Блокчейн-разработка на Tezos

От констант до публикации контракта

Предисловие

Это печатная версия онлайн-курса по блокчейн-разработке на Tezos от ForkLog и Tezos Ukraine. Он состоит из пяти уроков и рассчитан на новичков.

Мы начнем с азов: вы изучите структуру смарт-контрактов и базовый синтаксис языка LIGO, а затем научитесь писать контракты и взаимодействовать с блокчейном. В конце курса вы создадите тестовые токен и NFT в сети Tezos.

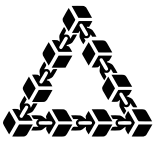
> **Важно:** сообщество регулярно обновляет протокол Tezos, а бейкеры отключают устаревшие тестнеты. Если тестнет из примеров не работает, перед публикацией смарт-контракта используйте актуальную RPC-ссылку из списка на сайте Taquito.

Содержание

Во время работы над курсом нас консультировали и поддерживали:

Tezos Ukraine

Некоммерческая организация, которая популяризирует Tezos в Украине и Восточной Европе.



Attic Lab

Разработчики обозревателя Teztracker и стейкингового сервиса Everstake. Консультируют Нацбанк Украины по цифровым валютам.

Baking Bad

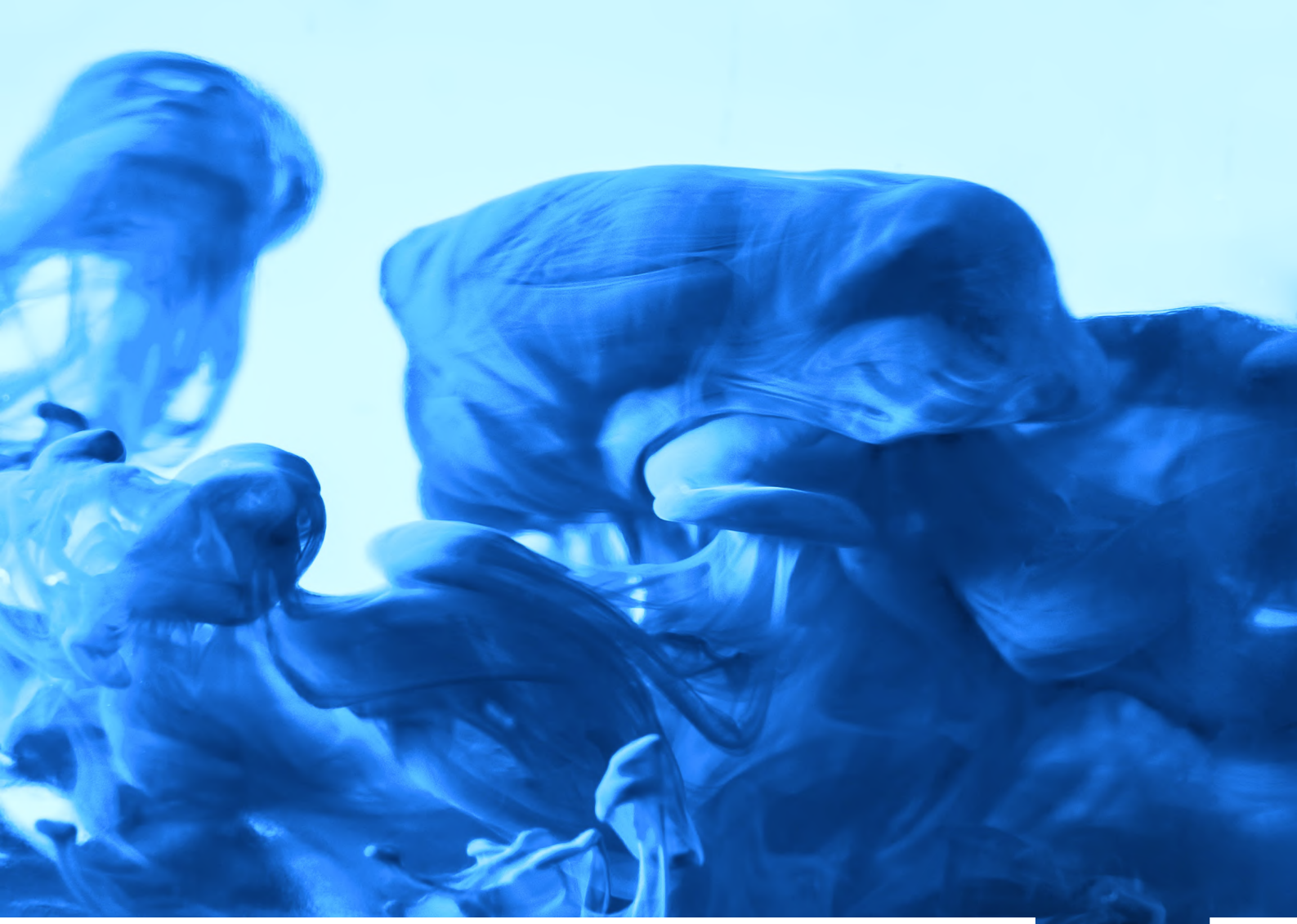
Разработчик одноименного сервиса аудита выплат за делегирование tez, биржи атомарных свопов Atomex, обозревателей TzKT и Better Call Dev.

Madfish Solutions

Разработчики веб-кошелька Temple, DEX Quipuswap и транспайлера Sol2Ligo.

Над онлайн-курсом работали:

Автор	**Павел Скоропляс**
Продюсер	**Светлана Лукина**
Стили	**Дмитрий Бойко**
Иллюстрации	**Кшиштоф Шпак**
Верстка	**Зара Аракелян**
Разработка	**Александр Пупко**
Руководитель	**Влад Лихута**



1

Что такое смарт-контракты и как они работают

Рассказываем о смарт-контрактах Tezos: на каком языке программирования их пишут и как их исполняет виртуальная машина.

В уроке почти нет информации о синтаксисе: на этом этапе важнее понять, как работает виртуальная машина блокчейна.

Что такое смарт-контракты

Смарт-контракт *(Smart contract)* — набор функций и данных, который хранится в блокчейне. Смарт-контракты выполняют разные задачи — от продажи токенов до управления децентрализованными организациями. Например:

- вестинг-контракты отправляют пользователям токены в указанное время. Ранние инвесторы Tezos получают средства из контракта KT1KCYaULopm8i2CBbGF5EHXeXLUr4R6r2dA;
- оракулы опрашивают источники данных и возвращают усредненные цены активов или другую информацию. Так работает оракул Harbringer, который получает цены криптовалют с торговых площадок;
- алгоритмические стейблкоины выпускают токены и меняют стоимость их выпуска в зависимости от курса нативного токена. Смарт-контракт Kolibri корректирует цену выпуска kUSD в зависимости от цены tez.

Смарт-контракты исполняет виртуальная машина (Virtual Machine или VM). Она использует вычислительную мощность блокчейна: смарт-контракты исполняют все узлы сети, но только самый быстрый записывает результат в блок.

Частые вызовы смарт-контрактов могут парализовать блокчейн. Чтобы этого избежать, разработчики протоколов ограничивают максимальный размер смарт-контрактов по объему кода и размеру комиссий.

Как работают виртуальные машины блокчейнов

Виртуальные машины «понимают» низкоуровневые языки программирования — байт-код. Они интерпретируют такой код быстрее, чем команды на языках вроде JavaScript или Python. Это можно показать на примере одной фразы, записанной разными словами:

- **низкоуровневый язык** — 2 + 2 = 4;
- **высокоуровневый язык №1** — два плюс два равно четыре;
- **высокоуровневый язык №2** — Если Сложить два и два, то получим четыре;
- **высокоуровневый язык №3** — в результате суммирования двойки и двойки получается четыре.

При чтении мы автоматически переводим последние строчки в самый удобный для понимания вариант — 2 + 2 = 4. В программировании так же: компилятор переводит высокоуровневый язык в байт-код.

Виртуальная машина Tezos работает с байт-кодом Michelson. Это низкоуровневый язык программирования с прямым доступом к стеку — структуре данных с быстрым доступом к информации. Код на Michelson выглядит так:

```
parameter unit ;
storage unit ;
code { CAR ;
       PUSH int 3 ;
```

```
      PUSH int 3 ;
      IFCMPEQ { DROP } { DROP } ;
      UNIT ;
      NIL operation ;
      PAIR }
// Функция if. Операторы PUSH дважды добавляют число 3 в стек. Оператор IFCMPEQ
   сравнивает два первых элемента в стеке, а затем исполняет команды { DROP }
   и выводит результат операции.
```

Опытные блокчейн-разработчики чаще всего пишут смарт-контракты на Michelson. Новичкам лучше использовать высокоуровневые языки программирования с набором библиотек и человекочитаемым синтаксисом.

Языки програмирования смарт-контрактов на Tezos: SmartPy, LIGO и Lorentz

Участники экосистемы Tezos разработали несколько высокоуровневых языков программирования. Самые популярные из них:

- **SmartPy** — объектно-ориентированный язык на основе Python. Поддерживает онлайн-компилятор SmartPy.io, в котором можно разрабатывать, тестировать и публиковать смарт-контракты.
- **LIGO** — императивный язык программирования с простой системой типов и онлайн-компилятором ide.ligolang.org. Существует несколько диалектов с синтаксисом распространенных языков: PascaLIGO (Pascal), CameLIGO (OCaml), ReasonLIGO (ReasonML) и jsLIGO (JavaScript).
- **Lorentz** — встраиваемый предметно-ориентированный язык на базе Haskell. С помощью Lorentz разработчики могут напрямую работать со стеком Michelson.

В этом курсе мы будем использовать язык с самым простым синтаксисом — PasaLIGO.

Простой смарт-контракт с одной точкой входа

Основа смарт-контракта на LIGO — точка входа *(entry point)*. Это главная функция смарт-контракта *(main function)*, которая принимает входящую транзакцию и вызывает другие функции.

```
function имя_функции (const параметр_входящей_транзакции : тип; const хранилище :
тип) : тип_результата is результат
```

Входные параметры main function:

- **параметр входящей транзакции.** Функция использует его при исполнении кода;
- **хранилище (storage).** Информация, которую нужно хранить в блокчейне до следующего исполнения смарт-контракта, например количество токенов или записи о пользователях. Разработчик задает значение storage при развертывании смарт-контракта. В дальнейшем содержимое хранилища может менять только смарт-контракт.

Выходные параметры main function:

- список выполненных операций, например детали транзакций, которые отправил смарт-контракт;
- результаты выполнения функций, которые прописал разработчик.

Пример смарт-контракта с функцией-инкрементом, которая принимает число и увеличивает его на 1:

```
function main (const number : int; const storage : int) : list (operation) * int
is ((nil : list (operation)), number + 1)
```

Код функции:

- *`function main (const number : int; const storage : int)`* — объявляем main function с входными параметрами number и storage. Когда пользователь отправляет контракту число, главная функция воспримет его как number и исполняет код;
- *`: list (operation) * int`* — определяем тип результата, который возвращает функция. В этом случае — пара из списка выполненных операций и число-результат;
- *`is ((nil : list (operation)), number + 1)`* — определяем результат выполнения функции:
  - *`nil : list (operation)`* — возвращает пустой список;
  - `,` — служит разделителем для значений типа tuple;
  - *`number + 1`* — однострочная функция, увеличивает число на 1.

Смарт-контракт можно модифицировать: реализовать декремент, возведение в квадрат или другую математическую операцию. У него одна точка входа, которая активирует единственную цепочку функций. Такой смарт-контракт не может выполнить несколько задач, например принять депозит, вернуть баланс и отправить транзакцию.

Смарт-контракт с несколькими точками входа

Виртуальная машина начинает исполнение смарт-контракта с *`main function`*. Она может выполнить только те функции, которые вызывает главная точка входа.

Разработчики увеличивают количество операций, создавая псевдо-точки входа внутри главной функции. Для этого нужно:

1. Объявить псевдо-точки входа и тип параметра, с которым они будут работать.
2. Описать функции, которые будет вызывать каждая псевдо-точка.
3. Использовать оператор case в главной функции. Он укажет виртуальной машине, какую функцию вызывать в случае обращения к псевдо-точке входа.

Пример смарт-контракта с псевдо-точками — калькулятор, который принимает название математической операции и два числа, а возвращает результат вычислений. Смарт-контракт не записывает значения в storage, потому что нам для выполнения математических операций не нужно помещать данные в блокчейн.

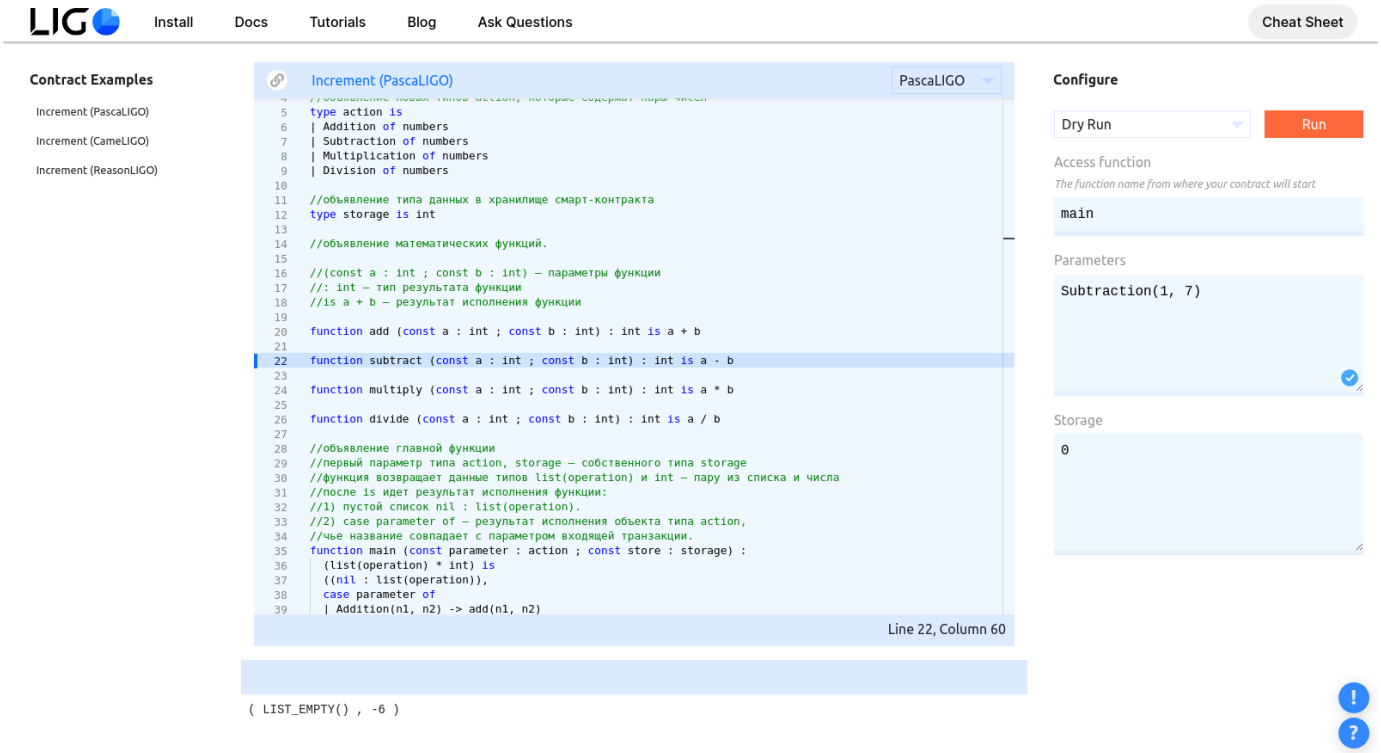
```
// объявление типа numbers, который содержит пару из двух чисел (tuple)
type numbers is (int * int)

// объявление типов action, которые содержат пары чисел
type action is
| Addition of numbers
| Subtraction of numbers
| Multiplication of numbers
| Division of numbers

// объявление типа данных в хранилище смарт-контракта
type storage is int

// объявление математических функций

// (const a : int ; const b : int) — параметры функции
// : int — тип результата функции
// is a + b — результат исполнения функции

function add (const a : int ; const b : int) : int is a + b

function subtract (const a : int ; const b : int) : int is a - b

function multiply (const a : int ; const b : int) : int is a * b

function divide (const a : int ; const b : int) : int is a / b

// объявление главной функции
// назначаем первому параметру тип action, параметру store — тип storage
// функция возвращает данные типов list(operation) и int — пару из списка и числа
   после is идет результат исполнения функции:
// 1) пустой список nil : list(operation).
```

```
// 2) const result : int = – запись результата исполнения функции в константу
      result.
// 3) case parameter of – результат исполнения объекта типа action,
      чье название совпадает с параметром входящей транзакции.
function main (const parameter : action ; const store : storage) :
  (list(operation) * int) is block {
    const result : int =
    case parameter of
  | Addition(n1, n2) -> add(n1, n2)
  | Subtraction(n1, n2) -> subtract(n1, n2)
  | Multiplication(n1, n2) -> multiply(n1, n2)
  | Division(n1, n2) -> divide(n1, n2)
  end;

  // вывод результата исполнения главной функции: пустой список операций
     и значение result
  } with ((nil : list(operation)), result)
```

Если отправить смарт-контракту запрос с параметром *`Multiplication(3, 9)`*, он вернет число 27. Виртуальная машина исполнит контракт так:

1. Сопоставит параметр *`Multiplication`* с вариантами псевдо-точек под оператором case.
2. Перейдет к функции *`multiply`*, которую разработчик описал в начале контракта.
3. Подставит вместо констант *a* и *b* параметры запроса — 3 и 9.
4. Исполнит функцию *`multiply`* и запишет ее в *`store`*.
5. Завершит исполнение главной функции — вернет пустой список и новое значение *`store`*.

Для проверки смарт-контракта скопируйте код калькулятора в редактор на ide.ligolang.org. Выберите значение Dry Run из выпадающего списка на панели **Configure**. В поле **Parameters** введите название математической операции и два параметра, а затем нажмите кнопку **Run**.



Измените значение в полях Parameters и Storage и посмотрите, как меняется результат.

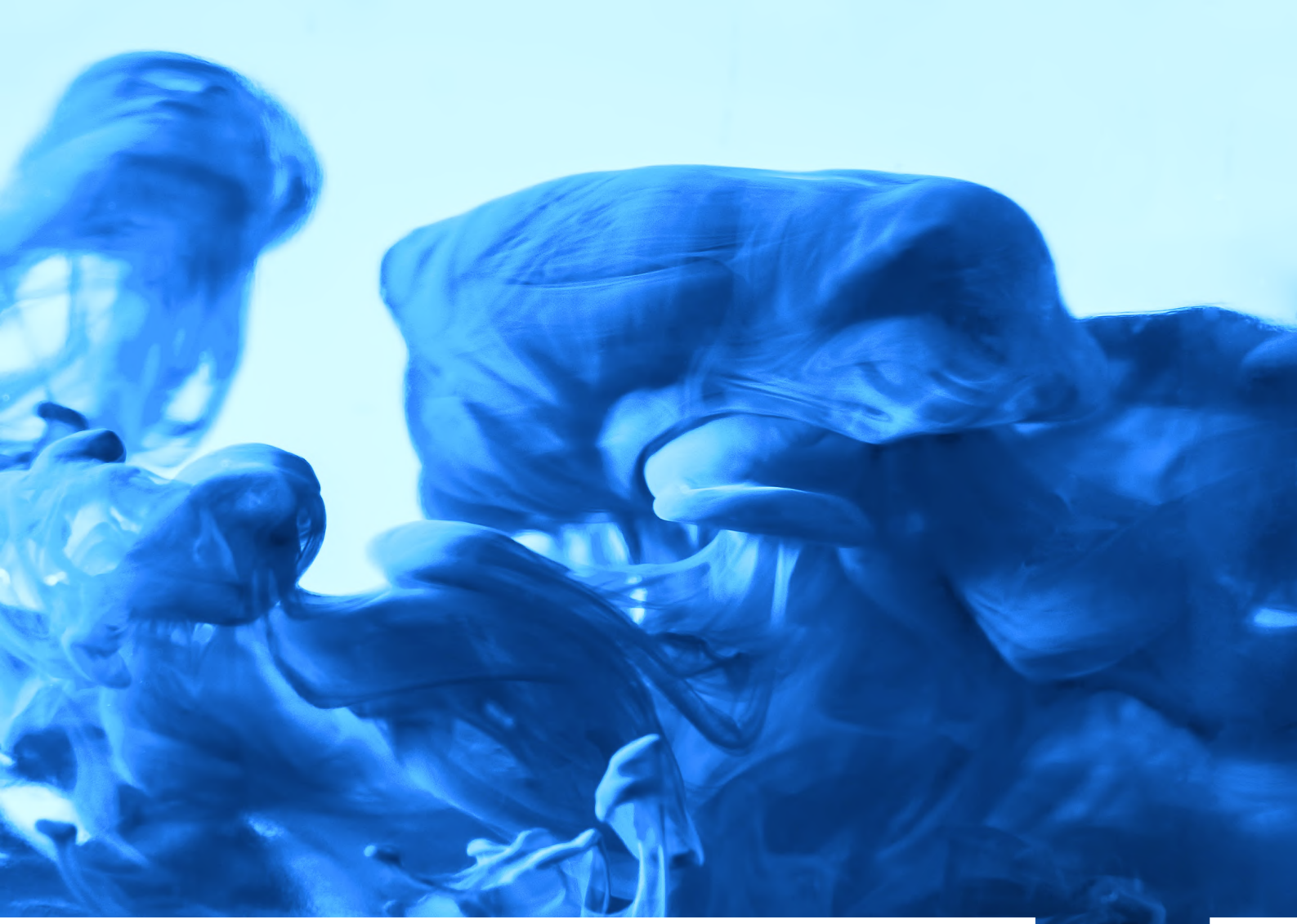
Подводим итоги

Смарт-контракт — это код в блокчейне. Виртуальная машина исполняет его, когда получает транзакцию с нужными параметрами.

Разработчики пишут смарт-контракты на высокоуровневых языках программирования с синтаксисом Python, Pascal, JS или Haskel. Опытные разработчики часто пишут байт-код на Michelson.

Смарт-контракты на языке PascalLIGO состоят из переменных и функций. Виртуальная машина исполняет контракт начиная с точки входа — главной функции *`main`*. В нее можно вставить псевдо-точки входа — дополнительные функции.

Смарт-контракт всегда возвращает результат исполнения: список операций и значение хранилища **storage**.



2

Среда разработки и базовые типы данных LIGO

На прошлом уроке мы запустили простой смарт-контракт в онлайн-компиляторе LIGO. В нем не получится создать большие проекты: там нельзя сохранить код, добавить стороннюю библиотеку или вызвать другой смарт-контракт.

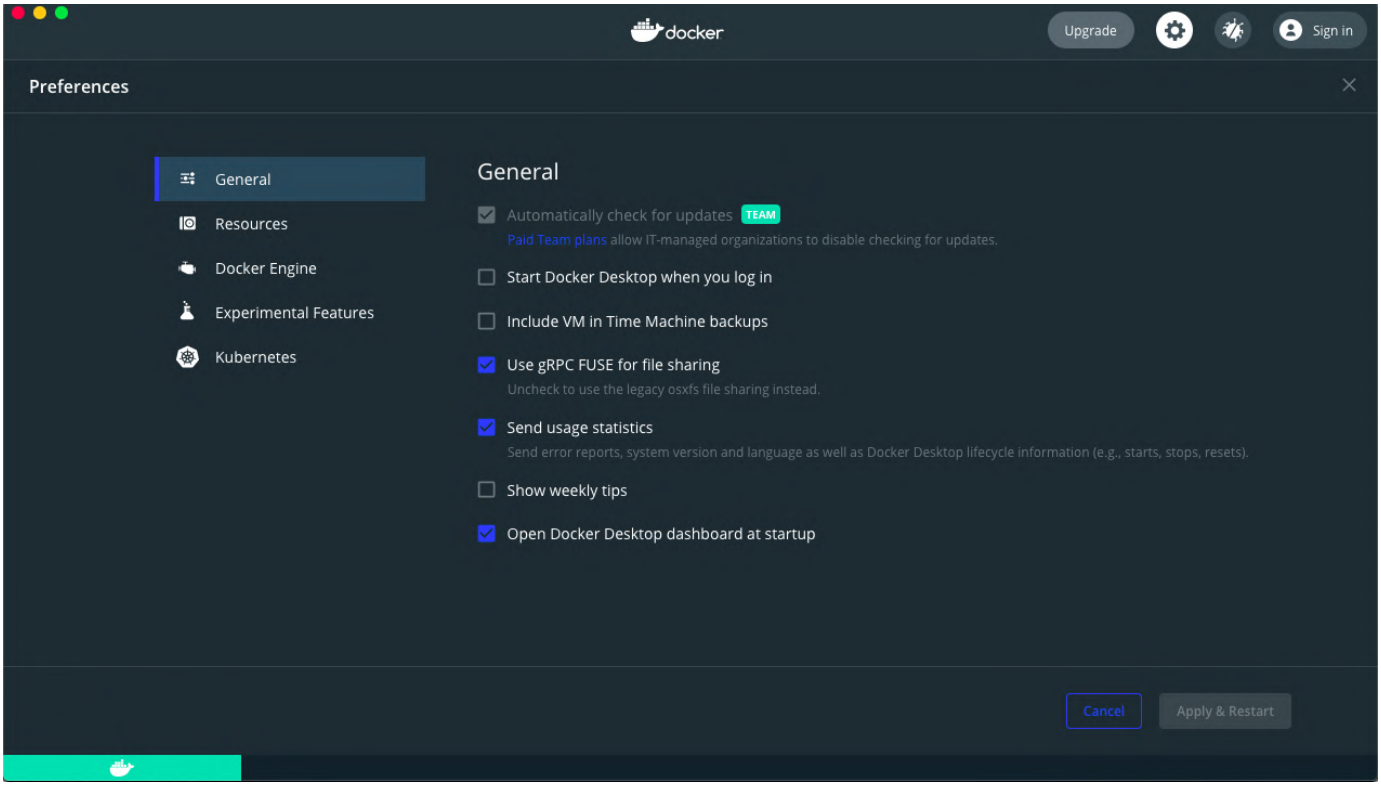
Программисты пишут код в средах разработки — текстовых редакторах с поддержкой плагинов и библиотек. На этом уроке мы установим среду разработки VS Code и расскажем о базовых типах данных LIGO.

