limit}
    transactions-by-address-limit = 1000
    distribution-address-limit = 1000
  }

  # Nodes synchronization settings
  synchronization {

    # How many blocks could be rolled back if fork is detected. If fork is
longer than this rollback is impossible.
    max-rollback = 100

    # Max length of requested extension signatures
    max-chain-length = 101

    # Timeout to receive all requested blocks
    synchronization-timeout = 60s

    # Time to live for broadcasted score
    score-ttl = 90s

    # Max baseTarget value. Stop node when baseTraget greater than this
param. No limit if it is not defined.
    # max-base-target = 200

    # Settings for invalid blocks cache
    invalid-blocks-storage {
```

```
      # Maximum elements in cache
      max-size = 30000

      # Time to store invalid blocks and blacklist their owners in advance
      timeout = 5m
    }

    # History replier caching settings
    history-replier {
      # Max microblocks to cache
      max-micro-block-cache-size = 50

      # Max blocks to cache
      max-block-cache-size = 20
    }

    # Utx synchronizer caching settings
    utx-synchronizer {
      # Max microblocks to cache
      network-tx-cache-size = 1000000

      # Max scheduler threads
      max-threads = 8

      # Max pending queue size
      max-queue-size = 5000

      # Send transaction to peers on broadcast request even if it's
already in utx-pool
      allow-tx-rebroadcasting = yes
    }

    # MicroBlock synchronizer settings
    micro-block-synchronizer {
      # How much time to wait before a new request of a microblock will be
done
      wait-response-timeout = 2s

      # How much time to remember processed microblock signatures
      processed-micro-blocks-cache-timeout = 3m

      # How much time to remember microblocks and their nodes to prevent
same processing
      inv-cache-timeout = 45s
    }
  }

  # Unconfirmed transactions pool settings
  utx {
    # Pool size
    max-size = 100000
    # Pool size in bytes
    max-bytes-size = 52428800 // 50 MB
    # Pool size for scripted transactions
```

```
    max-scripted-size = 5000
    # Blacklist transactions from these addresses (Base58 strings)
    blacklist-sender-addresses = []
    # Allow transfer transactions from the blacklisted addresses to these
recipients (Base58 strings)
    allow-blacklisted-transfer-to = []
    # Allow transactions from smart accounts
    allow-transactions-from-smart-accounts = true
    # Allow skipping checks with highest fee
    allow-skip-checks = true
  }

  features {
    auto-shutdown-on-unsupported-feature = yes
    supported = []
  }

  rewards {
    # desired = 0
  }

  extensions = [
    # com.wavesplatform.matcher.Matcher
    # com.wavesplatform.api.grpc.GRPCServerExtension
  ]

  # How much time to wait for extensions' shutdown
  extensions-shutdown-timeout = 5 minutes
}

# Performance metrics
kamon {
  # Set to "yes", if you want to report metrics
  enable = no

  # A node identification
  environment {
    service = "waves-node"

    # An unique id of your node to distinguish it from others
    # host = ""
  }

  metric {
    # An interval within metrics are aggregated. After it, them will be
sent to the server
    tick-interval = 10 seconds

    instrument-factory.default-settings.histogram {
      lowest-discernible-value = 100000 # 100 microseconds
      highest-trackable-value = 2000000000000 # 200 seconds
      significant-value-digits = 0
    }
  }
```

```
  # Reporter settings
  influxdb {
    hostname = "127.0.0.1"
    port = 8086
    database = "mydb"

    # authentication {
    #   user = ""
    #   password = ""
    # }
  }
}

# Non-aggregated data (information about blocks, transactions, ...)
metrics {
  enable = no
  node-id = -1 # ${kamon.environment.host}

  influx-db {
    uri = "http://"${kamon.influxdb.hostname}":"${kamon.influxdb.port}
    db = ${kamon.influxdb.database}

    # username = ${kamon.influxdb.authentication.user}
    # password = ${kamon.influxdb.authentication.password}

    batch-actions = 100
    batch-flash-duration = 5s
  }
}

# WARNING: No user-configurable settings below this line.

akka {
  loglevel = "INFO"
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"
  log-dead-letters-during-shutdown = false

  http.server {
    max-connections = 128
    request-timeout = 20s
    parsing {
      max-method-length = 64
      max-content-length = 1m
    }
  }

  io.tcp {
    direct-buffer-size = 1536 KiB
    trace-logging = off
  }
}
```

```
include "deprecated-settings.conf"
```

Процесс майнинга (Waves NG)

Процесс майнинга является ключевым для ноды, в конце концов ее основная задача в том, чтобы производить блоки с транзакциями. Чтобы это эффективно делать, нода также должна получать информацию о блоках от других нод и отправлять им свои блоки. Давайте рассмотрим упрощенную модель майнинга в Waves. Более подробная информация о процессе майнинга, включая формулы, есть в статье Fair Proof of Stake.

Proof of Stake

В основе майнинга лежит алгоритм Proof-of-Stake, который подразумевает, что вероятность сгенерировать блок каким-либо аккаунтом прямо пропорциональна балансу этого аккаунта. Давайте рассмотрим простейший случай: допустим, у нас есть аккаунт с балансом 10 млн Waves (из 100 млн выпущенных в момент создания). Вероятность смайнить блок будет 10%, иными словами мы будем генерировать 144 блока в сутки (1440 всего блоков в сети за сутки).

Теперь немного усложним. Хоть и выпущено всего 100 миллионов токенов, не все из них участвуют в майнинге (например, токены могут быть на бирже, а не на аккаунте ноды). Если в майнинге участвует 50 миллионов, то нода с балансом в 10 млн уже будет генерировать 288 блоков в сутки. Но на самом деле количество токенов, которые участвуют в майнинге, постоянно меняется, поэтому прямо предсказать, сколько будет смайнено блоков, не получится.

Вопрос, который возник у самых любопытных - *в каком порядке ноды будут генерировать блоки?*. Для ответа на этот вопрос потребуется углубиться в особенности реализации PoS в Waves, поэтому пристегнитесь и взбодритесь.

Можно сказать, что для ответа на вопрос "Кто будет следующим генератороом блока?" ноды используют информацию о балансах, времени между блоками и генератор псевдо-случайных чисел. Начнем с последнего, использовать `urandom` в данном случае не получится, так как он недетерминированный, и каждая нода получит свой результат. Поэтому ноды "договариваются" о рандоме. Каждый блок в цепочке содержит наряду с транзакциями, адресом ноды, сгенерировавшей блок, версией и времеменем, поле, называемое `generation-signature`. Взгляните, как выглядит блок номер 1908853 в мейннете в JSON представлении (без транзакций):

```
{
  "blocksize": 22520,
  "reward": 600000000,
  "signature":
"2kCWg8HMhLPXGDi94Y6dm9NRx4aXjXpVmYAE4y4KaPzgt1Z5EX9mevfWoiBLLr1cc1TZhTSqp
ozUJJZ3BpA5j3oc",
  "generator": "3PEFQiFMLm1gTVjPdfCErG8mTHRcH2ATaWa",
  "version": 4,
  "reference":
"3Jcr6m6SM3hZ1bu6xXBmAVhA2VEUHMvE6omhEiRFn3VhEuDkgb6sgeJUC1VNRB3vTSwPb5qh5
```

```
76a8DwGt3Ts72Tx",
  "features": [],
  "totalFee": 28800000,
  "nxt-consensus": {
    "base-target": 74,
    "generation-signature": "6cVJBZsjzuSqp7LPD3ZSw5V1BZ25hZQHioh9gHjWPKNq"
  },
  "desiredReward": 600000000,
  "transactionCount": 70,
  "timestamp": 1580458301503,
  "height": 1908853
}
```

Обратите внимание: для удобства структуры данных в этой книге представлены в формате JSON, но сами ноды работают с блоками, транзакциями, подписями и т.д. в бинарном формате. Для этого есть описания бинарных структур данных в документации, а с недавнего времени бинарный формат данных представляет из себя Protobuf.

Generation signature является SHA256 хэшом от `generation-signature` предыдущего блока и публичного ключа генератора блока. Первые 8 байт хэша `generting-signature` конвертируются в число и используется как некий рандом. Значение `base-target` отвечает за среднее время между блоками и пересчитывается во время генерации каждого блока. Если бы в сети постоянно были все ноды со всем стейком сейти, готорые сгенерировать блок, то `base-target` не был бы нужен, но коль это не так, нужен синтетический параметр, который меняется в зависимости от текущего времени между блоками и автоматически выравнивать среднее время между блоками в 60 секунд.

Итак, у нас есть параметры `hit`, который является псевдо-случайным числом, баланс каждого аккаунта и значение `base-target`, но что делать со всем этим ноде? Каждая нода, в момент получения нового блока по сети, запускает функцию проверки, когда будет ее очередь генерировать блок.

```
\delta t = f(hit, balance, baseTarget)
```

В результате выполнения этой функции, нода получает число секунд до момента, когда наступит ее время генерировать блок. Фактически, после этого нода устанавливает таймер, при наступлении которого начнет генерировать блок. Если она получит следующий блок до наступления таймера, то операция будет выполнена заново и таймер будет переставлен на новое значение \delta t.

Валидация блоков происходит таким же образом, за одним исключением, что в формулу подставляется баланс не этой ноды, а сгенерировавшей блок.

Waves NG

Если вы вообще что-то знаете про Waves, то могли слышать про Waves NG, который делает блокчейн Waves быстрым и отзывчивым. Waves-NG получил свое названия от статьи Bitcoin-NG: A Scalable Blockchain Protocol, которая была опубликована в 2016 году и предлагала способ масштабирования сети Bitcoin за счет изменнеия протокола генерации блоков. NG в названии расшифровывается как Next Generation, и действительно предложение помогло бы сети Bitcoin выйти на новый уровень по

пропускной способности, но эта инициатива так никогда и не была реализована в Bitcoin. Зато была воплощена в протоколе Waves в конце 2017 года. Waves NG влияет на то, как генерируются блоки и ноды общаются друг с другом.

В момент наступления своего времени майнинга, нода генерирует так называемый ключевой блок (key block), становясь лидером. Ключевой блок не содержит транзакций, он является только началом блока, который будет меняться. Дальее лидер получает право генерировать так называемые микроблоки, которые добавляют новые транзакции в конец блока и меняют его сигнатуру. Например, лидер генерирует ключевой блок со следующими параметрами:

```
{
  "blocksize": 39804,
  "reward": 600000000,
  "signature":
"4oBqMB7szmsbSYYguiaAXSE7ZLy13e4x97EKMmA4gs6puRqPKzCVJkuC6Py9eTpiovhcLAYuU
SsnEYAi4i73tvoA",
  "generator": "3P2HNUd5VUPLMQkJmctTPEeeHumiPN2GkTb",
  "version": 4,
  "reference":
"4KEFeMDQgPdntzqmSNZ92NBSMcNft1o4EyQexNLXEdN3976XbdYwDgqaucd9gu2PJWt9tpt1w
uvRcTMiiDtkZaX7",
  "features": [],
  "totalFee": 0,
  "nxt-consensus": {
    "base-target": 66,
    "generation-signature": "HpFc5qqVftyjKbqhADkQGWBg38CVR9Bz29c7uDZKKvYV"
  },
  "desiredReward": 600000000,
  "transactionCount": 0,
  "timestamp": 1580472824775,
  "height": 1909100
}
```

В блоке нет транзакций, что видно из значения `transactionCount`, но основные параметры вроде подписи и ссылки на предыдущий блок (поле `reference`) уже есть. Создатель этого блока сможет через несколько секунд сгенерировать микроблок со всеми транзакциями, которые появились в сети за эти секунды, и разослать остальным нодам. При этом в блоке поменяются некоторые поля:

```
{
  // неизмененные параметры опущены
  "blocksize": 51385,
  "signature":
"4xMaGjQxMX2Zd4jMUUUs5cmemkVwT8Jc5sqx6wzMUokVqWg5jvWSDF6SBF1P7x4UNQjYsgsCs
4csa2qtRmG8j3g4",
  "totalFee": 65400000,
  "transactionCount": 167,
  "tranasctions": [{...}, {...}, ..., {...}]
}
```

В блок добавились 167 транзакций, которые увеличили размер блока, так же поменялась подпись блока и комиссия, которую заработает лидер.

Несколько важных моментов, которые важно понимать:

- Микроблок содержит только транзакции и подпись лидера, параметры консенсуса не дублируются
- Время генерации микроблоков зависит от настроек майнера (поле `waves.miner.micro-block-interval` в конфигурации задает значение для каждой ноды). По умолчанию лидер будет генерировать микроблоки каждые 5 секунд.
- При каждом новом микроблоке меняются данные последнего блока, поэтому последний блок называют "жидким" (liquid) блоком
- Ключевой блок и все микроблоки, которые к нему относятся, объединяются в один блок так, что в блокчейне не остается никаких данных о микроблоках. Можно сказать, что они используются только для передачи информации о транзакциях между нодами.

Лидер блока будет генерировать микроблоки и менять жидкий блок до тех пор, пока не будет сгенерирован другой ключевой блок в сети (то есть у какой-то другой ноды сработает таймер начала майнинга) или достигнуты лимиты блока на размер (1 МБ).

Что дает Waves NG?

Благодаря Waves NG сокращается время попадания транзакции в блок. То есть можно в своем приложении обеспечивать гораздо лучший пользовательский опыт. Пользователь может получать обратную связь по своей транзакции за ~5 секунд, если нет большой очереди за попадание в блок. Только надо понимать, что попадание в блок не является гарантией финализации и блок может быть отменен (до 100 блоков в глубину, но на практике 2-3 блока в крайне редких случаях).

Waves NG делает нагрузку на сеть более равномерной. В случае отсутствия Waves NG блоки генерировались бы раз в минуту (сразу 1 МБ данных) и отправлялись бы по сети целиком. То есть можно представить ситуации, когда 50 секунд ноды (кроме майнера) ничего не делают и ждут, а потом принимают блок и валидируют его на протяжении 10 секунд. С Waves NG эта нагрузка более размазана по времени, ноды получают каждые 5 секунд новую порцию данных и валидируют их. Это в целом повышает пропускную способность.

Waves NG однако может себя иногда вести не очень удобно. Как вы помните, каждый блок содержит в себе поле `reference`, которое является ссылкой на поле `signature` предыдущего блока. `reference` фиксируется в момент генерации ключевого блока, и может случиться такое, что новый майнер поставит в своем ключевом блоке ссылку не на последнее состояние жидкого блока. Иными словами, если новый майнер блока `N` не успел получить и применить последний микроблок блока `N - 1` от предыдущего майнера, то он сошлется на "старую" версию блока `N - 1`, транзакции из последнего микроблока будут удалены из блока `N - 1` для всей сети.

Но не пугайтесь, это приведет только к тому, **что исключенные транзакции попадут в блок** `N`, вместо блока `N - 1`, в котором мы уже могли успеть увидеть эти транзакции в своем клиентском коде.

Waves NG так же влиет на распределение комиссий в блоке. Майнер получает 60% от комиссий из предыдущего блока и 40% из своего блока. Сделано это для того, чтобы исключить возможную "грязную игру" узлов, когда они будут специально ссылаться на самую первую версию предыдущего

блока, чтобы забрать все транзакции оттуда и положить в свой блок, а соответственно получить и комиссии.

Получаемая комиссия может быть потрачена майнером в этом же блоке. Он может добавить в блок транзакцию, за которую получит комиссию в 0.1 Waves и следующей же транзакцией положить в блок, переводящую эти 0.1 Waves с его аккаунта.

Обновления протокола и другие голосования

Как вы могли понять из предыдущего раздела, протокол работы блокчейна, в особенности Waves NG, совсем нетривиальная штука. Но как и любой протокол, он может и должен меняться со временем, чтобы становиться лучше. Но тут не все так просто. Команда разработки Waves не может просто так выпустить обновления и приказать всем обновиться или сказать, что те, кто не обновится, перестанут работать - это противоречит принципам децентрализации. Многие блокчейны идут по пути жестких "форков", просто выпускается новый функционал с новой версией ноды, дальше кому надо - устанавливает и начинает майнить с поддержкой новой фичи. Кто согласен - переключается на новую цепочку, кто нет - продолжает майнить старую. Этот путь далеко не лучший и может вести к бесчисленному количеству форков, поэтому добавление нового функционала или изменение правил консенуса в Waves возможно только через процедуру предложения нового функционала и голосование.

Процедуру изменения параметров консенсуса с помощью голосований часто называют гаверненсом (governance). Мы не будем использовать это слово, потому что governance в Waves сейчас ограничен двумя типами голосований, в то время как во многих других блокчейнах возможных изменений гораздо больше (баны аккаунтов, жесткие изменения балансов аккаунтов и другие зверства, плохо уживающиеся с принципами децентрализации).

Процедура предложения нового функционала

В экосистеме Waves есть неписанное правило (писанного быть не может, децентрализация ведь), что все новые функции и изменения консенсуса должны проходить через процедуру обсуждения. Все предложения по изменения являются так называемыми Waves Enhancement Proposal или WEP. У каждого WEP есть порядковый номер, четка определенная структура и вопросы, на которые это предложение должно отвечать. Форма WEP была предложена на форуме Waves в специальном разделе, но в данный момент основные обсуждения и предложения на github.

Итак, каждое предложение формулируется в виде WEP, далее это предложение обсуждается всеми заинтересованными сторонами, вносятся корректировки, уточняются формулировки и т.д. В конечно итоге, любой человек может реализовать предложенный WEP в коде ноды (репозиторий ведь открытый) и отправить Pull Request на добавление изменений в основную ветку разработки. Команда Waves отвечает за качество кода в репозитории, поэтому при отсутствии проблем с качеством, код будет добавлен в основную ветку и попадет в сборку следующего релиза, которые так же публикуются на github. Но это не говорит о том, что новый код начнет работать, потому что каждый новый функционал (далее фича, от слова "feature") должен быть одобрен не только разработчиками, но и сообществом, для этого запускается процедура голосования.

Голосования за правила консенсуса

Как только владелец ноды устанавливает новую версию, у него появлется возможность голосовать за то, чтобы новая фича была активирована. У каждой фичи есть порядкой номер, по которому идет голосование и идентификация, обычно номера новых фич перечислены в описаниях к релизу на Github, но так же можно посмотреть в API ноды. Владелец ноды может добавить номер поддерживамой фичи в свою конфигурацию в массив `waves.features.supported`. После этого (а точнее после перезапуска ноды) в каждый генерируемый блок от этой ноды начинает добавляться номер поддерживаемой фичи. То есть, в бинарном представлении блока (в котором и идет работа), появляется новое значение с номером фичи.

Для того, чтобы фича была активирована, необходимо, чтобы ее поддержка была не менее 80% - не менее 80% блоков за период голосования должны в себе содержать информацию о поддержке фичи. Периоды голосования начинаются на каждом кратном десяти-тысячном блое (блок номер 100 000, 110 000, 120 000 и т.д.). Проще говоря, из блоков с номерами 100000-109999 не менее 80% содержать информацию о поддержке новой фичи.

Фактически гаверненс в Waves устроен очень близко к тому, как работает система выборов президента США. Жители страны напрямую не голосуют за президента, они выбирают представителей от каждого штата (количество представителей разнится от штата к штату. Представители уже непосредственно голосуют за президента США.

В Waves владельцы токенов могут напрямую голосовать за активацию фичи, однако, в большинстве случаев они не имеют своих нод и сдают свои токены в лизинг, передавая, таким образом, свое право голоса, владельцу лизингового пула. Важно понимать, что они в любой момент могут отменить лизинг, если владелец пула голосует не так, как они хотели бы, что снизит количество блоков, которые сгенерирует данный пул. Самые большие пулы в Waves часто делают голосования среди лизеров с помощью децентрализованных приложений (мы рассмотрим пример такого голосования в разделе 7), так что и представительская *демократия*/децентрализация могут быть прозрачными.

Если фича была поддержана более чем 80% стейка, то она будет активирована через 10000 блоков с момента завершения периода голосования. Например, если голосование была во время блоков 10000-19999, то фича станет активной на высоте 30000. Данная задержка позволяет еще раз проверить, что все нормально и новая фича не вызовет форки.

Если посмотреть в код ноды или в API (`/activation/status`), то можно заметить, что у каждой фичи есть следующие возможные статусы:

- **VOTING** - идет голосование по фиче
- **APPROVED** - фича одобрена, но пока не активирована (из-за задержки в 10 000 блоков)
- **ACTIVATED** - фича активирована, все ноды на этой цепочке должны поддерживать данную фичу

Как видите, у фичи нет статуса `REJECTED`, то есть голосование за фичу может идти бесконечно.

Голосование за вознаграждение за блок

В Waves есть еще один вид голосования нодами, который не встречается практически нигде - голосование за вознаграждение за блок. В 2016 году блокчейн Waves запускался с ограниченной эмиссией токенов - 100 миллионов, коорые уже были созданы на момент запуска сети. Но осенью 2019 года комьюнити осознало, что все-таки модель с эмиссией является более живой, поэтому было предложение обновление - WEP-7, которое прошло через процедуру голосования и на блоке N фича

была активирована. Теперь за каждый сгенерированный блок, майнер получает не только комиссии (и то 40% от своего блока и 60% от предыдущего, вы ведь помните про Waves NG?), но и получает Waves, "генерируемые из воздуха". Примерно каждую минуту общее количество Waves в природе увеличивается на определенное значение. На какое именно значение - предмет голосования нод.

Каждые 100 тысяч блоков (примерно 2 с половиной месяца) начинается голосование за величину вознаграждения. В момент активации фичи N было установлено значение в 6 Waves. Каждые 100 000 блоков это значение может меняться не больше, чем на 0.5 Waves, и то при условии поддержки более чем 50% майнеров.

Голосование за вознаграждение за блок осуществляется немного по другому принципу, не как в случае с фичами. Владельцы нод могут в своем конфигурационном файле в параметре `waves.reward.desired` установить значение вознаграждения, которое хотели бы увидеть в долгосрочной перспективе. По истечении периода голосования, подсчитывается сколько блоков содержат желаемое вознаграждение больше, чем текущее, и если больше 50%, то вознаграждение увеличивается на 0.5 Waves на следующие 110 000 блоков (наступление нового периода голосования + сам период голосования).

Некоторые участники сообщество спрашивали, почему в `waves.reward.desired` просто не указывать `+` или `-`, раз все равно вознаграждение не будет изменено больше, чем на 0.5 Waves. Указание желаемого вознаграждения в долгосрочной перспективе избавляет от необходимости частого изменения конфигурации. Вы можете поставить значение `10` и не лезть каждый период голосования в конфигурацию, так как вы будете голосовать за увеличение до тех пор, пока вознаграждение не достигнет 10 Waves за блок. А как только достигнет (если достигнет), нода перестанет голосовать за увеличение вознаграждения. Так просто.

Ключи в блокчейне Waves

Первое, с чем сталкивается человек, когда начинает пользоваться блокчейном - работа с ключами. В отличие от классических веб приложений, где у нас есть логин и пароль, в блокчейнах есть только ключи, которые позволяют идентифицировать пользователя и валидность его действий.

У каждого аккаунта есть публичный ключ и соответствующий ему приватный. Публичный ключ является фактически идентификатором аккаунта, в то время как приватный позволяет сформировать подпись. В Waves используется подписи с использованием кривой Curve25519-Ed25519 с ключами X25519 (что иногда является проблемой, потому что поддержка ключей X25519 есть далеко не во всех библиотеках).

Публичный и приватный ключи представляют из себя 32 байтовые значения, которые соответствуют другу по определенным правилам (подробнее можете найти в описании EdDSA). Важно понимать несколько вещей, которые отличаются по сравнению с другими блокченами:

- не любые 32 байта могут быть приватным ключом
- приватный ключ не содержит в себе публичный ключ (например, в Ethereum приватный ключ содержит приватный, поэтому имеет размер в 64 байта, в Waves публичный ключ вычисляется каждый раз для приватного ключа)
- подпись с помощью EdDSA является недетерменированной, то есть одни и те же данные можно подписать одним и тем же ключом и получать разные подписи, так как используются и случайные

значения

Путешествия ключа

Большинство пользователей все-таки сталкивается с ключами не в виде массива байт, а в виде сид-фразы, часто так же называемой мнемонической. Любая комбинация байт может быть сидом, но в клиентах Waves обычно используется 15 английских слов. На основе сид фразы вычисляется приватный ключ следующим образом:

1. строка переводится в массив байт
2. вычисляется хэш `blake2b256` для данного массива байт
3. вычисляется хэш `keccak256` для результата предыдущего шага
4. вычисляется приватный ключ на основе предыдущего шага, пример функции для этого шага представлен ниже

```
func GenerateSecretKey(hash []byte) SecretKey {
    var sk SecretKey
    copy(sk[:], hash[:SecretKeySize])
    sk[0] &= 248
    sk[31] &= 127
    sk[31] |= 64
    return sk
}
```

Иными словами, а точнее кодом: `privateKey = GenerateSecretKey(keccak256(blake2b256(accountSeedBytes)))`

Публичный и приватный ключи обычно представляют в виде `base58` строк вроде `3kMEhU5z3v8bmer1ERFUUhW58Dtuhyo9hE5vrhjqAWYT`.

При отправке транзакций (например, отправке токенов) пользователь имеет дело с адресом, а не публичным ключом получателя. Но адрес генерируется из публичного ключа получателя с некоторыми дополнительными параметрами: версия спецификации адреса, байт сети и чек-сумма. В данный момент в сети Waves есть только одна версия адресов, поэтому первым байтом в этой последоствальности является `1`, второй байт - уникальный идентификатор сети, который позволяет отличать адреса в разных сетях (mainnet, testnet, stagenet). Байты сети для перечисленных выше сетей `W`, `T`, `S` соответственно. Благодаря байту сети невозможно ошибиться и отправить токены на адрес, которого не может существовать в сети, в которой отправляется транзакция. После первых двух служебных байт идут 20 байт, полученных в результате функций хэширования blake2b256 и keccak256 над публичным ключом. Эта операция `keccak256(blake2b256(publicKey))` возвращает 32 байта, но последние 12 байт отбрасываются. Последние 4 байта в адресе являются чек-суммой, которая считается как `keccak256(blake2b256(data))`, где `data` это первые 3 параметра (версия, байт сети и 20 байт хэша публичного ключа). Полученная последовательность байт переводится в base58 представление, чтобы получилось похожее на это: `3PPbMwqLtwBGcJrTA5whqJfY95GqnNnFMDX`.

Опытные разработчики на Waves пользуются особенностями формирования адресов, чтобы по одному виду определять к какой сети относится адрес. Благодаря тому, что первые 2 байта в адресе похожи

для всех адресов в одной сети, можно примерно понимать к какой сети относится адрес. Если адрес выглядит как `3P...`, то адрес с большой долей вероятности относится к mainnet, а если адрес начинается с `3M...` или `3N`, то перед вами скорее всего адрес из testnet или stagenet.

Работа с ключами

Если по какой-то причине, приложение необходимо генерировать ключи для пользователя, то можно воспользоваться библиотеками для разных языков программирования. Например, в библиотеке `waves-transactions` для JavaScript/TypeScript сгенерировать seed фразу можно с помощью следующего кода:

```
import {seedUtils} from '@waves/waves-transactions'

const seedPhrase = seedUtils.generateNewSeed(24);

console.log(seedPhrase);

// infant history cram push sight outer off light desert slow tape correct
chuckle chat mechanic jacket camp guide need scale twelve else hard cement
```

В консоль выведется строка из 24 слов, которые являются seed фразой нового аккаунта. Эти слова являются случайным подмножеством из словаря, который есть в коде библиотеки @waves/ts-lib-crypto и в котором содержится 2048 слов.

В данном примере я сгенерировал 24 слова, но по умолчанию во многих приложениях Waves генерируется набор из 15 слов. Почему именно 15 и увеличивается ли безопасность, если сгенерировать больше слов?

15 слов из 2048 в любом порядке достаточно, для того, чтобы вероятность генерации двух одинаковых seed фраз была пренебрежительно мала. В то же время, 24 слова еще уменьшают такую вероятность, почему бы не использовать большие значения? Ответ прост - чем больше слов мы используем, тем больше надо записывать и/или запоминать пользователю и тем сложнее ему будет. Смысл использования seed фразы (а не приватного ключа) именно в упрощении опыта пользователя, а с 24 словами мы заметно ухудшаем user experience.

Имея seed фразу можно получить приватный ключ, публичный и адрес. Я снова покажу как это сделать на JS, но вы же помните, что есть библиотеки и для других языков?