Установка и настройка среды разработки

Мы будем использовать редактор Visual Studio Code (VS Code) с плагином pascaligo-vscode.

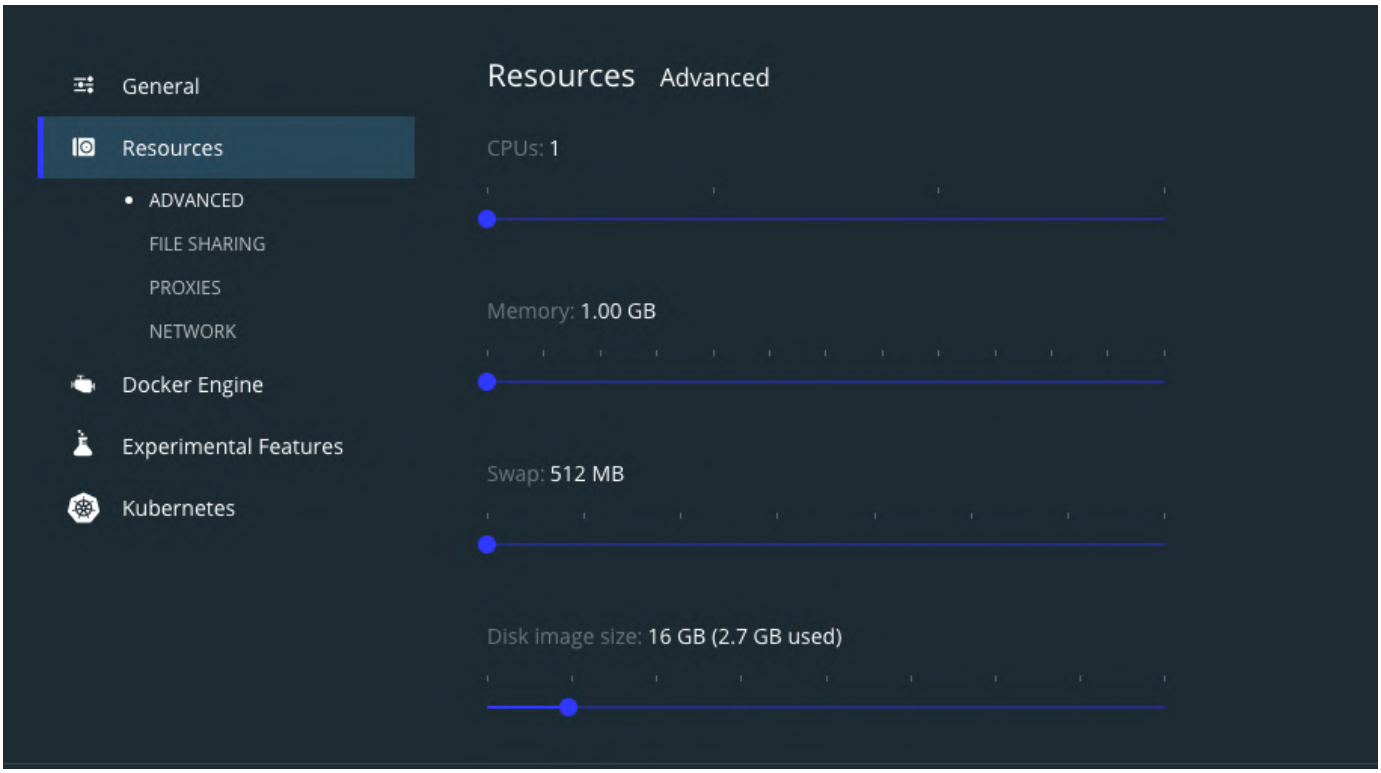
Порядок установки среды разработки:

1. Скачайте редактор с официального сайта Visual Studio Code.
2. Установите приложение и загрузите плагин pascaligo-vscode.
3. Установите Docker. Он нужен для запуска библиотеки LIGO в виртуальной машине. Так компьютер сможет компилировать и исполнять код на PascalLIGO.
4. Запустите Docker и дождитесь, пока в левом нижнем углу приложения появится зеленая полоска.



Во время установки могут возникнуть такие проблемы:

- **при запуске Docker выдает ошибку «This computer doesn't have VT-X/AMD-v enabled».** Загрузите BIOS. Включите виртуализацию VT-X или AMD-V;
- **после запуска Docker компьютер медленно работает.** Откройте настройки программы, перейдите во вкладку **Resources** и уменьшите ресурсы компьютера, выделенные для Docker. Нажмите кнопку **Apply and restart** в нижнем правом углу, чтобы применить изменения.



Установите контейнер LIGO. Для этого откройте терминал и выполните команду:

- для MacOS и Linux:
  `docker run --rm -v "$PWD":"$PWD" -w "$PWD" ligolang/ligo:0.19.0`
- для Windows:
  `docker run --rm -v "%CD%":/cd -w /cd ligolang/ligo:0.19.0`

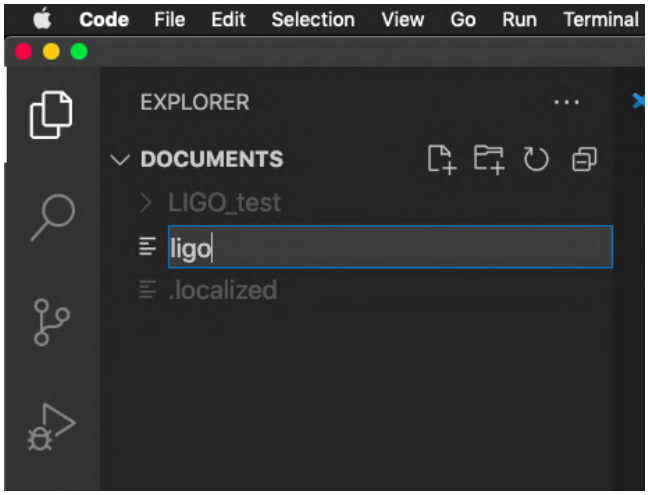
Docker загрузит контейнер и выдаст встроенную справку LIGO.

```
Unable to find image 'ligolang/ligo:0.19.0' locally
0.19.0: Pulling from ligolang/ligo
801bfaa63ef2: Pull complete
d5fb037205c7: Pull complete
837a824806a0: Pull complete
b97f32fe54fe: Pull complete
Digest: sha256:c4ca5868b40edc96fe9112f07da1340f006b21e2412dc53b833aa2dbc615ab02
Status: Downloaded newer image for ligolang/ligo:0.19.0
NAME
       ligo

SYNOPSIS
       ligo COMMAND ...

MORE HELP
       Use `ligo COMMAND --help' for help on a single command.

DOCUMENTATION
       https://ligolang.org/docs/intro/introduction
```

Проверьте, все ли работает. Выполните в терминале команду
`ligo --version`

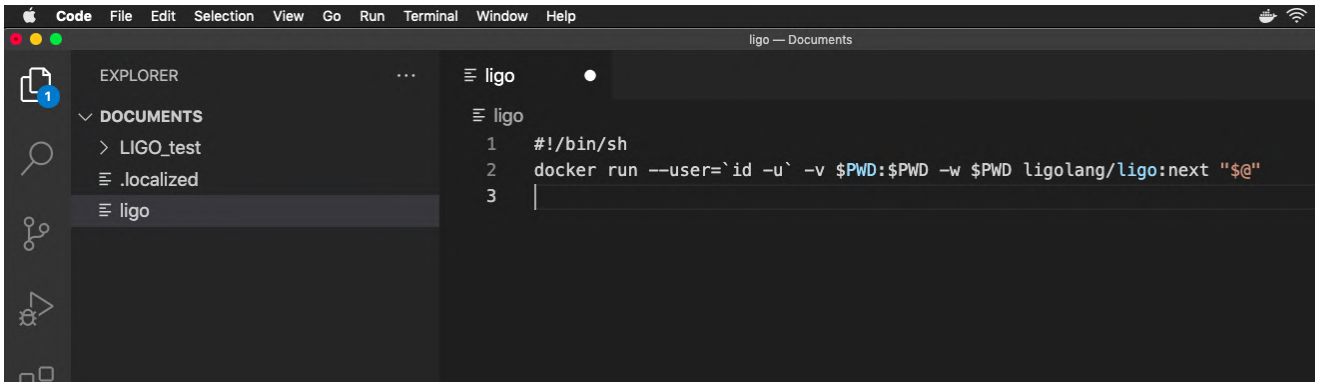
Если терминал выдает ошибку *«ligo: command not found»*, значит Docker преждевременно закрывает контейнер LIGO. Решение проблемы:

1. Откройте VS Code и создайте файл с названием ligo.



2. Вставьте в него следующий код:

```
#!/bin/sh
docker run --user=`id -u` -v $PWD:$PWD -w $PWD ligolang/ligo:next "$@"
```



3. Сохраните документ и откройте терминал. С помощью команды *cd* перейдите в папку с файлом ligo:

```
cd ~/Documents
```

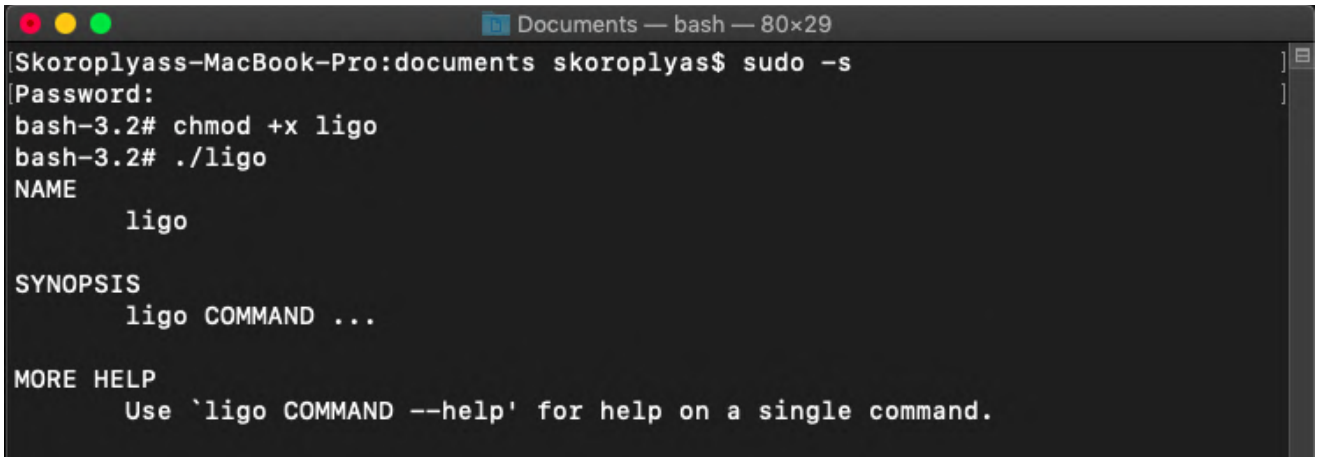
4. Если у вас Linux или Mac OS, включите режим суперпользователя:

```
sudo -s
```

5. В терминале выполните команды:

```
chmod +x ligo
./ligo
```

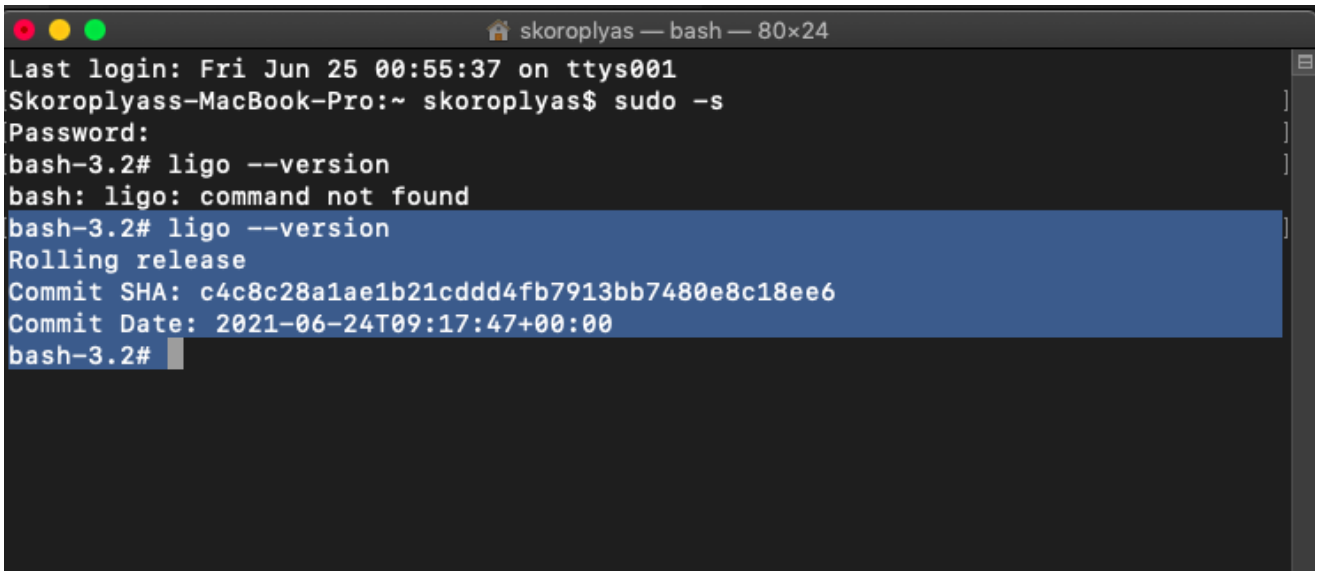
Они привяжут запуск контейнера LIGO к команде ligo.



6. Сделайте файл исполняемым на уровне системы. В Linux и Mac OS откройте проводник, и перенесите файл ligo в папку /usr/local/bin на системном диске. В Windows откройте терминал и выполните команду:

   ```
   $env:path += ";c:\[путь к папке с файлом ligo]"
   ```

7. Проверьте работу LIGO: снова введите в терминале команду *`ligo --version`*.



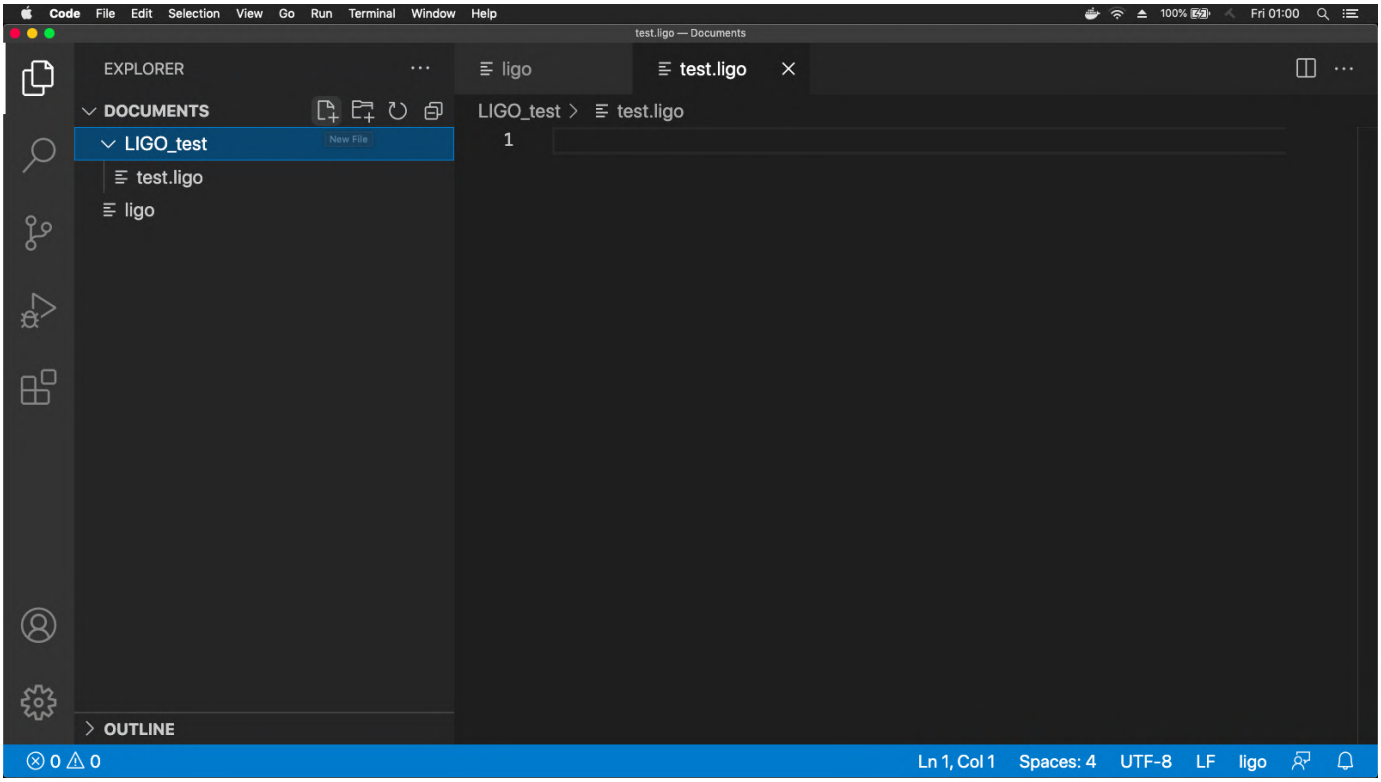
На черном фоне — ошибка при выполнении команды ligo --version до переноса файла ligo в папку bin.
На синем фоне — успешное выполнение команды после переноса файла.

Компиляция пробного контракта

Откройте VS Code и создайте новый проект:

1. Нажмите на кнопку **Open...**.
2. Выберите папку, в которой сохраните тестовый проект LIGO.
3. Создайте в ней папку LIGO_test.
4. Откройте новую папку.

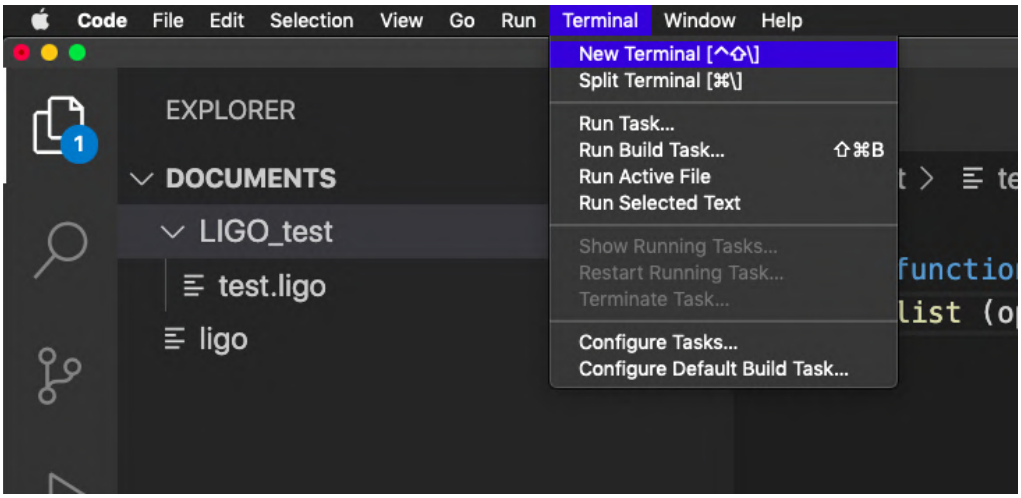
Слева отображаются папки и файлы текущего проекта. Создайте в этом списке пустой файл с названием test.ligo. По расширению .ligo VS Code поймет, что нужно задействовать плагин для подсветки синтаксиса и использовать команды PascaLIGO.



Вставьте в файл test.ligo функцию из первого урока.

```
function main (const number : int; const storage : int) : list (operation) * int
is ((nil : list (operation)), number + 1)
```

Попробуйте скомпилировать этот контракт. Откройте терминал VS Code.

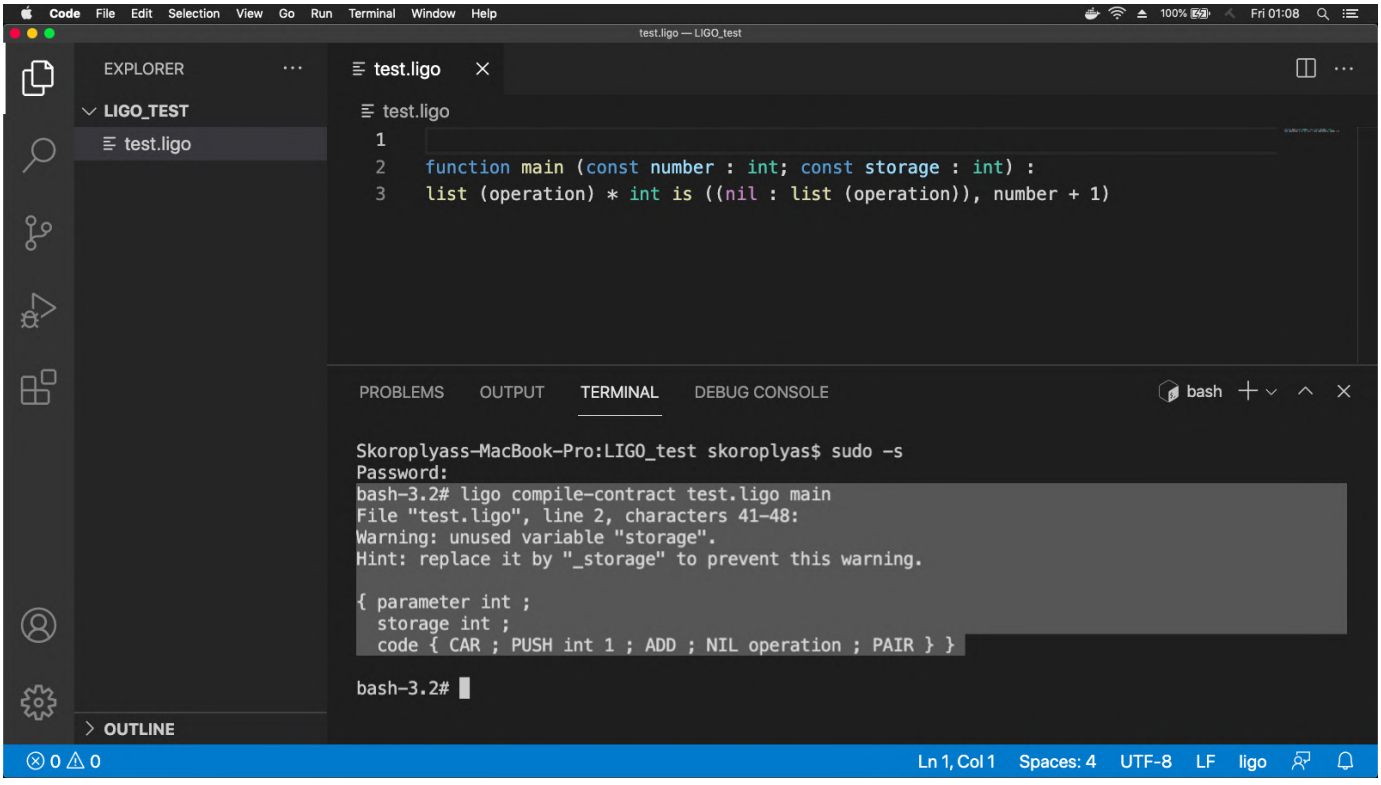


Введите в нем следующую команду и нажмите Enter:

```
ligo compile-contract test.ligo main
```

Если у вас Linux или Mac OS, терминал может выдать ошибку «root/ligo». В таком случае сначала выполните команду `sudo -s`, чтобы дать терминалу право исполнить скрипт ligo в папке bin.

VS Code переведет код PascaLIGO из test.ligo в код Michelson и проверит, можно ли исполнить функцию `main`.



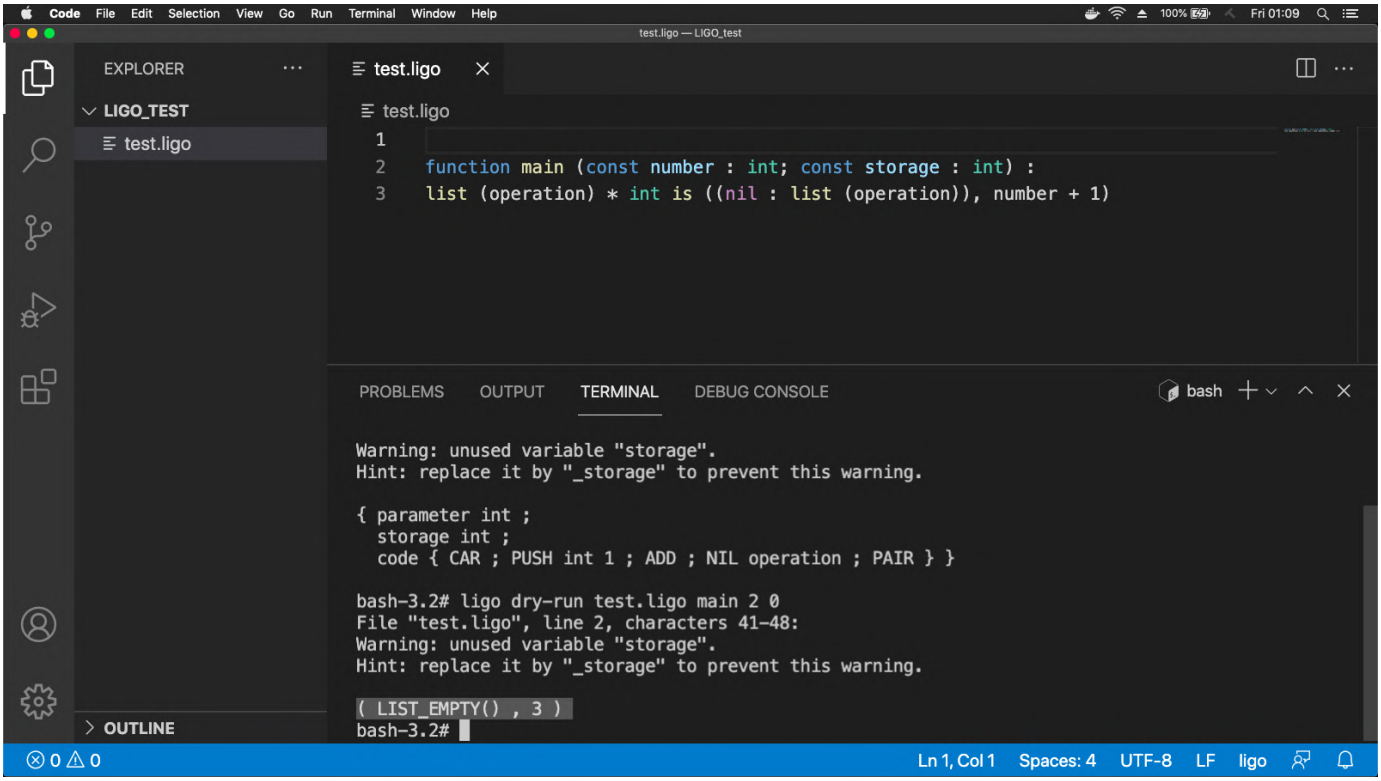
Теперь проверьте работу смарт-контракта с помощью команды *dry-run* (репетиции). Она выполнит контракт с заданными параметрами внутри виртуальной машины Docker.

Для «репетиции» тестового контракта введите в терминале команду:

```
ligo dry-run test.ligo main 2 0
```

- *main* — название точки входа;
- *2* — первый входящий параметр;
- *0* — значение хранилища.

Вы можете использовать другие числа для проверки этого контракта.



При исполнении dry-run компилятор выдаст результат исполнения функции main с заданными параметрами.

Константы, переменные и базовые типы данных LIGO

Для хранения информации LIGO использует:

- **const (константы)**. Смарт-контракт не может изменить значение константы во время исполнения функций. Чаще всего в них записывают параметры, которые поступают на вход главной функции;
- **var (переменные)**. Смарт-контракт может изменить значение переменной в любой момент. Обычно переменные используют в loop-функциях, которые контракт вызывает несколько раз.

Константам и переменным нужно присвоить тип данных, чтобы компилятор понимал, как с ними работать. LIGO использует такие базовые типы данных:

- *int* — неограниченное по размеру целое число, может быть отрицательным;
- *nat* — неограниченное по размеру натуральное число, не может быть отрицательным;
- *string* — строка, содержит печатные символы: латинские буквы, цифры, знаки препинания и некоторые спецсимволы вроде | и ~;
- *bool* — логическая переменная true или false.

Пример объявления констант и переменных:

```
// const [название] : [тип данных] = [значение]
// var [название] : [тип данных] := [значение]

const a : int = 50
const b : nat = 3n
var c : string := “Example”
var d : bool := True
```

Иногда константе или переменной нельзя присвоить значение до исполнения смарт-контракта. Пример: контракт получает входящий параметр, а затем использует его для расчетов. В таком случае нужно объявить пустую константу или переменную и присвоить ей значение позже.

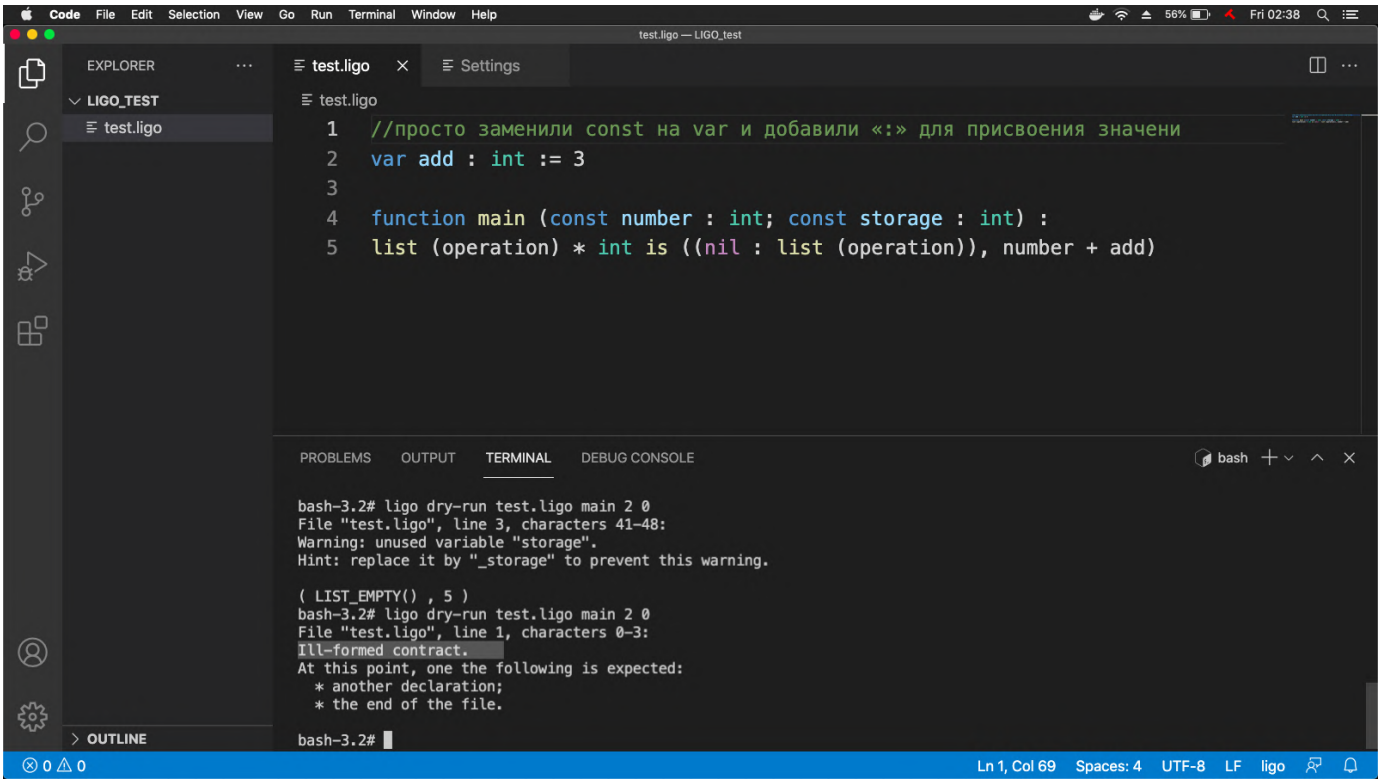
В примере ниже мы объявили константу **number** типа *int*. Во время исполнения кода смарт-контракт присвоит ей значение, которое отправит пользователь.

```
function main (const number : int; const storage : int) : list (operation) * int
is ((nil : list (operation)), number + 1)
```

Объявлять константы можно в любом месте смарт-контракта, а переменные — только внутри функций. Например, мы можем вынести число 1 из главной функции в примере выше. Для этого нужно объявить константу **add** в начале контракта, присвоить ей значение 1 и заменить единицу на **add**.

```
const add : int = 1

function main (const number : int; const storage : int) :
list (operation) * int is ((nil : list (operation)), number + add)
```

Но если сделать **add** переменной, при попытке исполнить код компилятор выдаст ошибку «ILL FORMED CONTRACT» — неправильно сформированный контракт.



Продвинутые типы и псевдонимы

Кроме базовых типов данных LIGO поддерживает несколько *продвинутых типов*:

- структуры данных — tuple, record и map для передачи нескольких значений в одном параметре;
- специальные типы — адреса, балансы tez, варианты, timestamp.

Разработчики создают такие типы с помощью псевдонимов (alias). Псевдонимы также упрощают чтение кода и позволяют сортировать параметры по способам применения.

Пример объявления псевдонимов:

```
// объявляем псевдоним типа int
type numbers is int

// объявляем константу типа numbers и присваиваем ей значение
const a : numbers = -5

// объявляем константу admin типа user_id
type user_id is nat
const admin : user_id = 1n
```

Часто используемые структуры данных

Главная функция принимает один параметр, поэтому базовых типов вроде int и nat недостаточно для работы сложных смарт-контрактов. Это ограничение можно обойти с помощью *структур данных*, которые возвращают несколько значений в одном параметре.

***tuple* или «кортеж»** хранит два и более значений заданного типа. В примере ниже мы задаем тип cube, в который можно записать длину сторон куба.

```
// объявляем псевдоним cube, чтобы хранить в нем длину трех граней куба
type storage is int

type cube is int * int * int

// объявляем в функции входящий параметр — константу side типа cube
function main (const side : cube ; const _store : storage) :
  (list(operation) * int) is block {

// объявляем константу result и присваиваем ей значение: умножение трех значений,
   которые хранятся в кортеже side
    const result : int = side.0 * side.1 * side.2
    } with ((nil : list(operation)), result)
```

record хранит несколько констант или переменных разных типов. Например, в константу типа user можно записать аргументы: id типа int, is_admin типа bool и name типа string:

```
// объявляем тип user
type user is
  record [
    id       : nat;
    is_admin : bool;
    name     : string
  ]

// объявляем константу alice и присваиваем ей значения типа user
const alice : user =
  record [
    id       = 1n;
    is_admin = True;
    name     = "Alice"
  ]
```

map связывает данные в набор пар «ключ–значение». Также в LIGO есть тип данных **big_map**, который оптимизирует загрузку большого количества «ключей-значений» и экономит газ, но не поддерживает итерацию.

```
// объявление псевдоним dims типа tuple, затем – cube_dimensions типа map
type dims is int * int * int
type cube_dimension is map (string, dims)

// объявление map cube, которая содержит два значения: название куба и значения
   длины его граней
const cubes : cube_dimension =
    map [
    "big cube" -> (123243, 1251, 823);
    "small cube" -> (3, 3, 7)
    ]
// записываем в константу big типа dims. Для этого получаем значения из map по
   обращению к ключу “big cube”
const big : option(dims) = cubes["big cube"]
```

unit — тип данных без значения. Он нужен, когда по правилам синтаксиса функция должна принять параметр, но этот параметр не важен для работы кода. Мы используем unit для объявления типов Labeouf, Nike и Yoda в примере ниже.

variant позволяет использовать переменные нескольких типов в зависимости от ситуации и логики смарт-контракта. В примере объявляем variant для создания псевдо-точек входа.

```
// объявляем псевдоним speach и записываем несколько вариантов. Для каждого
   варианта указываем тип параметра, который он будет принимать. В нашем случае
   главная функция ничего не делает с принятым параметром, поэтому для вариантов
   указываем пустой тип unit
type speach is
    Labeouf of unit
   |Nike of unit
   |Yoda of unit

// мы хотим, чтобы контракт возвращал строку, поэтому в типе return указываем,
   что он возвращает список операций и строку
type return is (list(operation) * string)

// объявляем, что функция main должна принять от пользователя параметр и записать
   его в константу word типа speach
```

```
function main (const word : speach; const _store : string) : return is
 ((nil : list (operation)),
// "case word of" сравнивает принятый параметр с вариантами типа speach. Если
   вариант совпадет, контракт вернет указанную строку
  case word of
    Labeouf (_n) -> "DO IT!"
  | Nike (_n) -> "Just do it"
  | Yoda (_n) -> "Do it you can"
  end)
```

Специальные типы данных

Кроме продвинутых структур данных, LIGO использует несколько «блокчейновых» типов данных и функцию *failwith*.

tez — обозначает количество токенов XTZ (tez). При объявлении, присвоении или оперировании типом tez нужно добавлять к числу суффикс «tz» или «tez».

В LIGO также используется суффикс «mutez» для обозначения одной миллионной части tez. Для удобства чтения можно разделять большие числа на порядки с помощью нижнего подчеркивания. Например, «1_000_000tez» виртуальная машина Tezos воспримет как 1 млн tez.

```
// 1 000 000 mutez = 1 tez
// присваиваем t значение 1,5003 tez с помощью суффикса mutez
const t : tez = 1_500_300mutez
```

address — тип данных для хранения адресов Tezos.

```
const my_account : address =
("tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" : address)
```

failwith — функция обработки ошибок и исключений. Например, мы можем применить *failwith*, чтобы калькулятор не делил числа на 7.

```
Division(n1, n2) -> if n2 = 7 then failwith("Error: you can not divide by 7!")
else divide(n1, n2)
```

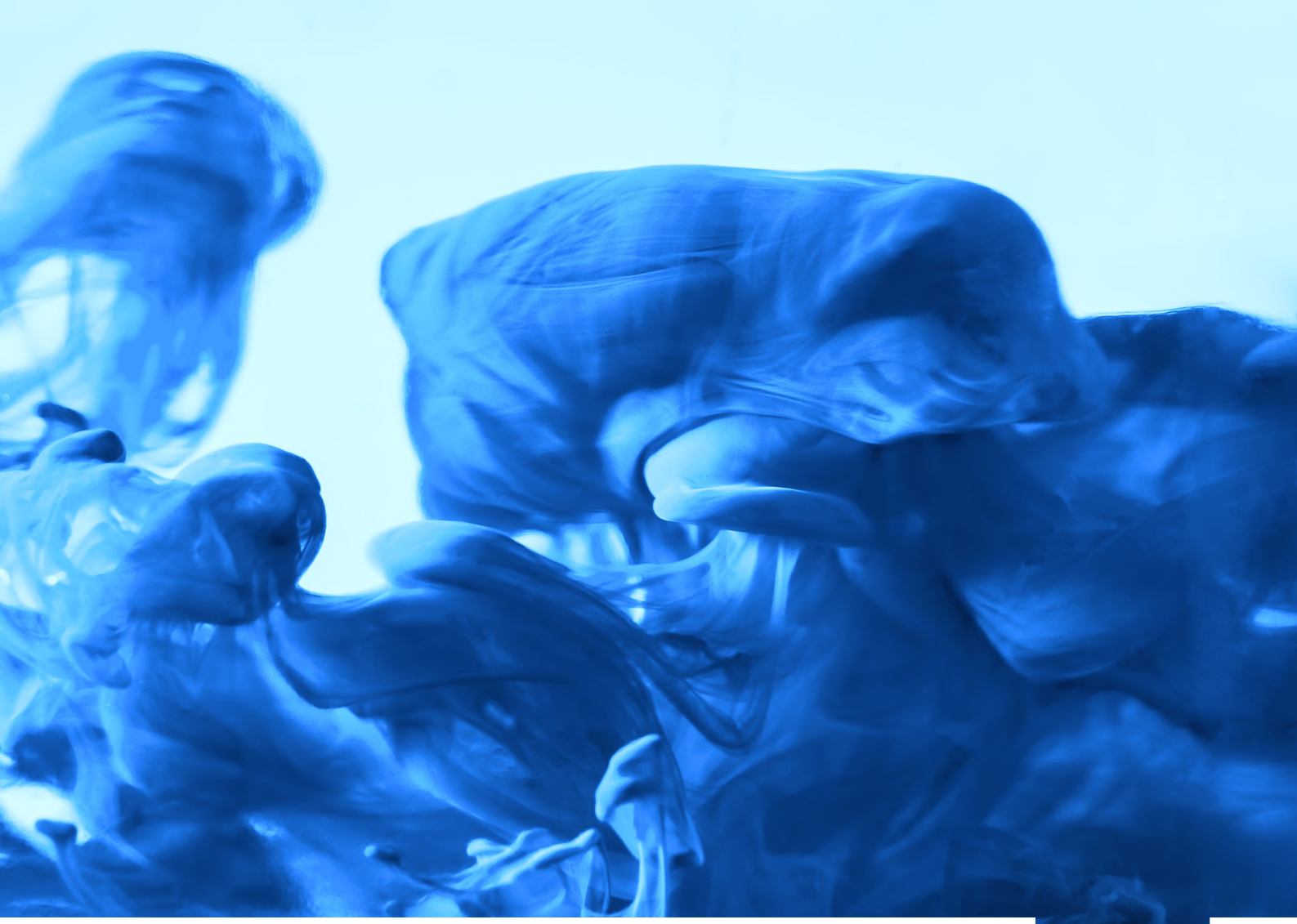
Больше примеров операций с разными типами данных — в документации LIGO.

Подводим итоги

В онлайн-среде разработки нельзя сохранять код, подгружать библиотеки и вызывать другие контракты, поэтому программисты используют Visual Studio Code и контейнер LIGO в Docker.

Базовые типы LIGO — `int`, `nat`, `string` и `bool`. Специальные структуры для хранения данных:

- `tuple` — несколько значений в одной константе или переменной;
- `record` — несколько констант с присвоенными значениями в одной константе или переменной;
- `map` — несколько констант продвинутых типов с привязкой к константе-ключу;
- `tez` — баланс XTZ;
- `address` — адрес в блокчейне Tezos;
- `unit` — тип данных без значения.



3

Развертывание смарт-контракта в тестовой сети с помощью Taquito

На прошлом уроке мы изучили базовые и продвинутые типы данных языка программирования LIGO. Этих знаний хватит, чтобы написать код простого смарт-контракта. Но чтобы другие пользователи могли взаимодействовать с контрактом, его нужно разместить в блокчейне.

Для развертывания смарт-контрактов в блокчейне разработчики используют специальные инструменты. В этом уроке мы установим библиотеку Taquito и с ее помощью опубликуем контракт в тестовой сети Tezos.

Установка и пробный запуск Taquito

Taquito — это TypeScript-библиотека, которая упрощает взаимодействие с блокчейном Tezos и смарт-контрактами. Например, так выглядит развертывание контракта вручную через консоль клиента Tezos:

```
tezos-client originate contract <название контракта>
for <user> transferring <количество tez> from <from_user>
running <путь к файлу с кодом>
--init '<состояние хранилища storage>'
--burn-cap <максимальная комиссия>
```

Скрипт Taquito для развертывания контракта выглядит проще:

```
try {
   const op = await tezos.contract.originate({
     // код смарт-контракта
     code: `{
           `,
     // значение хранилища
     init: ``,
   });
```

Также с помощью Taquito можно отправлять токены, проверять балансы адресов и разработать простой веб-интерфейс для смарт-контракта.

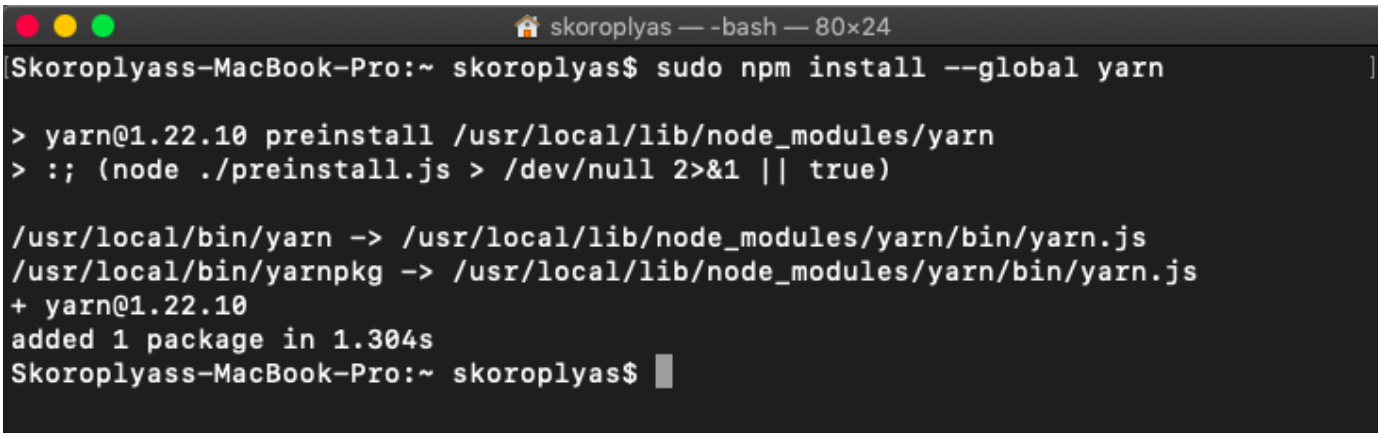
Установка Taquito

Для работы Taquito нужна платформа nodeJS. Скачайте ее с официального сайта и установите на свой компьютер.

Затем установите пакетный менеджер yarn. Он загрузит Taquito и автоматически пропишет его в системе.

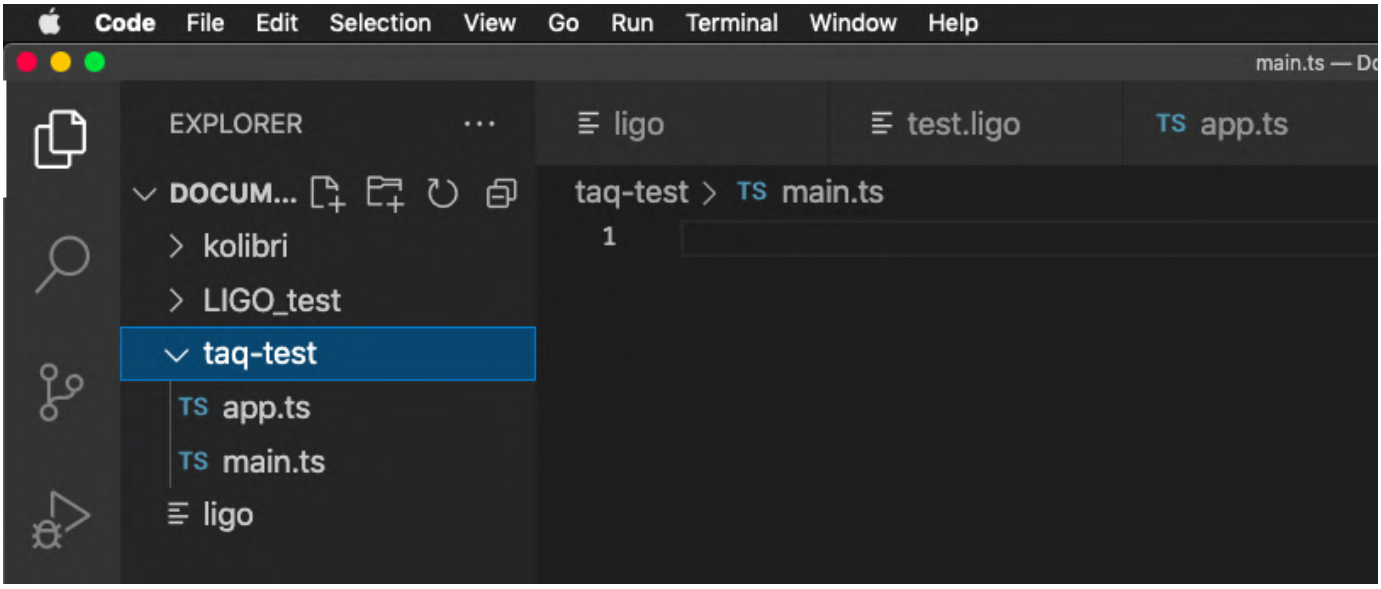
Для этого откройте консоль и выполните команду:

```
sudo npm install --global yarn
```



Средства для работы Taquito готовы, осталось протестировать ее работу. Откройте Visual Studio Code, создайте новую папку taq-test, а в ней — два файла с названиями app.ts и main.ts.

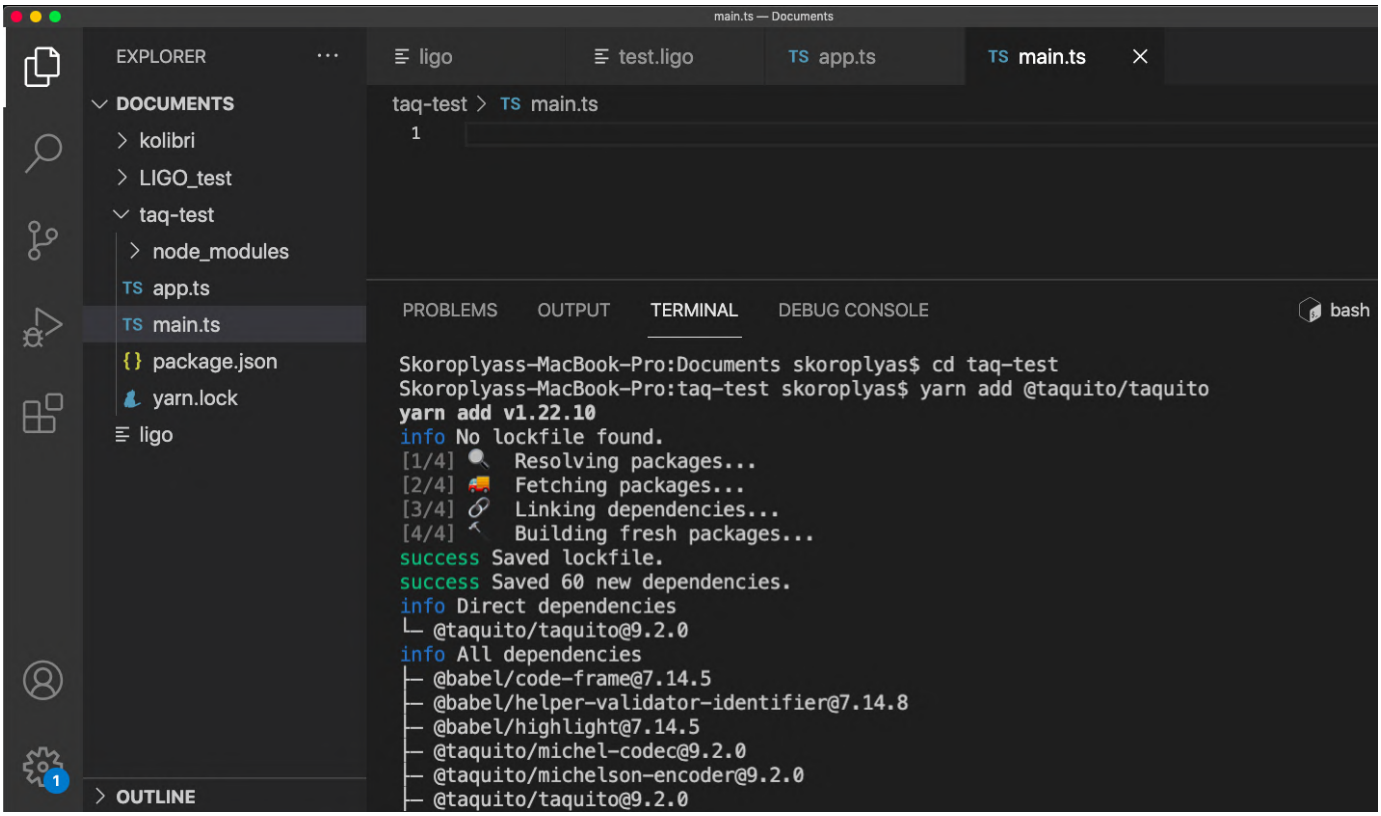
В app.ts мы будем записывать функции Taquito и основной код, а в main.ts — параметры и вызовы функций. Такой подход позволяет упростить структуру кода и не вызывать функции до их объявления.



Откройте терминал в VS Code и с помощью команды cd перейдите в папку taq-test.