```
import {seedUtils} from '@waves/waves-transactions';
import {
  address,
  privateKey,
  publicKey
} from '@waves/ts-lib-crypto'


const seedPhrase = seedUtils.generateNewSeed(24);
```

```
console.log(privateKey(seedPhrase));   //
3kMEhU5z3v8bmer1ERFUUhW58Dtuhyo9hE5vrhjqAWYT
console.log(publicKey(seedPhrase));    //
HBqhfdFASRQ5eBBpu2y6c6KKi1az6bMx8v1JxX4iW1Q8
console.log(address(seedPhrase, 'W')); //
3PPbMwqLtwBGcJrTA5whqJfY95GqnNnFMDX
```

Обратите внимание, что в функции `privateKey` и `publicKey` мы передаем только сид фразу, в то время как в `address` передаем еще один параметр `chainId` (он же байт сети). Как вы помните из объяснения выше, адрес в себе содержит такой дополнительный параметр.

Как аккаунт появляется в блокчейне

Мы разобрали как работают ключи, как связаны сид фраза, приватный и публичный ключ, а также как к ним относится адрес, но я не упомянул один очень важный момент, о котором забывают некоторые начинающие разработчики. До момента совершения какого-либо действия с аккаунтом (отправка с него или на него транзакции), блокчейн ничего не знает об этом аккаунте. Если вы сгенерировали аккаунт (у себя локально или в любом клиенте), но в блокчейне не блоы транзакций, связанных с этим аккаунтом (входящих или исходящих), вы не сможете найти никакую информацию о вашем аккаунте в эксплорере или с помощью API. Это отличается от поведения в централизованных системах и API, поэтому может быть не так интуитивно понятным и простым, но об этом важно помнить.

Обычные аккаунты vs. смарт аккаунты

Как вы уже наверняка поняли из предыдущего раздела, в Waves используется модель аккаунтов, а не входов и выходов (input b output) как в Bitcoin. Но отличия модели Waves есть не только от Bitcoin, но, например, и от Ethereum, где тоже есть аккаунты. Основное отличие в том, что в Waves есть несколько типов аккаунтов. Давайте разберем по порядку.

Обычные аккаунты

Работа обычных аккаунтов максимальна проста и интуитивно понятна. Каждый аккаунт (на самом деле публичный ключ) "обладает" некими токенами и сохраняет данные в свое хранилще. Чтобы сделать действие с аккаунта, необходимо сформировать транзакцию, подписать и отправить в сеть. **Валидная подпись транзакции позволяет делать транзакции с этого аккаунта.** Ноды при валидации транзакции фактически прооверяют соответствие подписи и тела транзакции. Так же они должны проверить некоторые параметры из блокчейна, например, есть аккаунт хочет отправить токены, есть ли они у него на балансе. Проще всего понять принцип работы с помощью аналогии - аккаунт является амбаром, с которым можно сделать что угодно (унести оттуда ~сыр~ токены, например), если есть ключ от амбарного замка.

Но обычный аккаунт можно превратить в смарт-аккаунт, который ведет себя по-другому.

Смарт аккаунт

Если на обычный аккаунт поставить скрипт, который задает другие правила валидации **исходящих** транзакций, то он станет **смарт-аккаунтом.** Смарт-аккаунт так же будет "владеть" токенами, но чтобы что-то с ними сделать (перевести, сжечь, обменять и т.д.) надо не предоставить ключ, а удовлетворять условиям, описанным в теле скрипта. Модель во многом похожа на то, что есть в Bitcoin скрипт, за исключением, что в Waves используется не примитивный Bitcoin script, а гораздо более мощный язык Ride. В этой книге будет отдельный раздел, посвященный Ride, поэтому сейчас больше поговорим про концепцию смарт-аккаунтов. Код на Ride отправляется в сеть с помощью транзакции установки скрипта (`SetScript`) и превращает обычный аккаунт в смарт.

Смарт-аккаунты являются разновидностью смарт-контрактов. В целом можно выделить 3 вида смарт-контрактов, которые встречаются в природе:

- Простые смарт-контракты для упраления аккаунтами (мультиподпись, эскроу и т.д.)
- Сложные контракты с нетривиальной логикой (Crypto-Kitties, Bancor и т.д.)
- Контракты токенов (ERC-20, ERC-721)

Смарт-аккаунты являются представителями первой категории, будучи предназначенными для базовых операций с аккаунтом. В Waves есть инструменты для создания сложных контрактов, которые будут рассмотрены в разделе 6, а для создания токенов в Waves вообще не требуются контракты, что так же рассмотрим в главе 4.

Смарт-аккаунты **позволяют валидировать только исходящие транзакции** (не входящие). Скрипт смарт-контракта является предикатом, который выполняется при попытке отправить транзакцию с аккаунта, и транзакция считается валидной только в том случае, если тело скрипта возвращает `true`. Тело скрипта может содержать различную логику, опирающуюся на:

- Параметры транзакции (например, размер комиссии, получатель транзакции перевода, тип транзакции и т.д.)
- Данные из блокчейна (номер последнего блока в блокчейне, данные из хранилища **любого** аккаунта)
- Подписи транзакции

Продолжая аналогию с амбаром, можно сказать, что смарт-аккаунт является амбаром с другим типа замка, который открывается не по ключу (или не только по ключу), но и может описаться на время, содержимое амбара (или любых других амбаров), личность открывающего и т.д. Думаю, многие амбары во многих колхозах были бы в большей безопасности, если там были замки, открыть которые можно только 5 ключами от разных людей.

Когда использовать смарт-аккаунты

Смарт-аккаунты очень легковесны, не требуют много вычислительных мощностей от нод, в то же время покрывает большое количество кейсов. Самыми частыми случаями использования смарт-аккаунтов являются:

- **Мультиподпись.** Например, есть аккаунт с токенами от 3 сторон, и тратить их можно только если согласны 2 стороны из 3.
- **Эскроу.** Часто при совершении операций в реальной жизни между 2 сторонами необходимо наличие некоего арбитра, который фиксирует факт совершения действия с одной стороны и

необходимость передачи ей денег. Продажа квартиры - отличный приме, при передаче ключей необходимо передавать деньги. Такое можно реализовать с помощью эскроу контракта.
- **Атомарный обмен** Обмен токенов в двух сетях, когда токены блокируются сторонами в 2 сетях. Пример атомарного обмена мы рассмотрим в главе 6.

Токены в Waves

После того, как мы поговорили об аккаунтах вполне логично поговорить про другую важную сущность в блокчейне Waves - про токены. Мне лично кажется необходимым начать с истории вопроса, потому что многие все еще знают Waves как платформу для выпуска токенов. В эру бума ICO (2017 год) Waves был второй по популярности платформой для выпуска токенов, потому что позволяла сделать это очень просто. На первом месте был Ethereum, в котором для выпуска токенов необходимо писать смарт-контракт (очень простой и чаще всего в соответствии с ERC-20, но все же). В Waves же выпуск токена делается крайне просто - отправкой одной транзакции специального типа `Issue`.

В некоторых клиентах для Waves (например, в Waves.Exchange) достаточно заполнить одну небольшую форму для выпуска простого токена, который автоматически будет доступен для переводов между аккаунтами, работы с децентрализованными приложениями или торговли на DEX. В данный момент в блокчейне Waves выпущено более 20 000 различных видов токенов.

Принципы работы токенов

В Waves все токены являются "граждами первого сорта" (first-class citizens), они есть прямо в ядре блокчейна, как есть, например, аккаунты. Некоторые (особенно с опытом работы с Ethereum) удивляются этому, но такой подход имеет ряд преимуществ:

- **Простота выпуска.** В 2017 году Waves занимал второе место именно благодаря простоте выпуска, что не надо было разбираться как работает Solidity или EVM, чтобы сделать токен, привязать к каким-то активам в реальной жизни, и начать использовать.
- **Быстрота работы.** Простые токены в Waves не исполняют никакого кода смарт-контрактов для своей работы, поэтому будут работать быстрее, чем в случае с Solidity. Фактически простой токен в Waves - запись в базе данных LevelDB.
- **Возможность торговли из коробки.** Выпускаемые токены на Waves автоматически поддерживаются для торговли на децентрализованных биржах на базе матчера Waves. Про работу матчера мы поговорим отдельно.

У вас уже должен был возникнуть вопрос, а что если я хочу не просто выпустить токен, но сделать для него свою логику?

Такое тоже возможно с помощью создания смарт-ассетов, которые мы рассмотрим позже в этой главе.

Много новых разработчиков спрашивают, чем отличается **ассет** от **токена**. В коде ноды Waves вы гораздо чаще будете встречать слооово `asset`, нежели `token`, но для удобства в рамках этой книги предлагаю считать эти 2 понятия взаимозаменяемыми. Да, в реальной жизни `asset` это скорее актив, а `токен` скорее что-то близкое к жетону, но в мире блокчейна граница между понятиями размылась.

В Waves есть только один токен, который не является ассетом - это сам токен Waves, который платится как комиссия майнерам. Можно сказать, что все токены в Waves равны по возможностям, но токен Waves чуть "ровнее" и его поведение отличается от других ассетов.

Выпуск токена

Как я уже писал выше, для выпуска токена достаточно отправить транзакцию типа `Issue`, что можно легко сделать с помощью библиотеки на JS:

```
const { issue } = require('@waves/waves-transactions')

const seed = 'seed phrase of fifteen words'

const params = {
  name: 'Euro',
  description: 'It is an example of token',
  quantity: 1000000,
  //senderPublicKey: 'by default derived from seed',
  reissuable: false,
  decimals: 2,
  script: null,
  timestamp: Date.now(),
  fee: 100000000,
}

const signedIssueTx = issue(params, seed)
console.log(signedIssueTx)
```

В результате выполнения этого кода в консоль выведется следующий JSON:

```
{
  "id": "CZw4KCpPUv5t1Uym3rLc9yEaQyDsP3VVPspdpmWKvVPE",
  "type": 3,
  "version": 2,
  "senderPublicKey": "HRQUmzJKgHDGbsfS23kSA1VRuudy5MY3wGCroUmNhKuJ",
  "name": "Euro",
  "description": "It is an example of token",
  "quantity": 1000000,
  "decimals": 2,
  "reissuable": false,
  "script": null,
  "fee": 100000000,
  "timestamp": 1575034734086,
  "chainId": 87,
  "proofs": [

"2ELbuezHiaHUDCuWpfhULwqSA8SUm4vGzWQe5QUmLEPZTA5WMXctiaXaoF9aUbr8TBSBreQxa
8WYMsp6Sy2qSSGU"
```

```
    ]
}
```

Давайте разберем основные параметры:

- **name** - название токена (4-16 байт)
- **description** - описание для токена (0-1000 байт)
- **quantity** - количество выпускаемых токенов
- **decimals** - количество знаков после запятой. Обратите внимание, что если поставить значение равное `0`, то токен будет неделимый. В примере выше минимальной единицой будет не 1 токен, а одна стоая от токена, что логично, так как мы выпускаем аналог евро как пример. Если быть точнее, то выпускаем 1 миллион токенов под название `Euro`, минимальной единицей которой будет евроцент.
- **reissuable** - флаг перевыпускаемости токена. Если значение равно `true`, то владелец токена в любой момент может довыпустить сколько угодно новых токенов такого вида. В момент перевыпуска, владелец может поменять значение этого флага, таким образом зафиксировав количество токенов в блокчейне.
- **script** - скомпилированный Ride скрипт, описывающий логику работы токена. В нашем примере значаение равно `null`, так как мы не хотим задавать никаких правил обращения токена.
- **fee** - комиссия за выпуск токена. В Waves минимальный транзакция для выпуска обычного токена (не NFT) составляет 1 Waves. Почему же тогда в транзакции указано 100000000, где аж 8 нолей? Все просто, у токена Waves количество decimals равно 8, а комиссия указывается в самых маленьких единицах, в случае с Waves минимальные единицы называют иногда ***waveslet***.

Отправив такую подписанную транзакцию, можно создать новый токен с названием `Euro`. Конечно, никакой ценности в таком токене нет, но ценность это уже следующий вопрос. Новосозданный токен получит уникальный идентификатор `assetId` равный ID транзакциии, которая его породила, в нашем случае `CZw4KCpPUv5t1Uym3rLc9yEaQyDsP3VVPspdpmWKvVPE`.

Данное правило может быть крайне полезным, поэтому предлагаю запомнить - `assetId` токена равен `ID` issue транзакции, которая его создала. В дальнейшем при работе с этим токеном придется использовать именно его assetId в подавляющем боольшинстве случаев.

Другой важный параметр, который надо запомнить, у токена Waves (нативного/системного для оплаты транзакций) нет `assetId`, поэтому в местах, где для других токенов вставляется длинная строка, для Waves необходимоо ставить `null`.

Выпуск NFT токена

Non-fungible токены очень часто используются для различных механик, чаще всего игровых. NFT отличаются тем, что каждый токен уникален и имеет свой уникальный идентификатор. В Waves выпуск Non-fungible токена осуществляется так же, как и выпуск fungile токенов, за несколькими ограничениями:

- `quantity` обязательно должно быть равно единице
- `decimals` всегда должно быть 0
- `reissuable` должно быть задано `false`

При соблюдении условий выше уже можно выпускать токен с комиссией не в 1 Waves, а в тысячу раз меньше - 0.001 Waves. Для удобной работы с NFT токенами существует JavaScript библиотека `@waves/waves-games`, которая упрощает создание и сохранение мета-информации о токене. Пример выпуска NFT с помощью этой библиотеки найдете ниже:

```
import { wavesItemsApi } from '@waves/waves-games'
const seed = 'my secret backend seed'

const items = wavesItemsApi('T') //testnet, use 'W' for mainnet
const item = await items
  .createItem({
    version: 1,
    quantity: 100,
    name: 'The sword of pain',
    imageUrl:
'https://i.pinimg.com/originals/02/c0/46/02c046b9ec76ebb3061515df8cb9f118.
jpg',
    misc: {
      damage: 22,
      power: 13,
    },
  }).broadcast(seed)
console.log(item)
```

Обратите внимание, что в примере выше выпускается 100 токенов, не токен с количеством 100, а 100 разных, у каждого из которых будет уникальный ID. Другими словами, библиотека отправит 100 issue транзакций. Минимальная комиссия за каждый токен составит 0.001 Waves, а для всех 100 - 0.1 Waves. Больше примеров по работе с библиотекой для NFT вы найдете в их туториалах.

Перевыпуск токенов

Если у токена в момент создания стояло значение `true` для поля `reissuable`, то создатель может отправлять транзакции типа `Reissue`, который позволит довыпустить еще токенов. Пример генерации reissue транзакции во многом похож на пример с issue:

```
const { reissue } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  quantity: 1000000,
  assetId: 'CZw4KCpPUv5t1Uym3rLc9yEaQyDsP3VVPspdpmWKvVPE',
  reissuable: false
}
```

```
const signedReissueTx = reissue(params, seed)
```

Главное отличие в том, что название или описание мы поменять не можем. В нашем примере мы довыпускаем к уже выпущенному миллиону токенов с названием Euro, еще один миллион.

Вы так же можете заметить, что в этой транзакции тоже есть флаг `reissuable`. Если отправить reissue транзакцию с полем `reissuable` равным `true`, то в дальнейшем отправлять такие транзакции для этого токена уже будет нельзя.

В примере выше поле комиссия опущена, но библиотека `@waves/waves-transactions` автоматически подставит минимальное значение в 1 Waves. Я часто пишу "минимальное значание" комиссии, чтобы показать, что это значение можно сделать больше, но сейчас сеть Waves не испытывает проблем с пропускной способностью, поэтому даже транзакции с минимальной комиссией почи моментально попадают в блоки.

Обратите внимание, что в истории Waves недолгое время был баг, который позволял перевыпускать токены, у которых `reissuable` был равен `false`. Баг был оперативно поправлен, но в блокчейне могут встратиться некоторые токены, которые всегда были неперевыпускаемые, но были перевыпущены. Удалить их оттуда не получится, ведь блокчейн иммутабелен. Так что об этом только стоит знать, если вы вдруг делаете эксплорер или какую-то аналитику.

Сжигание токенов

Иногда бывает, что токен мешает и не хочется видеть его в портфолио, а очень часто встречается необходимость сжигать по какой-то бизнес логике. Для этого в Waves есть транзакция типа `Burn`, которая позволяет сжечь токены (но только со своего аккаунта, конечно).

```
const { burn } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  assetId: 'CZw4KCpPUv5t1Uym3rLc9yEaQyDsP3VVPspdpmWKvVPE',
  quantity: 100
}

const signedBurnTx = burn(params, seed)
```

Транзакция burn максимально прооста и позволяет задать assetId токена, который хотим сжечь и количество. Собственно, все.

Изменение информации о токене

Очень много пользователей спрашивают, можно ли поменять название или описание своего токена. Причин для этого может быть много - переименование компании, изменение адреса веб-сайта (адрес

сайта мог быть в описании токена). До недавнего времени такое изменение было невозможно и заданное в самом начале жизни название и описание были навсегда с токеном, но в начале 2020 года появилась транзакция `UpdateAssetInfo`, которая позволяет обновить название и описание, но не чаще, чем раз в 100 000 блоков, что примерно равно 2,5 месяцев. На момент написания этих строк функционал был активирован только в stagenet и транзакция `UpdateAssetInfo` еще не поддерживалась библиотеками.

А дальше что?

Выпуск токена в большинстве случаев является только началом интеграции, поэтому в дальнейшем мы поговорим о том, как использовать Ride для задания логики и API для интеграций вашей бизнес логики с блокчейном.

Спонсирование транзакций

Среди разработчиков децентрализованных приложений есть несколько тем, обсуждение которых приводит к явно выраженной боли на лицах. Такими темами являются:

1. **Работа с ключами.** Просить у пользователя ключи нельзя, но он должен как-то подписывать транзакции.
2. **Необходимость платить комиссию в токенах за каждую транзакцию**. Как объяснить пользователям, что каждая транзакция требует комиссии в токене платформы, и что не менее важно - откуда они возьмут комиссии для своих первых транзакций?

Обозначенные проблемы приводят к очень высокой стоимости привлечения одного пользователя. Например, один из популярных dApp в экосистеме Waves имел стоимость привлечения клиента около $80 (!), при LTV меньше $10. Конверсию портили именно барьеры с расширением и комиссиями.

Первая проблема часто решается с помощью браузерных расширений вроде Metamask и Waves Keeper, но это решение не дружественное для пользователей и требует большого количества усилий, поэтому в экосистеме Waves появился Signer. Он не требует предоставлять ключи dApp, и в то же время не заставляет устанавливать браузерные расширения. В статье @Vladimir Zhuravlev рассказывается об этом и как интегрировать Waves Signer в свое приложение.

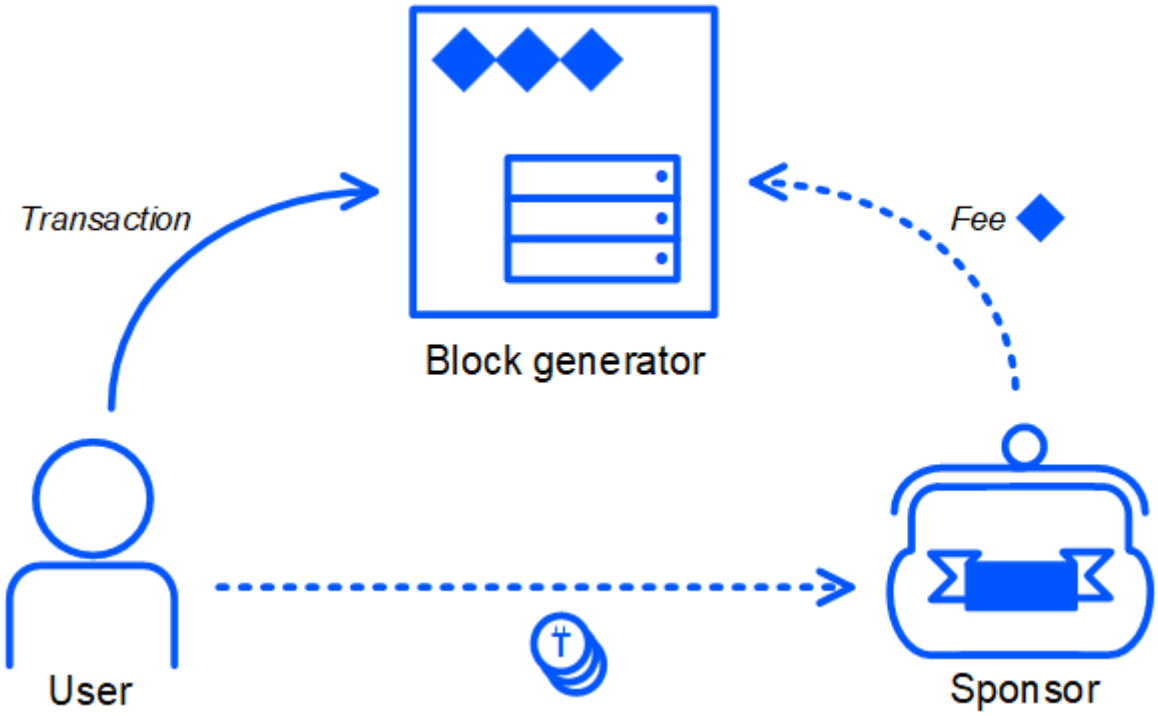
А что же по поводу второй проблемы? Многие создатели dApp просто не заботятся этим вопросом, пользователи должны откуда-то взять токены для комиссий. Другие требуют привязывать банковские карты во время регистрации, что очень сильно снижает мотивацию.

Сейчас я расскажу как решить проблему с комиссиями. Как сделать такой dApp, который не требует наличия нативного токена у пользователя. Это позволяет делать триальные периоды.

Как работает спонсирование

Если у вас есть свой токен, который нужен пользователям вашего dApp, то вы можете использовать механизм спонсирования транзакций. Пользователи будут платить комиссию в вашем токене, но так как майнеры всегда получают комиссию только в Waves, то фактически Waves будут списываться с аккаунта, выпустившего токен. Давайте еще раз по шагам, так как это важно понимать:

- Пользователь платит комиссию за транзакцию в вашем токене (например, он отправляет 10 ваших токенов, дополнительно платит 1 токен в виде комиссии, в итоге с его аккаунта списывается 11 токенов)
- Вы получаете эти токены (1 в нашем примере)
- С вашего аккаунта списываются WAVES в необходимом количестве и уходят майнерам(количество спонсируемых токенов и их соотетствие Waves настраивается в момент отправки транзакции SetSponsorship)



Вопрос, который должен был сразу возникнуть - сколько токенов заплатит пользователь и сколько токенов спишется с аккаунта спонсора?

Ответ: владелец может сам установить соотношение. В момент начала спонсирования создатель токена задает сколько его токенов соответствуют минимальной комиссии (0.001 Waves или 100 000 в минимальной фракции). Давайте перейдем к примерам и коду, чтобы было понятнее.

Для включения спонсирования, необходимо отправить транзакцию типа `Sponsorship`. С помощью пользовательского интерфейса можно сделать в Waves.Exchange, а с помощью waves-transactions можно выполнить следующий код:

```
const { sponsorship } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  assetId: 'A',
  minSponsoredAssetFee: 100
}

const signedSponsorshipTx = sponsorship(params, seed)
```

Код выше сформирует (но не отправит в блокчейн) транзакцию:


```
{
  "id": "A",
  "type": 14,
  "version": 1,
  "senderPublicKey": "3SU7zKraQF8tQAF8Ho75MSVCBfirgaQviFXnseEw4PYg",
  "minSponsoredAssetFee": 100,
  "assetId": "4uK8i4ThRGbehENwa6MxyLtxAjAo1Rj9fduborGExarC",
  "fee": 100000000,
  "timestamp": 1575034734209,
  "proofs": [

"42vz3SxqxzSzNC7AdVY34fM7QvQLyJfYFv8EJmCgooAZ9Y69YDNDptMZcupYFdN7h3C1dz2z6
keKT9znbVBrikyG"
  ]
}
```


Самым главным параметром в транзакции является `minSponsoredAssetFee`, который задает соответствие 100 токенов `A` равны 0.001 Waves. Таким образом, чтобы отправить `Transfer` пользователь должен будет в качестве комиссии приложить 100 токенов `A`.

Важно понимать некоторые ограничения, связанные со спонсированием. Использовать спонсированные токены как комиссию можно только для транзакций типов `Transfer` и `Invoke`. Спонсировать токен может только аккаунт, выпустивший этот токен. То есть, вы не сможете спонсировать токены, выпущенные не вами. Как только баланс создателя токена станет меньше 1.005 Waves, спонсирование автоматически выключитс (и обратно включится, когда баланс снова станет больше этого значения).

Безопасность

Прежде чем включать спонсирование, надо понимать несколько важных моментов.

1. Пользователь может использовать спонсируемые токены для операций не только с этим токеном. Например, аккаунт с токенами `A` на балансе может отправлять токены `B`, а как комиссию приложить токены `A`.
2. Пользователь может платить не минимальную комиссию за транзакцию. Например, если у пользователя есть 100 000 ваших токенов, а вы поставили параметр `minSponsoredAssetFee` равным 100, то пользователь сможет все свои 100 000 токенов указать в качестве комиссии. Вы получите 100 000 токенов `A`, а майнер получит 1000 Waves с вашего аккаунта (100 000 / 100 = 1000), если они есть на вашем аккаунте.

Функция спонсирования есть в Waves долгое время и отлично работает, но есть WEP-2 Customizable Sponsorship, в котором высказывались идеи по его улучшению. Если вам есть что добавить - присоединяйтесь к обсуждению.

Smart Assets

Токены на Waves по умолчанию не являются смарт-контрактом (в отличие от Ethereum), поэтому они свободно обращаются и владельцы токенов могут делать с ними все, что хочется. Но если сильно хочется изменить поведение токена, то это можно сделать, превратив обычный ассет в смарт-ассет. Мы уже рассматривали смарт-аккаунты, чье поведение отличается от обычных аккаунтов тем, что перед отправкой транзакции с такого аккаунта, выполняется его скрипт и он должен вернуть `true`. Смарт-ассеты очень сильно похожи на смарт-аккаунты. Точно так же к ассету добавляется скрипт на Ride, который выполняется при каждой операции с этим токеном и должен вернуть `true`, чтобы транзакция считалась валидной. Но есть несколько отличительных особенностей смарт-ассета.

Добавление скрипта к ассету

Главное отличие смарт-ассетов от смарт-аккаунтов заключается в том, что если токен выпущен без скрипта, то он не может быть к нему добавлен позже. Сделано это для того, чтобы создатели токенов не имели возмоожности обманывать пользователей, например, отправляя им токены, правила обращения которых могут поменяться. Важно так же пояснить, что если токен выпущен со скриптом и этот скрипт прямо не запрещает, то такой скрипт можно будет обновлять. Вы можете сказать, что владельцы токенов так тоже могут обманывать, но в таком случае пользователи с самого начала хотя бы будут видеть, что токен является смарт-ассетом и могут иметь это ввиду.

Другая причина невозможности добавления скрипта к ассетам, выпущенным без этого заключается в том, что функционал смарт-ассетов появился на третьем году жизни блокчейна Waves, и давать простым токенам, уже несколько лет живущим в сети Waves возможность менять правила игры на ходу без учета мнения пользователей было бы не совсем правильно.

А что же делать, если мы хотим выпустить токен, но не написали еще его скрипт? Достаточно в виде скрипта поставить `true` (а точнее скомпилированную версию такого скрипта в формате base64 - `AwZd0cYf`) как скрипт. Такой скрипт не будет запрещать никакие операции с токеном, но позволит в дальнейшем обновить скрипт и задать нужные вам правила.

В разделах 5 (Транзакции) и 6 (Ride) мы подробнее рассмотрим особенности задания скриптом для ассетов и отличительные особенности Ride для токенов.

Торговля ассетами и DEX

После появления возможности создания своих токенов, было логичным сделать возможность торговли ими (а если быть точнее - обмена) без участия посредников. Для этого в Waves был создан матчер (от англ "match" - соответствовать, подходить под пару), долгое время являвшийся частью ноды, по умолчанию выключенной (достаточно было в конфигурации ноды включить флаг `waves.matcher.enabled`).

Как работает матчер

Матчер принимает от пользователей заявки на обмен токенов, в экосистеме Waves такие заявки принято называть order. Пример такой "заявки" или "намерения" пользователя совершить обмен, представлен ниже:

```
{
  "version": 3,
  "senderPublicKey": "FMc1iASTGwTC1tDwiKtrVHtdMkrVJ1S3rEBQifEdHnT2",
  "matcherPublicKey": "7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy",
  "assetPair": {
    "amountAsset": "BrjUWjndUanm5VsJkbUip8VRYy6LWJePtxya3FNv4TQa",
    "priceAsset": null
  },
  "orderType": "buy",
  "amount": 150000000,
  "timestamp": 1548660872383,
  "expiration": 1551252872383,
  "matcherFee": 300000,
  "proofs": [

"YNPdPqEUGRW42bFyGqJ8VLHHBYnpukna3NSin26ERZargGEboAhjygenY67gKNgvP5nm5ZV8V
GZW3bNtejSKGEa"
  ],
  "id": "Ho6Y16AKDrySs5VTa983kjg3yCx32iDzDHpDJ5iabXka",
  "sender": "3PEFvFmyyZC1n4sfNWq6iwAVhzUT87RTFcA",
  "price": 1799925005,
}
```


Помимо информации об отправителе, служебных полей и подписи, каждый order содержит в себе информацию о том, в какой паре токенов должен произойти обмен, тип ордера (`buy` или `sell`), срок истечения действия ордера, количество токенов для обмена и цену, по которой пользователь хочет совершить обмен. Посмотрев на пример выше, можно понять, что пользователь хочет обменять `Waves`, потому что `assetPair.priceAsset` равен `null` и тип ордера `buyd`, на токен с assetId `BrjUWjndUanm5VsJkbUip8VRYy6LWJePtxya3FNv4TQa` и названием `Zcash`, которое можно найти в эксплорере.

Количество токенов для обмена указано 150000000 (всегда помним, что у Waves 8 знаков после запятой, поэтому фактически он хочет обменять 1.5 Waves) на Zcash по цене 17.99925005 за единицу (у Zcash количество знаков после запятой тоже 8). Иными словами, если найдется желающий продать 1 Zchah токен в обмен на 17.99925005 не позднее указанной даты экспирации (1551252872383 или 02/27/2019 @ 7:34am UTC), то будет совершен обмен.

Давайте представим, что другой пользователь отправил контр-ордер для этой же пары со следующими параметрами:

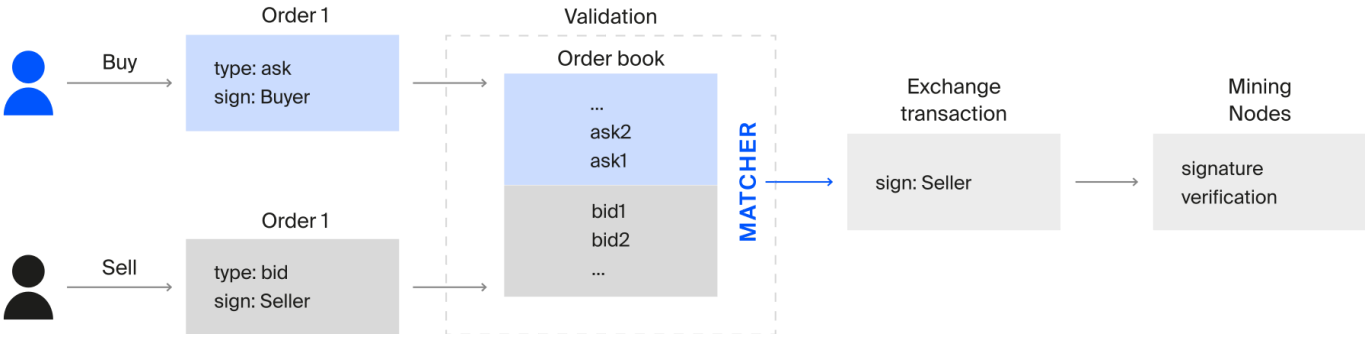

```
{
  "version": 3,
  "senderPublicKey": "FMc1iASTGwTC1tDwiKtrVHtdMkrVJ1S3rEBQifEdHnT2",
  "matcherPublicKey": "7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy",
  "assetPair": {
    "amountAsset": "BrjUWjndUanm5VsJkbUip8VRYy6LWJePtxya3FNv4TQa",
    "priceAsset": null
  },
  "orderType": "sell",
```

```
  "amount": 3000000000,
  "timestamp": 154866085334,
  "expiration": 1551252885334,
  "matcherFee": 300000,
  "proofs": [

"YNPdPqEUGRW42bFyGqJ8VLHHBYnpukna3NSin26ERZargGEboAhjygenY67gKNgvP5nm5ZV8V
GZW3bNtejSKGEa"
  ],
  "id": "Ho6Y16EFvFmyyZC1n4sfNWq6iwAVhzUT87RTFcAabXka",
  "sender": "3PAKDrySs5VTa983kjg3yCx32iDzDHpDJ5i",
  "price": 1799925005,
}
```

Отправитель этого ордера хочет сделать обратную операцию обмена (`Zcash` -> `Waves`) по такой же цене, но хочет обменять 30 Zcash.

Оба ордера отправляются на один матчер с публичным ключом `7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy`, который увидев совпадение параметров (пара, цена) и валидность подписи и даты экспирации, сформирует транзакцию обмена - `Exchange`. При этом, первый оредр будет исполнен полностью (все 1.5 Waves будут обменены на Zcash), а второй только частично и будет дальше ждать подходящий ордеров для совершения обмена. Примерная схема работы представлена на рисунке:



Пример `Exchange` транзакции мы рассмотрим в главе про транзакции, давайте сейчас поговорим про особенности матчера.

Функции матчера

Матчер является сердцем децентрализованных бирж (DEX) на базе Waves, самой популярной из которых сейчас является waves.exchange. Давайте разберемся как работает матчер и вся процедура децентрализованного обмена.

Матчер принимает от всех желающих их ордера на покупку или продажу токенов, хранит их в стакане (orderbook) и при нахождении соответствия формирует транзакцию обмена и отправляет в блокчейн (отправляет ноде, которая уже добавляет в блок, непосредственно производя обмен токенов на балансах пользователей).

Давайте опишем весь путь для обмена токенов:

1. Пользователь формирует ордер на совершения обмена, указывая пару токенов, тип ордера (что на что хочет обменять), цену обмена, количество токенов для обмена, срок действия, размер комиссии для матчера и на какой матчер хочет отправить свой ордер.
2. Пользователь подписывает ордер и отправляет на матчер по API.
3. Матчер проверяет валидность подписи ордера, правильность указанных дат, указанную пользователем комиссию и наличие токенов для обмена и комиссии на балансе у пользователя (для этого делает запрос к блокчейн ноде).
4. Если обмен совершается в паре, где один или оба токена являются смарт-ассетами, матчер выполняет скрипт ассета и только при получении `true` считает ордер валидным. В случае получения `false` или исключения, матчер считает ордер не валидным.
5. В случае нахождения в стакане (orderbook) контр-ордера, с которым можно совершить операцию обмена, матчер формирует `Exchange` транзакцию и отправляет блокчейн ноде. Если подходящего ордера не было в стакане, то свежесозданный ордер добавляется в стакан, где будет находиться до тех пока, не найдется правильный контр-ордер или не закончится срок действия ордера. Стоит заметить, что транзакция обмена делается от имени матчера и с подписью матчера, а не от имени пользователей, соответственно, комиссию за попадание в блокчейн платится матчером.
6. Блокчейн нода при получении `Exchange` транзакции валидирует транзакцию и входящие в него ордера (транзакция обмена в себя включает сами ордера тоже) и добавляет в блок.
7. Состояние балансов аккаунтов в блокчейне меняется в соответствии с параметрами `Exchange` транзакции.

Особенности обмена в блокчейне Waves

Децентрализованный обмен в Waves может осуществляться и без матчера: два пользователя могут свои ордера объединить в `Exchange` транзакцию и отправить в сеть от имени третьего аккаунта (или одного из них), но в виду неудобности такого способа, большинство транзакций обмена совершается с помощью матчера.

Матчер является централизованной сущностью и контролируется одним лицом или командой, но почему мы тогда называем обмен децентрализованным, а биржи с использованием матчера - DEX? Надо разобраться в главном отличии обычных централизованных бирж от DEX - контроль средств. Централизованные биржи имеют прямой доступ к средствам пользователей и их ключам, поэтому могут делать с ними все, что хотят, в то время как матчер в Waves имеет доступ только к намерениям пользоваталей (ордерам) и не могут ничего сделать с вашими токенами напрямую. Самое плохое, что может сделать матчер - обменять по не самой выгодной цене, которая есть рынке или не совершить транзакцию обмена, хотя контр-ордер был в стакане.

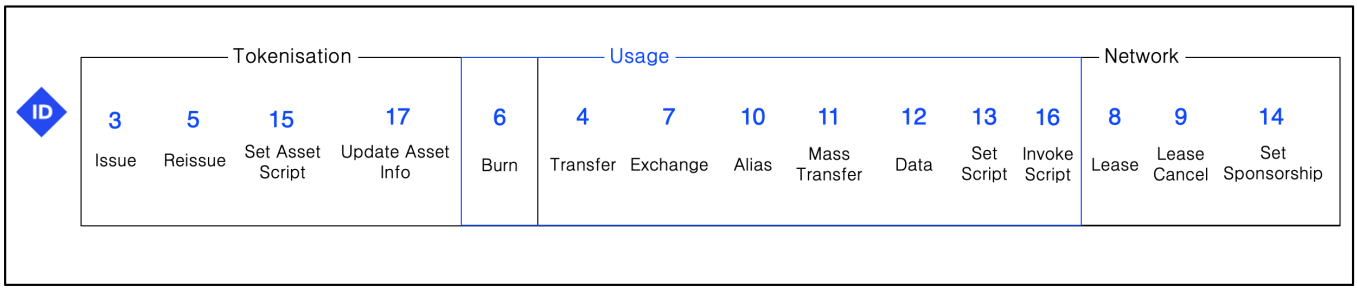
Есть ли более децентрализованные решения? Конечно есть, есть полностью децентрализованные биржи, однако при полной децентрализации невозможно решить проблему фронт-раннинга блокчейн нод (в описанной выше схеме есть риск фронт-раннинга матчера).

Другой особенностью обмена является то, что матчеров в экосистеме Waves много, но они не обмениваются ордерами друг с другом. Фактически, вы доверяете одному матчеру, когда отправляете ему свой ордер. Вы ему доверяете, что он сделает операцию и он сделает это честно (например, не пустит вперед вашего ордер, который пришел позже). Именно это доверие мешает сделать обмен ордерами: скорее всего, вы готовы довериться одному матчеру, но не готовы довериться всем, потому что любой из множества может оказаться "вредителем".

Наличие централизованного матчинга позволяет достичь отличной пропускной способности в тысячи формируемых `Exchange` транзакций в секунду. Максимально возможность скорость работы матчера сейчас намного выше, чем пропускная способность блокчейна. Конечно, торговать в режиме высокочастотной торговли (HFT, high-frequency trading) не получится, но есть большое количество ботов, которые делает сотни транзакций в секунду. Примеры ботов вы можете найти в github, самые популярные - Scalping Bot и Grid Trading Bot.

Транзакции в Waves

В отличие от многих других блокчейнов, где есть 1 (Bitcoin) или 2 (Ethereum) типа транзакций, в Waves их нассчитывается 17 на момент написания этих строк. Ниже представлена схема с условным разделением всех актуальных типов транзакций на категории:



У вас уже могли возникнуть вопросы: "Почему у транзакций такой хаотичный порядок нумерации?, почему нумерация не идет последовательно хотя бы в рамках одной категории?".

Дело в том, что транзакции получали номера (они же ID) по мере их добавления в протокол. В этой части мы будем рассматривать транзакции по мере их появления в блокчейне.

Важно: у многих типов транзакций есть несколько версий, в этой книге мы будем рассматривать последние актуальные версии в Mainnet.

Работа с транзакциями осуществуется с помощью API ноды, который позволяет как получать транзакции, так и отправлять их. Для вас, как для разработчиков, транзакция в большинстве случаев будет выглядеть как простой JSON:

```
```

Сама нода хранит транзакции в бинарном представлении, а не в виде JSON, но в момент запроса по API кодирует в JSON и отдает в таком виде. Принимает она ее тоже в виде JSON. В REST API ноды есть следующие полезные эндпоинты:

`GET /transactions/info/{id}` - получить информацию об одной транзакции `GET /transactions/address/{address}/limit/{limit}` - получить транзакции по адресу `GET /blocks/at/{height}` - получить список всех транзакций в блоке

Подпись транзакций

У всех транзакций есть важное поле - `senderPublicKey`, которое определяет от имени какого аккаунта совершается действие. Чтобы транзакция ("действие") считалась валидной, необходимо, чтобы подпись транзакции соответствовала этому публичному ключу (случаи со смарт-аккаунтами сейчас не рассматриваем).

Криптографические функции подписи ничего не знают про транзакции, так как они работают с байтами. В случае Waves, для подписания транзакций необходимо байты транзакции расположить в правильном порядке и передать функции подписи вместе с приватным ключом, в итоге получим подпись.

```
signature = sign(transactionBytes, privateKey)
```

Правильный порядок байтов для каждой транзакции описан в документации. Криптография выходит за пределы этой книги, но мы можем рассмотреть как формируется подпись для одного из типов транзакции в JS библиотеке:

```
```

Подписание транзакций делается обычно на стороне клиентского приложения, но сама нода так же умеет подписывать транзакции через API. Надо понимать, что нода подпишет тем приватным ключом, который задан у нее в конфигурации. **Подписать транзакцию от произвольного отправителя с помощью REST API нельзя.** Многие разработчики думают, что им необходимо получить API key, чтобы подписать свою транзакцию с помощью ноды, но это будет работать только в том случае, есть в конфигурации задан приватный ключ от того аккаунта, который должен совершать действие.

Жизненный цикл транзакции

Давайте разберем все стадии работы с транзакцией на примере одного действия - отправки токена от одного пользователя другому. У нас возникло желание отправить токены с нашего аккаунта, от которого мы знаем seed фразу (`A` в нашем примере). Отправлять будем на аккаунт с публичным ключом `B`. Первым делом нам необходимо задать параметры транзакции:

```
const params = {
  amount: 300000000,
  recipient: address('B'),
  feeAssetId: null,
  assetId: null,
  attachment: 'TcgsE5ehTSPUftEquDt',
  fee: 100000,
}
```

Поля транзакции мы разберем в следующей части этой главы. Сейчас сконцентрируемся на последовательности действий. Чтобы получить транзакцию вместе с подписью для наших параметров, мы используем библиотеку `waves-transactions`. Функции `transfer` мы передаем обозначенные

выше параметры и сид-фразу. В итоге мы получим JS объект, который будет содержать все указанные нами поля, а так же подпись в массиве `proofs`, время подписания транзакции (`timestamp`) и публичный ключ отправителя (аккаунта с сид фразой `A`) в поле `senderPublicKey`.

```
const signedTransferTx = transfer(params, 'A');
broadcast(signedTransferTx);
```

Библиотека от нас скрывает криптографию и подготовительный этап - формирование правильного порядка байт для подписи. Подписанная транзакция в форме JS объекта может быть отправлена в любую ноду, у которой открыт API. Запрос отправляется на `POST /transactions/broadcast` в виде JSON. Нода примет транзацию, если нет никаких проблем - подпись валидная, хватает токенов на балансе нашего аккаунта для совершения транзакции и т.д. Провалидированная транзакция попадет в UTX ноды, куда мы отправляли запрос, а она уже дальше будет рассылать информацию об этой транзакции всем нодам, с которыми она соединена.

UTX

UTX - список транзакций, которые находятся в ожидании попадания в блок. То есть кто-то их отправил и нода приняла транзакцию, но транзакция в блок не попала. В Waves есть определенные особенности, связанные с тем как такие транзакции обрабатываются. Как транзакция может попасть в UTX? Существует всего 2 способа для этого:

- Кто-то отправит транзакцию на эту ноду (с помощью REST API или gRPC)
- Нода получит транзакция по бинарному протоколу от другой ноды в сети

В конечном итоге можно сказать, что транзакции в сеть приходят через API, но не объязательно, чтобы это был API данной конкретной ноды.

У транзакции, которая попала в UTX, есть 2 варианта дальнейшего развития событий:

- В какой-то момент времени она будет добавлена в блок одним из майнеров и попадет в блок
- Транзакция станет невалидной и будем удалена из UTX (и никогда не сможет попасть в блок). Транзакция может стать невалидной по нескольким причинам - изменилось состояние блокчейна (другая транзакция попала в блок и изменила баланс отправителя, скрипт на аккаунте или скрипте теперь уже возвращает `false` и тд), истекло время жизни транзакции (сейчас в сети Waves `timestamp` транзакции может отличаться на -2 или +1.5 часа от текущего времени блокчейна).

Время жизни транзакции может истечь только по причине загрузки сети на все 100%. Ноды в Waves добавляют в блок транзакции поочередно, начиная с самых выгодных для них с наибольшей комиссией. Если в момент отправки нашей транзакции перевода токенов, в UTX было много транзакций с большей комиссией, то майнеры не будут добавлять в блок нашу, ведь у блока есть лимит на размер (1 МБ) и количество (6000). Майнеры будут производить блоки максимального размера с самыми выгодными для них транзакциями. Если такое продолжится на протяжении 90 минут, то наша транзакция станет невалидной. На самом деле сортировка транзакция в UTX майнерами производится не только на основе размера коммиссии, поэтому особенности работы UTX мы рассмотрим в дальнейшем.

Для многих новичков становится неожиданностью, что в Waves в блоки могут попадать транзакции "из прошлого" и "из будущего", у которых `timestamp` на 120 минут меньше или 90 минут больше настоящего времени. В некоторых случаях необходимо это учитывать при разработке своих приложений.

Типы транзакций

В Waves есть два типа транзакций, которые сейчас не используются и которые вам точно не пригодятся при работе в основной сети - `Genesis` и `Payment`. Долго останавливаться на этих типах транзакций не будем.

Genesis транзакция (type = 1) [deprecated]

`Genesis` транзакции были только в самом первом блоке блокчейна и отвечали за распределение предвыпущенных токенов. Их сразу было 100 миллионов. Давайте посмотрим как выглядел `genesis` блок.

Примечание: Многие путают `genesis` блок и `genesis` транзакции. `Genesis` блок - самый первый блок в блокчейн сети (во всех блокчейнах принято так называть), который отличается от остальных блоков только отсутствие ссылки на предыдущий блок, ведь его попросту не было. `Genesis` блок содержит `genesis` транзакции, которые отвечают за первоначальное распределение выпущенных токенов Waves.


```
{
  "reference":
"67rpwLCuS5DGA8KGZXKsVQ7dnPb9goRLoKfgGbLfQg9WoLUgNY77E2jT11fem3coV9nAkguBA
CzrU1iyZM4B8roQ",
  "blocksize": 500,
  "signature":
"FSH8eAAzZNqnG8xgTZtz5xuLqXySsXgAjmFEC25hXMbEufiGjqWPnGCZFt6gLiVLJny16ipxR
NAkkzjjhqTjBE2",
  "totalFee": 0,
  "nxt-consensus": {
    "base-target": 153722867,
    "generation-signature": "11111111111111111111111111111111"
  },
  "fee": 0,
  "generator": "3P274YB5qseSE9DTTL3bpSjosZrYBPDpJ8k",
  "transactionCount": 6,
  "transactions": [
    {
      "type": 1,
      "id":
"2DVtfgXjpMeFf2PQCqvwxAiaGbiDsxDjSdNQkc5JQ74eWxjWFYgwvqzC4dn7iB1AhuM32WxEi
Vi1SGijsBtYQwn8",
      "fee": 0,
      "timestamp": 1465742577614,
      "signature":
"2DVtfgXjpMeFf2PQCqvwxAiaGbiDsxDjSdNQkc5JQ74eWxjWFYgwvqzC4dn7iB1AhuM32WxEi
```

```
Vi1SGijsBtYQwn8",
      "recipient": "3PAWwWa6GbwcJaFzwqXQN5KQm7H96Y7SHTQ",
      "amount": 9999999500000000
    },
    {
      "type": 1,
      "id":
"2TsxPS216SsZJAiep7HrjZ3stHERVkeZWjMPFcvMotrdGpFa6UCCmoFiBGNizx83Ks8DnP3qd
wtJ8WFcN9J4exa3",
      "fee": 0,
      "timestamp": 1465742577614,
      "signature":
"2TsxPS216SsZJAiep7HrjZ3stHERVkeZWjMPFcvMotrdGpFa6UCCmoFiBGNizx83Ks8DnP3qd
wtJ8WFcN9J4exa3",
      "recipient": "3P8JdJGYc7vaLu4UXUZc1iRLdzrkGtdCyJM",
      "amount": 100000000
    },
    {
      "type": 1,
      "id":
"3gF8LFjhnZdgEVjP7P6o1rvwapqdgxn7GCykCo8boEQRwxCufhrgqXwdYKEg29jyPWthLF5cF
yYcKbAeFvhtRNTc",
      "fee": 0,
      "timestamp": 1465742577614,
      "signature":
"3gF8LFjhnZdgEVjP7P6o1rvwapqdgxn7GCykCo8boEQRwxCufhrgqXwdYKEg29jyPWthLF5cF
yYcKbAeFvhtRNTc",
      "recipient": "3PAGPDPqnGkyhcihyjMHe9v36Y4hkAh9yDy",
      "amount": 100000000
    },
    {
      "type": 1,
      "id":
"5hjSPLDyqic7otvtTJgVv73H3o6GxgTBqFMTY2PqAFzw2GHAnoQddC4EgWWFrAiYrtPadMBUk
oepnwFHV1yR6u6g",
      "fee": 0,
      "timestamp": 1465742577614,
      "signature":
"5hjSPLDyqic7otvtTJgVv73H3o6GxgTBqFMTY2PqAFzw2GHAnoQddC4EgWWFrAiYrtPadMBUk
oepnwFHV1yR6u6g",
      "recipient": "3P9o3ZYwtHkaU1KxsKkFjJqJKS3dLHLC9oF",
      "amount": 100000000
    },
    {
      "type": 1,
      "id":
"ivP1MzTd28yuhJPkJsiurn2rH2hovXqxr7ybHZWoRGUYKazkfaL9MYoTUym4sFgwW7WB5V252
QfeFTsM6Uiz3DM",
      "fee": 0,
      "timestamp": 1465742577614,
      "signature":
"ivP1MzTd28yuhJPkJsiurn2rH2hovXqxr7ybHZWoRGUYKazkfaL9MYoTUym4sFgwW7WB5V252
QfeFTsM6Uiz3DM",
      "recipient": "3PJaDyprvekvPXPuAtxrapacuDJopgJRaU3",
```

```
      "amount": 100000000
    },
    {
      "type": 1,
      "id":
"29gnRjk8urzqc9kvqaxAfr6niQTuTZnq7LXDAbd77nydHkvrTA4oepoMLsiPkJ8wj2SeFB5KX
ASSPmbScvBbfLiV",
      "fee": 0,
      "timestamp": 1465742577614,
      "signature":
"29gnRjk8urzqc9kvqaxAfr6niQTuTZnq7LXDAbd77nydHkvrTA4oepoMLsiPkJ8wj2SeFB5KX
ASSPmbScvBbfLiV",
      "recipient": "3PBWXDFUc86N2EQxKJmW8eFco65xTyMZx6J",
      "amount": 100000000
    }
  ],
  "version": 1,
  "timestamp": 1460678400000,
  "height": 1
}
```

Можно заметить, что было 6 публичных ключей-получателей свеже выпущенных токенов Waves. У всех транзакций одинаковый timestamp и они все были бесплатными (`fee` равен нулю), потому что нечем еще было платить `fee` на момент создания этих транзакций.

Эти транзакции созданы не вручную, они генерируются автоматически специальной утилитой `genesis-generator`, который есть в репозитории ноды. Вам это может понадобится сделать, если вы захотите запустить свой приватный блокчейн. Как это сделать (и зачем) мы рассмотрим в одной из следующих глав.

Внимательные читатели могут спросить, почему в самой первой транзакции отправляется `9999999500000000` токенов, если было выпущено всего 100 миллионов? В Waves во всех транзакциях счет идет минимальными неделимыми единицами токена (fraction). У токена Waves количество знаков после запятой (decimals) равно 8, поэтому минимальная единица - одна сто миллионная. Если в поле `amount` любой транзакции стоит значение `100000000` (10^8), это обозначает на самом деле один целый токен Waves. В случае с `genesis` транзакцией, `9999999500000000` означает 99 999 995 токенов или 9999999500000000 минимальных единиц. Минимальные единицы Waves часто называют WAVELET.

Payment транзакция (type = 2) [deprecated]

В момент запуска блокчейна Waves было реализовано всего 2 типа транзакций - уже рассмотренный тип `genesis` и `payment`, который позволял переводить токены Waves с одного аккаунта на другой. Примеры транзакции `payment` в JSON представлении можно посмотреть в блоке под номером 2000.