Затем добавьте в папку библиотеку Taquito:

```
cd ~/Documents/taq-test
yarn add @taquito/taquito
```



Если вы правильно установили NodeJS и yarn, терминал выдаст длинный список установленных файлов.

Откройте файл app.ts и добавьте в него следующий код:

```
import { TezosToolkit } from '@taquito/taquito';
export class App {

  public async main() { }

}
```

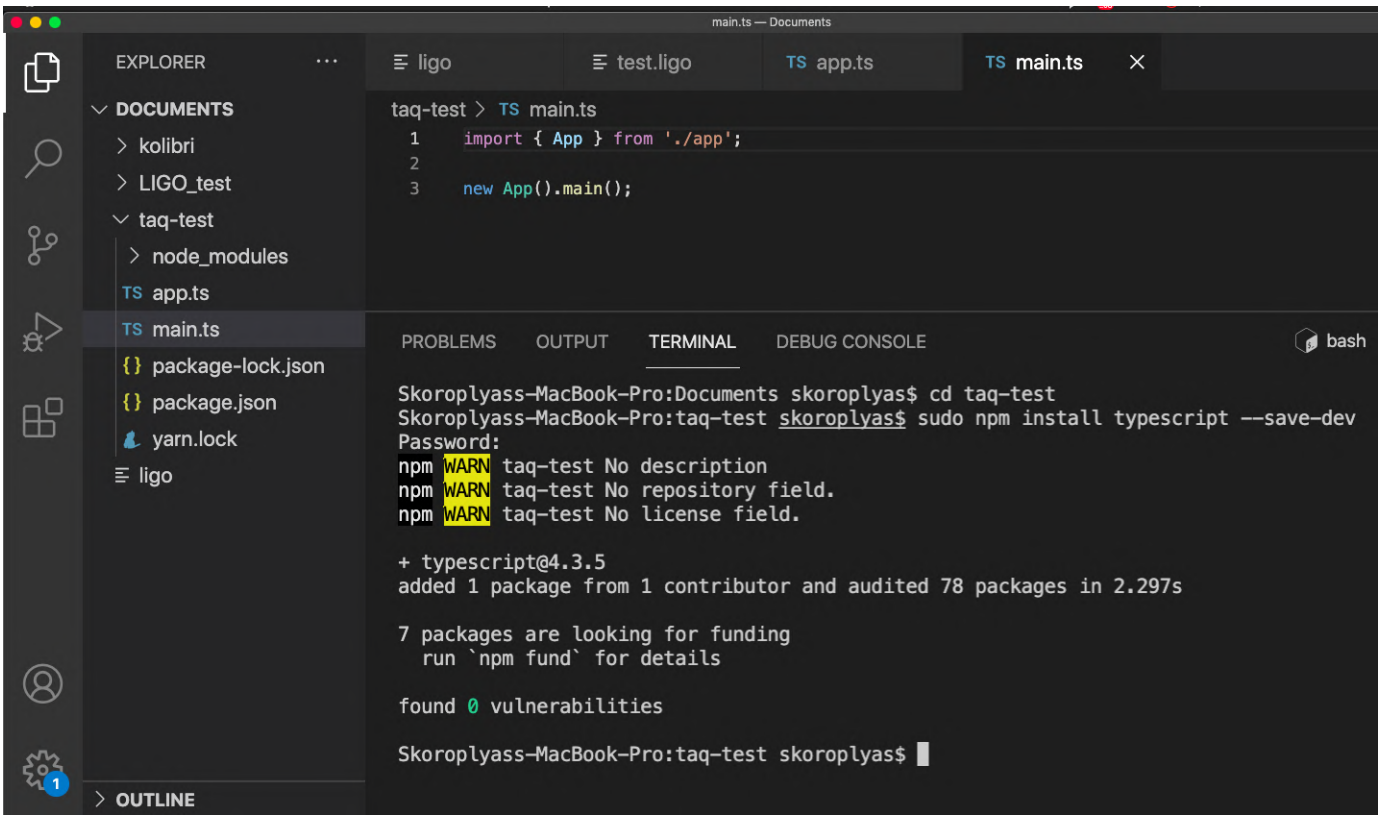
В файл main.ts добавьте:

```
import { App } from './app';

new App().main();
```

Не забудьте сохранить файлы с помощью ctrl+s или ⌘+s, иначе среда исполнения будет считать их пустыми.

Расширение .ts указывает на то, что в файлах содержится код TypeScript. Для его работы нужно установить библиотеку TypeScript. Для этого в терминале VS Code выполните:

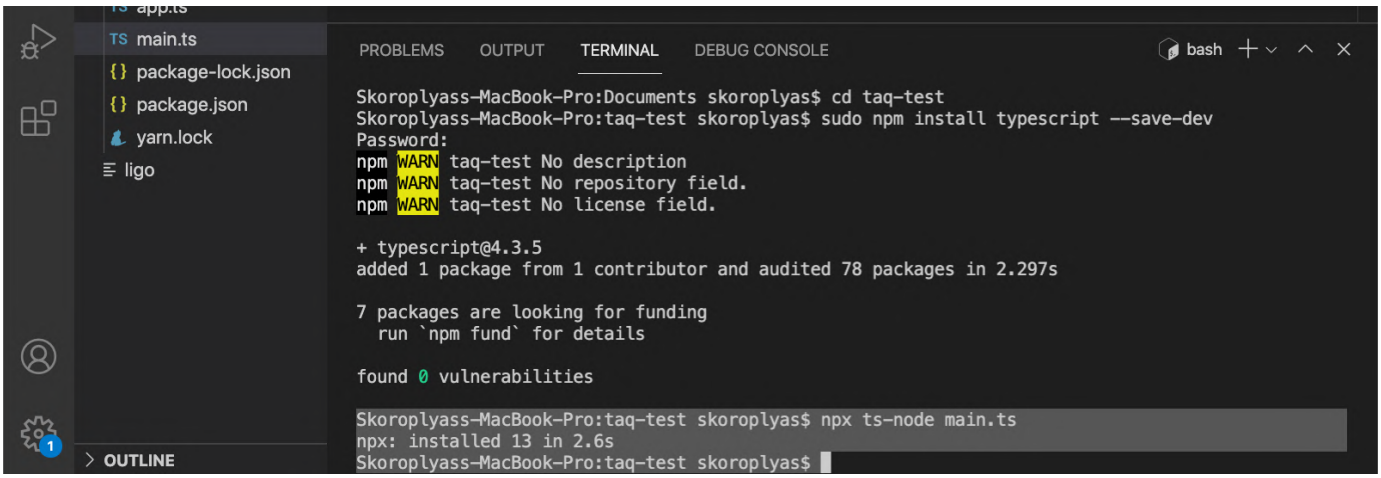
```
sudo npm install typescript --save-dev
```



При успешной установке терминал выдаст 3-4 предупреждения об отсутствии лицензии и описания проекта. Если в списке больше пунктов — вы забыли добавить sudo в начале команды или не находитесь в папке taq-test.

Последний шаг — выполните скрипт main.ts в виртуальном узле. Для этого введите в терминале команду:

```
npx ts-node main.ts
```



Терминал отчитается о запуске npx одной строкой. Это значит, что npx удачно импортировал в приложение Taquito. Так как главная функция app.ts пустая, npx сразу ее завершил. Другой результат при выполнении этого шага означает, что вы допустили ошибку во время установки TypeScript или Taquito.

При работе над смарт-контрактами вам нужно будет каждый раз устанавливать Taquito и TypeScript в папку проекта. В отличие от JavaScript, их нельзя установить глобально.

Получаем данные из блокчейна Tezos через Taquito

С помощью Taquito можно считывать балансы адресов в сети Tezos, отправлять токены и публиковать смарт-контракты. Для этого не нужно поднимать собственный узел Tezos: Taquito подключается к публичным узлам через RPC (Remote Procedure Call или вызов удаленных процедур).

Давайте подключимся к мейннету и запросим баланс адреса одного из бейкеров — валидаторов сети Tezos. Для этого в файле app.ts включим библиотеку TezosToolkit и подготовим константу для RPC-ссылки публичного узла Tezos:

```
import { TezosToolkit } from '@taquito/taquito';
export class App {

// объявляем приватный модификатор tezos типа TezosToolkit
  private tezos: TezosToolkit;

// объявляем конструктор rpcUrl, который будет передавать адрес публичного узла
Tezos в TezosToolkit
  constructor(rpcUrl: string) {
      this.tezos = new TezosToolkit(rpcUrl);
  }

  public async main() { }

}
```

В файл main.ts пропишем RPC-ссылку на публичный узел Tezos Mainnet и передадим ее классу App:

```
import { App } from './app';
// объявляем константу с адресом узла
const RPC_URL = "https://mainnet.smartpy.io"
// запускаем App, который передает главной функции адрес узла
new App(RPC_URL).main();
```

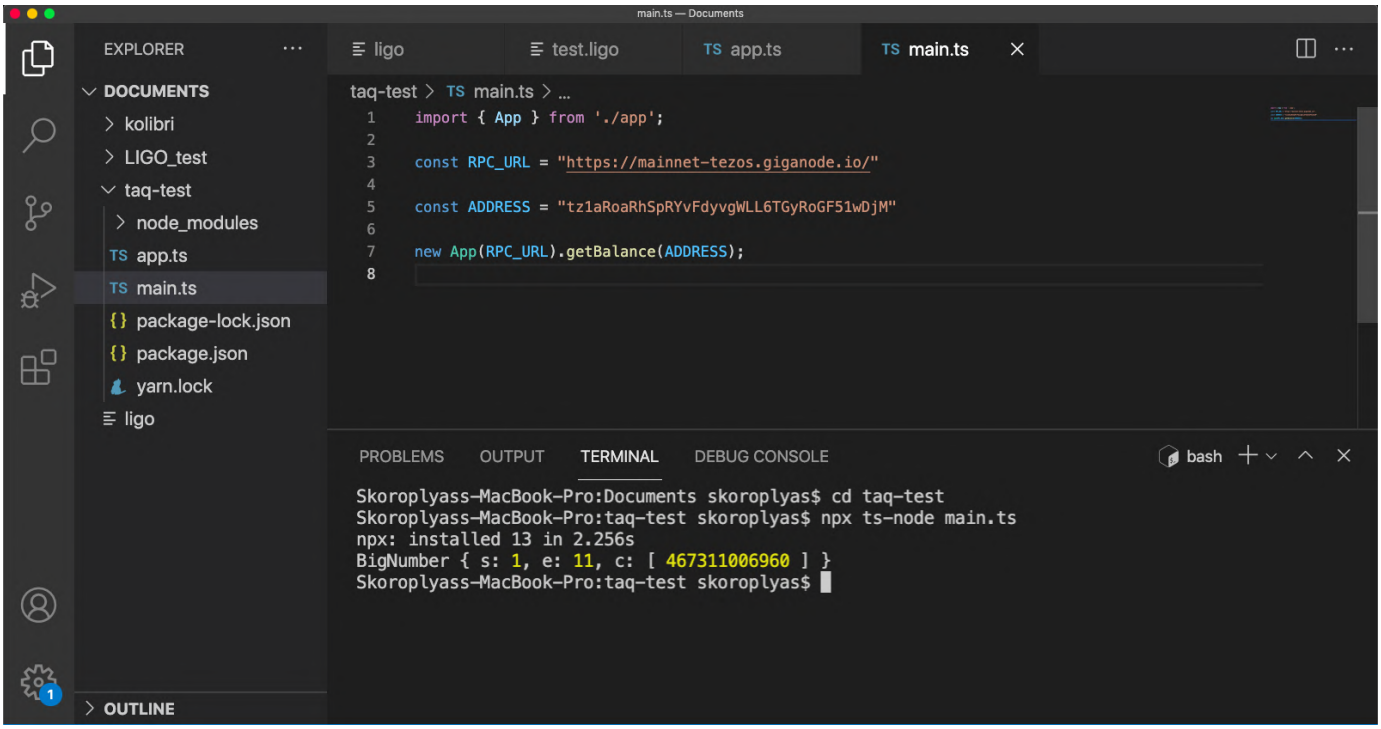
При исполнении команды npx ts-node main.ts Taquito получит ссылку на узел, но ничего с ней не сделает, потому что в app.ts нет методов для взаимодействия с сетью. Добавим в app.ts метод для получения баланса адреса:

```
import { TezosToolkit } from '@taquito/taquito';
export class App {

  private tezos: TezosToolkit;

  constructor(rpcUrl: string) {
      this.tezos = new TezosToolkit(rpcUrl);
  }

   // объявляем метод getBalance с входящим параметром address
  public getBalance(address: string) : void {

// Taquito отправляет узлу запрос баланса указанного адреса. Если узел исполнил
   запрос, скрипт выводит полученное значение в консоль. Если произошла ошибка –
   выдает «Address not found»
  this.tezos.rpc
      .getBalance(address)
      .then(balance => console.log(balance))
      .catch(e => console.log('Address not found'));
  }

  public async main() { }

}
```

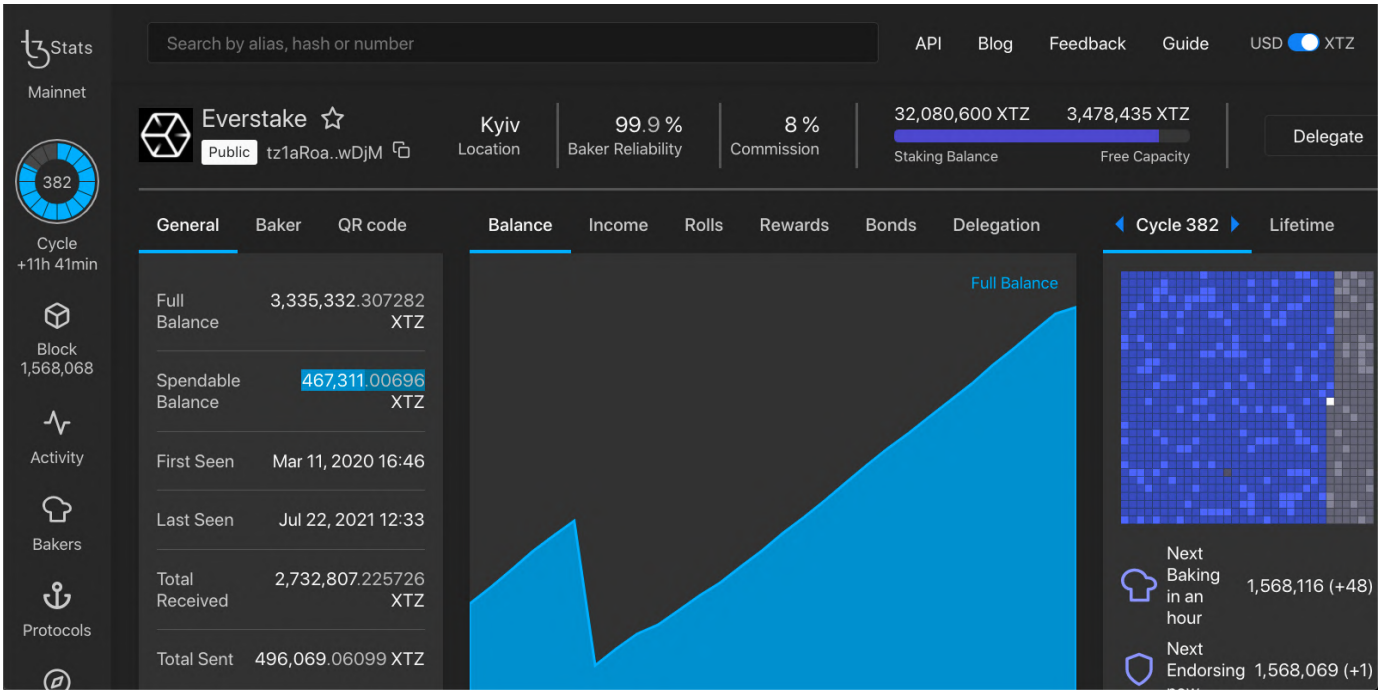
Добавим адрес бейкера и вызовем метод getBalance в скрипте main.ts:

```
import { App } from './app';

const RPC_URL = "https://mainnet.smartpy.io"
// объявляем константу с адресом бейкера Everstake
const ADDRESS = "tz1aRoaRhSpRYvFdyvgWLL6TGyRoGF51wDjM"
// запускаем App, передаем ему ссылку на узел, вызываем метод getBalance и
   передаем ему адрес
new App(RPC_URL).getBalance(ADDRESS);
```

Запустим скрипт с помощью команды npx ts-node main.ts.



Taquito вернет количество свободных tez на адресе Everstake.

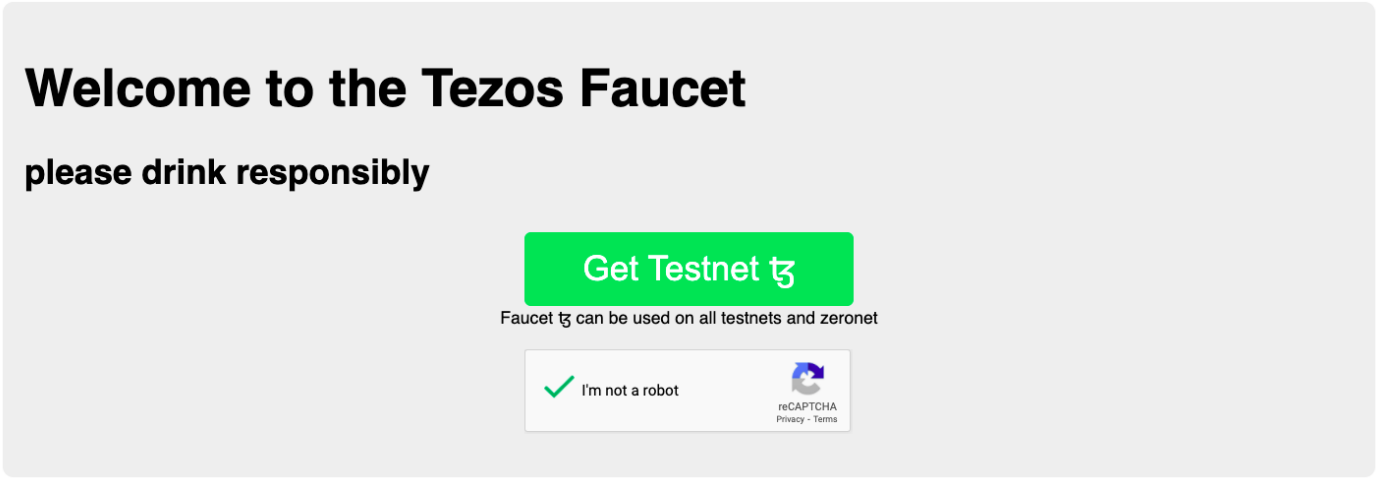


Полученный баланс совпадает с данными обозревателя tzStats.

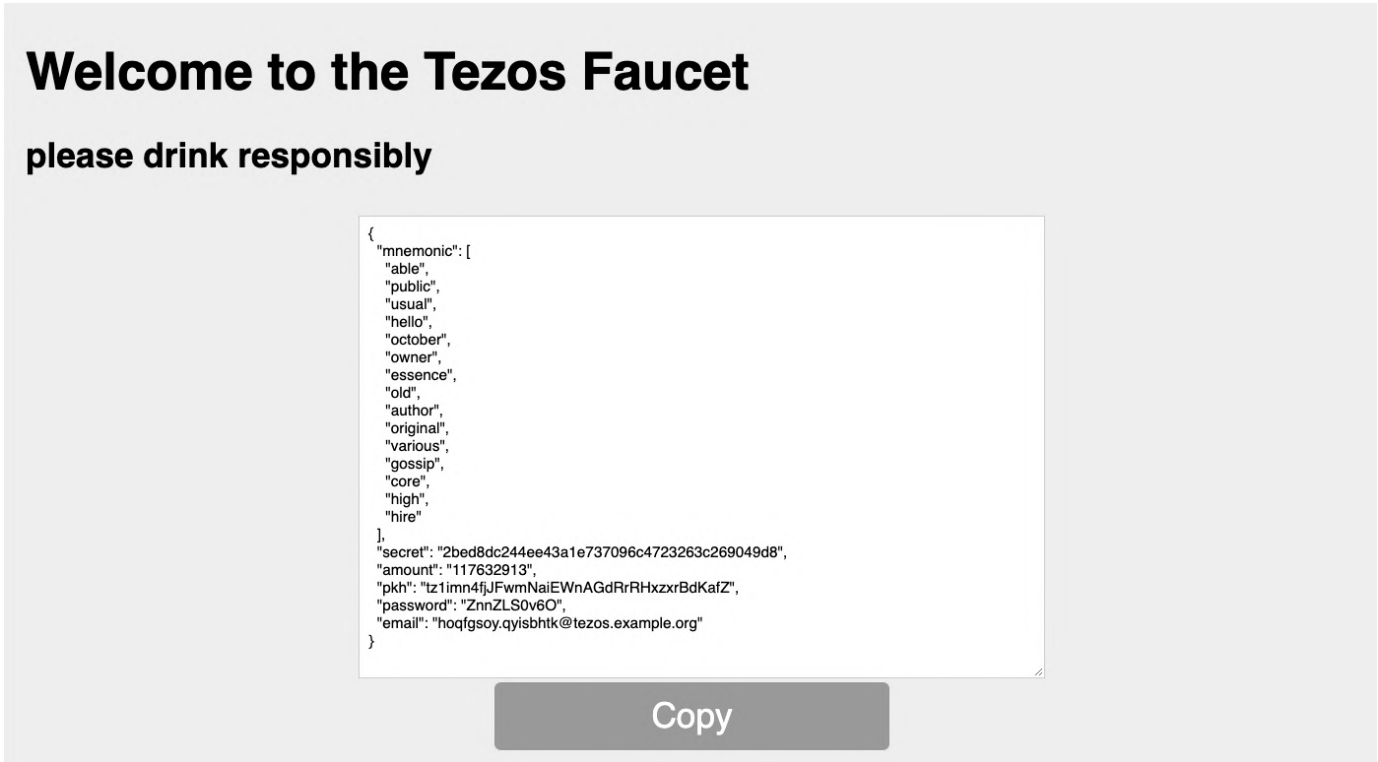
Создаем аккаунт в тестовой сети Tezos с помощью Taquito

Транзакции нужно подписывать закрытым ключом. Для этого Taquito использует модуль signer, который работает с данными блокчейн-аккаунтов.

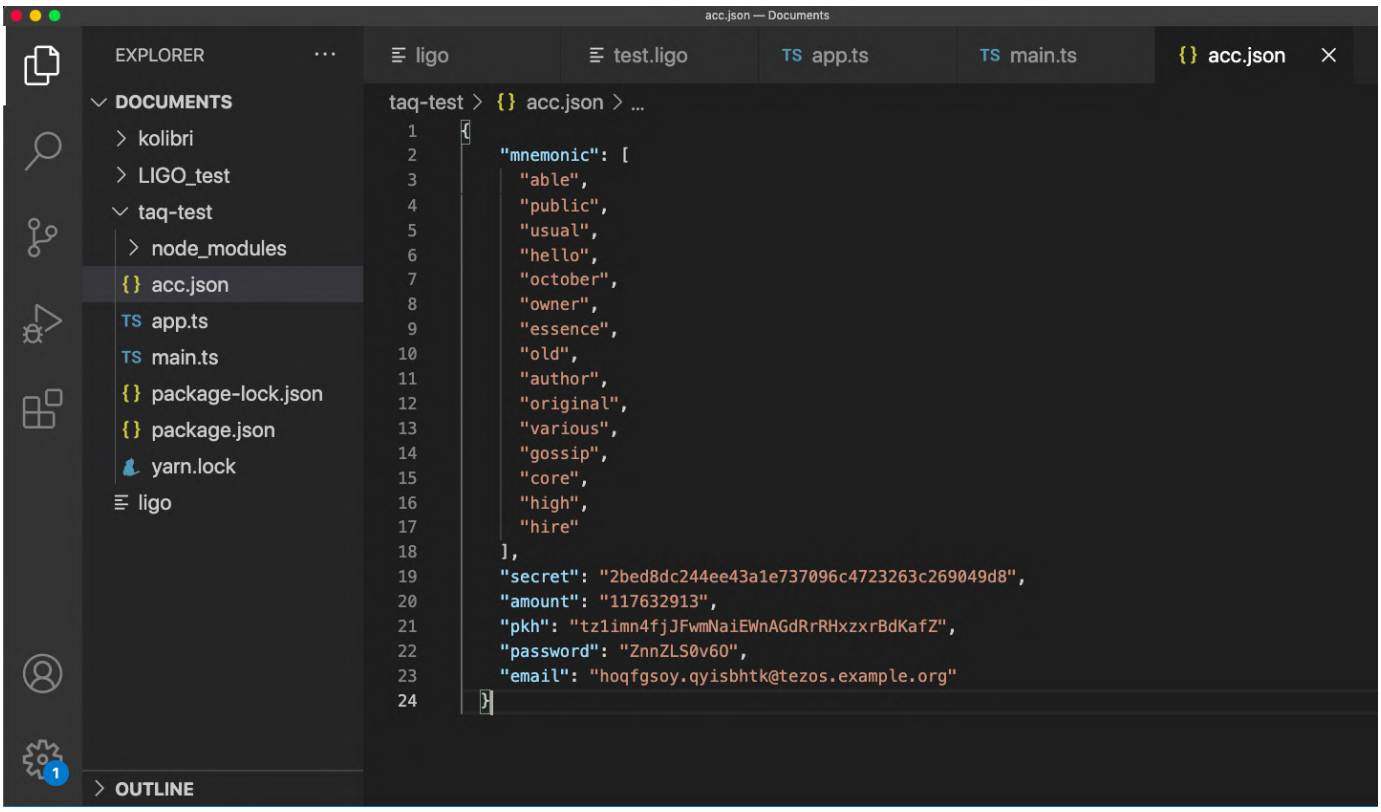
Чтобы не тратить настоящие tez, мы подключимся к Tezos Testnet. Сначала нужно получить адрес и тестовые tez на кране tzalpha. Перейдите по ссылке, подтвердите, что вы не робот, и нажмите кнопку **Get Testnet tez**.



Кран tzalpha создаст новый адрес и выдаст данные аккаунта: мнемоническую фразу, закрытый ключ и публичный адрес.



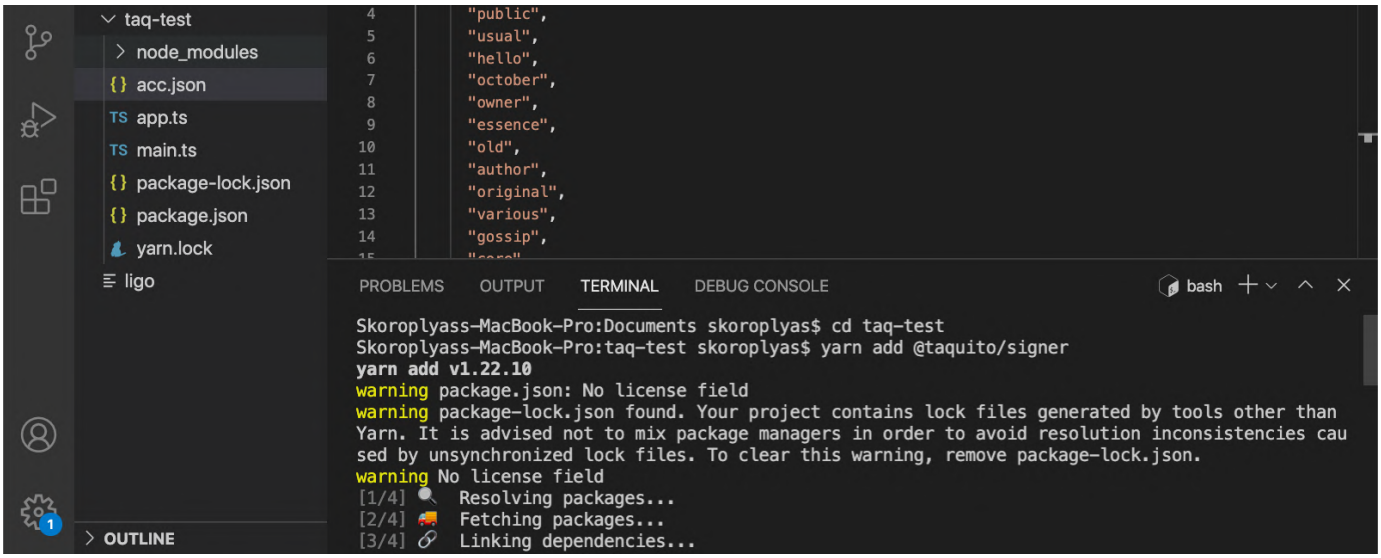
Данные аккаунта нужно сохранить в отдельном файле. Создайте в папке taq-test файл acc.json и вставьте в него код из tzalpha.



Теперь можно добавить в проект модуль signer. В VS Code откройте терминал, перейдите в папку taq-test и выполните команду:

```
yarn add @taquito/signer
```

Как и при установке Taquito, терминал выдаст предупреждения об отсутствующих лицензиях и список установленных файлов.



Создадим новый файл tx.ts и добавим в него модуль signer:

```
import { TezosToolkit } from '@taquito/taquito';
// импортируем inMemorySigner. Он сохранит приватный ключ в оперативной памяти
   и будет подписывать им транзакции
import { InMemorySigner } from '@taquito/signer';
// объявляем константу acc, которая направит скрипт к файлу acc.json
const acc = require('./acc.json');
export class Tx {
 private tezos: TezosToolkit;
  rpcUrl: string;
 constructor(rpcUrl: string) {
  this.tezos = new TezosToolkit(rpcUrl);
  this.rpcUrl = rpcUrl

// объявляем параметры с помощью метода fromFundraiser: почту, пароль
   и мнемоническую фразу, из которой можно получить приватный ключ
this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
acc.password, acc.mnemonic.join(' ')))
 }
// получаем публичный и приватный ключи и активируем аккаунт
public async activateAccount() {
     const {pkh, secret} = acc;
     try {
         const operation = await this.tezos.tz.activate(pkh, secret);
         await operation.confirmation();
     } catch (e) {
         console.log(e)
     }
 }
 public async main() { }
}
```

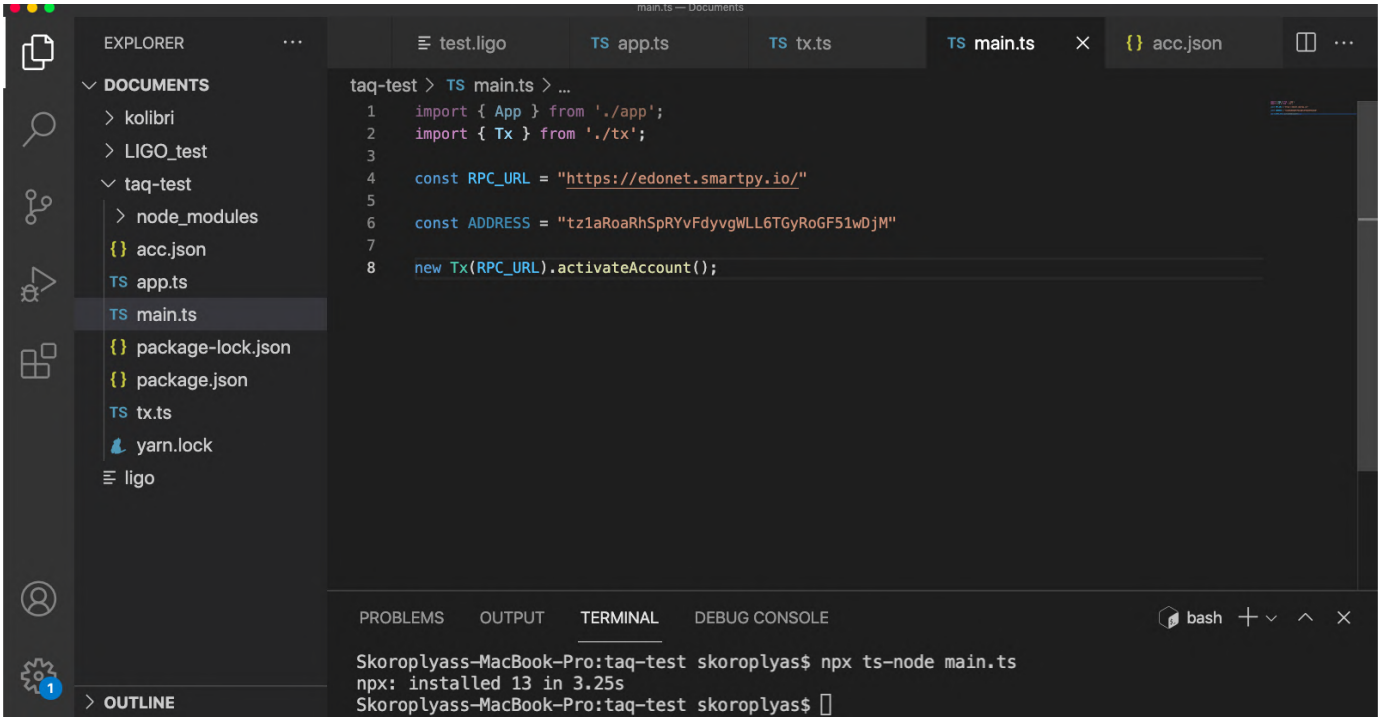
В файле main.ts вызываем метод *activateAccount()*:

```
import { App } from './app';
// импортируем Tx.ts
import { Tx } from './tx';
// меняем RPC-ссылку из мейннета на тестовую сеть. Не пугайтесь smartpy
```

```
    в ссылке - это просто адрес сервера
const RPC_URL = "https://florencenet.smartpy.io/"
const ADDRESS = "tz1aRoaRhSpRYvFdyvgWLL6TGyRoGF51wDjM"
// вызываем функцию Tx, передаем ей ссылку на тестовую сеть и просим активировать
    аккаунт
new Tx(RPC_URL).activateAccount();
```

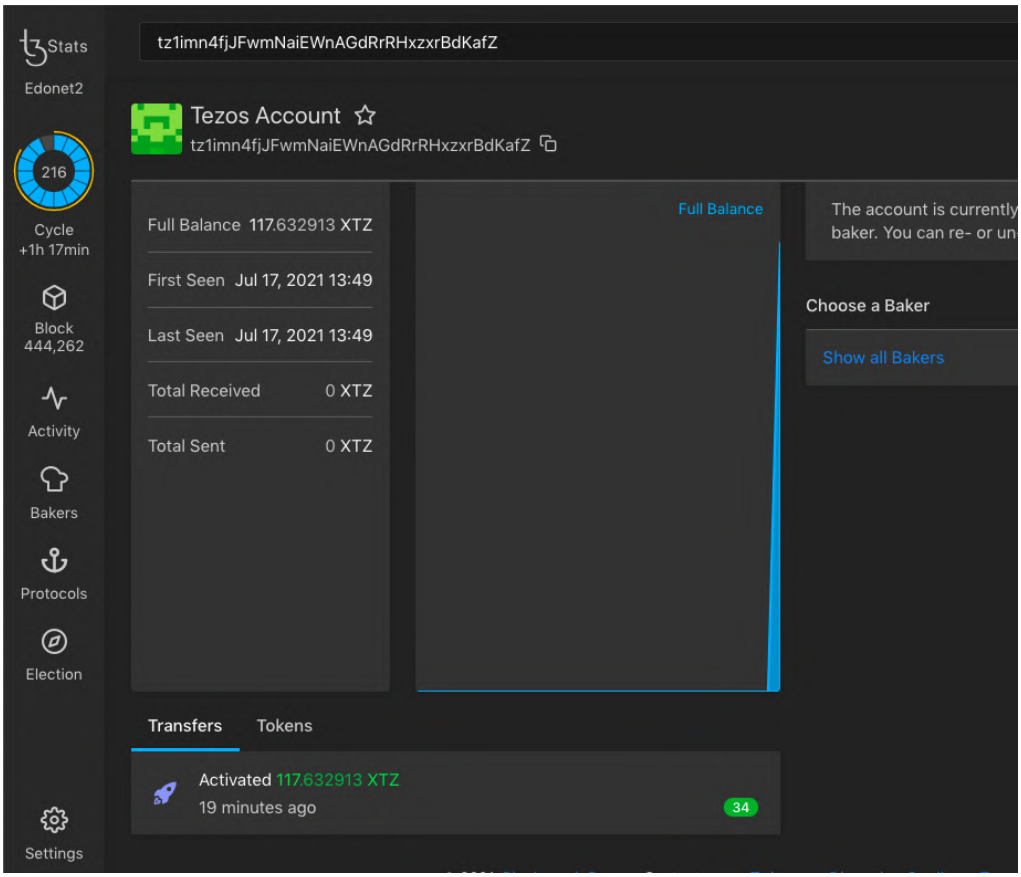
Осталось исполнить main.ts в терминале:

```
npx ts-node main.ts
```



Если тестовая сеть активирует аккаунт, консоль отчитается о запуске npx без сообщений об ошибках. Теперь можете проверить активацию в обозревателе тестовых сетей.

Откройте acc.json и скопируйте публичный ключ из поля «*pkh*». Перейдите на florence.tzstats и найдите свой аккаунт по публичному ключу. На балансе должно быть около сотни тестовых tez, а в истории транзакций — запись об активации.



Мы настроили Taquito и активировали тестовый аккаунт. Для работы в мейннете нужно будет сделать то же самое: создать файл .json с данными аккаунта и подключить его с помощью signer.

Публикуем контракт в тестнете

На прошлом уроке мы запускали смарт-контракт с помощью контейнера Docker и команды *dry-run*. Виртуальная машина с контрактом сразу закрывалась, и при этом не использовала хранилище storage.

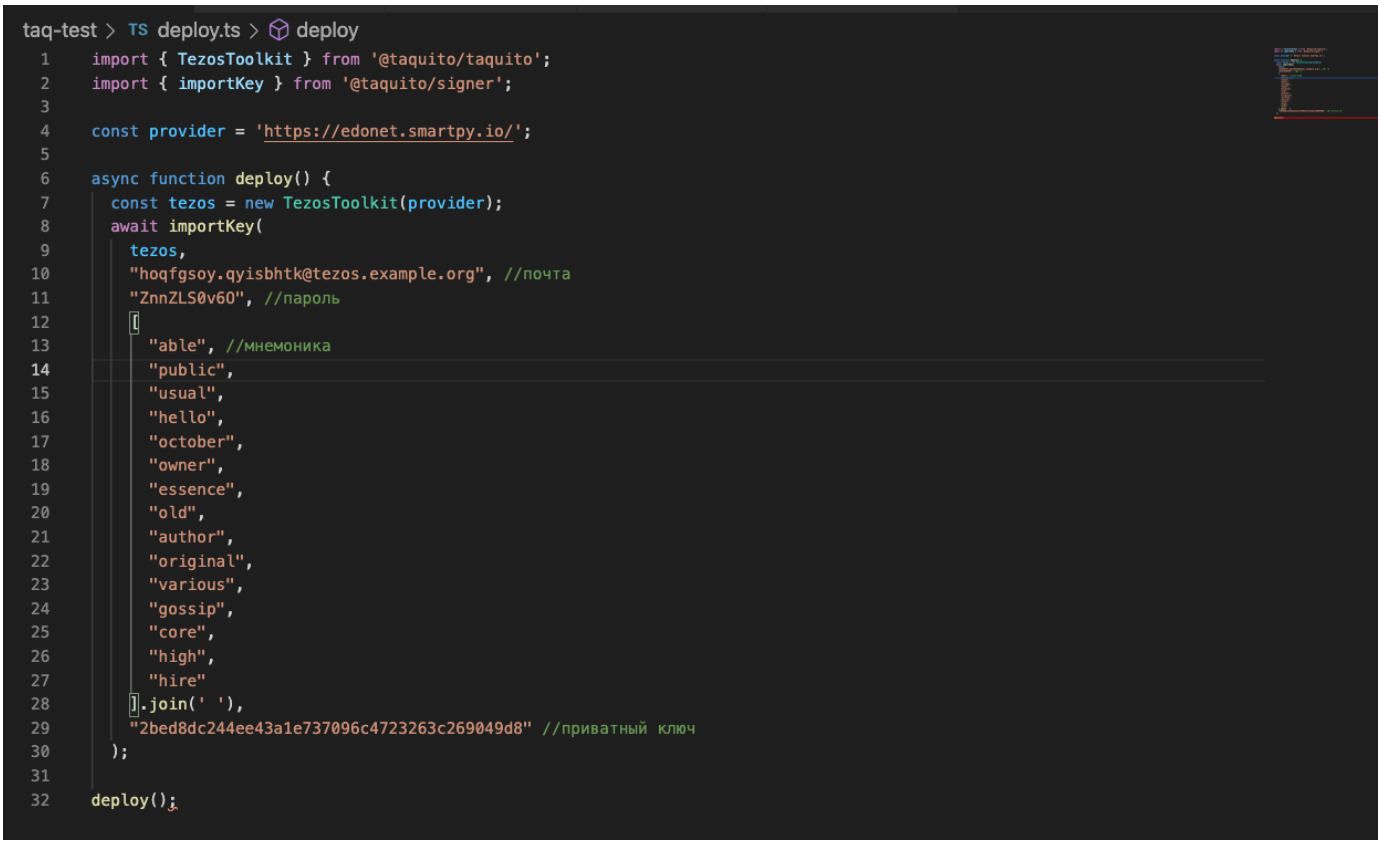
Настоящие смарт-контракты токенов и децентрализованных приложений постоянно используют storage для хранения данных пользователей. Мы подробнее расскажем об этом в следующем уроке. А пока опубликуем контракт в тестовой сети и посмотрим, как storage сохраняет информацию.

Перейдите в VS Code и создайте файл deploy.js. Вставьте в него следующий код:

```
import { TezosToolkit } from '@taquito/taquito';
import { importKey } from '@taquito/signer';

const provider = 'https://florencenet.smartpy.io/';

async function deploy() {
```

```
const tezos = new TezosToolkit(provider);
await importKey(
  tezos,
  "", // почта
  "", // пароль
  [
    "", // мнемоника
  ].join(' '),
  "" // приватный ключ
);

deploy();
```

Впишите в пустые места данные аккаунта: почту, пароль, слова мнемонической фразы и приватный ключ из файла acc.json.



После данных аккаунта добавьте код контракта. Мы будем использовать модифицированный контракт из первого урока:

```
function main (const num : int; const store : int) is
 ((nil : list (operation)),  store + num)
```

Этот смарт-контракт получает от пользователя параметр num типа int и суммирует его с числом в хранилище.

Скомпилируйте код контракта на язык Michelson. Чтобы не запускать Docker лишний раз, воспользуйтесь онлайн средой разработки.

Вставьте код в редактор, в выпадающем списке выберите пункт **Compile Contract** и нажмите кнопку **Run**. Компилятор выдаст код контракта на языке Michelson.

Вернитесь к файлу deploy.ts и после данных аккаунта вставьте следующий код:

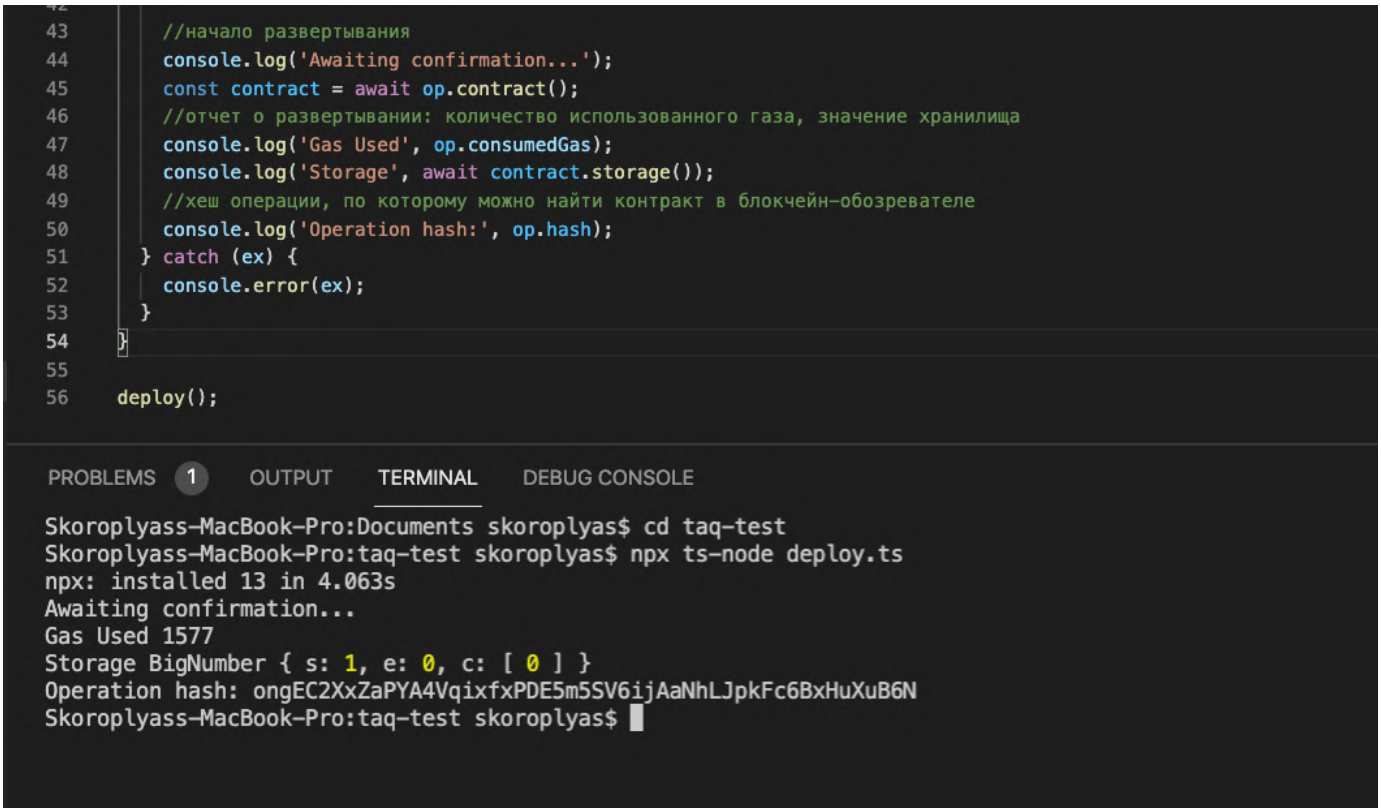
```
try {
  const op = await tezos.contract.originate({
    // код смарт-контракта
    code: `{ parameter int ;
      storage int ;
      code { UNPAIR ; ADD ; NIL operation ; PAIR } }
          `,
    // значение хранилища
    init: `0`,
  });
```

Теперь опишите процесс развертывания контракта. Финальный код будет выглядеть так:

```
import { TezosToolkit } from '@taquito/taquito';
import { importKey } from '@taquito/signer';

const provider = 'https://florencenet.smartpy.io/';

async function deploy() {
 const tezos = new TezosToolkit(provider);
 await importKey(
   tezos,
   "hoqfgsoy.qyisbhtk@tezos.example.org", // почта
   "ZnnZLS0v6O", // пароль
   [
     "able", // мнемоника
     "public",
     "usual",
     "hello",
     "october",
     "owner",
     "essence",
     "old",
     "author",
     "original",
     "various",
     "gossip",
     "core",
     "high",
     "hire"
   ].join(' '),
   "2bed8dc244ee43a1e737096c4723263c269049d8" // приватный ключ
 );

 try {
   const op = await tezos.contract.originate({
     // код смарт-контракта
     code: `{ parameter int ;
       storage int ;
       code { UNPAIR ; ADD ; NIL operation ; PAIR } }
           `,
```

```
    // значение хранилища
    init: `0`,
  });

  // начало развертывания
  console.log('Awaiting confirmation...');
  const contract = await op.contract();
  // отчет о развертывании: количество использованного газа, значение хранилища
  console.log('Gas Used', op.consumedGas);
  console.log('Storage', await contract.storage());
  // хеш операции, по которому можно найти контракт в блокчейн-обозревателе
  console.log('Operation hash:', op.hash);
 } catch (ex) {
  console.error(ex);
 }
}

deploy();
```

Откройте консоль и выполните команду `npx ts-node deploy.ts`. Терминал выдаст отчет о публикации контракта.



Скопируйте хеш операции и найдите транзакцию развертывания контракта в обозревателе florence.tzstats. В ней будет указан адрес смарт-контракта в тестовой сети.

Попробуйте вызвать опубликованный контракт. Для этого создайте скрипт, который вызовет главную функцию и передаст ей число.

Создайте новый файл call.ts и вставьте в него код:

```
import { TezosToolkit } from '@taquito/taquito';
import { InMemorySigner } from '@taquito/signer';
const acc = require('./acc.json');
export class Call {

   private tezos: TezosToolkit;
   rpcUrl: string;

  constructor(rpcUrl: string) {
   this.tezos = new TezosToolkit(rpcUrl);
   this.rpcUrl = rpcUrl

   // объявляем параметры с помощью метода fromFundraiser: почту, пароль
      и мнемоническую фразу, из которой можно получить приватный ключ
   this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
acc.password, acc.mnemonic.join(' ')))
  }


   public add(add: number, contract: string) {
       this.tezos.contract
           .at(contract) //обращаемся к контракту, чтобы получить его точки входа
           .then((contract) => {
               console.log(`Adding ${add} to storage...`);
               // обращаемся к главной функции. В отличие от синтаксиса ligo,
главная точка входа называется не main, а default
               return contract.methods.default(add).send();
           })
           .then((op) => {
               console.log(`Awaiting for ${op.hash} to be confirmed...`);
               return op.confirmation(1).then(() => op.hash); // ждем одно
подтверждение сети, чтобы быстрее получить результат
           })
           .then((hash) => console.log(`Call done}`)) // успешный вызов
```

```
            .catch((error) => console.log(`Error: ${JSON.stringify(error, null,
2)}`));
    }
}
```

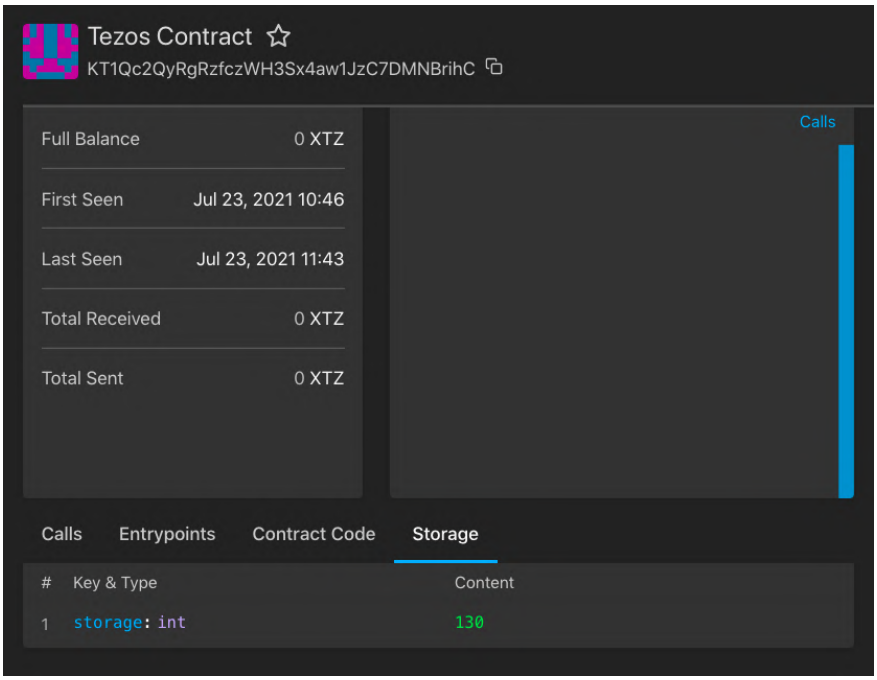
Теперь сделайте отдельный скрипт для вызова call.ts, передачи адреса контракта и значения аргумента. Для этого создайте файл maincall.ts и скопируйте в него код:

```
import { Call } from './call';

const RPC_URL = "https://florencenet.smartpy.io/"
const CONTRACT = "" // адрес опубликованного контракта
const ADD = 5 // число, которое получит главная функция. Можете изменить его на
другое
new Call(RPC_URL).add(ADD, CONTRACT);
```

Присвойте константе CONTRACT адрес опубликованного тестового контракта.
Откройте консоль и выполните команду:

```
npx ts-node maincall.ts
```

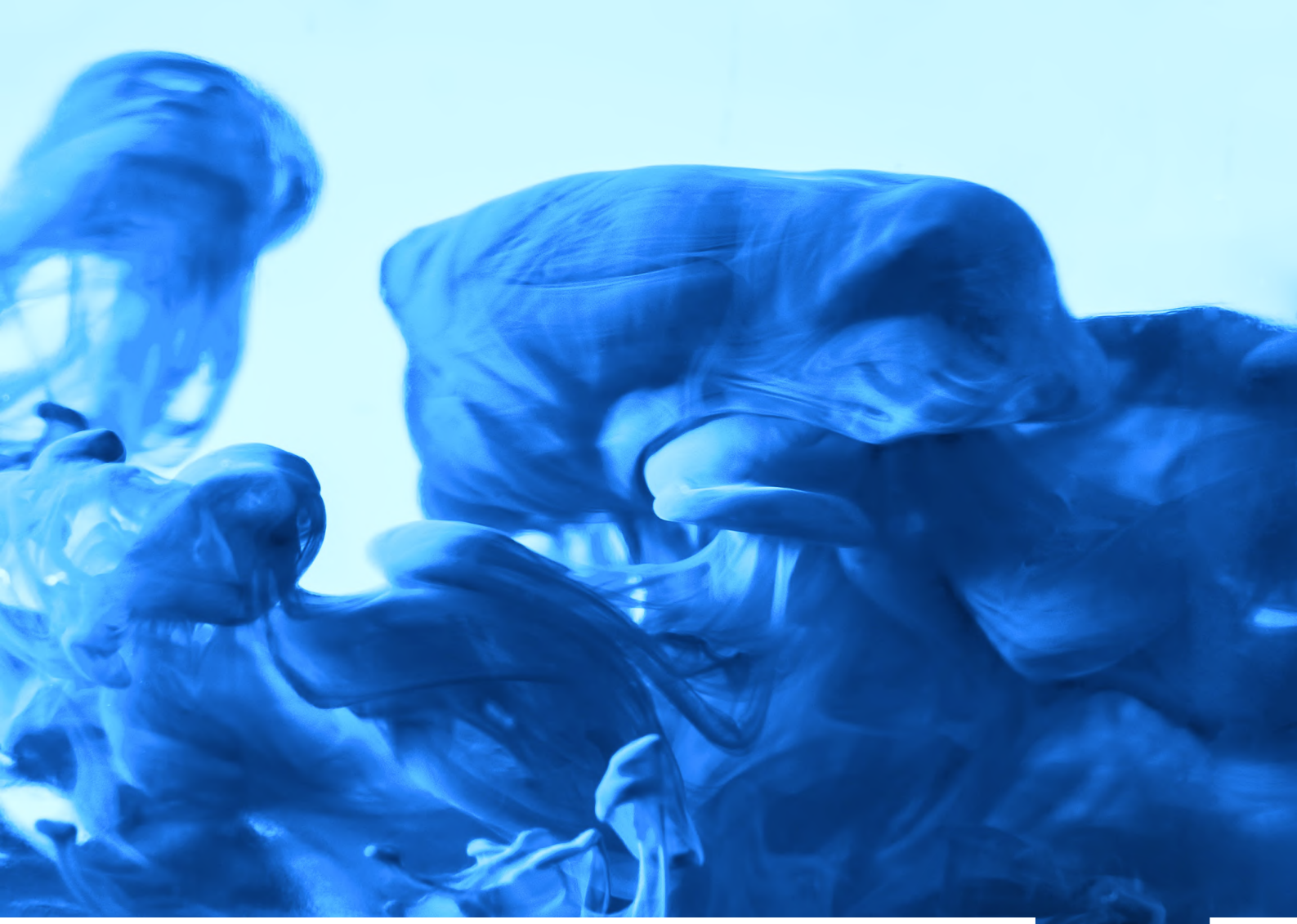
Когда терминал выдаст сообщение о завершении операции, перейдите по ссылке или найдите свой контракт по адресу. Как видите, значение хранилища изменилось.