```
    {
      "senderPublicKey": "6q5VhGeTanU5T8vWx6Jka3wsptPKSSHA9uXHwdvBMTMC",
      "amount": 10000000000,
      "sender": "3PGj6P4Mfzgo24i8cG3nhLU6uktF6s5LVCT",
```

```
      "feeAssetId": null,
      "signature":
"3gzk9QyfqQGvsU8A4zMMorpKTcFpdG7UtC4c5E7ds9MGMCMSyp6JZymQJoCjUSJQ8AaSWQDQw
NmQ5F46ud4ofA5o",
      "proofs": [

"3gzk9QyfqQGvsU8A4zMMorpKTcFpdG7UtC4c5E7ds9MGMCMSyp6JZymQJoCjUSJQ8AaSWQDQw
NmQ5F46ud4ofA5o"
      ],
      "fee": 1,
      "recipient": "3P59ixWkqiEnL7RJoXtZewgbatKBZo8bG15",
      "id":
"3gzk9QyfqQGvsU8A4zMMorpKTcFpdG7UtC4c5E7ds9MGMCMSyp6JZymQJoCjUSJQ8AaSWQDQw
NmQ5F46ud4ofA5o",
      "type": 2,
      "timestamp": 1465865163143
    }
```

`Payment` транзакция умеет отправлять только Waves токены (не кастомные ассеты) с одного адреса на другой. Она стала устаревшей с появлением `Transfer` транзакций, которые умеют отправлять как токены Waves, так и кастомные токены, поэтому сейчас `Payment` уже нигде не используется.

Issue транзакция (type = 3)

D разделе про токены мы уже подробно рассматривали как выпустить свой ассет с помощью `Issue` транзакции, поэтому сейчас не буду останавливаться на том, как его использовать. Стоит только сказать, что отличительная особенность `Issue` транзакции в наличии 2 принципиально разных вариантов выпуска токена:

- выпуск уникального токена (он же не взаимозаменяемый токен, non-fungible token, NFT)
- выпуск обычного токена

Выпуск уникального токена отличается тем, что параметры `amount`, `reissuable`, `decimals` должны иметь заранее известные значения - `1`, `false` и `0` соответственно. При соблюдении этого условия минимальная комиссия составит 0.001 Waves. Если данные параметры отличаются (хотя бы один из параметров), то токен считается обычным и минимальная комиссия выпуска составит 1 Waves.

Пример JSON представления `Issue` транзакции представлен ниже:

```
{
    senderPublicKey: "7nSKRN4XZiD3TGYsMRQGQejzP7x8EgiKoG2HcY7oYv6r",
    quantity: 210000000,
    signature:
"3Vj8M9tkVZmnjdYAKKN3GzAtV9uQDX5hhgUfXQDdvZsk2AmvqQum3oGBJqdjALVHXX2ibLAZH
eruwjNXR46WgBnm",
    fee: 100000000,
    description: "",
    type: 3,
    version: 1,
    reissuable: true,
```

```
    sender: "3PAJ6bw7kvSPf6Q9kAgfSLzmpFspZmsi1ki",
    feeAssetId: null,
    proofs: [
"3Vj8M9tkVZmnjdYAKKN3GzAtV9uQDX5hhgUfXQDdvZsk2AmvqQum3oGBJqdjALVHXX2ibLAZH
eruwjNXR46WgBnm"
],
    script: null,
    assetId: "oWgJN6YGZFtZrV8BWQ1PGktZikgg7jzGmtm16Ktyvjd",
    decimals: 1,
    name: "ihodl",
    id: "oWgJN6YGZFtZrV8BWQ1PGktZikgg7jzGmtm16Ktyvjd",
    timestamp: 1528867061493,
    height: 1039500
}
```

> Важно: если токен выпущен без скрипта, то он не может быть к нему добавлен позже, поэтому если вы хотите добавить скрипт в будущем, но пока у вас нет этого скрипта, то в качестве скрипта указывайте `AwZd0cYf` (`true` в скомпилированном base64 варианте)

Tranfer транзакция (type = 4)

`Tranfser` транзакция пришла на замену `Payment` транзакции, потому что `Payment` не позволял отправлять токены, созданные с помощью `Issue` транзакции. В данный момент `Transfer` транзакция является наиболее часто встречающейся по данным dev.pywaves.org и составляет порядка 70% транзакций в сети. Отправка `Transfer` транзакции похожа на отправку большинства транзакций, связанных с токенами:

```
const { transfer } = require('@waves/waves-transactions');

const seed = 'example seed phrase';

//Transfering 3 WAVES
const params = {
  amount: 300000000,
  recipient: '3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs',
  feeAssetId: null,
  assetId: null,
  attachment: 'TcgsE5ehTSPUftEquDt',
  fee: 100000,
}

const signedTransferTx = transfer(params, seed);
broadcast(signedTransferTx);
```

Пример выше сгенерирует транзакцию от аккаунта с сид фразой `example seed phrase`, автоматически подставит в созданную транзакцию дополнительные поля

(`timestamp`,`senderPublicKey`, `proofs`), подпишет приватным ключом от указанной сид-фразы и добавит подпись транзакции в массив `proofs`.

Получаетелем транзакции является адрес `3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs`, а отправляем мы токены Waves. Чтобы вычислить сколько отправляется токенов нам надо вспомнить, что в транзакции указывается значение `amount` в минимальных фракциях этого токена. Чтобы получить в целых единицах, надо `300000000` разделить на `10^decimals`. `300000000 / (10^8) = 3`.

У `Transfer` транзакции есть несколько интересных особенностей:

- Она поддерживает спонсирование транзакций, поэтому в поле `feeAssetId` можно указать assetId какого-либо токена, который есть у вас и спонсируется создателем, тогда вы заплатите комиссию в этом токене. В нашем случае указано `null`, поэтому комиссия будет уплачиваться в Waves.
- У транзакции есть поле `attachment`, который может содержать до 140 байт информации. В библиотеке `waves-transactions` значение `attachment` надо передавать в формате `base58`, поэтому вы видите `TcgsE5ehTSPUftEquDt`, хотя в "человеческом" представлении можно прочитать как `HelloWavesBook`.

`Transfer` транзакция позволяет указать в поле `amount` 0, то есть отправить 0 токенов получателю. Некоторые пользователи используют такую особенность для отправки `Transfer` транзакций как "сообщений" или событий, которые могут вызывать другие действия уже не в рамках блокчейна.

Пример `Transfer` транзакции представлен ниже:

```
{
    senderPublicKey: "CXpZvRkJqBfnAw3wgaRbeNjtLJcithoyQQQSzGQZRF3x",
    amount: 32800000000,
    signature:
"4cR2NAor9WjeTbysg2QMerkgymc5RLrX8PPjdXkUkWEc7BFBKMCCj8RKF7X1UchbvtEGoqGyQ
h62MDq5KoXsnCzg",
    fee: 100000,
    type: 4,
    version: 1,
    attachment: "",
    sender: "3P4FoAakEyk78TxUBcXH4uZXLaSE5BiDgjz",
    feeAssetId: null,
    proofs: [
"4cR2NAor9WjeTbysg2QMerkgymc5RLrX8PPjdXkUkWEc7BFBKMCCj8RKF7X1UchbvtEGoqGyQ
h62MDq5KoXsnCzg"
],
    assetId: null,
    recipient: "3PNX6XwMeEXaaP1rf5MCk8weYeF7z2vJZBg",
    feeAsset: null,
    id: "JAutkv1Nk4xVrkb4fkacS4451VvyHC3iJtEDfBRD7rwr",
    timestamp: 1528867058828,
    height: 1039500
}
```

Reissue транзакция (type = 5)

Если при выпуске транзакции указать флаг `reissuable` в значение `true`, то создатель токена получает возможность перевыпускать токен. История `reissuable` транзакций в Waves немного странная, так как вы можете найти в блокчейне токены, которые в момент создания имели флаг `reissuable` равный `false`, но были перевыпущены. Таких токенов было всего 4, вот их assetId: `6SGeUizNdhLx8jEVcAtEsE7MGPHGYyvL2chdmPxDh51K`, `UUwsxTvvG7LiN7yaAKvNU48JHcSwQ3q1HvsXyAgc9fL`, `3DhpxLxUrotfXHcWKr4ivvLNVQUueJTSJL5AG4qB2E7U`, `CH1LNr9ASLVqSHDb482ZzSA5rBVLDtF5QbfECGgwE8bh`. Такое стало возможным благодаря багу в коде ноды, он позволял перевыпускать неперевыпускаемые токены. Данный был просуществовал непродолжительное время. Не удивляйтесь, если найдете перевыпущенные `non-reissuable` токены в истории mainnet Waves.

Пример `Reissue` транзакции представлен ниже:

```
{
    senderPublicKey: "4X2Fv5XaDwBj2hjRghfqmsQDvBHqSa2zBUgZPDgySSJG",
    quantity: 10000000000000000,
    signature:
"5nNrLV46rVzQzeScz3RmZF4rzaV2XaSjT9kjtHoyrBzAj3iVZM9Gy6t5Paho7xRx9dyqzj1AK
yWYQsgL2nFa7jYU",
    fee: 1000000,
    type: 5,
    version: 1,
    reissuable: true,
    sender: "3P6ms9EotRX8JwSrebeTXYVnzpsGCrKWLv4",
    feeAssetId: null,
    chainId: null,
    proofs: [
"5nNrLV46rVzQzeScz3RmZF4rzaV2XaSjT9kjtHoyrBzAj3iVZM9Gy6t5Paho7xRx9dyqzj1AK
yWYQsgL2nFa7jYU"
],
    assetId: "AC3KZWmywTEYrcQwpjg4sQiWxkZ2TZmv81JAvDmsoQvy",
    id: "6qd8QbnFrKEibTr26JyNh1hc4KaafGQYStyShtXdNk3v",
    timestamp: 1528733511933,
    height: 1037381
}
```

Burn транзакция (type = 6)

Транзакция сжигания токенов позволяет сжечь любое количество токенов одного вида. Единственное условие - эти токены должны быть на вашем аккаунте.

Пример `Burn` транзакции представлен ниже:

```
{
    senderPublicKey: "EhuzuzEWHhZGo1th6YGy34AecoRP4sVi863xXCQUmgUT",
    amount: 10000000000,
    signature:
"5HdfqY47Pm4G6h67K9ZpN7jQ4NKr9hsNsmTAtyFD5FhBPr3J9kNxodhYn6hMSieKE7UmYZvSo
```

```
hv7XJpyjKvGCfTC",
    fee: 100000,
    type: 6,
    version: 1,
    sender: "3PAjApsrjJWGmRDbGo65gGgrN2hFJroAZDC",
    feeAssetId: null,
    proofs: [
"5HdfqY47Pm4G6h67K9ZpN7jQ4NKr9hsNsmTAtyFD5FhBPr3J9kNxodhYn6hMSieKE7UmYZvSo
hv7XJpyjKvGCfTC"
],
    assetId: "56w2Jbj8MGKwSWyTXvCzkqKKHiyX7C2zrgCQb2CEwM52",
    id: "EzeiYzYPwyJNEgofQrE23rpqaYERjUSnCaXZ84vUDoec",
    timestamp: 1528814759445,
    height: 1038647
}
```

Exchange транзакция (type = 7)

В предыдущей главе мы достаточно много говорили про процедуру обмена токенов, работу матчера и `Exchange` ордера. В том числе затронули тему, что транзакция содежит в себе ордера, и именно поэтому данная транзация является наиболее сложной в JSON представлении:

```
{
    senderPublicKey: "7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy",
    amount: 74,
    signature:
"2p1BS5BPkMW4C3C6vL8MsrQ8CBQRQqDoYieaZcxeMAq5zvAsm6T4N5DDN6MfPx8emVmbHfibZ
Rsok2v2Ss45e1mj",
    fee: 300000,
    type: 7,
    version: 1,
    sellMatcherFee: 63610,
    sender: "3PJaDyprvekvPXPuAtxrapacuDJopgJRaU3",
    feeAssetId: null,
    proofs: [
"2p1BS5BPkMW4C3C6vL8MsrQ8CBQRQqDoYieaZcxeMAq5zvAsm6T4N5DDN6MfPx8emVmbHfibZ
Rsok2v2Ss45e1mj"
],
    price: 103526336,
    id: "GHKhG3CWNfXAPWprk9bHSE4rxN6QfNDe3d3rZGaDLWhm",
    order2: {
    version: 1,
    id: "5C8qLi2eK92CJtBqXbL9pMuQ2R9VpRMaJ6NGACfxMBCn",
    sender: "3P7DsCo8TN5t1PNz45exhLe6vKFkTQJYrNb",
    senderPublicKey: "6mYVd69bZsLYW9gpxu3Vjneaf4xpZPnKYiLFuGXJQKQw",
    matcherPublicKey: "7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy",
    assetPair: {
    amountAsset: "725Yv9oceWsB4GsYwyy4A52kEwyVrL5avubkeChSnL46",
    priceAsset: null
},
    orderType: "sell",
```

```
    amount: 349,
    price: 103526336,
    timestamp: 1528814695617,
    expiration: 1528814995617,
    matcherFee: 300000,
    signature:
"4DSQvXBLA4U4mtTRzjz62Ci757TZsys8phWbfnCmwvrKDhYFfB8kEknJ9fknAfWkJua7wN4EP
bdrSLPgRShaxTsj",
    proofs: [
"4DSQvXBLA4U4mtTRzjz62Ci757TZsys8phWbfnCmwvrKDhYFfB8kEknJ9fknAfWkJua7wN4EP
bdrSLPgRShaxTsj"
]
},
    order1: {
    version: 1,
    id: "Eiy6wSzu3aZu3V5Mi7VN54Vmu5KQE18nEQ3j5bJU2WYK",
    sender: "3PMFLMN9GG1coCXRn26vUmF2vtCCd4RDWRR",
    senderPublicKey: "Dk3r1HwVK1Ktp3MJCoAspNyyRpLFcs2h5SKsoV5F3Rvd",
    matcherPublicKey: "7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy",
    assetPair: {
    amountAsset: "725Yv9oceWsB4GsYwyy4A52kEwyVrL5avubkeChSnL46",
    priceAsset: null
},
    orderType: "buy",
    amount: 74,
    price: 103526336,
    timestamp: 1528814695596,
    expiration: 1528814995596,
    matcherFee: 300000,
    signature:
"5kM8NRVxu4xtDUwz7GCVqyHbeszjXheJn1f7Q5Kpa4zdkeXe8k1kNENAU1YVNXyxNjMHCwtY9
mwUkBpZWPo2CHWf",
    proofs: [
"5kM8NRVxu4xtDUwz7GCVqyHbeszjXheJn1f7Q5Kpa4zdkeXe8k1kNENAU1YVNXyxNjMHCwtY9
mwUkBpZWPo2CHWf"
]
},
    buyMatcherFee: 300000,
    timestamp: 1528814695635,
    height: 1038644
}
```


Как вы можете заметить, транзакция содержит поля `order1` (ордер типа `buy`) и `order2` (транзакция типа `sell`). Так же присутствует подпись в массиве `proofs`, которая является подписью матчера (не отправителей ордеров), размер комиссии для матчера (`sellMatcherFee`), комиссия для ноды, которая смайнит блок (`fee`).

Значения полей `matcherPublicKey` в ордерах должно совпадать с полем `senderPublicKey` для `Exchange` транзакции, что гарантирует невозможность совершения операции обмена с помощью этих ордеров другим матчером.

Формирование `Exchange` транзакции в большинстве случаев не нужно пользователям и разработчикам, поэтому не поддерживается во многих библиотеках для разных языков программирования. Другое дело - ордера, формирование которых необходимо для ботов и многих пользовательских интерфейсов. Формирование ордера с помощью `waves-tranasctions` принципиально не отличается от формирования транзакции:

```
const { order } = require('@waves/waves-transactions')

const seed =
'b716885e9ba64442b4f1263c8e2d8671e98b800c60ec4dc2a27c83e5f9002b18'

const params = {
  amount: 100000000, //1 waves
  price: 10, //for 0.00000010 BTC
  priceAsset: '8LQW8f7P5d5PZM7GtZEBgaqRPGSzS3DfPuiXrURJ4AJS',
  matcherPublicKey: '7kPFrHDiGw1rCm7LPszuECwWYL3dMf6iMifLRDJQZMzy',
  orderType: 'buy'
}


const signedOrder = order(params, seed)
```

Обратите внимание, что в отличие от примеров с транзакциями, в примере не используется функция `broadcast` для отправки транзакции, потому что `broadcast` отправляет транзакцию в ноду, а нам необходимо отправлять в матчер. Информацию про API матчера можете найти в документации waves.exchange.

Lease и Lease Cancel транзакции (type 8 и 9)

В самом начале этой книги мы немного затронули тему лизинга, который позволяет сдавать свои токены другим нодам "в аренду" для генерации блоков. Чтобы сделать это необходимо отправить транзакцию типа `Lease`.

```
const { lease } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  amount: 100,
  recipient: '3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs',
  fee: 100000
}

const signedLeaseTx = lease(params, seed)
broadcast(signedLeaseTx);
```

Как видите, транзакция предельно простая, указываем получателя в поле `recipient` в виде адреса или алиаса (про них поговорим ниже) и сумму, которую ходим отдать в лизинг. Необходимо учитывать, что участвовать в майнинге эти токены будут только спустя 1000 блоков после того, как они будут отправлены в лизинг.

Отправитель лизинга может в любой момент отменить лизинг, снова получая к ним доступ для торговли, переводов или майнинга на своем адресе. Для этого необходимо отправить транзакцию `LeaseCancel`:

```
const { cancelLease } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  leaseId: '2fYhSNrXpyKgbtHzh5tnpvnQYuL7JpBFMBthPSGFrqqg',
  senderPublicKey: '3SU7zKraQF8tQAF8Ho75MSVCBfirgaQviFXnseEw4PYg',
//optional, by default derived from seed
  timestamp: Date.now(), // optional
  fee: 100000, //minimal value
  chainId: 'W' // optional
}

const signedCancelLeaseTx = cancelLease(params, seed)
broadcast(signedCancelLeaseTx);
```

Транзакция отмены лизинга требует передавать id транзакции отправки в лизинг. Отменять можно только всю транзакцию лизинга целиком. Например, если вы отправите в лизинг 1000 Waves любой ноде одной транзакций, вы не сможете забрать часть этой сумму - отмена может быть только целиком.

Обратите так же внимание, что в данной транзакции указывается `chainId`, в то время как в транзакции отправки лизинга, такого не требуется. Попробуйте угадать почему.

Ответ прост: в транзакции отправки лизинга есть поле `recipient`, куда указывается адрес и который и так содержит `chainId` в себе, а в транзакции отмены такого поля нет, поэтому, чтобы сделать невозможным отправку одной и той же транзакции в разных сетях, приходится указывать байт сети. Но если вы используете библиотеку `waves-transactions`, то она сама подставит байт сети для Mainnet, так что не переживайте.

Другое отличие отмены лизинга от отправки в лизинг в том, что, отмена начинает действовать сразу же, как попадает в блокчейн, без ожидания 1000 блоков.

Alias транзакция (type = 10)

В Waves есть уникальная особенность, которой нет во многих других блокчейнах - наличие алиасов. Использовать адреса для совершения операций порой крайне неудобно, они длинные и их невозможно запомнить, поэтому каждый аккаунт может сделать себе алиас. Он может быть простым и легкозапоминаемым. В любой транзакции в сети Waves в поле `recipient` можно указывать не только адрес, но и алиас.

В Ethereum есть немного похожая концепция ENS, которая построена по принципам DNS, с разными уровнями (namespace) и управлением через смарт-контракты. В Waves алиасы являются частью протокола и все находятся в глобальном пространстве имен, не имея разделения на домены и поддомены. Один аккаунт может создавать неограниченное количество алиасов с помощью отправки специального типа транзакции:

```
const { alias } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  alias: 'new_alias',
  chainId: 'W'
}

const signedAliasTx = alias(params, seed)
broadcast(signedAliasTx)
```

Алиас может состоять из:

- буквы латинского алфавита в нижнем регистре
- цифры
- точка
- нижнее подчеркивание
- знак дефиса
- знак @

Алиас должен быть длиной от 4 до 30 символов. Проблема алиасов в сети Waves в том, что они все находятся в глобальном пространстве и не могут повторяться, поэтому есть аккаунтами с более чем 2000 алиасов - своеобразная форма киберсквоттинга в блокчейне.

Mass transfer транзакция (type = 11)

На заре своей истории Waves был известен как блокчейн с очень легким выпуском токенов, и закономерным желанием сообщества стало упрощение следующего шага многих кампаний по выпуску токенов - распределение токенов среди получателей. Для удовлетворения этого спроса была создана транзакция, которая позволяет отправить токены с одного адреса на множество. Есть только 2 ограничения - получаетелей может быть до 100, а отправляется им всем только 1 вид токена (нельзя сделать `MassTransfer` и отправить первой половине адресов токен `A`, а второй `B`).

```
const { massTransfer } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  transfers: [
    {
      amount: 100,
```

```
      recipient: '3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs',
    },
    {
      amount: 200,
      recipient: '3PPnqZznWJbPG2Z1Y35w8tZzskiq5AMfUXr',
    },
  ],
  //senderPublicKey: 'by default derived from seed',
  //timestamp: Date.now(),
  // fee: 100000 + transfers.length * 50000,
}

const signedMassTransferTx = massTransfer(params, seed);
broadcast(signedMassTransferTx);
```

Кроме удобства работы с такой транзакцией, по сравнению с отправкой 100 транзаций типа `Transfer`, такая транзакция получается еще и дешевле. Если минимальная комиссия для `Transfer` составляет 0.001 Waves (100000 Wavelet), то размер минимальной комиссии для `MassTransfer` вычисляется по формуле:

```
100000 + transfers.length * 50000
```

То есть, отправка 100 `Transfer` транзакций нам обойдется в 0.1 Waves, в то время как отправка одной `MassTransfer` со 100 получателями всего лишь в 0.051 Waves - почти в 2 раза дешевле.

Data транзакция (type = 12)

Особенность Waves, которая делает его крайне удобным блокчейном для работы с данными, является наличие `Data` транзакций, которые появились в апреле 2018 года и позволили записывать данные в блокчейн в очень удобном формате.

С введением `Data` транзакций, у каждого аккаунта появилось key-value хранилище, в которое можно записывать данные четырех типов: строки, числа, булевые значения и массивы байт.

Хранилище аккаунта не имеет ограничения по общему размеру данных, которое можно туда записывать, но есть ограничения на:

- размер одной транзакции записи данных в хранилище не более 140 килобайт. Комиссия за транзакцию зависит от размера транзакции и считается по формуле `100000 + bytes.length * 100000`.
- размер данных на один ключ не более 32 килобайт
- размер ключа не более 100 символов. Ключами в хранилище могут быть только строки в формате UTF-8.

Давайте посмотрим как записать данные с помощью JavaScript библиотеки:

```
const { data } = require('@waves/waves-transactions')

const seed = 'example seed phrase'
```

```
const params = {
  data: [
    { key: 'integerVal', value: 1 },
    { key: 'booleanVal', value: true },
    { key: 'stringVal', value: 'hello' },
    { key: 'binaryVal', value: [1, 2, 3, 4] },
  ],
  //senderPublicKey: 'by default derived from seed',
  //timestamp: Date.now(),
  //fee: 100000 + bytes.length * 100000
}

const signedDataTx = data(params, seed);
broadcast(signedDataTx);
```

Надо понимать, что состояние хранилища со всеми ключами и значениями может прочитать любой пользователь, более того, значение по любому ключу доступно всем смарт-контрактам в сети, будь то dapp, смарт ассет или смарт аккаунт.

Данные по ключу могут перезаписываться неограниченное количество раз, если обратное не указано в контракте аккаунта. В дальнейшем мы рассмотрим, как реализовать на аккаунте read-only пары, которые могут быть записаны только один раз и не могут быть изменены или удалены.

Многие пользователи ожидают, что у ассетов тоже есть свои key-value хранилища, однако это не так. Только аккаунт имеет такое хранилище, поэтому если вам необходимо записывать данные для использования ассетом - записывайте в аккаунт, который выпустил токен, или любой другой аккаунт, все равно можно читать любые ключи любых аккаунтов в коде вашего смарт-ассета.

Другой частый вопрос, можно ли удалить из хранилища ключ. До недавнего времени такое было невозможно, но с релизом Ride v4 это становится возможно. Чтобы сейчас не смешивать и Ride и транзакции, давайте отложим рассмотрение кода Ride до следующего раздела, но сейчас поговорим по получение данных их хранилища аккаунта. Это можно сделать с помощью REST запроса к API ноды:

1. Эндпоинт `/addresses/data/{address}?matches={regexp}` позволяет получить все данные из хранилища, при необходимости фильтрую ключи по регулярному выражению, передаваемому как параметр `matches`. Фильтрация по значениям пока не поддерживается в ноде.
2. Эндпоинт `/addresses/data/{address}/{key}` позволяет получить значение одного ключа в хранилище одного аккаунта.

В библиотеке `waves-transactions` есть дополнительные методы, которые позволяют делать это без необходимости писать самому запрос к API. Ниже пример получения всего состояния хранилища и значения по одному ключу:

```
const { nodeInteractions } = require('@waves/waves-transactions')

const address = '3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs'

const wholeStorage = await accountData(address);
const oneKeyValue = await accountDataByKey(address, "demoKey");
```

```
console.log(wholeStorage, oneKeyValue);
```

Как видите, все достаточно несложно. У API ноды Waves есть несколько особенностей, некоторые из которых хорошо бы знать до начала работы, чтобы в самый неподходящий момент не получить ошибку в рантайме. К таким особенностям работы я бы отнес следующее:

1. Нода предназначена в первую очередь для поддержания работы блокчейна, а не оптимальной работы с API, поэтому запросы всего хранилища для аккаунтов с большим количеством данных могут приводить к проблемам. Я бы никогда и никому не рекомендовал запрашивать весь стейт аккаунта никогда, потому что если вы это делаете - вы что-то делаете не так.
2. Нода возвращает результаты в JSON, но в JSON нет возможности передавать массив байт как есть, поэтому в отличие от других типов данных (строк, чисел и булевых значений), они кодируются в `base64` представление. На самом деле, при записи данных типа массив байт в блокчейн с помощью `waves-transactions` он так же конвертирует байты в `base64` строку и отправляет это, а не массив байт в виде чиел. Вот, например, как выглядит сформированная транзация для отправки в API с помощью `POST` запроса:

```
```

Будучи DevRel компании Waves я получал много вопросов относительно потенциально некотролируемого роста размера блокчейна. Многих людей, особенно у которых есть опыт работы с другими блокчейнами, смущает факт возможности записывать много данных по фиксированной и достаточно низкой цене и масштабируемость такого решения. В некоторых случаях (особенно долгосрочного хранения) блокчейн Waves может быть экономически эффективнее, чем хранение в Amazon S3, что потенциально опасно для масштабирования сети. Простого ответа на этот вопрос действительно нет, пока размер блокчейна Waves составляет порядка 40 гигабайт (не ~2.8 ТБ как в Ethereum), так что проблема не актуальна, зато простота записи позволяет делать "блокчейн для людей", о чем мы говорили в самом начале книги. Проблема станет актуальной только в случае быстрого роста популярности блокчейна Waves, но в таком случае будет расти и цена токенов, соответственно, стоимость хранилища тоже, что будет приводить к меньшему количеству желающих писать в блокчейн большие объемы данных. Там, где технология не могут полностью решить проблему, приходит на помощь экономика, что и будет происходить в случае роста популярности.

SetScript транзакция (type = 13)

Транзакции типа `SetScript` мы косвенно затрагивали, когда говорили про смарт-аккаунты. Логику поведения смарт-аккаунта и децентрализованных приложений мы описываем с помощью языка Ride, который компилируется в `base64` представление одним из доступных способов (JS бибиотека `ride-js`, API ноды, Java пакет в Maven, online IDE или плагин для Visual Studio Code) и отправляется в составе `SetScript` транзакции:

```
const { setScript } = require('@waves/waves-transactions')
```

```
const seed = 'example seed phrase'
const params = {
  script: 'AQa3b8tH', // TRUE в base64 представлении
  //senderPublicKey: 'by default derived from seed',
  //timestamp: Date.now(),
  //fee: 100000,
  //chainId: 'W'
}

const signedSetScriptTx = setScript(params, seed)
broadcast(signedSetScriptTx);
```

`SetScript` транзакция используется только для аккаунтов, чтобы сделать из них Smart Account или dApp, но не для токенов. Установка скрипта с помощью `SetScript` транзакции меняет поведение аккаунта не только с точки зрения того какие транзакции будут попадать в блокчейн, но и с точки зрения комиссии. Смарт-аккаунт платит на 0.004 Waves больше за каждый вид транзакции, по сравнению с обычным аккаунтом.

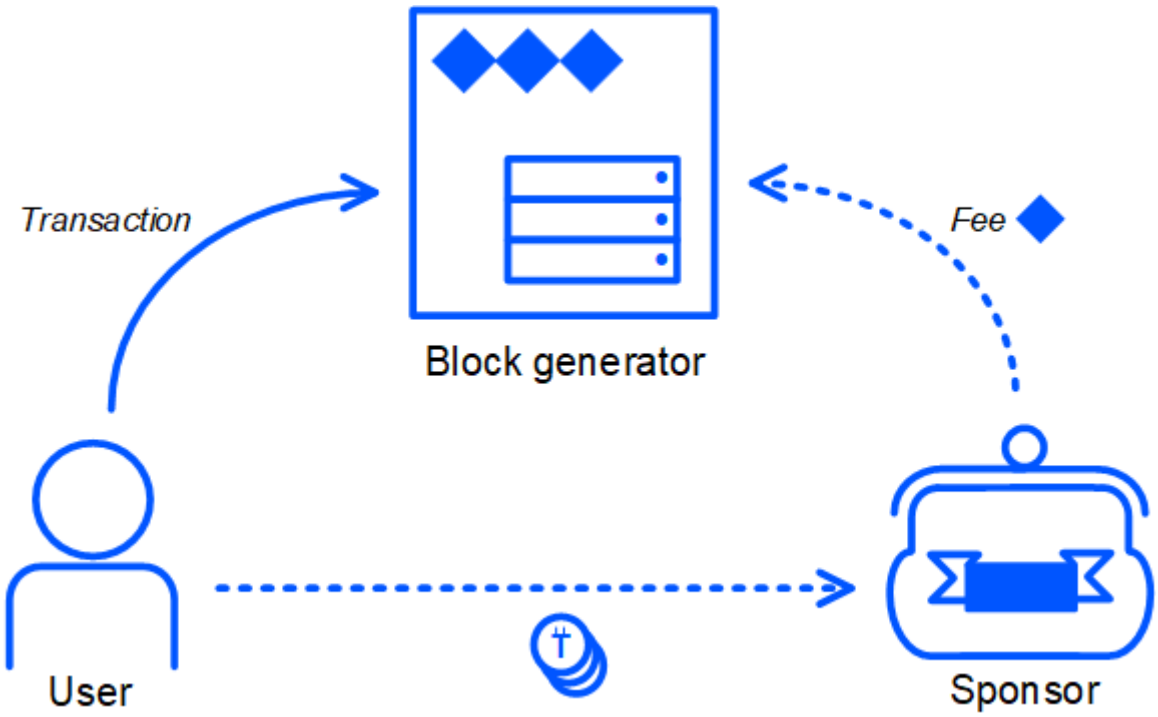
Чтобы превратить смарт-аккаунт в обычный аккаунт без скрита, необходимо отправить транзакцию `SetScript` с параметром `script` равным `null`. Но не каждый смарт-аккаунт может снова стать обычным аккаунтов. Скрипт смарт-аккаунта может прямо запрещать делать транзакцию `SetScript` или накладывать другие ограничения.

SetSponsorship транзакция (type = 14)

Особенности спонсирования транзакций и пример SetSponsorship транзакции мы рассматривали в разделе 4.2, но давайте кратко вспомнил основную суть.

Создать токена имеет возможность отправить транзакцию, которая включает спонсирование транзакций с использованием этого токена. Пользователи будут платить комиссию в токене, но так как майнеры всегда получают комиссию только в Waves, то фактически Waves будут списываться с аккаунта, выпустившего токен.

- Пользователь платит комиссию за транзакцию спонсируемым токеном (например, он отправляет 10 токенов, дополнительно платит 1 токен в виде комиссии, в итоге с его аккаунта списывается 11 токенов)
- Создатель токена получает комиссию в его токене (1 в нашем примере)
- С вашего аккаунта списываются WAVES в необходимом количестве и уходят майнерам (количество спонсируемых токенов и их соотетствие Waves настраивается в момент отправки транзакции SetSponsorship)



Отправить транзакцию включения спонсирования можно достаточно просто:

```
const { sponsorship } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {
  assetId: 'A',
  minSponsoredAssetFee: 100
}

const signedSponsorshipTx = sponsorship(params, seed)
```

Код выше сформирует (но не отправит в блокчейн) транзакцию:

```
{
  "id": "A",
  "type": 14,
  "version": 1,
  "senderPublicKey": "3SU7zKraQF8tQAF8Ho75MSVCBfirgaQviFXnseEw4PYg",
  "minSponsoredAssetFee": 100,
  "assetId": "4uK8i4ThRGbehENwa6MxyLtxAjAo1Rj9fduborGExarC",
  "fee": 100000000,
  "timestamp": 1575034734209,
  "proofs": [

"42vz3SxqxzSzNC7AdVY34fM7QvQLyJfYFv8EJmCgooAZ9Y69YDNDptMZcupYFdN7h3C1dz2z6
keKT9znbVBrikyG"
  ]
}
```

Чтобы отменить спонсирование транзакций, достаточно отправить транзакцию с полем `minSponsoredAssetFee` равным `null`.

SetAssetScript транзакция (type = 15)

Данная транзакций по своей сути похожа на `SetScript` транзакцию, за одним исключением - она позволяет менять скрипт для токена.

```
const { setAssetScript } = require('@waves/waves-transactions')
const seed = 'example seed phrase'
const params = {
  script: 'AQa3b8tH', // TRUE в base64 представлении
  assetId: '4uK8i4ThRGbehENwa6MxyLtxAjAo1Rj9fduborGExarC',
  //senderPublicKey: 'by default derived from seed',
  //timestamp: Date.now(),
  //fee: 100000,
  //chainId: 'W'
}

const signedSetAssetScriptTx = setAssetScript(params, seed)
broadcast(signedSetAssetScriptTx);
```

`SetAssetScript` возможна только для ассетов, на которых уже есть скрипт. Если вы с помощью `Issue` транзакции выпустили транзакцию, которая не имеет скрипта, то установить на нее скрипт в дальнейшем не удастся.

Установка скрипта на токен увеличивает минимальную комиссию для операций с этим токеном на 0.004 Waves (прямо как в случае со смарт-аккаунтами и децентрализованными приложениями).

Например, минимальная комиссия `Transfer` транзакции составляет 0.001, но для смарт-ассетов составляет 0.005 Waves. Если мы захотим сделать перевод смарт-ассета со смарт-аккаунта, то придется уже заплатить не менее 0.009 Waves (0.001 базовой стоимости, 0.004 прибавки за выполнение скрипта смарт-аккаунта/децентрализованного приложения и смарт-ассета).

InvokeScript транзакция (type = 16)

`InvokeScript` транзакция является одной из самых важных транзакций в сети, так как она предназначена для вызова функций в децетрализованных приложениях.

```
const { invokeScript } = require('@waves/waves-transactions')

const seed = 'example seed phrase'

const params = {

  call: {
```

```
      args: [{ type: 'integer', value: 1 }],
      args: [{ type: 'binary', value: 'base64:AAA=' }],
      args: [{ type: 'string', value: 'foo' }],
      args: [{ type: 'boolean', value: true }],
      function: 'foo',
    },
    payment: [
      {
        amount: 16,
        assetId: '73pu8pHFNpj9tmWuYjqnZ962tXzJvLGX86dxjZxGYhoK'},
      {
        amount: 10,
        assetId: null
      }
    ],
    dApp: '3Fb641A9hWy63K18KsBJwns64McmdEATgJd',
    chainId: 'W',
    fee: 500000,
    feeAssetId: '73pu8pHFNpj9tmWuYjqnZ962tXzJvLGX86dxjZxGYhoK',
    //senderPublicKey: 'by default derived from seed',
    //timestamp: Date.now(),
    //fee: 100000,
    //chainId:
  }

  const signedInvokeScriptTx = invokeScript(params, seed)
  console.log(signedInvokeScriptTx)
```

Пример выше вызовет функцию `foo` децентрализованного приложения на аккаунте с адресом `3Fb641A9hWy63K18KsBJwns64McmdEATgJd`. При вызову функции передаются 4 аргумента. Аргументы в `InvokeScript` не именованные, но их порядок должен совпадать с порядком, объявленным в коде децентрализованного приложения. `InvokeScript` позволяет так же прикрепить к вызову до 2 видов токенов в качестве платежа. В примере выше в качестве оплаты прикрепляются токен `73pu8pHFNpj9tmWuYjqnZ962tXzJvLGX86dxjZxGYhoK` и Waves (с `assetId=null`).

`InvokeScript` наряду с `Transfer` могут быть спонсированы, поэтому в примере выше вызов контракта оплачивается токеном `73pu8pHFNpj9tmWuYjqnZ962tXzJvLGX86dxjZxGYhoK`, который должен быть спонсированным.

При работе с некоторыми приложениями может возникать желание отправлять транзакции типа `InvokeScript` с большими аргументами, но сделать это не получится, так как ограничение на размер транзакции составляет 5кб. Если функции в децентрализованном приложении надо передавать аргументы, которые больше этого ограничения, то возможен следующий сценарий:

1. Отправить `Data` транзакцию (до ~140кб данных)
2. При вызове функции с помощью `InvokeScript` передавать в качестве аргумента ключи, которые были записаны с помощью `Data` транзакции.
3. В коде децентрализованного приложения читать значения по переданным ключам и их обрабатывать.

UpdateAssetInfo транзакция (type = 17) [stagenet]

Новая транзакция `UpdateAssetInfo` (type = 17) доступна только в сети Stagenet на момент написания этих строк. Она позволяет обновлять данные о выпущенном токене. В протоколе давно существует транзакция перевыпуска (`Reissue`), которая позволяет довыпустить токены и запретить перевыпуск в дальнейшем, но возможности изменить название или описание токена раньше не было. Однако смена названия и описания может понадобиться многим проектам - например, при смене названия компании, домена (если он упоминается в описании) или продаже компании/бренда. Чтобы избежать недопониманий, давайте зафиксируем отличия транзакций перевыпуска (`Reissue`) и `UpdateAssetInfo`:

- `Reissue` позволяет довыпустить токен (количество задается создателем) и поменять флаг `reissuable` (только на false), если в момент выпуска токена создатель поставил `reissuable=true`
- `UpdateAssetInfo` позволяет обновить название и описание токена, но не чаще, чем раз в 100 000 блоков.

```
const { updateAssetInfo } = require('@waves/waves-transactions')
const seed = 'example seed phrase'
const params = {
  script: 'AQa3b8tH', // TRUE в base64 представлении
  assetId: '4uK8i4ThRGbehENwa6MxyLtxAjAo1Rj9fduborGExarC',
  description: "New description",
  name: "New name"
  //senderPublicKey: 'by default derived from seed',
  //timestamp: Date.now(),
  //fee: 100000,
  //chainId: 'W'
}

const updateAssetInfoTx = updateAssetInfo(params, seed)
broadcast(updateAssetInfoTx);
```

Особенности работы с транзакциями

При формировании транзакций с использованием библиотек часто хочется указывать минимальное количество параметров, чтобы библиотека сама заполнила все остальные. Библиотека `waves-transactions` так и делает, предлагаю заполнить только самые важные поля и подставляя остальные параметры по умолчанию. Однако существуют поля, заполнять которые не обязательно, но понимать их и знать об их существовании желательно.

additionalFee

Для всех типов транзакций есть дополнительное поле `additionalFee`, который позволяет добавить дополнительную комиссию к значениям по умолчанию. Это может быть полезно в 2 случаях:

- Указать дополнительную комиссию при работе со смарт-ассетами и смарт-аккаунтами. Например, минимальная комиссия за `Transfer` транзакция по умолчанию составляет 0.001 Waves и именно это значение укажет библиотека `waves-transactions`, но в случае работы со смарт-ассетами необходимо дополнительно заплатить 0.004. Библиотека не знает, что транзакция отправляется с использованием смарт-ассета, поэтому разработчику необходимо самому предусматривать дополнительную комиссию. Конечно, можно использовать поле `fee`, чтобы указать всю комиссию целиком, но использование `additionalFee` удобнее, ведь не надо самому помнить минимальные комиссии за каждый тип транзакции.
- Отправлять транзация с повышенной комиссией для быстрого попадания в блок. Загрузка сети Waves сейчас сильно меньше пропускной способности, поэтому необходимость указывать повышенную комиссию встает крайне редко, но такая возможность существует. В следующей главе мы поговорим про сортировку транзакций в UTX (листе ожидания для попадания в блок).

В таблице ниже представлены минимальные комиссии за транзакции разных типов (при отправке с обычного аккаунта и без взаимодействия со смарт-ассетами):

Transaction type	Transaction type ID	Minimum transaction fee in WAVES
Issue transaction	3	• 1 for regular token • 0.001 for non-fungible token
Transfer transaction	4	0.001
Reissue transaction	5	0.001 – starting from node version 1.2.0
Burn transaction	6	0.001
Exchange transaction	7	0.003
Lease transaction	8	0.001
Lease cancel transaction	9	0.001
Create alias transaction	10	0.001
Mass transfer transaction	11	0.001 + 0.0005 × N N is the number of transfers inside of the transaction. The value is rounded up to the thousandths
Data transaction	12	0.001 per kilobyte
Set script transaction	13	0.01
Sponsor fee transaction	14	1
Set asset script transaction	15	1
Invoke script transaction	16	0.005 + K K is the number of assets issued as a result of dApp script invocation that are not non-fungible tokens.
Update asset info transaction	17	0.001

chainId

В примерах транзакций выше вы могли замечать поле `chainId`, которое чаще всего было указано как `W`. Каждая транзакция в сети `Waves` содержит в себе байт сети либо в прямом виде, либо опосредованно (когда в транзакциях задействован адрес получателя). Байт сети мы рассматривали, когда говорили про адреса в разделе 3.

Байт сети - уникальный идентификатор сети, который позволяет отличать адреса и транзакции в разных сетях (mainnet, testnet, stagenet). Байты сети для перечисленных выше сетей - `W`, `T`, `S` соответственно. Благодаря байту сети невозможно ошибиться и отправить токены на адрес, которого не может существовать в сети, в которой отправляется транзакция. Если бы не было байта сети, то была бы возможна атака на пользователей, которые используют одну пару приватного и публичного ключа в нескольких сетях (stagenet и mainnet, например). Злоумышленник мог бы скопировать транзакцию из сети stagenet от пользователя и отправить ее в сеть mainnet, произведя действие, которое пользователь не хотел делать в mainnet. Благодаря байту сети такое невозможно.

timestamp

У каждой транзакции есть время его создания, которое прописывается в транзакции и подписывается отправителем наряду с другими полями. `waves-transactions` по умолчанию поставит время, которое задано в операционной системе, где запускается код. В протоколе Waves ноды синхронизируют время друг с другом с помощью протокола NTP, поэтому отличие между ними составляет не больше 1 секунды. Можно сказать, что сеть Waves знает актуальное время, и актуальное время прописывается в теле блока в момент его создания нодой. Если какой-либо генератор попытается сделать блок "из прошлого" или "из будущего", то остальные генераторы и валидаторы такой блок не примут.

Что же касается времени транзакции, то оно может отличаться от времени блока не более, чем на 90 минут в прошлое и 120 в будущем. Вы можете отправить транзакцию, в которой `timestamp` будет из будущего на 120 минут и генераторы попробуют добавить ее в блок, но если отправить со временем, которое больше времени на нодах на 121 минуту, то транзакция уже будет отвергнута.

Параметр `timestamp` может использоваться для регулирования сколько максимально времени транзакция может находиться в списке ожидания на попадания в блок. Если сеть загружена, транзакции попадают в блок очень медленно и нам не хочется платить большую комиссию, но мы готовы подождать, то можно поставить `timestamp`, который на 120 минут больше времени на нодах. Такая транзакция будет валидной в течение 210 минут (3 с половиной часа) и только если она не попала в блок в течение этого времени, она будет отвергнута. Может быть и обратная ситуация, когда нам важно, чтобы транзакция могла только быстро попасть в блок или не попасть вовсе. В таком случае, установка `timestamp` на 85 минут меньше, чем актуальное время, гарантирует, что она будет валидной только 5 минут, и если в течение этих 5 минут не попала в блок, то будет вычищена из UTX и уже никогда не попадет в блок.

При использовании поля `timestamp` транзакций в коде смарт-контрактов необходимо помнить, что оно может отличаться от настоящего на [-90; +120] минут. В разделе 7 мы поговорим о том, как правильно использовать время, если оно вам все-таки надо в коде контракта.

proofs

Поле `proofs` является массивом, который предназначен для подписей транзакции. Подписей может быть до 8. На самом деле, в этом поле можно хранить не только подписи, но и использовать для передачи в качестве аргументов в смарт-аккауты или децентрализованные приложения. Это особенно может быть полезно при работе со смарт-аккаунтами, которые не могут принимать аргументы.

id

Каждая транзакция в сети имеет уникальный ID, который является хэшом на основе полей транзакции. В сети не может быть 2 одинаковых транзакций с двумя одинаковыми ID. ID транзакции вычисляется `waves-transactions` автоматически и оно может быть использовано для работы с API - для ожидания попададания в блок или проверки статуса.

version

В сети Waves есть не только много разных типов транзакций, но могут быть несколько разных версий для каждого типа. Например, для типов вроде `Transfer` или `Issue` имеют 3 версии. Важно учитывать, что JSON представление транзакций при работе с API может отличаться для разных версий одного и того же типа.

Подпись транзакций

Особенности обработки UTX

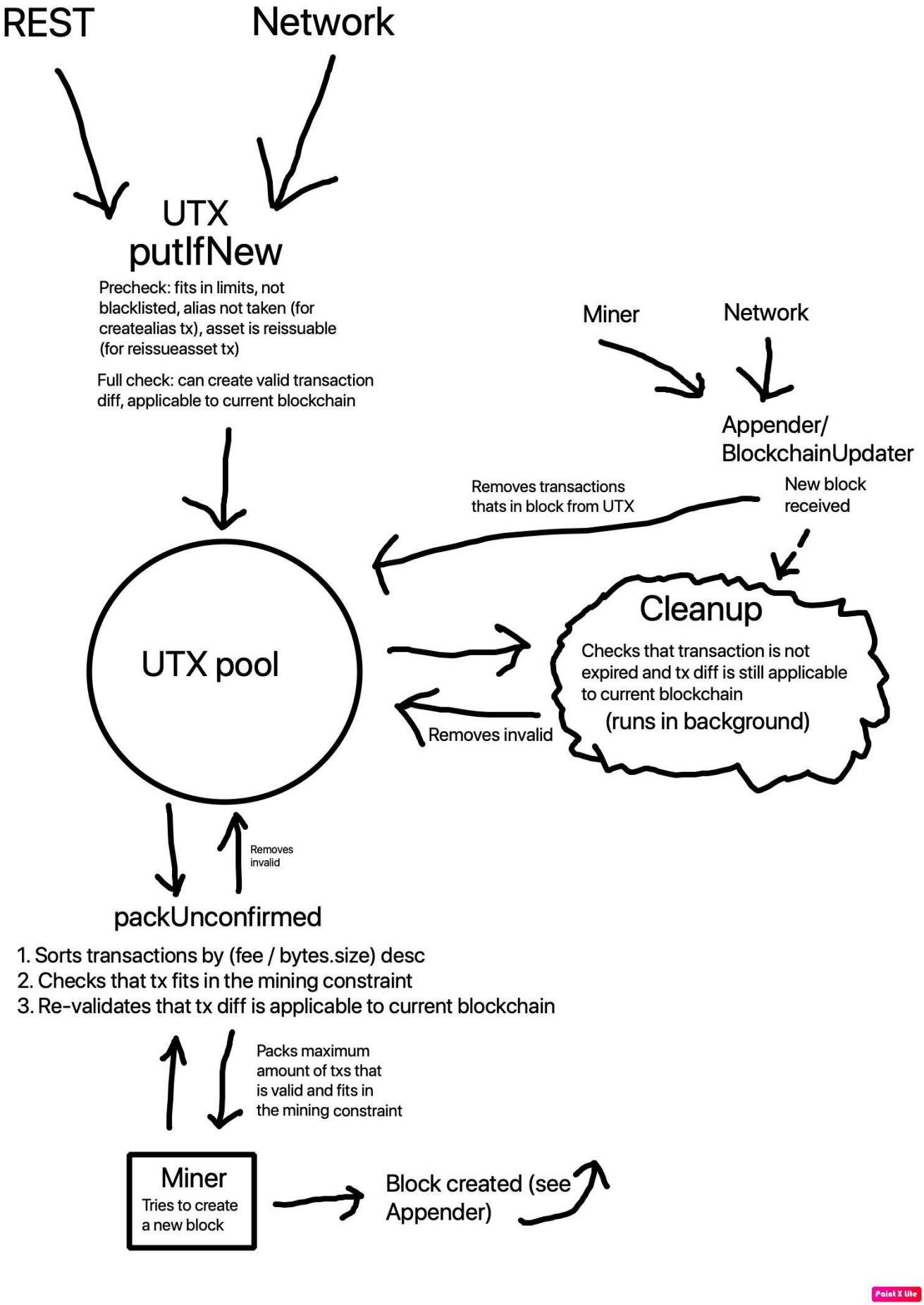
Транзакции в листе ожидания это ни разу не весело, они ведь хотят в блок! Как же определить какая транзакция должна первой попасть в блок. Можно было бы сделать простую очередь и руководствоваться принципом "Кто первый встал, того и тапки", но такой подход не является оптимальным для майнеров в сети. Им гораздо выгоднее класть в блок транзакции с большей комиссией. Но и тут не все так просто, вы ведь помните, что в Waves существует разные виды транзакций. У каждого вида своя минимальная комиссия, заданная в консенсусе, поэтому обработка в зависимости от размера комиссии тоже не приведет к ожидаемому результату. Например, отправка InvokeScript транзакций с минимальной комиссией в 0.005 Waves будет всегда попадать в блок раньше, чем транзакция Transfer с комиссией в 0.001 Waves. Что же делать?

Первое, что можно придумать, это сортировать транзакции в зависимости от стоимости на байт транзакции. Нода тратит ресурсы на валидацию подписи для транзакции, и чем больше транзакция по размеру, тем больше ресурсов на это потратится. Например, `Data` транзакция размером в 140 kb будет валидироваться в несколько десятков раз дольше, чем `Transfer` транзакция размером меньше килобайта. Давайте поговорим на примерах. Скажем, у нас есть 2 транзакции:

- Data транзакция размером в 100 kb и с комиссией в 0.01 Waves
- Transfer транзакция размером в 1kb и с комиссией в 0.001 Waves

Какая транзакция будет первой в очереди? Та, которая была получена первой, потому что в пересчете на 1 байт транзакции комиссия у этих 2 транзакций равная.

На схеме ниже показано, как именно нода обрабатывает транзакции до попадания в блок, а так же во время его нахождения там. На схеме вы можете видеть функцию `cleanup`, которая постоянно выполняется в фоне и проверяет, не стали ли транзакции, находящиеся в UTX, невалидными (истек срок жизни, баланс отправителя стал нулевым из-за другой транзакции и не может оплатить комиссию и т.д.) и нет ли необходимости их оттуда удалить.


REST
Network
UTX
putIfNew
Precheck: fits in limits, not blacklisted, alias not taken (for createalias tx), asset is reissuable (for reissueasset tx)
Full check: can create valid transaction diff, applicable to current blockchain
Miner
Network
Appender/
BlockchainUpdater
Removes transactions thats in block from UTX
New block received
Cleanup
Checks that transaction is not expired and tx diff is still applicable to current blockchain
(runs in background)
UTX pool
Removes invalid
Removes invalid
packUnconfirmed
1. Sorts transactions by (fee / bytes.size) desc
2. Checks that tx fits in the mining constraint
3. Re-validates that tx diff is applicable to current blockchain
Packs maximum amount of txs that is valid and fits in the mining constraint
Miner
Tries to create a new block
Block created (see Appender)

Введение

RIDE - это компилируемый статически типизированный функциональный язык программирования с ленивым исполнением, предназначенный для построения децентрализованных приложений. Язык создавался с целью помочь разработчикам писать код без ошибок.

RIDE не является Тьюринг-полным языком. В нем нет циклов (в классическом понимании), рекурсий и предусмотрено много ограничений, которые позволяют делать приложения простыми для понимания и отладки.

История создания

Основным идейным вдохновителем создания Ride является Илья Смагин. Язык изначально был нацелен на реализацию простых кейсов вроде мульти-подписи и впервые появился в Mainnet летом 2018, позволяя писать скрипты для создания смарт-аккаунтов. В январе 2019 года на Ride стало возможным писать скрипты для смарт-ассетов, а в июле 2019 уже появилась возможность писать полноценные децентрализованные приложения благодаря релизу под названием Ride4dapps.

Философия языка и его создателей сводится к нескольким простым правилам:

1. Язык программирования - это инструмент для реализации конкретных кейсов. Усложнение языка и добавление новых конструкций делаются только если есть кейсы и бизнесы, которым этого не хватает.
2. Тьюринг-полнота вместе с масштабируемостью мало достижимы в рамках публичной блокчейн системы, поэтому язык должен быть удобен и без этой характеристики.
3. Ride - специфичный язык для написания децентрализованных приложений, а не язык общего назначения, поэтому в языке есть специфичные конструкции.

Язык испытал большое влияние Scala и F#. Нельзя сказать, что Ride очень сильно похож на какой-то из этих языков, но людям, которые знают эти языки, будет проще начать писать на Ride. В то же время, язык прост и для разработчиков на всех других языках, первичное ознакомление с языком и инструментами занимает обычное около часа. К концу этой главы вы изучите почти весь язык и все основные конструкции.

Несмотря на то, что Ride прост, он дает много возможностей разработчику. Давайте перейдем к синтаксису языка.

Hello world

```
func say() = {
  "Hello world!"
}
```

Функции объявляются с помощью `func`. Функции возвращают типы, но их не нужно объявлять, за вас это сделает компилятор. В примере выше, функция `say` вернет строку `"Hello World!"`. В языке нет `return`, потому что RIDE основан на выражениях (все является выражением), а последний оператор является результатом функции.

Блокчейн

RIDE предназначен для использования внутри блокчейна, и нет никакого способа получить доступ к файловой системе.

Функции RIDE могут считывать данные из блокчейна и возвращать транзакции в результате, которые будут записаны в блокчейн.

Комментарии

```
# Это комментарий

# В языке нельзя создавать многострочные комментарии

"Hello world!" # Можно писать комментарии и в таких местах
```

Директивы

Каждый скрипт на Ride должен начинаться с директив для компилятора. Существует 3 возможных типа директив с различными возможными значениями.

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}
```

`STDLIB_VERSION` задает версию стандартной библиотеки. Последняя версия в mainnet - 3, в stagenet - 4.

`CONTENT_TYPE` задает тип файла, над которым вы работаете. На данный момент существуют типы `DAPP` и `EXPRESSION`. Тип `DAPP` позволяет объявлять функции, завершать выполнение скрипта некоторыми транзакциями (изменениями в блокчейне) и использовать аннотации, тогда как `EXPRESSION` позволяет завершать выполнение скрипта логическим выражением (`true` или `false`).