Подводим итоги

Taquito — это инструмент для работы с блокчейном Tezos через код на JavaScript и TypeScript. С помощью Taquito разработчики обращаются к точкам входа смарт-контрактов и к сети Tezos, как к обычным JS-методам. Также Taquito позволяет хранить данные нескольких аккаунтов в разных файлах и выносить часто используемый код в отдельные модули.

Библиотека Taquito нужна для разработки DApp на Tezos. Разработчикам проще интегрировать в веб-приложение JavaScript, чем переводить запросы пользователей в консольные команды клиента Tezos. В следующих уроках мы также будем использовать Taquito для создания взаимозаменяемого токена и NFT.



4

Создание токена стандарта FA 1.2

На прошлых уроках мы изучили синтаксис языка программирования LIGO и опубликовали простейший контракт в тестовой сети Tezos.

Теперь мы изучим механизм работы цифровых активов, разберемся со стандартом FA 1.2, и выпустим токен в тестовой сети.

Виды и стандарты токенов

Для начала определимся с терминами. Токенами называют два типа единиц учета: нативные токены и токены, которые выпускают смарт-контракты. Логика работы первых описана в коде протокола, а вторых — в коде смарт-контрактов (СК).

Нативные токены также называют криптовалютами. К ним относятся биткоин, Ether (ETH), Tezos (XTZ, tez), Litecoin и другие.

Токены смарт-контрактов — цифровые активы, созданные пользователями блокчейна, например USDT и DOGE на блокчейне Ethereum, kUSD — на Tezos.

В этом и следующем уроках мы будем говорить о токенах смарт-контрактов, но для простоты называть их токенами.

Мы выяснили, что логика работы токенов не описана в протоколе. Разработчики создают ее в коде смарт-контрактов: добавляют базовые функции вроде хранения балансов и передачи монет.

Так, 100 kUSD на кошельке пользователя — это запись в хранилище СК вида «Адрес TZ1 может отправить 100 токенов этого контракта». Другими словами, токены — это реестры внутри блокчейна.

Чтобы разработчики выпускали рабочие и устойчивые к атакам смарт-контракты, сообщество принимает стандарты токенов. Они содержат списки функций, необходимых для взаимодействия смарт-контракта с кошельками и блокчейн-приложениями.

Разработчики могут прописывать в смарт-контрактах дополнительные функции, например сжигание токенов. Но базовые функции стандарта обязательны.

Базовые функции стандарта FA 1.2

Стандарты Tezos называются Financial Application (FA). Два самых популярных из них:

- **FA 1.2** для взаимозаменяемых токенов;
- **FA 2** для взаимозаменяемых токенов и NFT.

На этом уроке мы рассмотрим более простой стандарт FA 1.2, а на следующем — разберемся с FA 2.

Стандарт FA 1.2 включает функции:

- **передача (transfer)** — проведение транзакций между пользователями;
- **одобрение (approve)** — разрешение Алисе отправить токены с адреса Боба. Позволяет контракту отправлять пользователем запросы на транзакции;
- **одобренное количество (getAllowance)** — просмотр количества токенов, которые Алиса может отправить с адреса контракта;
- **баланс пользователя (getBallance)** — просмотр баланса пользователя;
- **свободная эмиссия (getTotalSupply)** — просмотр количества токенов на адресах всех пользователей.

Кроме того, контракт токена должен хранить в storage информацию о полной эмиссии и балансах пользователей, а в коде — сообщения о стандартных ошибках вроде нехватки средств.

Разработчик может включить в хранилище метаданные с названием контракта и токена, а также ссылку на его логотип.

Что такое имплементация FA 1.2

Имплементация — это рабочий продукт на основе какой-либо идеи. В случае FA 1.2 — шаблон смарт-контракта для операций с токенами.

Разработчик может скопировать имплементацию, изменить ее под свои задачи, опубликовать в блокчейне и получить работоспособный токен.

Имплементация токена FA 1.2 состоит примерно из сотни строк. Код можно разделить на восемь частей:

- **FA 1.2** для взаимозаменяемых токенов;
- **FA 2** для взаимозаменяемых токенов и NFT.

На этом уроке мы рассмотрим более простой стандарт FA 1.2, а на следующем — разберемся с FA 2.

Стандарт FA 1.2 включает функции:

- псевдонимы, типы данных и псевдо-точки входа;
- функция *`getAccount`*, которая получает данные пользователя;
- функция *`getAllowance`*, которая получает количество токенов, доступных для транзакции;
- функция *`transfer`* для перевода токенов из одного адреса на другой;
- функция *`approve`*, которая подтверждает право пользователя на перевод токенов;
- функция *`getBalance`*, которая возвращает баланс пользователя;
- функция *`getTotalSupply`*, которая возвращает количество свободных токенов;
- функция *`main`*, которая принимает входящие параметры и передает их одной из предыдущих функций.

Ниже подробнее расскажем о каждой из этих частей.

Объявление псевдонимов и типов

В начале контракта нужно объявить все типы и псевдонимы, которые он будет использовать в функциях:

```
// объявляем псевдоним trusted типа address. Мы будем использовать его для
   обозначения адресов, у которых есть право отправлять токены
type trusted is address;

// объявляем псевдоним amt (amount) типа nat для хранения балансов
type amt is nat;
```

```
(* объявляем псевдоним account типа record. В нем будем хранить данные
   пользователей, которым можно передавать токены *)
type account is
  record [
    balance         : amt;
    allowances      : map (trusted, amt);
  ]

(* объявляем тип хранилища смарт-контракта. В нем хранится общее количество
   токенов, а также структура данных big_map, которая связывает публичные адреса
   и балансы пользователей *)
type storage is
  record [
    totalSupply     : amt;
    ledger          : big_map (address, account);
  ]

(* объявляем псевдоним для метода return, который будем использовать для
   возвращения операций. В коротких контрактах можно обойтись без него, но в
   контрактах с несколькими псевдо-точками входа проще один раз прописать тип
   возврата и использовать его в каждой функции *)
type return is list (operation) * storage

(* объявляем пустой список noOperations. Его будут возвращать функции transfer и
   approve *)
const noOperations : list (operation) = nil;
```

Компилятор Michelson автоматически сортирует содержимое продвинутых структур данных в алфавитном порядке. Сортировка может случайно сломать контракт. Например, если первый параметр — адрес отправителя — начинается с TZ19, а второй — адрес получателя — начинается с TZ11, компилятор поменяет их местами. В таком случае контракт попытается отправить токены не с того аккаунта.

Чтобы сохранить нужный порядок параметров, разработчики записывают важные структуры данных в типе *`michelson_pair`*. Такая структура включает два аргумента и их названия, например, два адреса: sender и receiver. Компилятор переносит в код на Michelson значения michelson_pair без сортировки.

```
(* объявляем псевдонимы входящих параметров для каждой базовой функции FA 1.2. *)

// функция transfer получает на вход адрес отправителя, адрес получателя и сумму
   транзакции
type transferParams is michelson_pair(address, "from", michelson_pair(address,
"to", amt, "value"), "")
// approve получает адрес пользователя и количество токенов, которые он может
   отправить с баланса смарт-контракта
type approveParams is michelson_pair(trusted, "spender", amt, "value")
// getBallance получает адрес пользователя и прокси-контракта, которому она
   отправит данные о балансе
type balanceParams is michelson_pair(address, "owner", contract(amt), "")
// getAllowance получает адрес пользователя, данные его аккаунта в смарт-
   контракте и прокси-контракт
type allowanceParams is michelson_pair(michelson_pair(address, "owner", trusted,
"spender"), "", contract(amt), "")
// totalSupply не использует michelson_pair, потому что первый входящий параметр–
   пустое значение unit — и так окажется первым после сортировки компилятора
   Michelson
type totalSupplyParams is (unit * contract(amt))

(* объявляем псевдо-точки входа: даем название и присваиваем им тип параметров,
которые описали выше*)
type entryAction is
  | Transfer of transferParams
  | Approve of approveParams
  | GetBalance of balanceParams
  | GetAllowance of allowanceParams
  | GetTotalSupply of totalSupplyParams
```

Функция *getAccount* получает входящий параметр типа address и значение storage из смарт-контракта:

```
function getAccount (const addr : address; const s : storage) : account is
  block {
      // присваиваем переменной acct значение типа account: нулевой баланс
         и пустую запись allowances
    var acct : account :=
      record [
```

```
        balance    = 0n;
        allowances = (map [] : map (address, amt));
      ];

    (* проверяем, есть ли в хранилище аккаунт пользователя. Если нет – оставляем
       в acct пустое значение из предыдущего блока. Если есть – присваиваем
       переменной acct значение из хранилища. Функция возвращает значение acct *)
    case s.ledger[addr] of
      None -> skip
    | Some(instance) -> acct := instance
    end;
  } with acct
```

Функция *getAllowance* спрашивает у пользователя, сколько токенов он разрешает перевести на другой адрес. Она получает адрес пользователя, адрес контракта (spender) и состояние хранилища, а возвращает аргумент **amt** — количество токенов разрешенных для расходования:

```
function getAllowance (const ownerAccount : account; const spender : address;
const s : storage) : amt is
  (* если пользователь разрешил отправить некоторое количество токенов, функция
     присваивает это количество переменной amt. Если не разрешил – количество
     токенов равняется нулю *)
  case ownerAccount.allowances[spender] of
    Some (amt) -> amt
  | None -> 0n
  end;
```

Функция *transfer* получает от пользователя адреса отправителя и получателя, количество токенов для перевода и состояние хранилища:

```
function transfer (const from_ : address; const to_ : address; const value : amt;
var s : storage) : return is
  block {

    (* вызываем функцию getAccount, чтобы присвоить переменной senderAccount
       данные аккаунта пользователя. Затем мы используем senderAccount, чтобы
       считывать баланс пользователя и разрешения *)
    var senderAccount : account := getAccount(from_, s);
```

```
(* проверяем, достаточно ли у пользователя средств для перевода. Если нет –
   виртуальная машина прерывает исполнение контракта, если достаточно –
   продолжает исполнять контракт *)
if senderAccount.balance < value then
  failwith("NotEnoughBalance")
else skip;

(* проверяем, может ли адрес-инициатор транзакции отправить токены. Если он
   запрашивает перевод из чужого адреса, функция запрашивает разрешение у
   настоящего владельца. Если инициатор и отправитель – один адрес,
   виртуальная машина продолжает исполнять контракт *)
if from_ =/= Tezos.sender then block {
(* вызываем функцию getAllowance, чтобы владелец адреса-отправителя указал,
   сколько токенов он разрешает отправить. Присваиваем это значение константе
   spenderAllowance *)
  const spenderAllowance : amt = getAllowance(senderAccount, Tezos.sender,s);

(* если владелец разрешил отправить меньше токенов, чем указано во входящем
   параметре, виртуальная машина прекратит исполнять контракт *)
  if spenderAllowance < value then
    failwith("NotEnoughAllowance")
  else skip;

  (* отнимаем от разрешенного для отправки количества токенов сумму
     транзакции *)
  senderAccount.allowances[Tezos.sender] := abs(spenderAllowance - value);
} else skip;

(* отнимаем от баланса адреса-отправителя количество отправленных токенов *)
senderAccount.balance := abs(senderAccount.balance - value);

(* обновляем запись о балансе отправителя в storage *)
s.ledger[from_] := senderAccount;

(* еще раз вызываем функцию getAccount, чтобы получить или создать запись
   аккаунта для адреса-получателя *)
var destAccount : account := getAccount(to_, s);

(* добавляем к балансу получателя количество отправленных токенов *)
destAccount.balance := destAccount.balance + value;
```

```
    (* обновляем запись о балансе получателя в storage *)
    s.ledger[to_] := destAccount;

  }
  // возвращаем пустой список операций и состояние storage после исполнения
     функции
  with (noOperations, s)
```

Функция approve запрашивает подтверждение количества токенов, которое spender-адрес может отправить с sender-адреса. Пример использования: блокчейн-приложение (spender) запрашивает у пользователя (sender) разрешение на отправку токенов.

Эта функция подвержена атаке опережающего расходования. Например, sender снижает количество разрешенных для отправки токенов с 20 до 10. Spender узнает об этом и создает транзакцию с расходованием 20 токенов. Он платит повышенную комиссию, чтобы транзакция с тратой попала в блок раньше, чем транзакция со сменой разрешений.

Spender получает 20 токенов и дожидается смены разрешения. После этого он создает еще одну транзакцию — на отправку 10 токенов. Функция разрешает отправку. В результате spender отправляет с sender-адреса 30 токенов вместо разрешенных 10.

Чтобы избежать подобной ситуации, разработчики реализуют в approve задержку смены разрешения. Если разрешенное количество токенов больше нуля, его можно изменить только на нуль, а если равно нулю — его можно изменить на натуральное число. В таком случае spender-адрес не может потратить токены дважды.

```
function approve (const spender : address; const value : amt; var s : storage) :
return is
  block {

    (* получаем данные аккаунта пользователя *)
    var senderAccount : account := getAccount(Tezos.sender, s);

    (* получаем текущее количество токенов, которое пользователь разрешил
       отправить *)
    const spenderAllowance : amt = getAllowance(senderAccount, spender, s);

    if spenderAllowance > 0n and value > 0n then
      failwith("UnsafeAllowanceChange")
    else skip;
```

```
    (* вносим в данные аккаунта новое разрешенное количество токенов для
       расходования *)
    senderAccount.allowances[spender] := value;

    (* обновляем хранилище смарт-контракта *)
    s.ledger[Tezos.sender] := senderAccount;

  } with (noOperations, s)
```

Функции *getBalance*, *getAllowance* и *getTotalSupply* относятся к обзорным (view). Они возвращают запрашиваемое значение не пользователю, а специальному промежуточному контракту (proxy-contract). Последний позволяет приложениям получать данные от пользовательских контрактов и отображать их в интерфейсе.

Функция *getBallance* возвращает значение баланса заданного адреса:

```
function getBalance (const owner : address; const contr : contract(amt); var s :
storage) : return is
  block {
      // присваиваем константе ownerAccount данные аккаунта
    const ownerAccount : account = getAccount(owner, s);
  }
  // возвращаем промежуточному контракту баланс аккаунта
  with (list [transaction(ownerAccount.balance, 0tz, contr)], s)
```

Функция *getAllowance* возвращает количество разрешенных для расходования токенов запрашиваемого аккаунта:

```
function getAllowance (const owner : address; const spender : address; const
contr : contract(amt); var s : storage) : return is
  block {
      // получаем данные аккаунта, а из них – количество разрешенных для
         расходования токенов
    const ownerAccount : account = getAccount(owner, s);
    const spenderAllowance : amt = getAllowance(ownerAccount, spender, s);
  } with (list [transaction(spenderAllowance, 0tz, contr)], s)
```

Функция *getTotalSupply* возвращает количество токенов на балансах всех пользователей:

```
function getTotalSupply (const contr : contract(amt); var s : storage) : return
is
  block {
    skip
  } with (list [transaction(s.totalSupply, 0tz, contr)], s)
```

Главная функция принимает название псевдо-точки входа и ее параметры:

```
function main (const action : entryAction; var s : storage) : return is
  block {
    skip
  } with case action of
    | Transfer(params) -> transfer(params.0, params.1.0, params.1.1, s)
    | Approve(params) -> approve(params.0, params.1, s)
    | GetBalance(params) -> getBalance(params.0, params.1, s)
    | GetAllowance(params) -> getAllowance(params.0.0, params.0.1, params.1, s)
    | GetTotalSupply(params) -> getTotalSupply(params.1, s)
  end;
```

Готовим смарт-контракт токена к публикации

Чтобы не тратить время на добавление Taquito и создание файлов проекта, воспользуемся папкой taq-test из прошлого урока.

Запустите редактор VS Code. Создайте в папке taq-test папку token, а в ней — файл token.ligo. Скопируйте в файл код токена.

```
// объявляем псевдоним trusted типа address. Мы будем использовать его для
   обозначения адресов, которые могут пересылать токены с контракта
type trusted is address;

// объявляем псевдоним amt (amount) типа nat для хранения балансов
type amt is nat;

(* объявляем псевдоним account типа record. В нем будем хранить данные
   пользователей, которым можно передавать токены.
*)
```

```
type account is
 record [
   balance         : amt;
   allowances      : map (trusted, amt);
 ]

(* объявляем тип хранилища смарт-контракта. Он хранит общее количество токенов, а
   также big_map, который связывает публичные адреса и данные account
   пользователей *)
type storage is
 record [
   totalSupply     : amt;
   ledger          : big_map (address, account);
 ]

(* объявляем псевдоним для метода return, который будем использовать для
   возвращения операций. В коротких контрактах можно обойтись без него.
   Но в контрактах с несколькими псевдо-точками входа проще один раз прописать
   тип возврата и использовать его для каждой функции *)
type return is list (operation) * storage

(* объявляем пустой список noOperations. Его будет возвращать метод return *)
const noOperations : list (operation) = nil;

(* объявляем псевдонимы входящих параметров для каждой базовой функции FA1.2 *)
type transferParams is michelson_pair(address, "from", michelson_pair(address,
"to", amt, "value"), "")
type approveParams is michelson_pair(trusted, "spender", amt, "value")
type balanceParams is michelson_pair(address, "owner", contract(amt), "")
type allowanceParams is michelson_pair(michelson_pair(address, "owner", trusted,
"spender"), "", contract(amt), "")
type totalSupplyParams is (unit * contract(amt))

(* псевдо-точки входа *)
type entryAction is
 | Transfer of transferParams
 | Approve of approveParams
 | GetBalance of balanceParams
 | GetAllowance of allowanceParams
 | GetTotalSupply of totalSupplyParams
```

```
(* функция getAccount получает входящий параметр типа address и значение storage
   из смарт-контракта. Это функция-помощник: ее вызывают другие функции контракта
   для получения данных о пользователе, а пользователь не может вызвать ее
   напрямую *)
function getAccount (const addr : address; const s : storage) : account is
 block {
     // присваиваем переменной acct значение типа account: нулевой баланс и
        пустую запись allowances
   var acct : account :=
     record [
       balance    = 0n;
       allowances = (map [] : map (address, amt));
     ];

   (* проверяем, есть ли в хранилище аккаунт пользователя. Если нет – оставляем в
      acct пустое значение из предыдущего блока. Если есть – присваиваем его
      переменной acct. Функция возвращает запись acct *)
   case s.ledger[addr] of
     None -> skip
   | Some(instance) -> acct := instance
   end;
 } with acct

(* getAllowance запрашивает у пользователя разрешение отрпавить токены из его
   адреса. Она получает адрес пользователя, адрес контракта spender и состояние
   хранилища *)
function getAllowance (const ownerAccount : account; const spender : address;
const _s : storage) : amt is
 (* если пользователь разрешил отправить некоторое количество токенов, функция
    присваивает это количество переменной amt. Если не разрешил – количество
    токенов равняется нулю *)
 case ownerAccount.allowances[spender] of
   Some (amt) -> amt
 | None -> 0n
 end;

(* Функция Transfer получает от пользователя адреса отправителя и получателя,
   количество токенов для перевода и состояние хранилища *)
function transfer (const from_ : address; const to_ : address; const value : amt;
var s : storage) : return is
 block {
```

```
(* вызываем фукнцию getAccount, чтобы присвоить ей данные аккаунта
   пользователя. Затем мы используем senderAccount чтобы считывать баланс
   пользователя и разрешения *)
var senderAccount : account := getAccount(from_, s);

(* проверяем, есть ли у пользователя достаточно средств для перевода. Если
   нет  – виртуальная машина прерывает исполнение контракта, если есть –
   продолжает исполнять контракт *)
if senderAccount.balance < value then
  failwith("NotEnoughBalance")
else skip;

(* проверяем, может ли адрес-инициатор транзакции отправить токены. Если
   адрес-инициатор запрашивает перевод из чужого адреса, функция запрашивает
   разрешение у настоящего владельца. Если инциатор и отправитель – один
   адрес, виртуальная машина продолжает исполнять контракт *)
if from_ =/= Tezos.sender then block {
(* вызываем функцию getAllowance, чтобы владелец адреса-отправителя указал,
   сколько токенов он разрешает отправить. Присваиваем это значение константе
   spenderAllowance *)
  const spenderAllowance : amt = getAllowance(senderAccount, Tezos.sender, s);

(* если владелец разрешил отправить меньше токенов, чем указано во входящем
   параметре, виртуальная машина прекратит исполнять контракт *)
  if spenderAllowance < value then
    failwith("NotEnoughAllowance")
  else skip;

  (* отнимаем от разрешенного для отправки количества токенов сумму
     транзакции *)
  senderAccount.allowances[Tezos.sender] := abs(spenderAllowance - value);
} else skip;

(* отнимаем от баланса адреса-отправителя количество отправленных токенов *)
senderAccount.balance := abs(senderAccount.balance - value);

(* обновляем запись о балансе отправителя в storage *)
s.ledger[from_] := senderAccount;

(* еще раз вызываем функцию getAccount, чтобы получить или создать запись
   аккаунта для адреса-получателя *)
```

```
    var destAccount : account := getAccount(to_, s);

    (* добавляем к балансу получателя количество отправленных токенов *)
    destAccount.balance := destAccount.balance + value;

    (* обновляем запись о балансе получателя в storage *)
    s.ledger[to_] := destAccount;

  }
  // возвращаем пустой список операций и состояние storage после исполнения
     функции
  with (noOperations, s)

(* функция Approve запрашивает подверждение на количество токенов, которое адрес-
   инициатор может отправить из адреса пользователей  *)
function approve (const spender : address; const value : amt; var s : storage) :
return is
  block {

    (* получаем данные аккаунта-инициатора *)
    var senderAccount : account := getAccount(Tezos.sender, s);

    (* получаем текущее количество токенов, которое пользователь разрешил
       отправить *)
    const spenderAllowance : amt = getAllowance(senderAccount, spender, s);

    (* защищаем контракт от атаки с опережающим расходованием. Допустим,
       пользователь изменяет разрешенное количество токенов для отправки с 20 до
       10. Владелец адреса-инициатора может об этом узнать и создать транзакцию с
       расходованием 20 токенов. Если он оплатит повышенную комиссию, то эта
       транзакция попадет в блок раньше, чем пользователь изменит разрешенное
       количество токенов. По обновлении разрешения владелец адреса-инициатора
       создаст еще одну транзакцию на 10 токенов. В результате он отправит из
       адреса пользователя 30 токенов вместо 10. Чтобы такого не случилось,
       разработчики добавляют эту защиту *)

    (* если старое разрешенное количество токенов для расходования больше нуля,
       его можно изменить только на ноль. Если оно равно нулю, его можно изменить
       на другое натуральное число *)
    if spenderAllowance > 0n and value > 0n then
      failwith("UnsafeAllowanceChange")
```

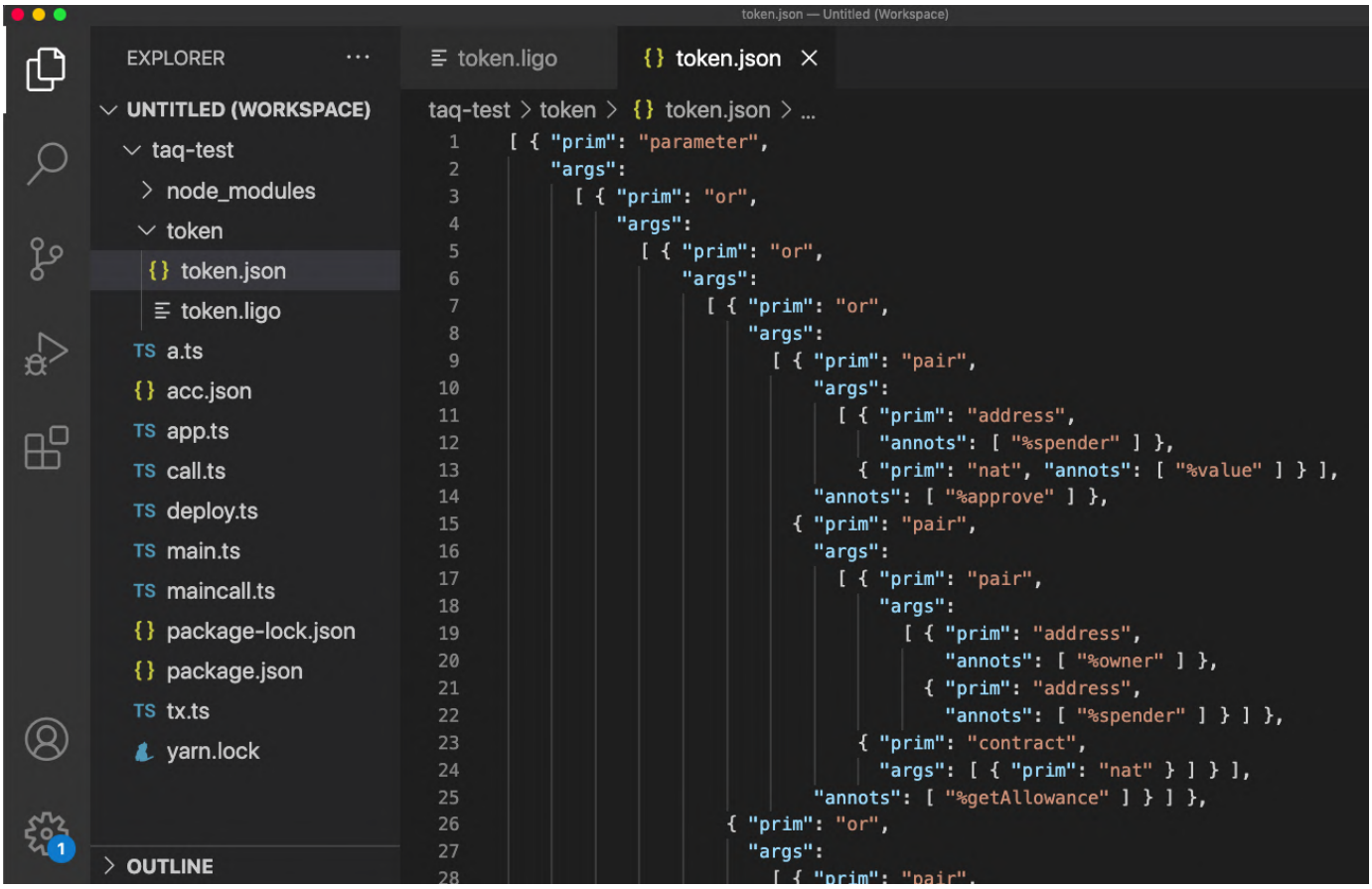
```
    else skip;

    (* вносим в данные аккаунта новое разрешенное количество токенов для
       расходывания *)
    senderAccount.allowances[spender] := value;

    (* обновляем хранилище смарт-контракта *)
    s.ledger[Tezos.sender] := senderAccount;

  } with (noOperations, s)

(* Функции getBalance, getAllowance и getTotalSupply относятся к обзорным (view).
   Они возвращают запрашиваемое значение не пользователю, а специальному
   промежуточному контракту (proxy-contract). Во входящих параметрах для вызова
   этих функций пользователь должен указать адрес промежуточного контракта *)

(* Функция getBallance возвращает значение баланса заданного адреса *)
function getBalance (const owner : address; const contr : contract(amt); var s :
storage) : return is
 block {
     // присваиваем константе ownerAccaunt данные аккаунта
    const ownerAccount : account = getAccount(owner, s);
 }
 // возвращаем промежуточному контракту баланс аккаунта
 with (list [transaction(ownerAccount.balance, 0tz, contr)], s)

(* Функция getAllowance возвращает количество разрешенных для расходования
   токенов запрашиваемого аккаунта *)
function getAllowance (const owner : address; const spender : address; const
contr : contract(amt); var s : storage) : return is
 block {
     // получаем данные аккаунта, а из них – количество разрешенных для
        расходования токенов
    const ownerAccount : account = getAccount(owner, s);
    const spenderAllowance : amt = getAllowance(ownerAccount, spender, s);
 } with (list [transaction(spenderAllowance, 0tz, contr)], s)

(* Функция getTotalSupply возвращает количество токенов на балансах всех
   пользователей *)
function getTotalSupply (const contr : contract(amt); var s : storage) : return
is
```

```
 block {
   skip
 } with (list [transaction(s.totalSupply, 0tz, contr)], s)

(* Главная функция принимает название псевдо-точки входа и ее параметры *)
function main (const action : entryAction; var s : storage) : return is
 block {
   skip
 } with case action of
   | Transfer(params) -> transfer(params.0, params.1.0, params.1.1, s)
   | Approve(params) -> approve(params.0, params.1, s)
   | GetBalance(params) -> getBalance(params.0, params.1, s)
   | GetAllowance(params) -> getAllowance(params.0.0, params.0.1, params.1, s)
   | GetTotalSupply(params) -> getTotalSupply(params.1, s)
 end;
```

Код на LIGO нужно скомпилировать в Michelson, чтобы виртуальная машина Tezos могла его исполнить. На прошлом уроке мы вставили код контракта из двух строчек напрямую в скрипт развертывания. Код токена включает 200 строк, поэтому лучше сохранить его отдельным файлом и импортировать в скрипт с помощью команды import.

Откройте онлайн-среду LIGO и вставьте код токена в поле редактора. В выпадающем списке выберите пункт Compile Contract и поставьте галочку в поле Output Michelson in JSON format. Нажмите кнопку Run. Компилятор выдаст готовый код под полем редактора.

Создайте в папке token файл token.json и вставьте в него JSON-код.



На прошлом уроке мы вписали в скрипт deploy.ts данные тестового аккаунта и настроили RPC-ссылку публичного узла тестовой сети Tezos. Мы можем использовать этот код для публикации токена.

Создайте в папке taq-test файл token-deploy.ts и вставьте в него код из deploy.ts. Затем его нужно доработать: добавить метод file sync для чтения других файлов, импортировать код контракта и указать начальное состояние хранилища.

После методов import добавьте метод file sync и константу **Tezos** для вызова методов Taquito:

```
const fs = require("fs");
const { Tezos } = require('@taquito/taquito')
```

Замените метод *try*. После этого задайте исходное состояние хранилища — общее количество токенов и баланс пользователя:

```
try {
   const op = await tezos.contract.originate({
```

```
        // считываем код из файла token.json
        code: JSON.parse(fs.readFileSync("./token.json").toString()),
        // задаем состояние хранилища на языке Michelson. Замените оба адреса на
           адрес своего аккаунта в тестовой сети,
        // а числа — на количество токенов, которое вы хотите выпустить
        init:
            '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" (Pair { Elt
"tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" 1000 } 1000) } 1000)',
    })
```

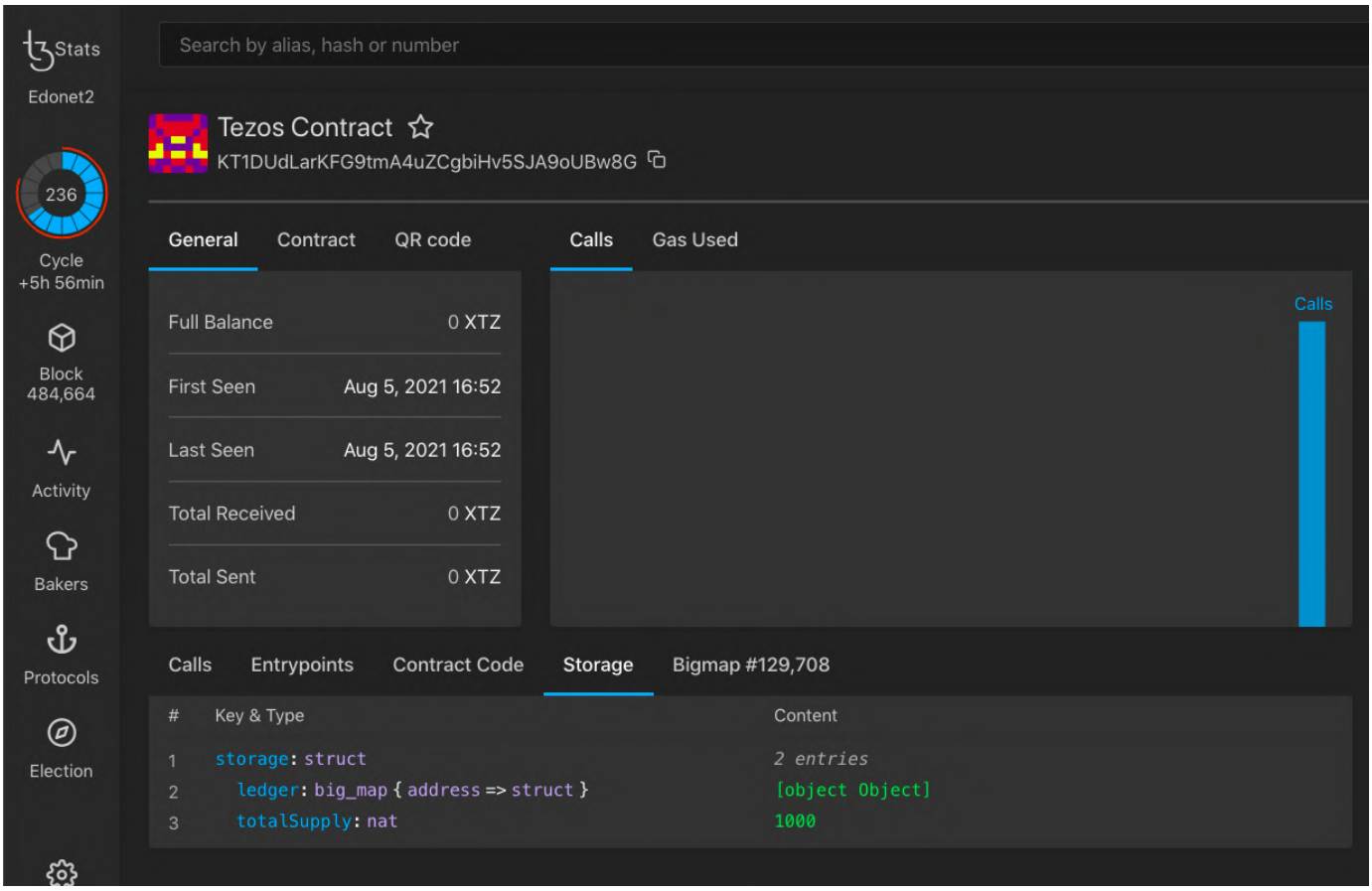
Готовый код выглядит так:

```
import { TezosToolkit } from '@taquito/taquito';
import { importKey } from '@taquito/signer';

const { Tezos } = require('@taquito/taquito')
const fs = require("fs");

const provider = 'https://florencenet.api.tez.ie';

async function deploy() {
 const tezos = new TezosToolkit(provider);
 await importKey(
   tezos,
   "hoqfgsoy.qyisbhtk@tezos.example.org", // почта
   "ZnnZLS0v6O", // пароль
   [
     "able", // мнемоника
     "public",
     "usual",
     "hello",
     "october",
     "owner",
     "essence",
     "old",
     "author",
     "original",
     "various",
     "gossip",
     "core",
```

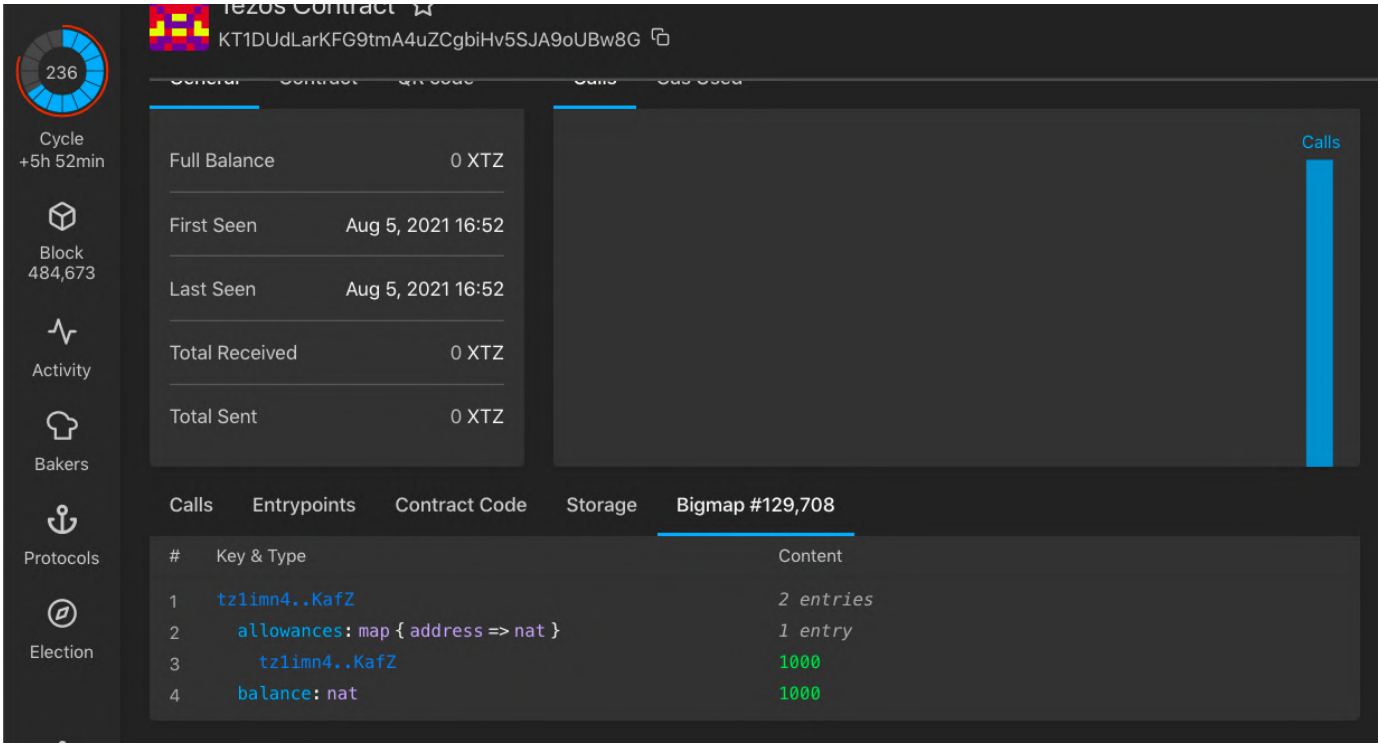
```
      "high",
      "hire"
    ].join(' '),
    "2bed8dc244ee43a1e737096c4723263c269049d8" // приватный ключ
  );

  try {
    const op = await tezos.contract.originate({
        // считываем код из файла token.json
        code: JSON.parse(fs.readFileSync("./token.json").toString()),
        // задаем состояние хранилища на языке Michelson. Замените оба адреса на
           адрес своего аккаунта в тестовой сети,
        // а числа – на количество токенов, которое вы хотите выпустить
        init:
            '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" (Pair { Elt
"tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" 1000 } 1000) } 1000)',
    })

    // начало развертывания
    console.log('Awaiting confirmation...');
    const contract = await op.contract();
    // отчет о развертывании: количество использованного газа, значение хранилища
    console.log('Gas Used', op.consumedGas);
    console.log('Storage', await contract.storage());
    // хеш операции, по которому можно найти контракт в блокчейн-обозревателе
    console.log('Operation hash:', op.hash);
  } catch (ex) {
    console.error(ex);
  }
}

deploy();
```

Откройте терминал и выполните команду npx ts-node token-deploy.ts. Через несколько минут Taquito опубликует контракт токена в тестовой сети и выдаст хеш операции.

Найдите ее в florence.tzstats и проверьте состояние хранилища контракта. У нас в нем записана тысяча токенов, у вас — ваше количество.



В поле Bigmap находятся записи об адресах владельцев токенов. При публикации контракта мы выпустили все токены на тестовый адрес.



Отправляем токены с помощью Taquito

Помните, как мы добавили число в хранилище контракта с помощью Taquito? Теперь усложним задачу: вызовем функцию transfer и передадим токены другому пользователю.

Установите кошелек Tezos и создайте аккаунт, если не сделали это на прошлом уроке. Мы рекомендуем Temple Wallet, потому что он из коробки поддерживает тестовые сети Tezos.

Откройте VS Code и создайте файл token-transfer.ts. Вставьте в него код:

```
// импортируем методы Taquito и файл с данными тестового аккаунта acc.json

import { TezosToolkit } from '@taquito/taquito';
import { InMemorySigner } from '@taquito/signer';
const acc = require('./acc.json');
export class token_transfer {

   // настраиваем ссылку на публичный узел тестовой сети
   private tezos: TezosToolkit;
   rpcUrl: string;

  constructor(rpcUrl: string) {
   this.tezos = new TezosToolkit(rpcUrl);
   this.rpcUrl = rpcUrl

   // считываем почту, пароль и мнемоническую фразу, из которой можно получить
      приватный ключ
   this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
acc.password, acc.mnemonic.join(' ')))
  }

  // объявляем метод transfer, который принимает параметры:
  //
  // 1) contract — адрес контракта;
  // 2) sender — адрес отправителя;
  // 3) receiver — адрес получателя;
  // 4) amount — количество токенов для отправки.

   public transfer(contract: string, sender: string, receiver: string, amount:
number) {
       this.tezos.contract
           .at(contract) // обращаемся к контракту по адресу
```

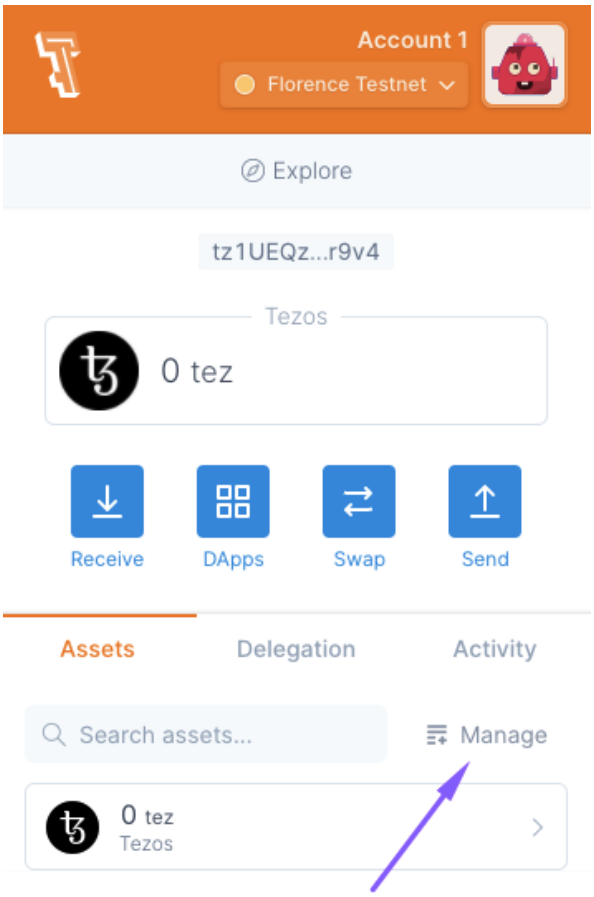
```
            .then((contract) => {
                console.log(`Sending ${amount} from ${sender} to ${receiver}...`);
                // обращаемся к точке входа transfer, передаем ей адреса
отправителя и получателя, а также количество токенов для отправки.
                return contract.methods.transfer(sender, receiver, amount).send()
            })
            .then((op) => {
                console.log(`Awaiting for ${op.hash} to be confirmed...`);
                return op.confirmation(1).then(() => op.hash); // ждем одно
подтверждение сети
            })
            .then((hash) => console.log(`Hash: https://florence.tzstats.com/
${hash}`)) // получаем хеш операции
            .catch((error) => console.log(`Error: ${JSON.stringify(error, null,
2)}`));

    }

}
```

Теперь создайте файл transfer.ts и вставьте в него код:

```
import { token_transfer } from './token-transfer';

const RPC_URL = "https://florencenet.api.tez.ie"
const CONTRACT = "KT1DUdLarKFG9tmA4uZCgbiHv5SJA9oUBw8G" // адрес опубликованного
контракта
const SENDER = "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" // публичный адрес
отправителя – возьмите его из acc.json
const RECEIVER = "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" // публичный адрес
получателя – возьмите его из кошелька Tezos, который вы создали
const AMOUNT = 3 // количество токенов для отправки. Можете ввести другое число
new token_transfer(RPC_URL).transfer(CONTRACT, SENDER, RECEIVER, AMOUNT);
```

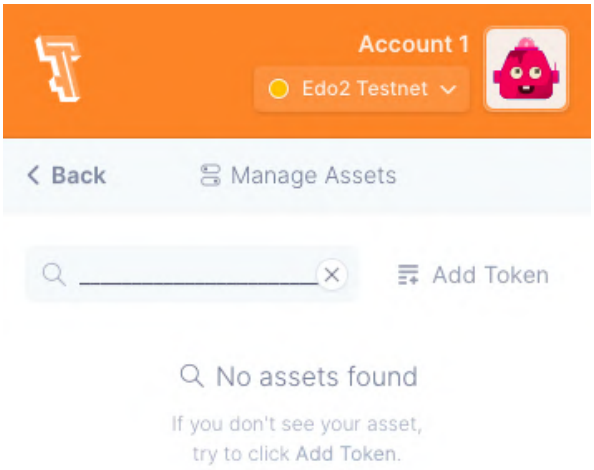
Откройте консоль и выполните команду npx ts-node transfer.ts. Подождите, пока консоль вернет хеш операции.

Теперь откройте кошелек Temple Wallet. Нажмите на кнопку **Tezos Mainnet** в верхней части окна и выберите пункт **Florence Testnet** в выпадающем списке.

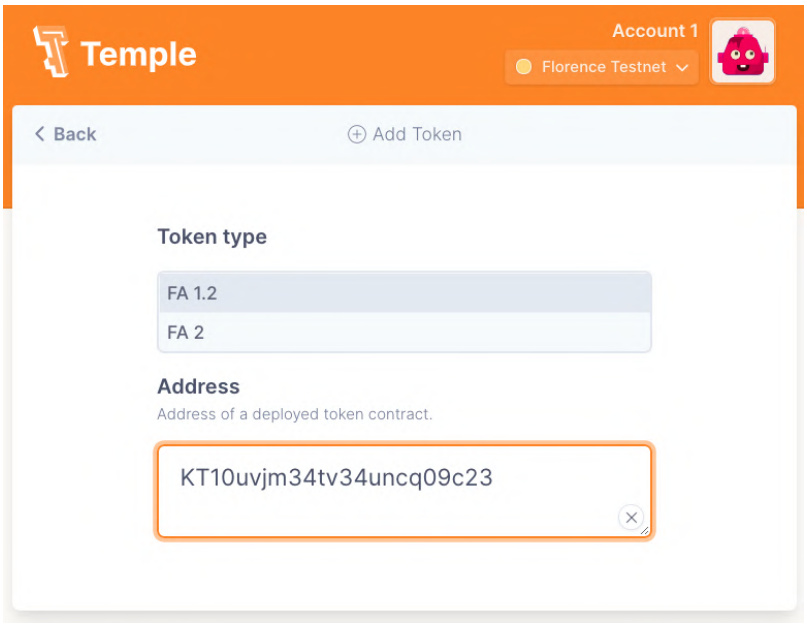


Кошелек выглядит пустым, но на самом деле токены уже пришли на ваш адрес. Temple не видит токены, потому что в коде контракта нет метаданных с названием и другими параметрами. Метаданные — тема следующего урока. Но сейчас мы можем добавить их вручную и увидеть токены на кошельке.

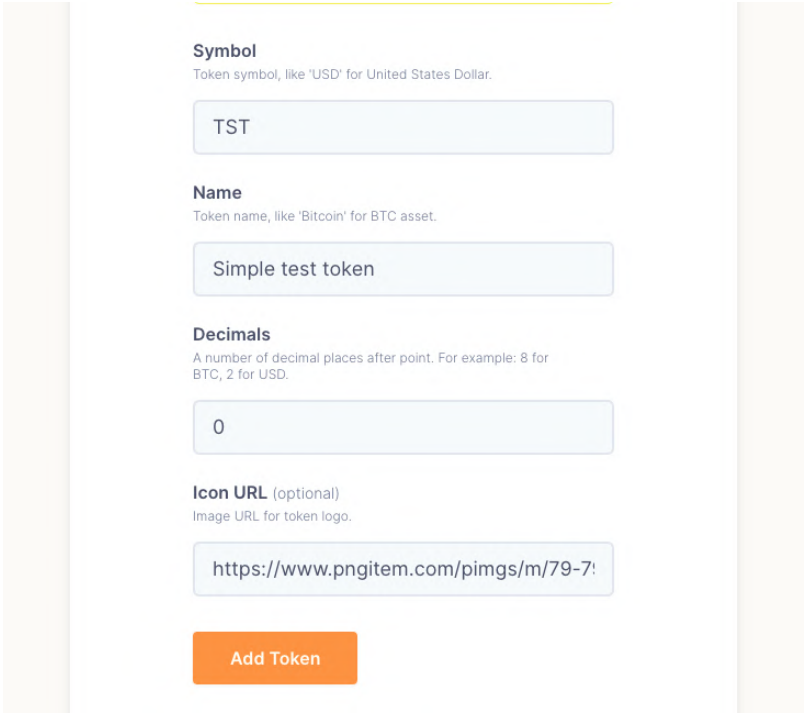
Скопируйте адрес контракта токена из файла transfer.ts. Откройте Temple Wallet и нажмите кнопку **Manage** справа от поля **Search Assets**, а затем — кнопку **Add Token**.