`SCRIPT_TYPE` задает тип сущности, к которой мы хотим добавить скрипт и изменить поведение по умолчанию. Скрипты на Ride можно прикрепить к `ACCOUNT` или `ASSET`.

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ASSET #-} # тип dApp нельзя использовать для ASSET
```

Не все комбинации директив являются правильными. Пример выше не будет работать, ибо тип контента `DAPP` допустим только для аккаунтов, в то время как тип `EXPRESSION` доступен для токенов (ассетов) и аккаунтов.

Функции

```
func greet(name: String) = {
  "Hello, " + name
}

func add(a: Int, b: Int) = {
  a + b
}
```

Тип должен следовать за именем аргумента.

Как и во многих других языках, функции не могут быть перегружены. Это помогает сохранить код читаемым и простым для изменений.

Функции можно использовать только после их объявления.

Переменные

```
let a = "Bob"
let b = 1
```

Переменные объявляются и инициализируются с помощью ключевого слова `let`. Это единственный способ объявить переменные в RIDE. Все переменные в RIDE являются иммутабельными. Это означает, что вы не можете изменить значение переменной после объявления.

Тип переменной определеляется, а если быть точнее, то выводится (type inference) исходя из значения в правой части.

RIDE позволяет определять переменные внутри любой функции или в глобальной области видимости.

```
a = "Alice"
```

Приведенный выше код не будет компилироваться, потому что переменная `a` не определена. Все переменные в Ride нужно объявлять с помощью ключевого слова `let`.

```
func lazyIsGood() = {
  let a = "Bob"
  true
}
```

Функция выше будет скомпилирована и вернет `true` в качестве результата, но переменная `a` не будет инициализирована, потому что RIDE ленивый, это означает, что все неиспользуемые переменные не вычисляются.

```

```

```
func callable() = {
  42
}

func caller() = {
  let a = callable()
  true
}
```

Функция `callable` также не будет вызвана, поскольку переменная `a` не используется.

В отличие от большинства языков, переиспользование переменных не допускается. Объявление переменной с именем, которое уже используется в родительской области видимости, приведет к ошибке компиляции.

Базовые типы

Основные базовые типы показаны ниже:

```
Boolean    #   true
String     #   "Hey"
Int        #   1610
ByteVector #   base58'...', base64'...', base16'...',
fromBase58String("...") etc.
```

Строки

```
let name = "Bob"
name + " is cool!" # строки будут соеденены, ибо используется знак +

name.indexOf("o")  # 1
```

В RIDE строка - это массив байт, доступный только для чтения. Строковые данные кодируются с помощью UTF-8.

Для обозначения строк можно использовать только двойные кавычки. Строки неизменяемы, как и все другие типы. Это означает, что функция поиска подстроки в строке очень эффективна: копирование не выполняется, дополнительные выделения не требуются.

Все операторы в RIDE должны иметь значения одного и того же типа с обеих сторон. Код ниже не будет компилироваться, потому что `age` имеет тип `Int`, а `"Bob is "` является строкой:

```
let age = 21
"Bob is " + age # не будет компилироваться
```

Чтобы заставить код работать, мы должны преобразовать `age` в `String`:

```
let age = 21
"Alice is " + age.toString() # вот так работает!
```

Специальные типы

```
List    # [16, 10, "hello"]
Nothing #
Unit    # unit
```

RIDE имеет несколько типов, которые "**выглядят** как ~~утки~~ в Scala, плавают как ~~утки~~ в Scala и крякают как ~~утки~~ в Scala". Например, типы `Nothing` и `Unit`.

В RIDE нет типа `null`, как во многих других языках. Обычно, встроенные функции возвращают тип `Unit` вместо `null`.

```
"String".indexOf("substring") == unit # true
```

Списки

```
let list = [16, 10, 1997, "birthday"]       # коллекция может содержать
различные типы данных
let second = list[1]                        # 10 - второе значение из
списка
```

Для правильной работы со списками в RIDE, у них всегда должен быть известен размер, потому что нет циклов и рекурсий.

`List` не имеет полей, но в стандартной библиотеке есть функции, которые позволяют работать с ними проще.

```
let list = [16, 10, 1997, "birthday"]

let last = list.getElement(list.size() - 1) # "birthday", постфиксный
вызов функции size()

let lastAgain = getElement(collection, size(collection) - 1) # то же, что
и выше
```

Функция `.size()` возвращает длину списка. Обратите внимание, что это значение только для чтения, и оно не может быть изменено.

```
let initList = [16, 10]            # init value
let newList = cons(1997, initList) # создает новый список с элементами
initLinit и новым значением – [1997, 16, 10]
```

Вы можете добавить новый элемент к существующему списку с помощью функции `cons`. Изначальный список не будет изменен, `cons` вернет новый список. Нет никакого способа объединить два списка или добавить несколько значений в список.

Union-типы

```
let valueFromBlockchain = getString("3PHHD7dsVqBFnZfUuDPLwbayJiQudQJ9Ngf",
"someKey") # Union(String | Unit)
```

Типы Union - это очень удобный способ работы с абстракциями, `Union(String | Unit)` показывает, что значение является пересечением этих типов.

Если бы в Ride были пользовательские типы, то можно было бы разобрать следующий пример:

```
type Human : { firstName: String, lastName: String, age: Int}
type Cat : {name: String, age: Int }
```

`Unioin(Human | Cat)` является объектом с одним полем `age`. Обычно `Union` возвращается в результате вызовов функций, когда в зависимости от параметров рантайм языка мог получить различные типы.

```
Human | Cat => { age: Int }
```

Мы можем использовать pattern matching, чтобы выяснить настоящий тип:

```
let t = ... # Union(Cat | Human)
let age = t.age                    # OK
let name = t.name                    # Compiler error
let name = match t {             # OK
  case h: Human => h.firstName
  case c: Cat   => c.name
}
```

Например, функция `getString` для чтения строк из хранища аккаунта возвращает `Union(String | Unit)` потому что некоторые ключи (и их значения соответсвенно) могут не существовать.

```
let valueFromBlockchain = getString("3PHHD7dsVqBFnZfUuDPLwbayJiQudQJ9Ngf",
"someKey")
let realStringValue = valueFromBlockchain.extract()

# or
let realStringValue2 = getStringValue(this, "someKey")
```

Чтобы получить реальный тип и значение от Union, можно использовать не только pattern matching, но и функцию `extract`, которая прервет скрипт в случае значения `Unit`. Другой вариант заключается в использовании специализированных функций, таких как `getStringValue`, `getIntegerValue` и др., чье поведение будет идентичным (ошибка если значения нет в хранилище или по заданному ключу хранится другой тип данных).

If

```
let amount = 1610
if (amount > 42) then "I claim that amount is bigger than 42"
else if (amount > 100500) then "Too big!"
else "I claim something else"
```

`if` операторы довольно просты и похожи на большинство других языков, за исключением двух отличий: `if` может использоваться как выражение (результат присваивается переменной) и ветвь `else` всегда обязательна.

```
let a = 16
let result = if (a > 0) then a / 10 else 0 #
```

Pattern matching

```
let readOrInit = match getInteger(this, "someKey") {
    case a:Int => a
    case _ => 0
}
```

Pattern matching - это механизм проверки значения по образцу. RIDE позволяет использовать pattern matching только для предопределенных типов.

Pattern matching в RIDE выглядит так же, как в Scala, но единственным вариантом использования сейчас является получение реального типа от переменной с типом `Union`. Pattern matching может быть полезен, когда в результате вызова переменной мы можем получить значение с типом `Union(Int | Unit)` или даже бывает такое `Union(Order | ReissueTransaction | BurnTransaction | MassTransferTransaction | ExchangeTransaction | TransferTransaction |`

`SetAssetScriptTransaction | InvokeScriptTransaction | IssueTransaction | LeaseTransaction | LeaseCancelTransaction | CreateAliasTransaction | SetScriptTransaction | SponsorFeeTransaction | DataTransaction)`.

```
let amount = match tx { # tx - текущий объект исходящей транзакции в
глобальной области видимости для смарт аккаунта
  case t: TransferTransaction => t.amount
  case m: MassTransferTransaction => m.totalAmount
  case i: InvokeScriptTransaction => if (i.payment)
i.payment.extract().amount else 0
  case _ => 0
}
```

Приведенный выше код показывает пример использования pattern matching, когда мы хотим получить количество передаваемых токенов в текущей транзакции от заданного аккаунта. В зависимости от типа транзакции, реальное количество передаваемых токенов может храниться в разных полях. Если транзакция типа `TransferTransaction`, `MassTransferTransaction` или `InvokeScriptTransaction`, мы возьмем правильное поле, во всех остальных случаях мы получим 0.

Чистые функции (pure functions)

Функции RIDE являются чистыми (pure) по умолчанию, что означает, что их возвращаемые значения определяются только их аргументами, и их оценка не имеет побочных эффектов. Честно говоря, относительно "чистоты" функций в Ride было огромное количество споров среди самих разработчиков Ride. Дело в том, что в Ride есть одна переменная в глобальной области видимости - `height`, которая хранит текущую высоту блокчейна (номер текущего блока, в который попадает данная транзакция). В теории, результат функции зависит не только ее параметров, но и от окружения (этой переменной `height`), поэтому некоторые скажут, что функции "не полностью чистые" или даже "не чистые совсем".

В любом случае, RIDE не является чистым функциональным языком, поскольку также существует функция `throw ()`, которая завершает выполнение скрипта в любой момент. То есть функция может не завершиться вовсе, а не просто завершиться с ошибкой, поэтому все-таки полностью функциональным язык назвать не получится.

```
let a = getInteger(this, "key").extract()
throw("I will terminate it!")
let result = if a < 0 then
    "a is negative"
else
    "a is positive or 0"
```

В приведенном выше примере скрипт завершится на строке 2 с сообщением `I will terminate it!` и никогда не достигнет выражения `if`.

Аннотации / модификаторы доступа

Функции могут быть определены только в скрипте с помощью `{-# CONTENT_TYPE DAPP #-}`. Они могут быть без аннотаций, либо с аннотациями `@Callable` или `@Verifier`.

```
func getPayment(i: Invocation) = {
  let pmt = extract(i.payment)
  if (isDefined(pmt.assetId)) then
    throw("This function accetps waves tokens only")
  else
    pmt.amount
}

@Callable(i)
func pay() = {
  let amount = getPayment(i)
  WriteSet([DataEntry(i.caller.bytes, amount)])
}
```

Функции с аннотацией `@Callable` могут быть вызваны извне блокчейна. Для вызова вызываемых функций необходимо отправить `InvokeScript` транзакцию.

Аннотации могут "привязывать" некоторые значения к функции. В приведенном выше примере переменная `i` была привязана к функции `pay` и хранила всю информацию о факте вызова (публичный ключ, адрес, платеж, прикрепленный к транзакции, комиссию, id транзакции и т.д.).

Функции без аннотаций **не** могут быть вызваны "извне", только сам скрипт может их вызывать. То есть `Callable` или `Verifier` функция начитают выполнение, в ходе которого могут вызвать функцию без аннотации.

```
@Verifier(tx)
func verifier() = {
  match tx {
    case m: TransferTransaction => tx.amount <= 100 # можно отправить не
больше 100 токенов
    case _ => false
  }
}
```

Функции с аннотацией `@Verifier` устанавливают правила для исходящих транзакций децентрализованного приложения (dApp) или смарт-аккаунта. Функции верификатора нельзя вызывать извне, но они выполняются каждый раз, когда предпринимается попытка отправить транзакцию с этого аккаунта.

Функции верификатора должны всегда возвращать значение `Boolean` в качестве результата, в зависимости от этого транзакция попадет в блокчейн или нет.

Функция верификатора использует переменную `tx`, которая является объектом со всеми полями текущей исходящей (и проверяемой) транзакции.

В одном скрипте может быть определена только одна функция-верификатор с аннотацией `@Verifier`.

```
@Callable(i)
func callMeMaybe() = {
  let randomValue = getRandomValue()
  [IntegerEntry("key", randomValue)]
}

func getRandomValue() = {
  16101997 # достаточно рандомное число
}
```

Этот код не будет компилироваться, потому что функции **без** аннотаций должны быть определены **перед** функциями с аннотациями.

Предопределенные структуры данных

RIDE имеет много предопределенных специфических структур данных для Waves Blockchain, таких как: `Address`, `Alias`, `IntegerEngry`, `StringEntry`, `Invocation`, `ScriptTransfer`, `AssetInfo`, `BlockInfo` и тд.

```
let keyValuePair = StringEntry("someKey", "someStringValue")
```

`StringEntry` это структура данных, которая описывает пару ключ-значение, как в хранилище аккаунта, где значением является строка.

```
let ScriptTransfer = ScriptTransfer("3P23fi1qfVw6RVDn4CH2a5nNouEtWNQ4THs",
amount, unit)
```

Все встроенные структуры данных могут быть использованы для проверки типов и pattern matching.

Результат выполнения

```
@Verifier(tx)
func verifier() = {
  "Returning some string"
}
```

Expression-скрипты (с директивой `{-# CONTENT_TYPE EXPRESSION #-}`) наряду с функциями `@Verifier` должны всегда возвращать логическое выражение. В зависимости от этого значения транзакция будет принята (в случае `true`) или отклонена (в случае `false`) блокчейном.

```
@Callable(i)
func giveAway(age: Int) = {

  let callerAddress = i.caller
  let reissuable = false
  let assetScript = Unit
  let decimals = 0
  let amount = 100
  let nonce = 0
  let newAsset = Issue(assetScript, decimals, "Description here",
reissuable, "MyCoolToken", amount, nonce)


  [
    ScriptTransfer(callerAddress, age, unit),
    IntegerEntry("ageof_" + callerAddress.toBase58String(), age),
    BooleanEntry("booleanKey", true),
    StringEntry("stringKey", "somevalue"),
    BinaryEntry("binaryKey", base58'3P'),
    DeleteEntry("ScriptTransfer(i.caller, 100,
newAsset.calculateAssetId()),"),
    newAsset,
    ScriptTransfer(i.caller, 100, newAsset.calculateAssetId()),
    Reissue(base58'81hNyHLFU7Z7PRUeKAfGVPca5CMmFWTxLByHcNAS8i9W',
reissuable, amount),
    Burn(base58'81hNyHLFU7Z7PRUeKAfGVPca5CMmFWTxLByHcNAS8i9W', amount)
  ]
}
```

Каждый, кто вызовет функцию `giveAway` получит столько Waves, сколько ему лет (количество лет сам пользователь передает в виде аргумента), и dApp будет хранить информацию о факте передачи в своем стейте, кроме этого скрипт запишет в хранилище данного децентрализованного приложения несколько пар ключ-значений (буловое значение, массив байт, строку) и удалит целиком пару с ключом `deleteKey`. Скрипт так же выпустит новый токен с названием `MyCoolToken`, отправит 100 таких токенов вызывающему аккаунту, довыпустит 100 токенов с `assetId` равным `81hNyHLFU7Z7PRUeKAfGVPca5CMmFWTxLByHcNAS8i9W` и сожжет 100 токенов с таким же `assetId`.

`@Callable` функции могут заканчиваться 5 типами изменений блокчейна:

1. Изменение состояния key-value хранилища, в том числе добавление и удаление пар
2. Выпуск токенов
3. Перевыпуск токенов
4. Передача токенов
5. Сжигание токенов

Каждый результирующий Issue увеличивает комиссию за вызов такой функции на 1 Waves (если выпускается не NFT токен).

Результирующий массив может содержать до 100 изменений данных в хранилище и 10 операций с токенами (выпуск, сожжение, отправка, перевыпуск).

Исключения

```
throw("Here is exception text")
```

`throw` функция немедленно завершит выполнение скрипта с предоставленным текстом. Нет никаких способов поймать брошенные исключения.

Основная идея `throw` заключается в том, чтобы остановить выполнение и отправить пользователю информативную обратную связь.

```
let a = 12
if (a != 100) then throw ("a is not 100, actual value is " + a.toString())
else throw("A is 100")
```

`throw` функция может быть использована для отладки кода при разработке dApps, так как дебаггера для Ride пока не существует.

Контекст выполнения

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ACCOUNT #-}

let a = this # Адрес текущего аккаунта
a == Address(base58'3P9DEDP5VbyXQyKtXDUt2crRPn5B7gs6ujc') # true если
скрипт выполняется на аккаунте с определенным адресом
```

RIDE скрипты в блокчейне waves могут быть привязаны к аккаунтам и токенам (директивой `{-# SCRIPT_TYPE ACCOUNT | ASSET #-}`), и в зависимости от `SCRIPT_TYPE` ключевое слово `this` может ссылаться на различные сущности. Для типа скрипта `ACCOUNT`, `this` это `Address`

Для типа `ASSET`, `this` это тип `AssetInfo`

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ASSET #-}

let a = this # AssetInfo для текущего токена, с которым совершается
операция
a == assetInfo(base58'5fmWxmtrqiMp7pQjkCZG96KhctFHm9rJkMbq2QbveAHR') #
true если скрипт выполняется для ассета с указанным ID
```

Макрос FOLD

Отсутствие Тьюринг полноты (об этом мы поговорим подробнее чуть позже) не позволяет в Ride иметь полноценные циклы, однако в языке есть макрос `FOLD`, который позволяет выполнять указанную функцию несколько раз и "собрать" результат в одну переменную.

```
func sum(acc:Int, el:Int) = acc + el
let arr = [1,2,3,4,5]
let sum = FOLD<5>(arr, 0, sum) # result: 15
```

Параметр в угловых скобках (5 в примере выше) задает сколько максимум раз будет вызвана функция `sum`. Каждый новый вызов будет передавать в качестве аргумента следующий элемент массива `arr`. Второй параметр макроса `FOLD` задает начальное значение. Функция `sum` принимает 2 аргумента:

- `acc` - сумма с предыдущего шага
- `el` - следующий элемент массива

`sum` будет вызываться со следующими параметрами:

```
sum(0, 1)  # 1
sum(1, 2)  # 3
sum(3, 3)  # 6
sum(6, 4)  # 10
sum(10, 5) # 15
```

`FOLD<N>` является макросом, то есть синтаксическим сахаром. Интерпретатор Ride ничего не знает про `FOLD`, потому что момент компиляции `FOLD` превращается в подобный код:

```
let restult = {
   let size = arr.size()
   if(size == 0) then acc0 else {
      let acc1 = function(acc0, arr[0])
      if(size == 1) then acc1 else {
         let acc2 = function(acc1, arr[1])
         if(size == 2) then acc2 else {
            let acc3 = function(acc2, arr[2])
            if(size == 3) then acc3 else {
               let acc4 = function(acc3, arr[3])
               if(size == 4) then acc4 else {
                  let acc5 = function(acc4, arr[4])
                  if(size == 5)
                     then acc5
                     else
                       throw("Too big array, max 5 elements")
}}}}}
```

Выглядит намного хуже, чем `FOLD<N>`. Параметр `N` всегда должен являться целым числом выше 0 и является обязательным. То есть, разработчик должен знать максимальный размер списка, который будет обрабатываться с помощью `FOLD`.

Если в `FOLD<N>` передать массив размерностью больше, чем `N`, то будет выброшено исключение.

Не все операции, возможные с другими циклами, можно реализовать с помощью `FOLD`.

Процесс разработки

Разработка любого приложения начинается с идеи, и децентрализованные приложения тут не исключения, однако, когда дело доходит до кода, хорошо бы иметь четкую последовательность шагов, как реализованную в коде идею запустить и дать на суд ощественности. В случае с децентрализованными приложениями на Ride, полный жизненный цикл выглядит следующим образом:

1. Написанный код на Ride компилируется в base64 представление. Существует 2 компилятора для Ride - на JavaScript и на Scala. Скомпилировать код можно с помощью разных инструментов - онлайн IDE, REST API ноды, библиотеки `surfboard` или `ride-js`, расширение для Visual Studio Code. Об этих инструментах мы поговорим чуть позже в этом разделе.
2. Скомпилированный скрипт отправляется в блокчейн в составе транзакции - `SetAssetScript` или `Issue` для смарт-ассетов, `SetScript` для смарт-аккаунтов и децентрализованных приложений или. Все эти транзакции имеют поле `script`, который принимает скомпилированный код в `base64` представлении.
3. После попадания транзакции со скриптом в блок, поведение аккаунта или ассета меняется в соответствии с тем, что написано в коде.
4. В случае со смарт-ассетами, смарт-аккаунтами и функциями `@Verifier`, код контракта будет исполняться каждый раз, когда аккаунт со скриптом отправляет транзакцию или совершается транзакция с токеном со скриптом.
5. В случае с `@Callable` функциями, их выполнение начинается в тот момент, когда любой пользователь вызывает функцию с помощью `InvokeScript` транзакции.
6. Обновление скрипта на новый возможно и для токенов и для аккаунтов в том случае, когда это не запрещено кодом установленного скрипта.

Особенности смарт-контрактов в Waves

Смарт-контракты и децентрализованные приложения в Waves отличаются от таковых в Ethereum и многих других блокчейнах. Давайте рассмотрим основные отличия и их причины.

Подсчет сложности

Все функции и операции в Ride, в том числе операции сложения, вычитания, деления, ветвления, а так же функции стандартной библиотеки имеют сложность. Сложность каждой операции выражается в условных единицах (назовем `complexity`, иначе придется называть `попуги`). Например, операция сложения имеет complexity 1, а функция проверки подписи `sigVerify()` имеет complexity 200.

Так как в каждом скрипте есть много вариантов исполнения из-за ветвлений, то `complexity` скрипта считается как сложность самой сложной ветки. Если вы используете, например, online IDE, то он будет показывать сложность скрипта в режиме реального времени.

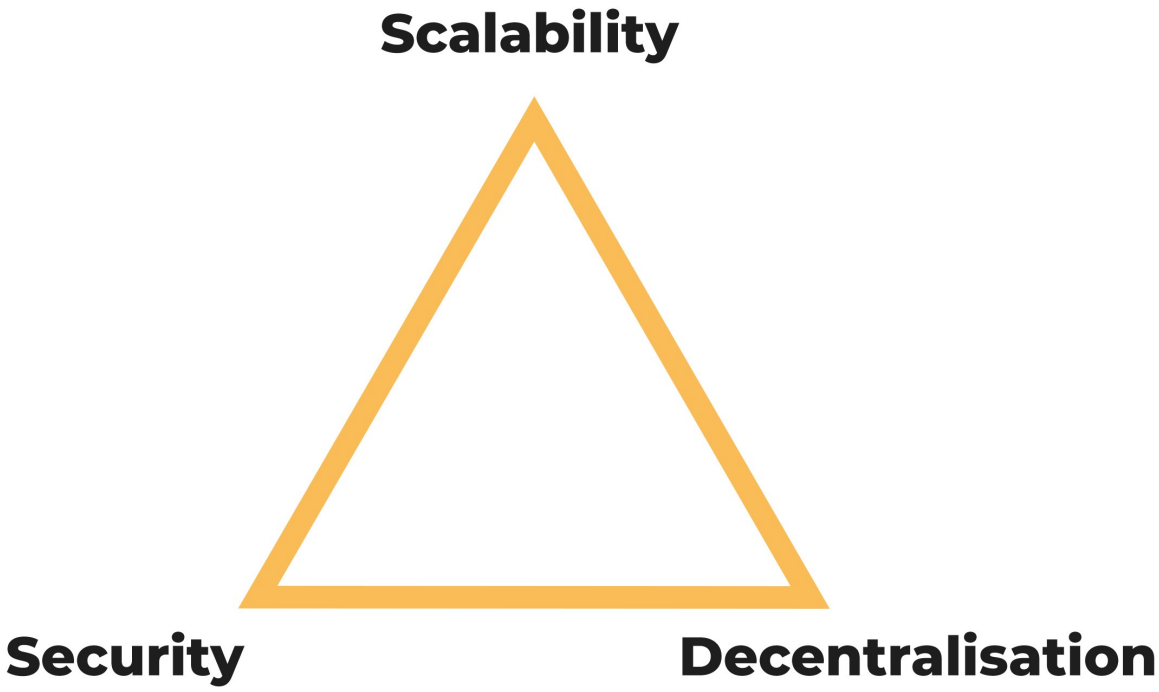
В Ride есть ограничение на максимальную сложность скрипта, и она разная для разных типов функций. Для функций `@Verifier`, смарт-аккаунтов и смарт-ассетов максимальная сложность скрипта составляет 3000 единиц, а для функций `@Callable` наиболее сложная ветка может иметь 4000 единиц. В отличие от других языков смарт-контрактов, например, Solidity в Ethereum, сложность скрипта всегда известна заранее, так как отсутствует Тьюринг-полнота. В случае Ethereum, довольно часто бывает, что мы используем цикл в коде, но не знаем сколько итераций будет в этом цикле в момент исполнения (код может читать коллекцию произвольной длины и итерировать по ней). Другой возможный в Ethereum сценарий - использование рекурсии. В Ride и Waves такое невозможно, так как отсутсвуют полноценные циклы - макрос `FOLD` заранее ограничивает максимальное количество исполнений, а рекурсий как таковых просто нет.

Заранее известная сложность избавляет от такой проблемы в Ethereum как `Out of gas`. Все, кто писал смарт-контракты и делал децентрализованные приложения на Solidity сталкивались с такой ситуацией, когда транзакция стала невалидной из-за того, что "закончился газ". В Waves такая ситуация попросту невозможна.

Кроме ограничения по максимальной сложности контракта, так же есть ограничение на максимальный размер контракта, на момент написания оно составляет 32 кб. То есть код децентрализованного приложения не может быть больше 32 кб.

Отсутствие Тьюринг полноты

Ride является *не* Тьюринг полным языком, но не потому, что сделать Тьюринг полноту сложно или долго, а потому что у такого подхода есть свои плюсы. Блокчейн является не самой высокопроизводительной системой, ведь все транзакции выполняется на каждой ноде, а на сетевые коммуникации уходит большое количество ресурсов. Есть различные подходы к масштабированию, например, шардинг, создание сайдчейнов и тд, но все они являются копромиссами - при увеличении пропускной способности всегда страдает уровень децентрализации или безопасность. Именно это утверждает блокчейн трилемма. Из 3 характеристик блокчейн - децентрализации, скорости и безопасности, полностью обеспечить можно только 2. Или другими словами, необходимо выбирать одну грань у треугольника:



Как вы возможно помните, в ценностях Waves всегда быть максимально дружелюбной платформой для разработчиков и пользователей, поэтому скорость работы не должна быть узким местом, но в то же время блокчейн Waves не позволит делать десятки тысяч транзакций в секунду, так как блокчейн должен оставаться безопасным и децентрализованным.

Отсутствие Тьюринг-полноты позволяет Waves предлагать оптимальное сочетание этих 3 характеристик:

1. Из-за отсутствия сложных скриптов, нода Waves может быть запущена на виртуальной машине за $40 в любом публичном облаке, что способствует децентрализации
2. Простота скриптов так же позволяет блокчейну иметь достаточную пропускную способность, чтобы даже при среднесуточном количестве транзакций в 100 000, не было конкуренции за попадание в блок и, соответственно, высоких комиссий.
3. Отсутствие Тьюринг-полноты делает смарт-контракты безопаснее. Ride является в какой-то мере DSL (domain specific language) или предметно-ориентированным языком, а не языком общего назначения, и именно DSL применяются в сферах, где требуется максимальная безопасность. Подробнее про это я рассказывал на одной из конференций в Сан-Франциско, с записью выступления вы можете ознакомиться здесь.

Таким образом, отсутствие Тьюринг-полноты несет в себе массу преимуществ, однако, это влияет на опыт разработки, давайте рассмотрим каким именно образом.

Последствия отсутствия Тьюринг полноты

Отсуствие Тьюринг полноты иногда не позволяет реализовать весь необходимый функционал в рамках одной функции, поэтому часто в Waves приходится разбивать логику децентрализованного приложения на несколько функций и последовательно вызывать их с помощью нескольких `InvokeScript`

транзакций. Например, одно из самых сложных приложений в сети Waves - стейблкоин Neutrino состоит из 5 контрактов.

Контракты не могут напрямую вызывать друг друга (как в Ethereum), но они могут общаться друг с другом благодаря сохранению данных и промежуточных состояний в key-value хранилища. Любой контракт может читать хранилище любого другого контракта или аккаунта, поэтому логика обработки сложных вычислений часто представляет из себя следующее:

1. Функция 1 децентрализованного приложения A вызывается с помощью `InvokeScript` транзакции, результат выполнения записывается в хранилище аккаунта A.
2. Функция 1 децентрализованного приложения B, вызванная с помощью `InvokeScript` транзакции, читает данные, записыванные в хранилище приложения A и использует для вычисления своего результата.

Возможность чтения состояния хранилища другого аккаунта в Waves является мощнейшим инструментом, позволяющим композировать логику, строить приложения, которые опираются на другие, уже существующие.

Особенности обработки UTX

В разделе 5 мы разбирали как именно происходит сортировка транзакций в UTX пуле, однако в тот момент мы опустили некоторые детали. Сейчас, когда вы знакомы с концепцией сложности скрипта, давайте разберемся во всех деталях.

Как мы уже говорили, сортировка транзакций в очереди на попадание в блок происходит по размеру комисси на 1 байт транзакции, однако есть и второй параметр, который необходимо учитывать - сложность исполнения скрипта. Задача майнера в том, чтобы максимизировать прибыль, получаемую с комиссий, поэтому майнеру может быть не выгодно валидировать транзакции со скриптом и тратить на них драгоценное время, когда можно положить в блок много транзакций без скрипта, просто проверив подпись. Поэтому, в ноде сортировка транзакций делается с помощью

В блокчейне Waves есть несколько параметров, которые ограничивают размеры блока, то есть, косвенно ограничивают максимальную пропускную способность:

- до 1 мегабайта транзакций в блоке (около 6000 транзакций)
- ограничение на максимальную суммарную сложность скриптов в блоке равна 1 000 000 (не более 250 транзакций вызова скрипта с максимальной сложностью). При достижении этого лимита в блок будут укладываться только транзакции, не связанные с исполнением скриптов, и ровно до тех пор, пока не будет достигнут лимит по размеру в 1 мегабайт.

Важно понимать, что эти параметры могут быть пересмотрены в будущем, если это будет необходимо для обслуживания всех пользователей. Однако это приведет к возрастанию системных требований к нодам.

Стандартная библиотека Ride

Стандартная библиотека Ride включается себя относительно небольшой набор функций, многие из которых связаны с криптографией (хэш функции, функции проверки подписей и т.д.). Давайте рассмотрим самые часто используемые и какие особенности есть у этих функций.

Инструменты для разработки децентрализованных приложений

Для удобства разработки децентрализованных приложений на Waves существует большое количество разных инструментов. Начать стоит, в первую очередь, с обозревателя блокчейна, который расположен по адресу wavesexplorer.com и позволяет анализировать данные в блоках, все транзакции и UTX как в основной сети, так и в stagenet и testnet.

Однако, если мы говорим про разработку децентрализованных приложений, а не просто использование, то встает несколько основных вопросов:

1. В какой среде писать код для смарт-аккаунтов, смарт-ассетов и децентрализованных приложений?
2. Как тестировать написанный код? Какие есть варианты для автоматического тестирования и ручного?
3. Как отлаживать код?
4. Как деплоить?

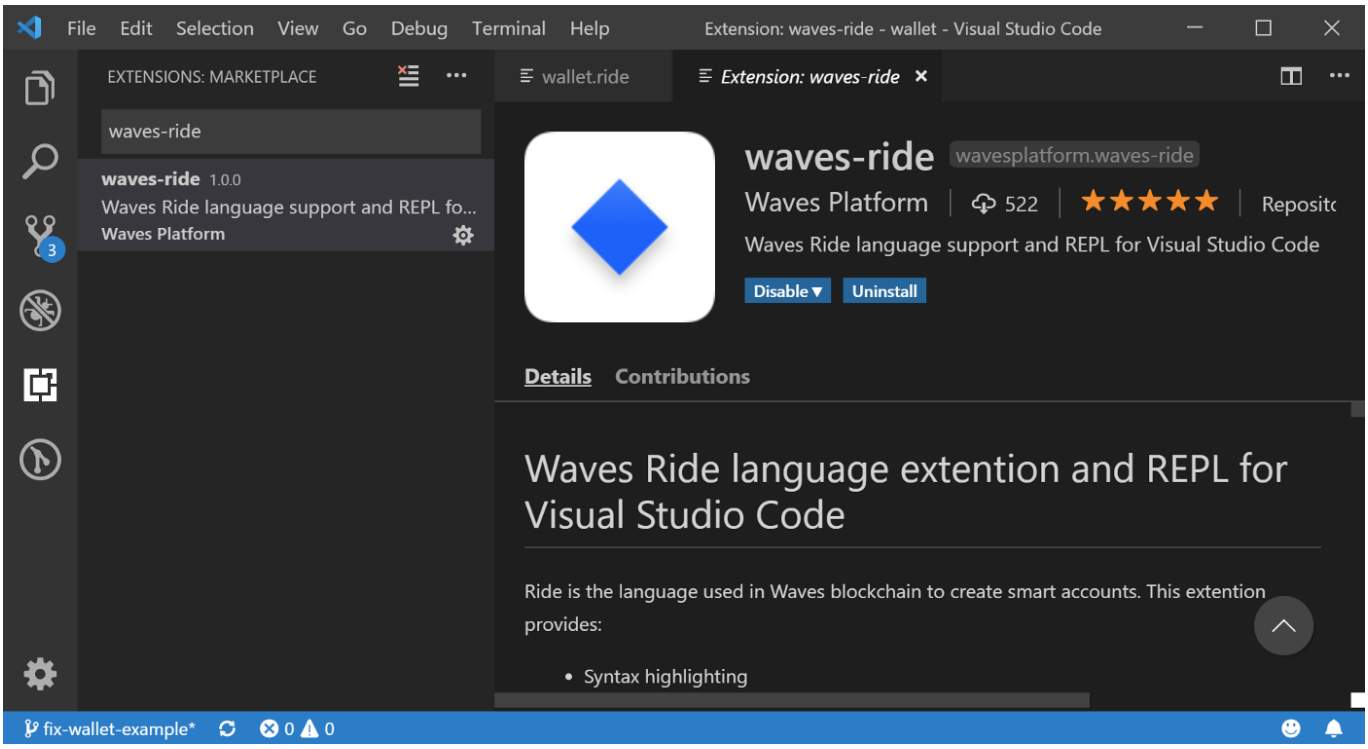
Давайте разберем по порядку какие есть инструменты для этого.

Среда разработки

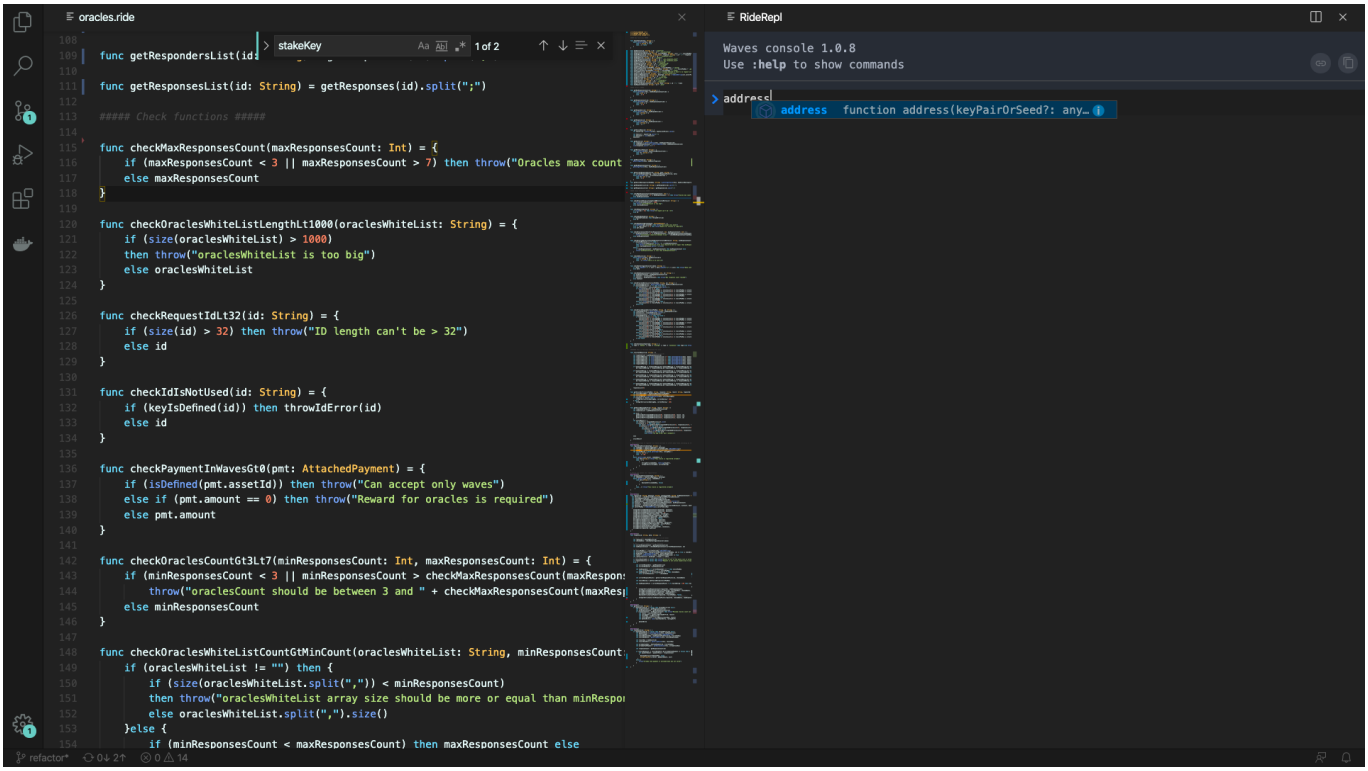
Самым простым вариантом начать писать код, тестировать и работать с аккаунтами, является использование онлайн IDE, который доступен по адресу https://ide.wavesplatform.com. В нем есть подсветка синтаксиса Ride, умные подсказки, вывод типов, компилятор, консоль для работы с библиотекой `waves-transactions` и даже REPL (read-eval-print loop) Ride, который позволяет выражения на Ride исполнять прямо в браузере. Так же есть примеры кода на Ride, примеры интеграционных тестов на JavaScript, возможность управления аккаунтами и отправки транзакций с помощью веб-интерфейса. Онлайн IDE отлично подходит для тестирования контрактов в stagenet и testnet. Токены Waves для этих сетей можно бесплатно получить с помощью крана в wavesexplorer по адресам `https://wavesexplorer.com/stagenet/faucet` и `https://wavesexplorer.com/testnet/faucet`, но не более 10 Waves каждые 10 минут.

Однако, для более профессиональной разработки контрактов, рекомендую использовать другие инструменты.

Расширение RIDE для Visual Studio Code является первым необходимым инструментом для профессиональной разработки с использованием блокчейна Waves. Установка этого расширения позволяет получить подсветку синтаксиса и подсказки для файлов с расширением `.ride`.



Кроме подсветски синтаксиса, расширение добавляет в VS Code еще и интерактивную консоль (прямо как в онлайн IDE), которая позволяет запускать функции из `waves-transactions`, `waves-crypto` и еще несколько специализированных.

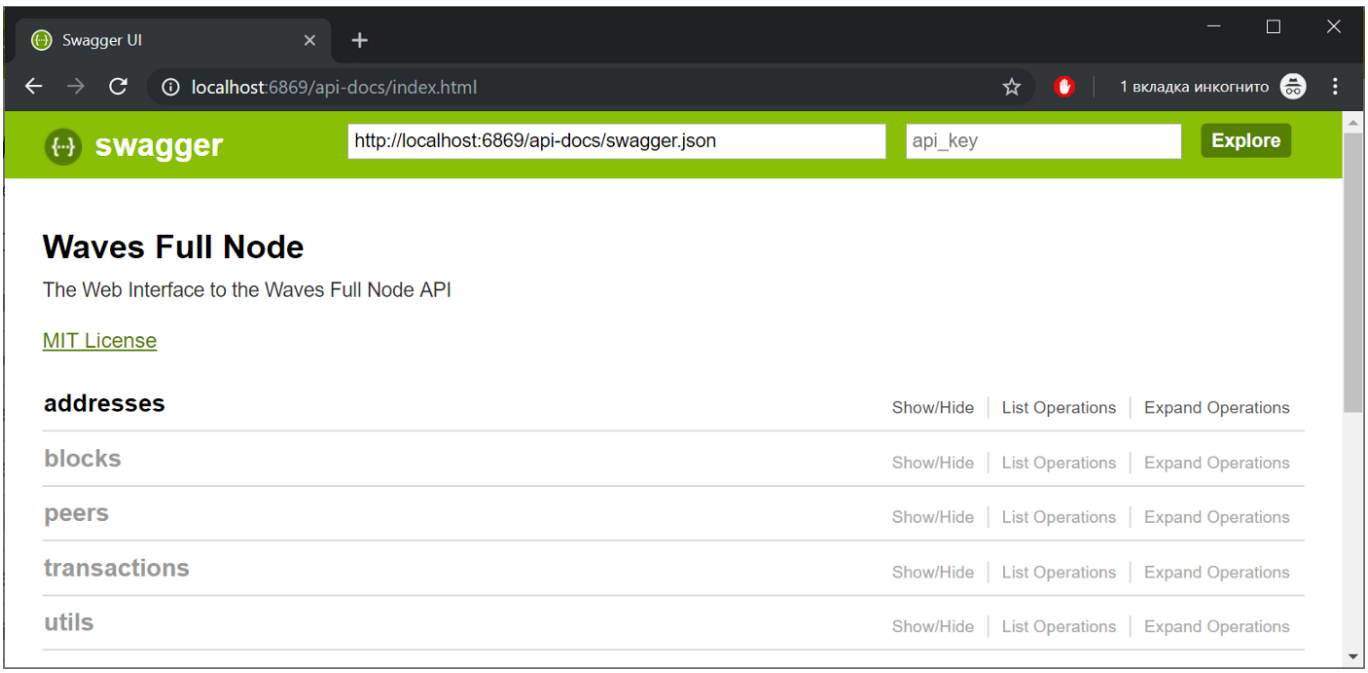


Локальное окружение

Во время разработки можно взаимодействовать с нодами из stagenet или testnet, которые доступны по адресам `http://nodes-stagenet.wavesnodes.com/` и `https://nodes-testnet.wavesnodes.com/`, однако наиболее удобным вариантом является использование приватного блокчейна из одной единственной ноды. Запустить такой блокчейн можно при условии наличия установленного Docker. Запуск осуществляется простой командой:

```
docker run -d -p 6869:6869 wavesplatform/waves-private-node
```

После запуска команды, локальный блокчейн будет запущен в виде Docker контейнера, API ноды будет доступен по адресу `http://localhost:6869/`, а все 100 миллионов токенов будут на балансе аккаунта с сид фразой `waves private node seed with waves tokens`.



Подробную информацию о Docker образе с этой нодой вы можете найти в этом репозитории.

Преимуществами такого подхода являются:

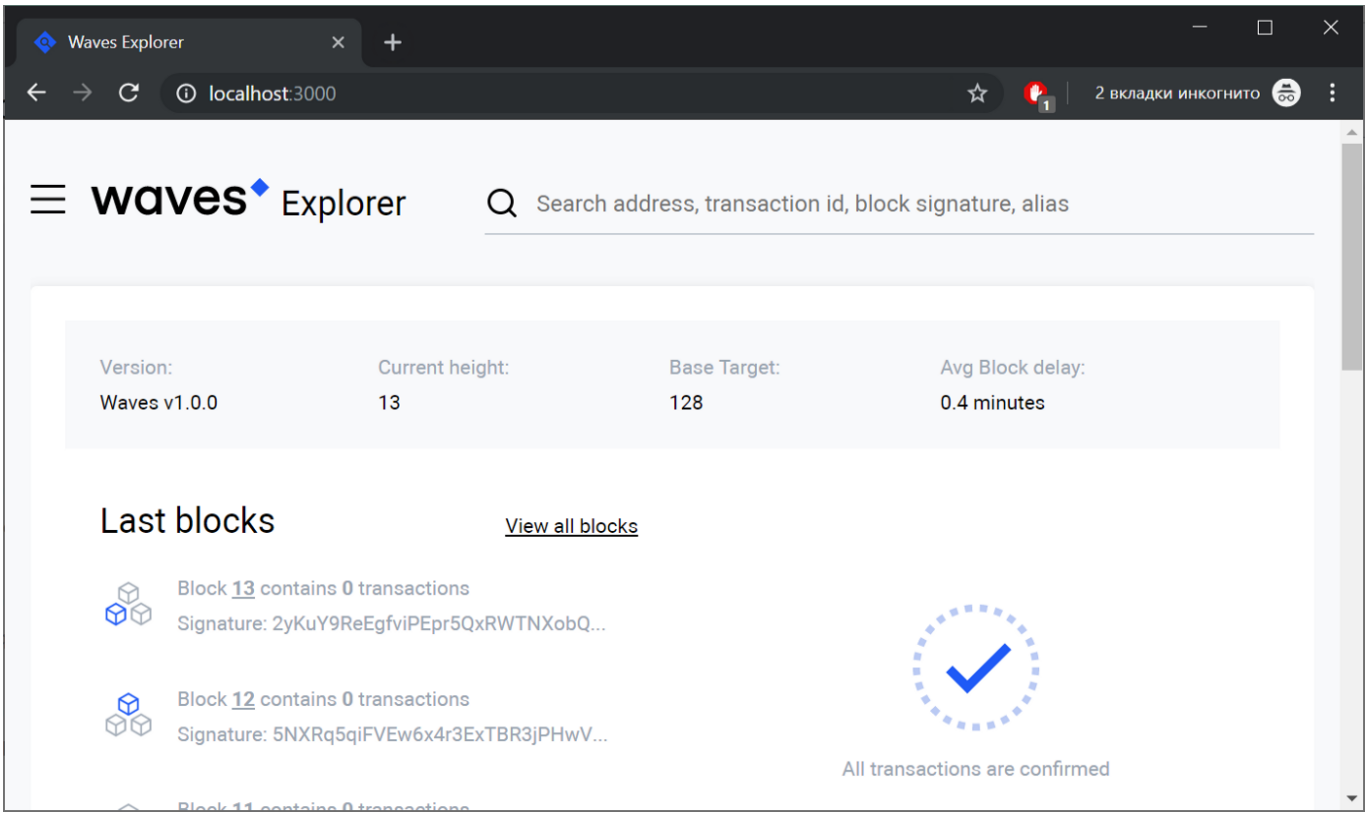
- Уменьшенное время блока: в testnet и stagenet блоки генерируются раз в минуту, тогда как в вашей приватной сети будут каждые 10 секунд. Это позволяет экономить время во время прогонов интеграционных тестов.
- Контроль над всеми токенами и отсутствие необходимости запрашивать токены с использованием крана
- Полный контроль над домой и работоспособностью API. Команда Waves Protocol старается обеспечить максимальную доступность публичных узлов, однако не всегда это возможно и в некоторых случаях API для stagenet или testnet могут быть недоступны.
- Ответы API публичных нод кэшируются, что может вызывать неожиданные ошибки.

После запуска ноды вы так же можете запустить локальный обозревательно блокчейна, который будет работать с вашей нодой. Делается это так же с помощью разворачивания Docker образа:

```
docker run -d -e API_NODE_URL=http://localhost:6869 -e
NODE_LIST=http://localhost:6869 -p 3000:8080 wavesplatform/explorer
```

Обратите внимание, что при разворачивании указывается адрес API нашей ноды с приватным блокчейном.

После разворачивания образа, обозревательно станет доступен по адресу `http://localhost:3000`:



Тестирование кода

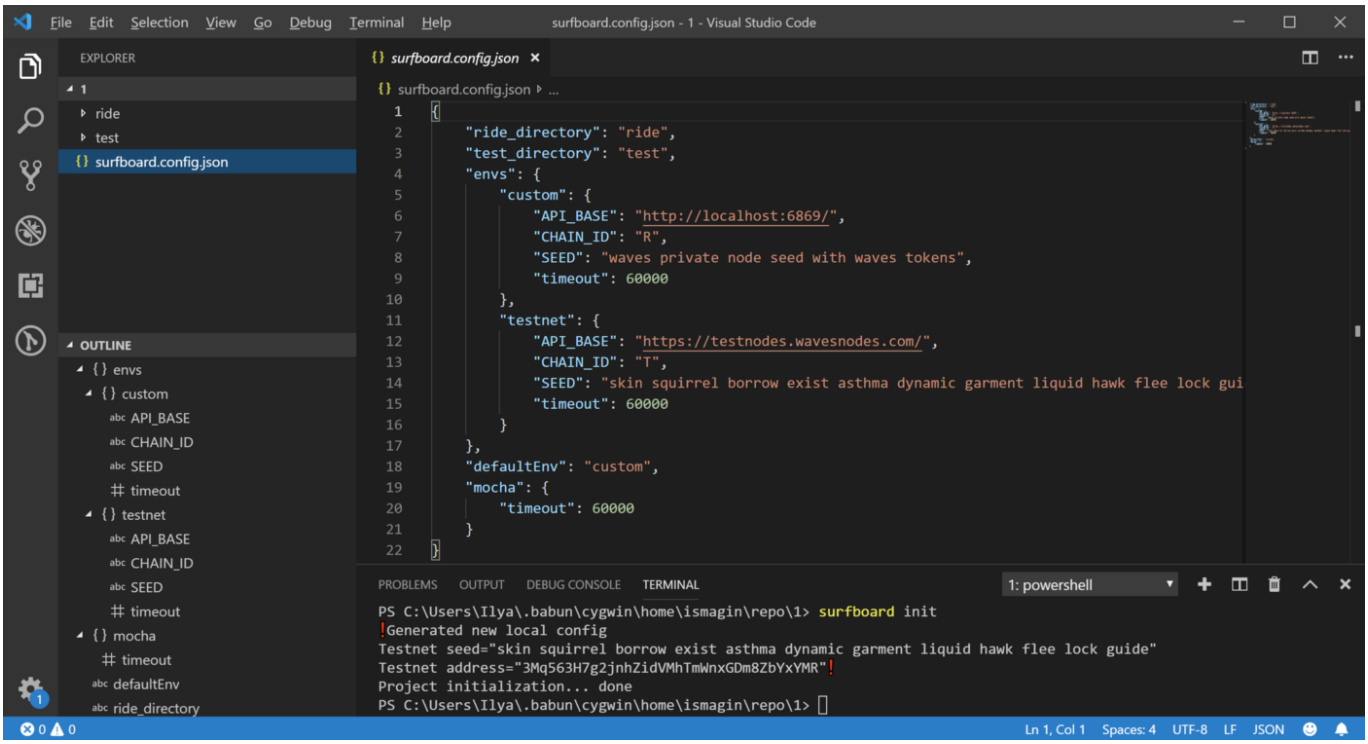
На момент написания этих строк существует возможность написания только интеграционных тестов для децентрализованных приложений на Ride. Инструментов для Unit тестов пока нет. Интеграционное тестирование в случае с Ride подразумевает, что написанный код компилируется, разворачивается с помощью `SetScript`, `SetAssetScript` или `Issue` транзакции на ассете или аккаунте и выполняются транзакции, которые проверяют корректность поведения скрипта. Другими словами, идет непосредственно работа с блокчейном (не эмуляцией!) и отправляются настоящие транзакции.

Интеграционные тесты могут быть написаны на Java с использованием библиотеки Paddle или на [JavaScript] с использованием онлайн IDE или библиотеки `surfboard`.

Surfboard можно установить из npm (при условии наличия у вас node.js и npm) слдующей командой:

```
npm install -g @waves/surfboard
```

После этого у вас станет доступна команда `surfboard`. Команда `surfboard init` позволит инициализировать новый проект, который будет содержать конфигурационный файл и директории для тестов (`./test`) и скриптов на Ride (`./ride`). Конфигурационный файл позволяет задать настройки для работы с разными типами сетей, параметры аккаунтов и т.д.



В директории `./test` можно создавать любые файлы с расширением `.js` и писать в них интеграционные тесты с использованием тестового фреймворка `Mocha`. Кроме непосредственно самой `Mocha` в тестовом файле доступны функции из `waves-transactions` и несколько дополнительных функций и переменных:

- `setupAccounts({[key: string]: number})` - позволяет в начале скрипта создать новые аккаунты и перечислить на них с мастер сида тоокена
- `compile(file: File): String` - позволяет скомпилировать содержимое файла
- `file(path: String): File` - позволяет получить содержимое файла
- `accounts` - объект, в которой хранятся сиды созданных функцией `setupAccounts` аккаунтов

Описание этих и других функций доступно в документации. Примеры тестов вы можете найти в онлайн IDE или в репозитории ride-examples.

Запуск тестов в директории можно запустить с помощью команды `surfboard test`, если же хочется запустить конкретный файл, а не все файлы в директории `./test`, то можно выполнить `surfboard test my-scenario.js`.

Отладка скриптов Ride

Для отладки скриптов Ride принято искользовать 2 основных приема.

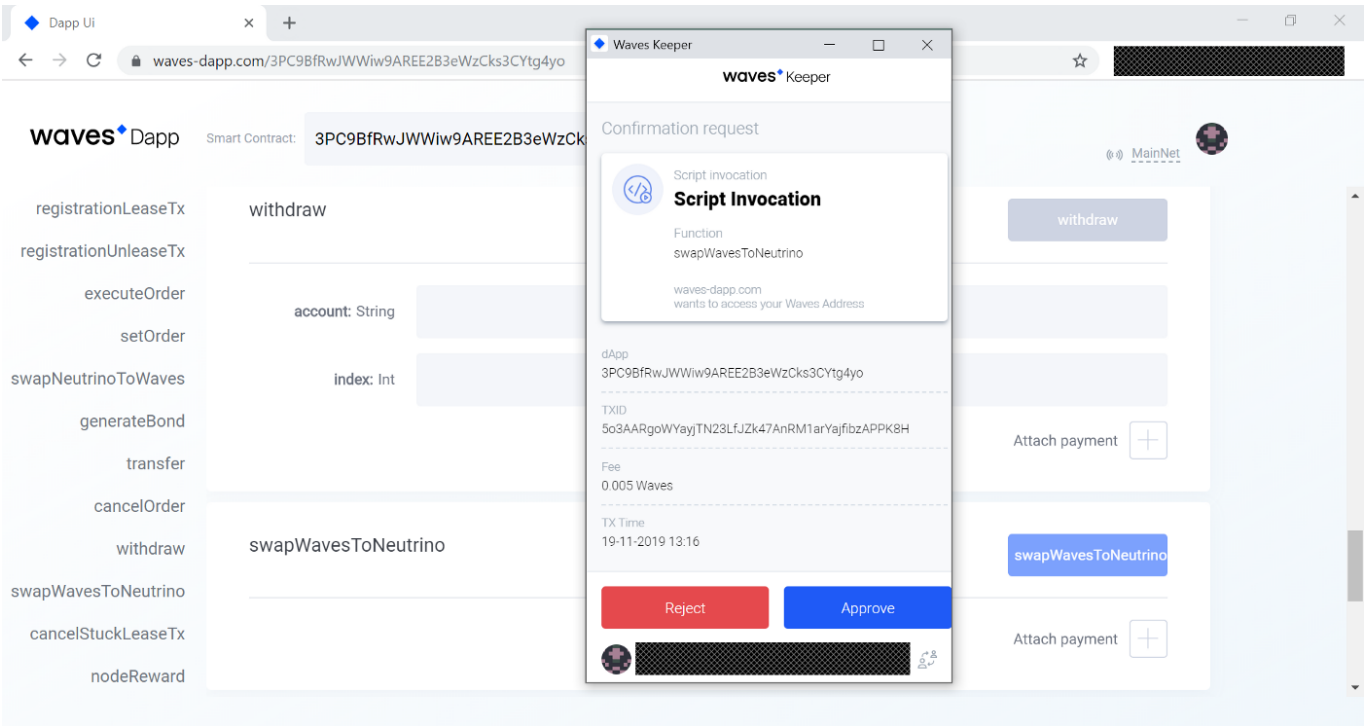
В онлайн IDE и Surfboard есть REPL, который позволяет ввести код и срзу же получить результат. REPL позволяет не только выполнять базовые операции, но и объявлять переменные, работать с настоящим блокчейном (например, читать стейт аккаунтов), вызывать функции стандартной библиотеки

```
PS > surfboard repl
Welcome to RIDE repl
Compiler version 1.1.3-5-gca4d146
Use ?{term} to look up a specific definition, ?? to get the full context
Use :clear to clear console, :reset to restart repl, .editor to enter multiline
RIDE > let x = 2 + 2
defined let x: Int
RIDE > x
res1: Int = 4
RIDE >
```

В более сложных ситуациях, когда у вас уже есть полноценный скрипт, который необходимо отлаживать, на помощь приходит функция `throw()` из стандартной библиотеки Ride, которая позволяет выбросить ошибку и текстовое описание к ней.

Ручное тестирование приложений

Если вам по душе больше ручное тестирование или вы хотите поиграться с приложениями, уже развернутыми в сети, вы можете воспользоваться веб-сайтом https://waves-dapp.com. Вы можете просто указать адрес аккаунта нужного децентрализованного приложения и Waves-Dapp покажет все доступные функции, какие параметры они принимают и позволит вызвать любую из них. Инструмент может быть полезен и для тестирования своего приложения, когда у вас нет интерфейса или тестов для некоторых функций, или, вам необходимо поменять настройки вашего приложения.



Oraculus

Блокчейн и данные из реального мира

Блокчейн отлично работает с данными внутри себя, но не имеет никакого представления о том, что происходит за границами его консенсуса, а ведь там вся жизнь. Блокчейн не может сходить во внешний API и получить данные оттуда, потому что должна быть детерменированность операций. Если

каждая нода будет ходить во внешний API в разное время, то какая-то может получить один результат, вторая - другой, третья - вообще получить ошибку, в итоге ноды блокчейна никогда не придут к консенсусу, ведь нельзя понять, какой ноде надо довериться. Чтобы решить эту проблему, вместо модели pull (когда ноды ходят за данными в реальный мир) принято применять модель push, когда есть какие-то поставщики данных, которые сохраняют данные в блокчейне, а дальше любые децентрализованные приложения в блокчейне используют эти данные. Сущности, которые сохраняют данные в блокчейне называются оракулами. Но с оракулами есть одна проблема - они централизованные, то есть мы доверяем одной сущности, которая поставляет нам какие-то данные. В общем случае, система является настолько централизованной, насколько централизованна самая "плохая" часть. То есть, децентрализованное приложение, которое использует для принятие критичных решений одного оракула на самом деле является централизованным приложением. Почему? Логика простая - повлияв на поведение одной сущности (оракула) можно добиться нужного поведения от такого приложения.

> Если децентрализованное приложение опирается на одного оракула, то такое приложение не является децентрализованным.

Идея сделать децентрализованных оракулов лежит на поверхности, но простого решения у этой проблемы нет, поэтому ее часто называют "проблемой оракулов". Давайте рассмотрим, что можно с этим сделать.

Простейший вариант децентрализованных оракулов

Самым простым решением является мультиподпись, когда несколько разных пользователей должны придти к консенсусу и подписать одни и те же данные. Например, мы хотим получать данные о курсе USD/EUR и у нас есть 5 оракулов, которые должны договориться о цене (каким-то образом за пределами блокчейна), подписать транзакцию и отправить ее в сеть на специальный аккаунт, который примет эту транзакцию только при наличии не менее 3 подписей из 5. Простейший контракт для такого случая выглядел бы так:

```
{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ACCOUNT #-}

# array of 5 public keys
let pks = [base58'', base58'', base58'', base58'', base58'']

# inner fold step for each signature
func signedBy(pk:ByteVector) = {
   # is signed by the public key or not
   func f(acc: Boolean, sig: ByteVector)
      = acc || sigVerify(tx.bodyBytes, sig, pk)
   FOLD<5>(tx.proofs, false, f)
}

# outer fold step for each public key
func signedFoldStep(acc:Int, pk:ByteVector)
   = acc + (if(signedBy(pk)) then 1 else 0)
```

```
# comparing total number of correct signatures
# to required number of correct signatures
FOLD<5>(pks, 0, signedFoldStep) >= 3
```

У такого решения вопроса есть несколько проблем:

- В случае отсутствия консенсуса у оракулов, данные просто не будут поставлены в блокчейн
- Отсутствует экономическая мотивация у поставщиков данных
- Заранее заданный и ограниченный список оракулов

Для решения этих проблем можно сделать полноценное децентрализованное приложение, которое будет из себя представлять маркетплейс, на котором встречаются 2 стороны:

1. Приложения, которым нужны данные
2. Оракулы, которые готовы поставлять данные в обмен на вознаграждение

Давайте сформулирует основные функциональные требования к такому приложению и реализуем его, используя Ride и блокчейн Waves.

Децентрализованный оракул как dApp

Основные принципы работы децентрализованных оракулов должны быть следующими:

- Каждый желающий владелец любого децентрализованного приложения должен быть в состоянии запросить данные в определенном формате и с определенным вознаграждением для оракулов
- Каждый желающий должен быть в состоянии зарегистрировать своего оракула и отвечать на запросы, зарабатывая таким образом
- Все действия оракулов должны быть доступны для аудита

Запрос данных

Любой аккаунт в блокчейне может "отправить запрос" оракулам. При отправке запроса необходимо прикладывать награду для оракулов за предоставление корректных данных (в токенах Waves). При задании вопросы обходимо указывать следуюшие параметры:

- `id` - уникальный идентификатор каждого вопроса, генерируется задающим вопрос. Требование - отсутствие такого же ключа у dApp, не больше 64 символов.
- `question` - непосредственно задаваемый вопрос. Вопрос формируется в специальном формате для каждого типа данных, в начале вопроса должен стоять тип данных, после разделителя `//` идут мета данные в формате JSON. Например, для типа данных `Temperature` вопрос выглядит следующим образом: `Temperature//{"lat": "55.7558", "lon": "37.6173", "timestamp": 150000000000, "responseFormat": "NN.NN"}`.
- `consensusType` - правило агрерации данных. Для строковых типов данных контракт предусматривает возможность только консенсуса (полное совпадение ответов), в то время как для числовых может быть `median` или `average`.
- `minOraclesCount` - минимальное количество оракулов, которые должны предоставить данные, чтобы можно было получить итоговый консенсус-результат. Значение не может быть меньше 3.
- `maxOraclesCount` - максимальное количество оракулов, которые могут ответить на вопрос. Не больше 6.

- `oraclesWhiteList` - список оракулов (публичные ключи через запятые), которые должны предоставить данные. Если значение параметра равно пустой строке, то любой оракул может ответить на запрос данных
- `tillHeight` - дедлайн до достижения консенсуса. Если до этого времени консенсус между оракулами не был достигнут (не набралось количество ответов > `minOraclesCount`), то отправитель запроса может забрать награду.

Формат типа запросов данных оставим на усмотрение отправителей запросов и оракулов, но в качестве примера предлагают следующие:

- `Temperature//{"lat": "55.7558", "lon": "37.6173", "timestamp": 150000000000, "responseFormat": "NN.NN"}`
- `Pricefeed//{"pair": "WAVES/USDN", "timestamp": 150000000000, "responseFormat": "NN.NNNN"}`
- `Sports//{"event": "WC2020", "timestamp": 150000000000, "responseFormat": "%s"}`
- `Random//{"round": 100, "responseFormat": "%s"}`

Сбор ответов оракулов

Любой аккаунт Waves может зарегистрироваться как оракул определенного типа данных. Для этого будет достаточно вызвать метод децентрализованного приложения и передать в качестве аргумента тип поставляемых данных. Пример вызова может выглядеть следующим образом - `registerAsOracle("Temperature")`. В этот момент в стейте dApp будет зафиксировано в какой момент произошла регистрация оракула в качестве поставщика определенного типа данных и записано следующее: `{oraclePublicKey}_Temperature={current_height}`.

Ответ оракулов осуществляется с помощью метода `response(id: String, data: String)` и `responseNumber(id: String, data: Integer)` .

Подсчет результатов

Для подсчета результатов необходимо вызвать метод `getResult(id: String)`. Подсчет результатов возможен только в том случае, когда ответили больше оракулов, чем указано в `minOraclesCount`. При выборе правильного ответа используется не простое большинство, а рейтинги оракулов. Рейтинг формируется по следующей логике:

- каждый оракул имеет рейтинг 100 при регистрации
- за каждый ответ, который в итоге стал результатом запроса, оракул получает +1 рейтинга, за каждый неверный -1

Давайте представим, что на запрос `Sports//{"event": "WorldCup2020", "timestamp": 150000000000, "responseFormat": "%s"}` ответили 5 оракулов со следующими рейтингами и ответами:

1. Oracle0, рейтинг = 102, ответ = "France"
2. Oracle1, рейтинг = 200, ответ = "Croatia
3. Oracle2, рейтинг = 63, ответ = "France"
4. Oracle3, рейтинг = 194, ответ = "France"

5. Oracle4, рейтинг = 94, ответ = "Croatia"

Итоговый результат будет `France`, ведь суммарный рейтинг оракулов с таким ответом составляет 359, в то время как рейтинг оракулов, ответивших `Croatia` 294.

В результате процедуры подсчета голосов рейтинги оракулов `Oracle0`, `Oracle2` и `Oracle3` будут увеличены на 1 и они смогут забрать вознаграждение, в то время как рейтинги `Oracle1` и `Oracle4` уменьшены на единицу и они не получат вознаграждения.

Реализация

Давайте шаг за шагом начнем реализацию такого децентрализованного приложения. Логичнее всего начать с метода регистрации оракулов, который будет принимать в качестве аргумента тип предоставляемых оракулом данных. Если один оракул с одним публичным ключом предоставляет несколько типов данных, то он должен регистрироваться несколько раз.

```
@Callable(i)
func registerAsOracle(dataType: String) = {
    let neededKey = inv.callerPublicKey.toBase58String() + "_" + dataType
    let ratingKey = i.callerPublicKey.toBase58String() + "_rating"

    let currentRating = match getInteger(this, ratingKey) {
        case v: Int => v
        case _ => 100
    }
    match (getString(i.caller, neededKey)) {
        case data:String => throw("This oracle is registered already")
        case _ => {
            [
                StringEntry(neededKey, toString(height)),
                IntegerEntry(ratingKey, currentRating)
            ]
        }
    }
}
```

Следующим логичным шагом будет реализация функционала по отправке запросов на предоставление данных. Как мы уже описывали ранее, запрос на предоставление данных должен включать в себя следующие аргументы:

- `id` - уникальный идентификатор каждого вопроса.
- `question` - непосредственно задаваемый вопрос в заранее оговоренном формате.
- `consensusType` - правило агрерации данных, `consensus`, `median` или `average`.
- `minOraclesCount` - минимальное количество оракулов.
- `maxOraclesCount` - максимальное количество оракулов.
- `oraclesWhiteList` - список оракулов (публичные ключи через запятые или пустая строка)
- `tillHeight` - дедлайн до достижения консенсуса.

Функция должна записывать в стейт контракта параметры запроса, сумму вознаграждения, а так же публичный ключ отправителя запроса и ключи, по которым в дальнейшем будем записывать количество ответов, сами ответы, публичные ключи ответивших оракулов и флаг завершения запроса.

В момент запроса данных необходимо проверять аргументы на следующие условия:

- Если задан white-list оракулов, то длина строки с их публичными адресами не должна быть больше 1000 символов (функция `checkOraclesWhiteListLengthLt1000`)
- Уникальный идентификатор запроса должен быть не более 32 символов (функция `checkRequestIdLt32`)
- Идентификатор запроса не должен быть раньше использован (функция `checkIdIsNotUsed`)
- У каждого запроса должно быть вознаграждение в токенах Waves (функция `checkPaymentInWavesGt0`)
- Минимальное количество оракулов должно быть не менее 3, а максимальное не более 6 (функция `checkOraclesCountGt3Lt6`)
- Значение минимального количества должно быть меньше либо равно, чем значение максимального количества (функция `checkOraclesWhiteListCountGtMinCount`)

Все листинга кода ниже включают в себя вызовы вспомогательных функций, которые в этой статье не показываются, однако вы модете найти их в репозитории Ride examples

```
@Callable(i)
func request(id: String, question: String, minResponsesCount: Int,
maxResponsesCount: Int, oraclesWhiteList: String, tillHeight: Int) = {
        let whiteList =
checkOraclesWhiteListLengthLt1000(oraclesWhiteList)
        let checkedRequestIdLt64 = checkRequestIdLt32(id)
        let requestId = checkIdIsNotUsed(checkedRequestIdLt64)
        let paymentAmount =
checkPaymentInWavesGt0(i.payments[0].extract())
        let minCount = checkOraclesCountGt3Lt6(minResponsesCount,
maxResponsesCount)
        let maxCount =
checkOraclesWhiteListCountGtMinCount(oraclesWhiteList, minCount,
maxResponsesCount)
        let callerPubKey = toBase58String(i.callerPublicKey)
        [
            StringEntry(keyQuestion(requestId), question),
            StringEntry(keyOraclesWhiteList(requestId), whiteList),
            StringEntry(keyRequesterPk(requestId), callerPubKey),
            StringEntry(keyResponders(requestId), ""),
            StringEntry(requestId, question),
            IntegerEntry(keyMinResponsesCount(requestId), minCount),
            IntegerEntry(keyMaxResponsesCount(requestId), maxCount),
            IntegerEntry(keyResponsesCount(requestId), 0),
            IntegerEntry(keyTillHeight(requestId), tillHeight),
            IntegerEntry(keyRequestHeight(requestId), height),
            IntegerEntry(keyPayment(requestId), paymentAmount),
            BooleanEntry(keyRequestIsDone(id), false)
        ]
```

```
}
```

Отлично! Теперь у нас есть функции регистрации оракулов и отправки запросов от пользователей. Давайте теперь реализуем функционал отправки ответа от одного оракула.

Ответ на запрос данных

Каждый оракул может ответить на запрос, если нет ограничений в виде white-list и запрос уже не закончился (по количеству ответов или длительности). В момент ответа на запрос, публичный ключ оракула и его ответ записываются в хранилище децентрализованного приложения, для этого в момент отправки запроса были созданы ключи `{id}_responders` и `{id}_responses`. Данный в этих ключах хранятся в виде строк с разделителем `;`.

```
@Callable(i)
func response(id: String, data: String) = {

    # Шаг 0 - проверка валидности предоставленных даннх
    let requestId = checkIdExists(id)
    let checkedData = checkDataIllegalCharacters(data)

    # Шаг 1 - проверка состояния запроса (количества ответов уже)
    let currentResponsesCount = getResponsesCount(id)
    let newResponsesCount = checkNewResponsesCount(currentResponsesCount,
id)

    # Шаг 2 - проверка на вхождение оракула в white list и не ответил ли
он уже на запрос
    let oraclePubKey = i.callerPublicKey.toBase58String()
    let oracleIsAllowed = checkOracleInWhiteList(oraclePubKey, id) == true
|| checkOracleResponded(oraclePubKey, id) == false
    let maxHeight = getIntegerValue(this, keyTillHeight(id))
    let isDone = getBooleanValue(this, keyRequestIsDone(id)) == true
    let requestIsActive = maxHeight > height || isDone

    if (oracleIsAllowed == false) then throw("Oracle is not in the white
list or already responded") else
    if (requestIsActive == false) then throw("Request is not active
anymore due to max height (" + maxHeight.toString() + "/" +
height.toString() + ") or it is just done (" + isDone.toString() + ")")
    else {

        let currentResponders = getResponders(id)
        let currentResponses = getResponses(id)

        let newResponders = if currentResponders == "" then oraclePubKey
            else currentResponders + ";" + oraclePubKey
        let newResponses = if currentResponses == "" then checkedData
            else currentResponses + ";" + checkedData
```

```
        let currentResponsePoints = getCurrentResponsePoints(id,
checkedData)

        let oracleRating = getOracleRating(oraclePubKey)

        let newResponsePoint = currentResponsePoints + if oracleRating <
200 then oracleRating / 3 else log(oracleRating, 0, 8, 0, 5, HALFEVEN)


        [
            IntegerEntry(keyCurrentResponsePoints(requestId, checkedData),
newResponsePoint),
            IntegerEntry(keyResponsesCount(requestId), newResponsesCount),
            StringEntry(keyResponseFromOracle(requestId, oraclePubKey),
checkedData),
            StringEntry(keyResponders(requestId), newResponders),
            StringEntry(keyResponses(requestId), newResponses),
            BooleanEntry(keyTookPayment(requestId, oraclePubKey), false),
            # StringEntry(keyOneResponse(requestId, i, checkedData),
newResponders),newResponsePoint)
        ]
    }
}
```

Так же во время ответа мы увеличиваем счетчик количества ответивших на запрос и какое количество баллов набирает данный ответ (баллы равны сумме рейтингов оракулов).

Сбор результатов

После отправки своих данных оракулами, должно быть подведение итогов, вырадающееся в выборе итогового результата (консенсусного решения, среднего или медианы) и изменении рейтингов оракулов. В этой функции мы так же могли бы выплатить сразу часть вознагрждения ответившим правильно, но в виду ограничения по сложности контракта в 4000, сделать в рамках одной функции не получится. Однако мы можем записать в хранилище аккаунта, кто имеет право забрать часть вознаграждения и позволить самим оракулам вызвать специальную функцию получения вознаграждения. Напомню, что только оракулы, чей ответ совпал с итоговым (или все оракулы с ответом в пределах 10% от результата если запрашивалось среднее или медиана) имеют право забрать часть ознаграждения.

```
@Callable(i)
func getResult(id: String) = {
    if (keyIsDefined(id) == false) then throwIdError(id) else {
        let responsesCount = getResponsesCount(id)
        let minResponsesCount = getMinResponsesCount(id)
        if (responsesCount < minResponsesCount) then throw("Minimum
oracles count not reached yet") else {
            let result = calculateResult(id)
            let ratingsDiff = getOracleRatingsDiff(id, result)
            let resultKey = keyResult(id)
```

```
            let resultDataEntry = StringEntry(resultKey, result)
            let dataToWrite = cons(resultDataEntry, ratingsDiff)

            dataToWrite
        }
    }
}
```

Получение вознаграждения

Функция получения вознаграждения для оракула должна позволять забрать средства только один раз и проверять, что этот оракул действительно отвечал на запрос и ответ его считается корректным.

```
@Callable(i)
func takeReward(id: String) = {
    if (keyIsDefined(id) == false) then throwIdError(id) else {
        let paymentValue = getIntegerValue(this, keyPayment(id))
        let oraclePubKey = i.callerPublicKey.toBase58String()
        let oracleResponseKey = keyResponseFromOracle(id, oraclePubKey)
        let oracleResponse = getStringValue(this, oracleResponseKey)

        let resultKey = keyResult(id)
        let resultDataEntry = getStringValue(this, resultKey)

        let alreadyTookKey = keyTookPayment(id, oraclePubKey)
        let alreadyTookPayment = getBooleanValue(this, alreadyTookKey)

        let responsesCount = getResponsesCount(id)

        if (oracleResponse == resultDataEntry && alreadyTookPayment ==
false) then {
            let paymentAmount = paymentValue / responsesCount
            [
              BooleanEntry(alreadyTookKey, true),
              ScriptTransfer(i.caller, paymentAmount, unit)
          ]
        }else {
            throw("Already took payment or provided data was not valid")
        }

    }
}
```

На этом основной функционал контракта для консенсуса оракулов готов. Примеры в виде тестов как работать с таким контрактом вы можете найти в репозитории Ride examples

Возможности для развития

Релизованный функционал является простейшим вариантом работы децентрализованных оракулов. Мы решили проблемы, которые были обозначены в начале:

- В блокчейне всегда будут данные, даже если не все оракулы достигнут консенсуса
- У участников процесса появляется экономическая и репутационная мотивация участвовать в предоставлении данных
- Список оракулов может быть максимально широким, но в то же время его можно ограничить для своего запроса, если мы не хотим получать данные от любых оракулов, а только тех, которым мы доверяем.

Благодаря тому, что форматы запросов являются типизированными, предоставление ответов можно автоматизировать, например, в виде браузерного расширения, которое следит за запросами на адресе децентрализованного приложения и отвечает на данные, если тип запрашиваемых данных поддерживается расширением. Можно представить сценарий, когда пользователи с открытым браузером могут зарабатывать на предоставлении данных, при этом не делая ничего для этого своими руками.

Во многих случаях данные необходимы не разово, а в виде постоянного потока. В нашем децентрализованном приложении функциональность подписки на данные не реализована, но мы будем рады участию нашего сообщества в доработке этого примера. Pull requests are welcome!

Billy

Лучший способ выучить язык - начать на нем писать. Заставить вас писать я не могу, поэтому предложу вам второй по крутизне вариант - читать как пишется код! Мы разберем несколько примеров контрактов, начнем с относительно простых, и закончим на достаточно сложных и разухабистых. Первым контрактом, который будем писать является контракт Oraculus, его идея очень проста - сделать децентрализованных оракулов.

Описание проекта

Billy - децентрализованное приложение на базе бота для корпоративного мессенджера Slack. Вы можете найти подробную информацию о том, как работает Billy, на официальном сайте проекта - https://iambilly.app. Но давайте я коротко расскажу о том из каких частей он состоит и как именно в нем используется блокчейн.

Billy является проектом для мотивации сотрудников компании. В ходе работы у нас возникает много ситуаций, когда мы помогаем коллегам или они помогают нам. И далеко не всегда такая помощь входит в рабочие обязанности коллеги. Чтобы стимулировать помощь вы можете добавить Billy в Slack одной кнопкой на официальном сайте проекта. Billy генерирует уникальный адрес для каждого сотрудника компании и сохраняет их в базе данных. Каждый месяц бот начисляет на сгенерированные адреса 500 "Спасибо"-токенов, которые могут быть потрачены с помощью этого бота. Для этого сотрудники компании могут отправлять специальные команды боту (`10 спасибо @username`) или реагировать на сообщения с использованием специалных emoji.

Неиспользованный остаток из 500 спасибо-токенов за месяц сгорают в конце этого месяца. Заработанные спасибо-токены не сгорают и могут быть использованы для 3 целей:

- **Перевод другим пользователям в благодарность.** Полученные токены можно переводить в любое время своим коллегам.

- **Покупка товаров во внутреннем магазине.** Внутренний магазин позволяет сотрудникам компании (или уполномоченным лицам) предлагать товары и услуги в обмен на спасибо-токены.
- **Для участия в голосованиях и crowdfunding компаниях.** Каждый пользователь может указать цель, для которой собирает токены (например, проведение внутреннего митапа) или руководство компании может инициировать голосования, где каждый токен будет считаться одним голосом, то есть более "полезный" и активный сотрудник может больше влиять на итоги голосования.

Видео-демонстрацию работы бота вы можете найти на официальном сайте проекта - https://iambilly.app

How to: бесплатные (для пользователя) децентрализованные приложения

Дисклеймер: данная возможность является НЕ документированной и будет заменена на WEP 2: Customizable Sponsorship. Но знать данный метод может не помешать.

Оплата комиссии за счет dApp

Этот вариант спонсирования подходит только для одного типа транзакций - `InvokeScript` и представляет собой достаточно простой механизм. Преимуществом по сравнению со спонсированием является отсутствие необходимости использовать свой токен.

Любой пользователь может вызвать dApp имея 0 Waves на своем аккаунте, но указав в качестве комиссии именно Waves. Скрипт начнет выполняться и, если в результата выполнения скрипта на аккаунт пользователя перечисляются Waves, то он получит их и уже оттуда спишется комиссия. Давайте на примере, так всегда ведь проще.

Например, есть dApp, который на основе адреса вызывающего отправляет или нет ему 1 Waves. **Данный пример совершенно не безопасен и приводится только для объяснения принципа работа. Ни в коем случае не используйте такую логику в своих децентрализованных приложениях.**

```
@Callable(invoke)
func callMeBaby() = {

    let callerAddress = invoke.caller.bytes.toBase58String()

    if (takeRight(invoke.caller.bytes, 4) == base16'ABCD') then
        TransferSet([
            ScriptTransfer(invoke.caller, 100000000, unit)
        ])
    else
    throw("You didn't win")
}
```

Если вызывать эту функцию с аккаунта, на котором 0 Waves баланса, указав `fee` в 0.005 Waves, скрипт все равно начнет выполняться. Если пользователь имеет право на получение 1 Waves, то транзакция будет считаться валидной и попадет в блокчейн. Пользователь СНАЧАЛА получит 1 Waves в результате

вызова, затем с него будет списано 0.005 комиссии. По итогу пользователь получит на своем аккаунте 0.995 Waves.

Внимательные читатели уже догадались, что скрипт выше является небезопасным, потому что можно вызывать его бесконечное количество раз - если аккаунт не выиграл, то транзакция не попадет в блокчейн, если аккаунт выиграл, то получит 0.995 Waves. Беспроигрышная лотерея получилась.

Правильный рецепт заключается в том, чтобы всегда проверять дополнительные условия имеет ли пользователь права вызывать скрипт, например, иметь свой белый список.

```
@Callable(invoke)
func callMeBaby(uuid: String) = {

    if (isInWhiteList(invoke.caller) && invoke.fee == 500000) then {

        let id = extract(invoke.transactionId)
        let callerAddress = toBase58String(invoke.caller.bytes)

        ScriptResult(
            WriteSet([
                DataEntry(toBase58String(id), callerAddress + uuid)
            ]),
            TransferSet([
                ScriptTransfer(addressFromStringValue(callerAddress),
invoke.fee, unit)
            ])
        )
    }
    else
        throw()
}
```

Хочется дополнительно отметить, что если в будущем модель исполнения контрактов в Waves изменится и транзакции с ошибками (`throw()`) тоже будут записываться в блокчейн и снимать комиссию, то использование этого функционала станет невозможным.

Обсуждение того, как работает описанный выше функционал и о возможных путях развития вы можете найти в этом issue на гитхаб.

Лучшие практики

Оба решения рекомендуется использовать только для триальных режимов продуктов и проверять все граничные условия, потому что неправильное использовать может привести к потере средств.