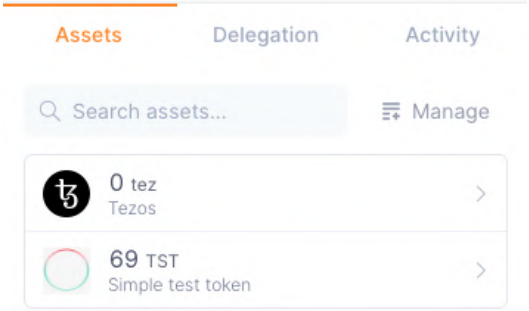
Temple откроет браузер с вкладкой, на которой можно добавить токен вручную. Выберите стандарт FA 1.2 в выпадающем списке **Token type**, а затем вставьте адрес смарт-контракта в поле **Address**.



Заполните данные о токене: тикер, описание, число знаков после запятой и ссылку на логотип. Пишите, что нравится, но оставьте значение «0» в поле Decimals. Нажмите кнопку Add Token, чтобы добавить в кошелек токен с заданными настройками.



Готово! Теперь Temple Wallet отображает ваш токен. Он подхватил точки входа FA 1.2, а значит вы можете просматривать баланс и отправлять токены из интерфейса кошелька. Однако получателю придется вручную настраивать метаданные, чтобы увидеть перевод.



На следующем уроке мы разберемся с метаданными и опубликуем полноценный токен, который кошельки будут видеть сразу. А еще выпустим NFT стандарта FA 2.

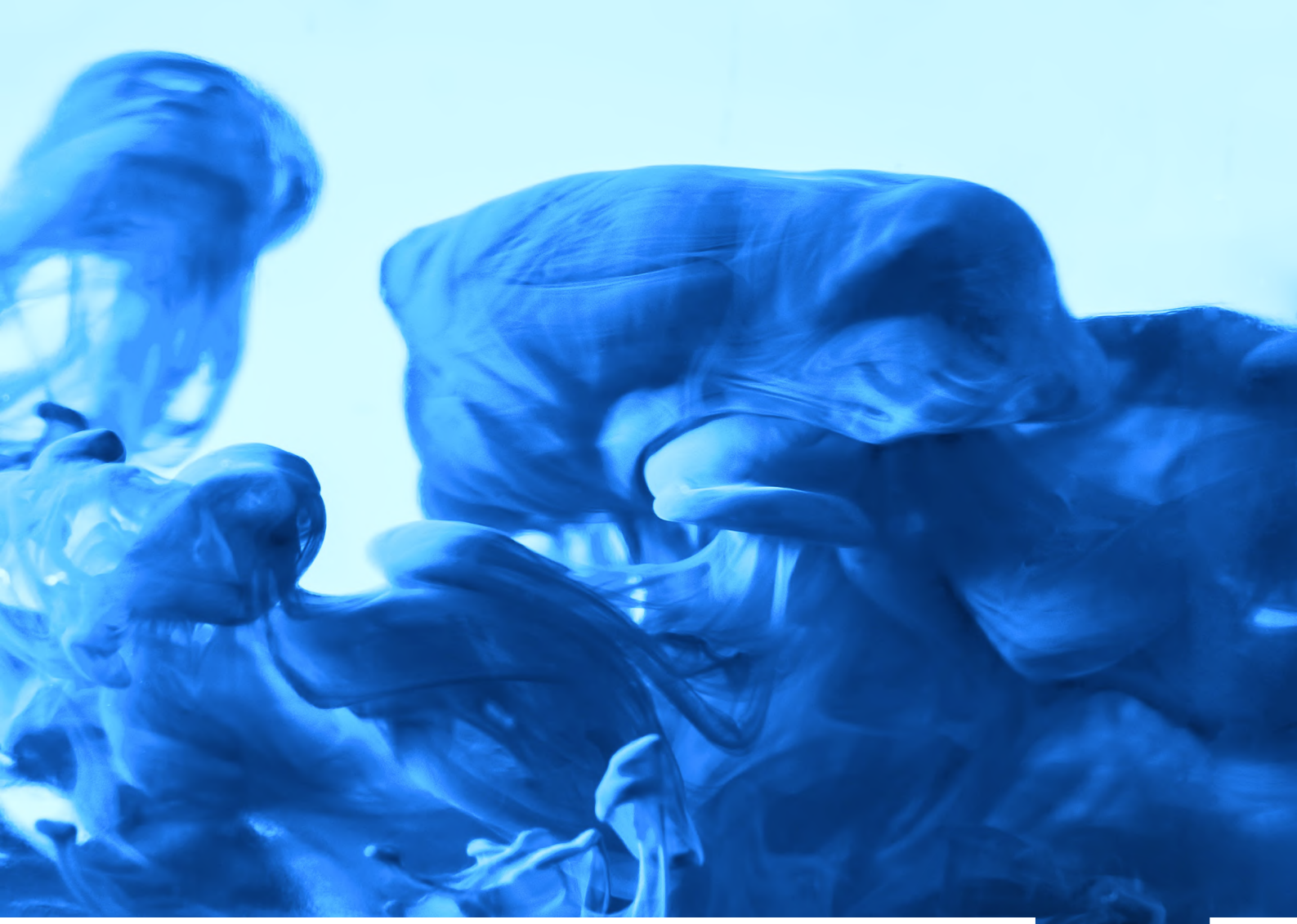
Подбиваем итоги

Токены — это записи в хранилище смарт-контрактов вроде «Адресу TZ1 принадлежит 1000 монет». Передача токенов происходит так:

1. Пользователь или приложение вызывает смарт-контракт.
2. Смарт-контракт проверяет, может ли пользователь отправить токены.
3. Смарт-контракт обновляет записи о балансах пользователей: уменьшает количество токенов отправителя на сумму перевода, а затем добавляет ее к балансу получателя.

Разработчики протокола создают стандарты, которые описывают основные функции токенов: хранение записей о пользователях, проверку балансов и проведение транзакций. В Tezos два популярных стандарта: FA 1.2 для взаимозаменяемых активов и FA 2 для NFT.

Разработчики могут писать смарт-контракты токенов с нуля или использовать имплементации — шаблоны с базовыми функциями. Продвинутые механизмы вроде выпуска и сжигания токенов нужно создавать самостоятельно.



5

Метаданные токенов и выпуск NFT на Tezos

На прошлом уроке мы изучили стандарт FA1.2 и опубликовали смарт-контракт токена в тестовой сети Tezos. Но мы не использовали метаданные, поэтому токен не отображался в кошельке автоматически.

На этом уроке мы добавим в смарт-контракт метаданные, чтобы его видели кошельки и блокчейн-обозреватели. Затем изучим стандарт NFT в сети Tezos и создадим невзаимозаменяемый токен.

Как приложения видят токены

Напомним, токены — это не монетки и не отдельные файлы, а записи в хранилище смарт-контрактов. Если в вашем кошельке есть 100 kUSD, то в хранилище контракта записано «Адресу TZ1 принадлежит 100 токенов».

Кроме балансов пользователей в хранилище находятся метаданные — информация о смарт-контракте: название и краткое описание токена, тикер, число знаков после запятой, идентификатор (id), ссылка на логотип, адрес владельца, псевдо-точки входа и другие параметры.

Метаданные нужны, чтобы приложения видели токены и правильно отображали информацию о них. Если метаданные отсутствуют, приложение не увидит токен, а если заполнены неправильно — отобразит его с ошибками.

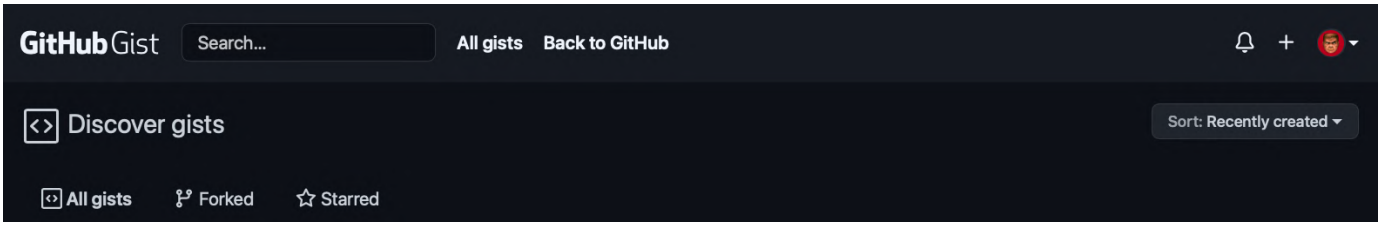
Создаем JSON-файл с метаданными

С 2020 года большинство разработчиков Tezos использует стандарт TZIP-16, согласно которому метаданные хранятся в JSON-файле.

Разработчик должен загрузить файл на любой публичный сервер — собственный сайт, IPFS или Github, а затем разместить ссылку на него в хранилище смарт-контракта.

Мы добавим метаданные в код токена token.ligo из прошлого урока. Для этого воспользуемся сервисом Gist от Github: создадим JSON-файл и получим прямую ссылку на него (Uniform Resource Identifier, URI).

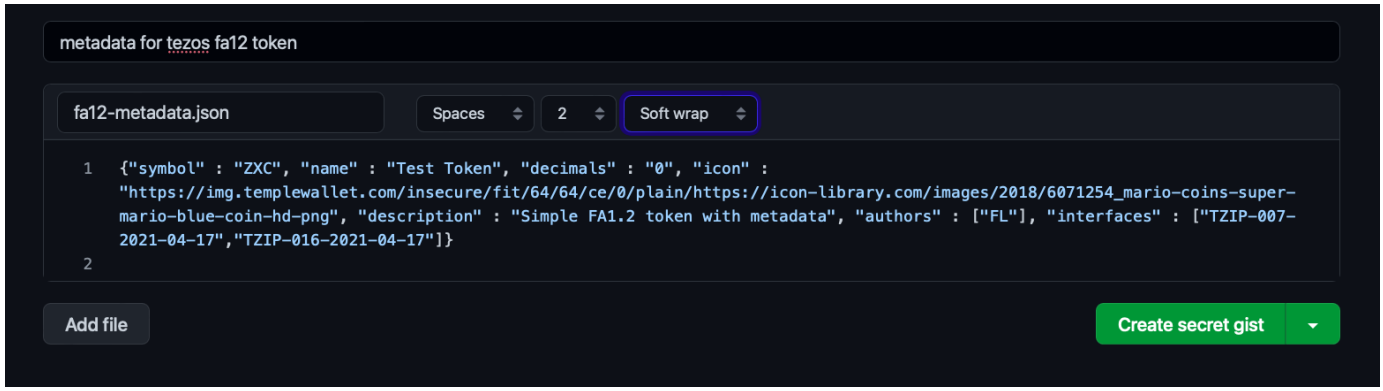
Зарегистрируйтесь или авторизируйтесь на Github, а затем перейдите в Gist. Нажмите на + в правом верхнем углу окна, чтобы создать новый файл.



Назовите файл fa12-metadata.json и вставьте в него шаблон:

```
{"symbol" : "Короткое название, тикер", "name" : "Полное название", "decimals" :
"0", "icon" : "ссылка на картинку. Она должна заканчиваться на .jpg, .png или
.svg", "description" : "Описание токена", "authors" : "автор", "interfaces" :
["TZIP-007-2021-04-17","TZIP-016-2021-04-17"]}
```

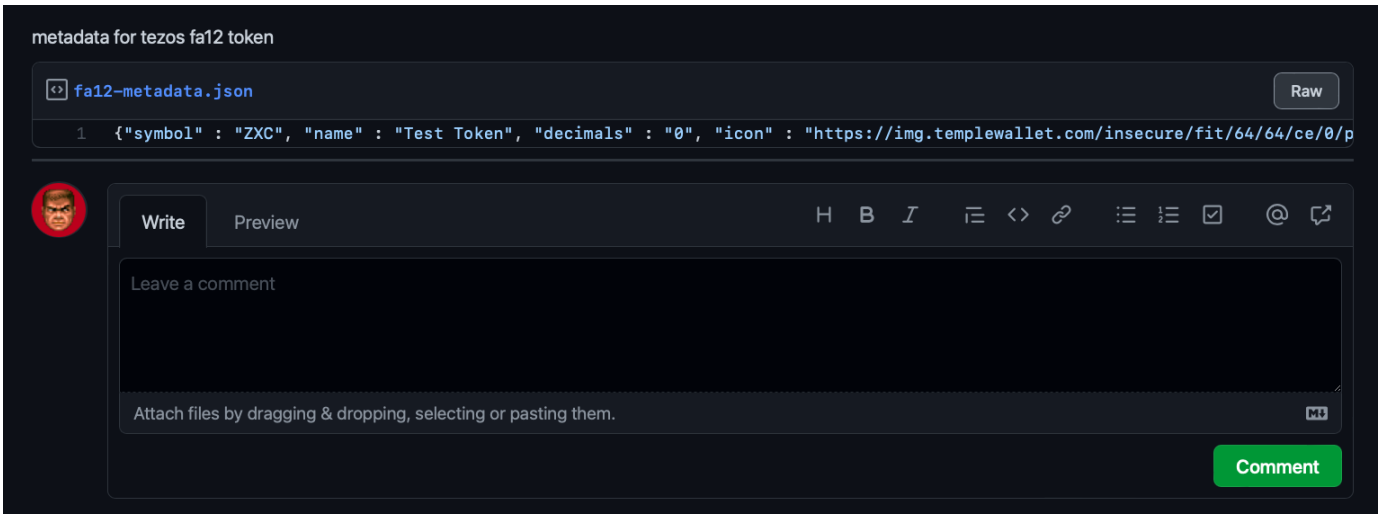
Заполните поля: введите тикер, название токена и добавьте ссылку на логотип. Поля **decimals** и **interfaces** оставьте как есть:



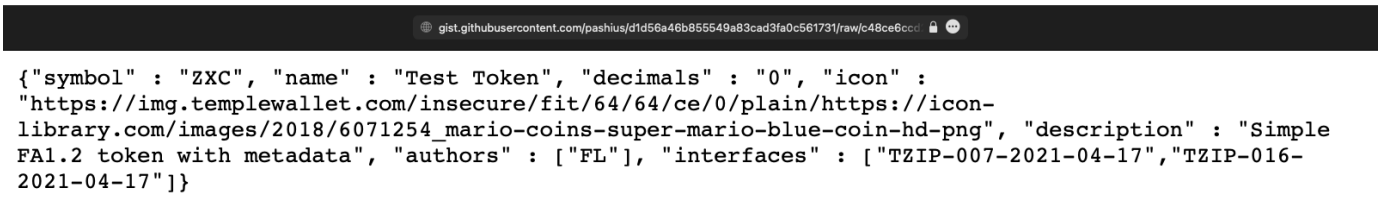
- **decimals** — количество знаков после запятой. Если установить его больше 0, например 3, кошельки отобразят токен в виде 0,001. Мы оставляем значение по умолчанию, чтобы не усложнять пример;
- **interfaces** — правила отображения метаданных и список стандартных точек входа, которые использует контракт. Изменение этого поля может привести к проблемам с отображением токена в некоторых кошельках.

Откройте выпадающий список на зеленой кнопке, и выберите пункт **Create public gist**. После этого нажмите саму кнопку. Так вы сохраните файл и сделаете его доступным для всех пользователей.

После обновления страницы нажмите кнопку **Raw** в правом верхнем углу окна, чтобы открыть файл по прямой ссылке.



Не закрывайте вкладку с файлом, она понадобится через несколько минут.



Добавляем метаданные в смарт-контракт

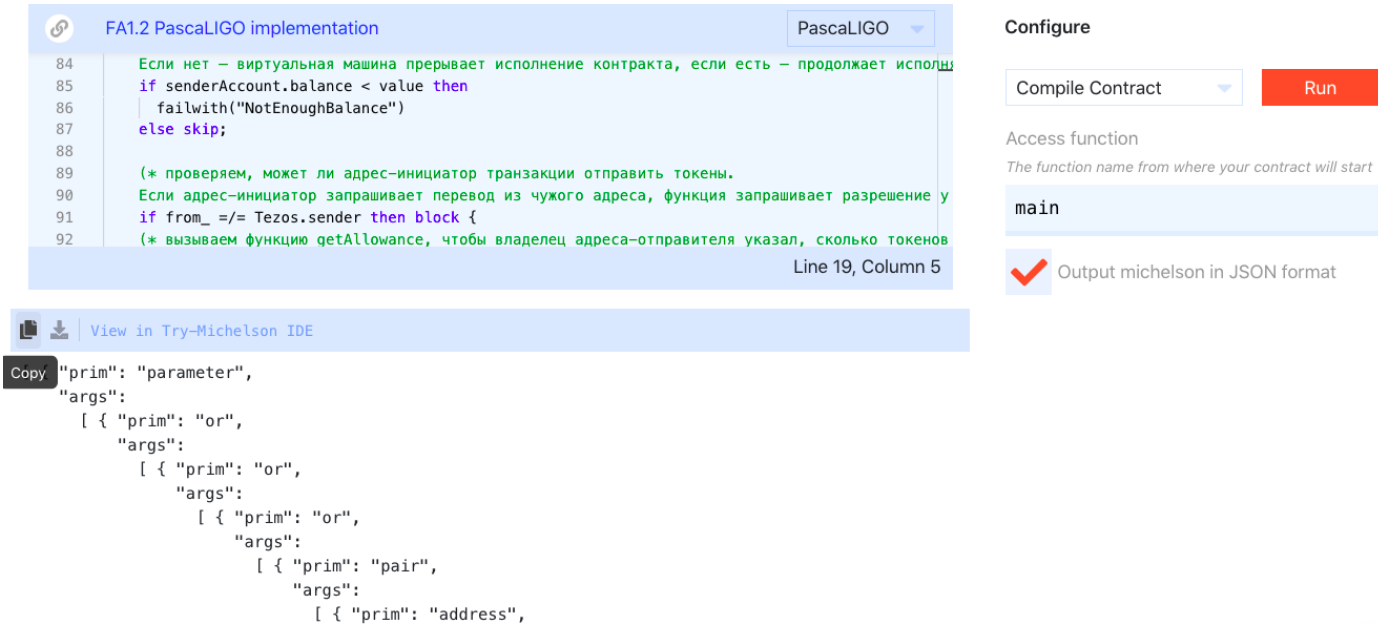
URI-ссылка на метаданные должна находиться в хранилище контракта в специальном формате — *`metadata : big_map`* (string, bytes). Добавим его в storage.

Запустите VS Code и откройте файл token.ligo. В начале контракта найдите объявление типа *`storage`* и добавьте в него формат метаданных.

```
(* объявляем тип хранилища смарт-контракта. Он хранит общее количество токенов, а также big_map,
который связывает публичные адреса и данные account пользователей *)
type storage is
  record [
    metadata        : big_map (string, bytes);
    totalSupply     : amt;
    ledger          : big_map (address, account);
  ]
```

Мы обновили контракт на LIGO. Теперь его нужно скомпилировать в формат JSON для публикации в тестнете.

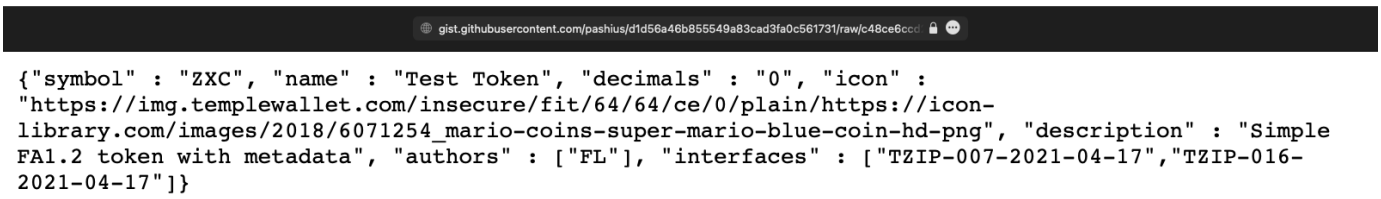
Скопируйте содержимое файла token.ligo и перейдите на сайт ide.ligolang.org. Выберите пункт **Compile Contract** в выпадающем списке и поставьте галочку в поле **Output Michelson** in **JSON format**. Нажмите кнопку Run. Компилятор выдаст готовый код под полем редактора.



Перейдите в VS Code, откройте файл token.json. Замените старый код новым и сохраните файл.

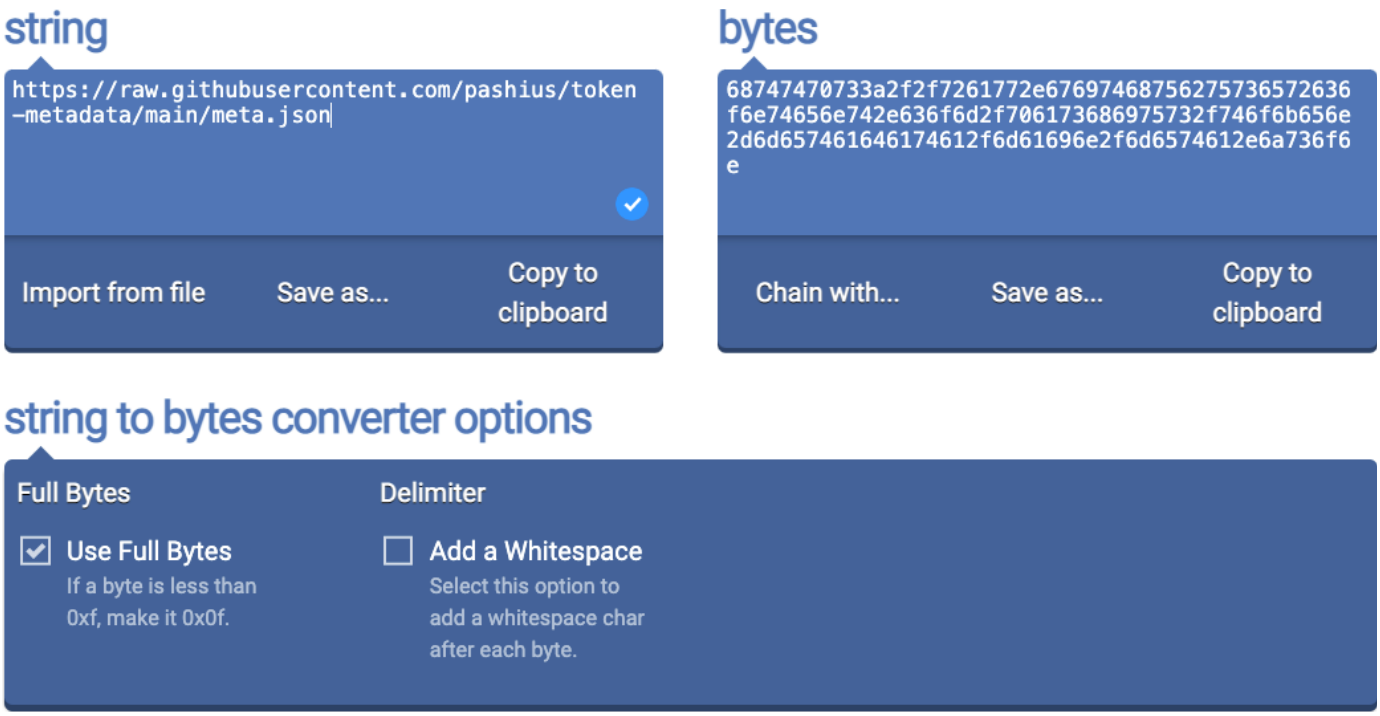
Теперь в **storage** есть поле **metadata**: кошельки и блокчейн-обозреватели будут искать в нем ссылку на JSON-файл с метаданными. Пока в этом поле пусто: мы укажем URI файла в хранилище контракта перед развертыванием.

Чтобы виртуальная машина Michelson могла прочесть ссылку, ее нужно перевести в байтовый формат. Вернитесь на вкладку с URI на meta.json и скопируйте содержимое адресной строки.



Откройте онлайн-конвертер вроде Onlinestringtools. Вставьте текст ссылки в поле **String** и снимите галочку с **Add Whitespaces**, чтобы убрать пропуски между байтами.

Конвертер выдаст длинное число в поле **Bytes** — это и есть ссылка в байтовом формате. Мы будем использовать ее позже, а пока не закрывайте вкладку.



Переключитесь на VS Code и создайте файл storage.tz. Вставьте в него шаблон хранилища:

```
'(Pair (Pair { Elt "публичный адрес из файла acc.json" (Pair { Elt "публичный
адрес из файла acc.json" количество токенов } количество токенов) } { Elt ""
0xссылка на meta.json в байтовом формате }) количество токенов)'
```

Заполните поля шаблона. Важно: перед ссылкой на meta.json нужно обязательно добавить пустую строку — "". Это ключ, по которому виртуальная машина Tezos понимает, что дальше будет ссылка на метаданные.

Публичный адрес обязательно должен быть в кавычках, а количество токенов и ссылка — без. Перед ссылкой в байтовом формате обязательно допишите *0x*, иначе компилятор ее не прочтет.

```
taq-test > stuff > storage.tz
1    '(Pair (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
2                       (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" 1000 } 1000) }
3                 { Elt ""
4                       0x68747470733a2f2f636c6f7564666c6172652d697066732e636f6d2f697066732f6261666b
5           1000)'
6
```

Скопируйте код из storage.js и переключитесь на token-deploy.ts. Замените старую запись состояния хранилища на новую.

```
try {
  const op = await tezos.contract.originate({
    // считываем код из файла token.json
    code: JSON.parse(fs.readFileSync("./token.json").toString()),
    // задаем состояние хранилища на языке Michelson. Замените оба адреса на адрес своего а
    // а числа — на количество токенов, которое вы хотите выпустить
    init:
    '(Pair (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" (Pair { Elt "tz1UEQzJbuaGJgwv
,
  })
```

Откройте терминал в VS Code. Убедитесь, что находитесь в папке taq-test, и выполните команду:

```
npx ts-node token-deploy-ts
```

Taquito выдаст ссылку на хеш операции. Перейдите на сайт tzstats.io, найдите по хешу адрес контракта и скопируйте его.

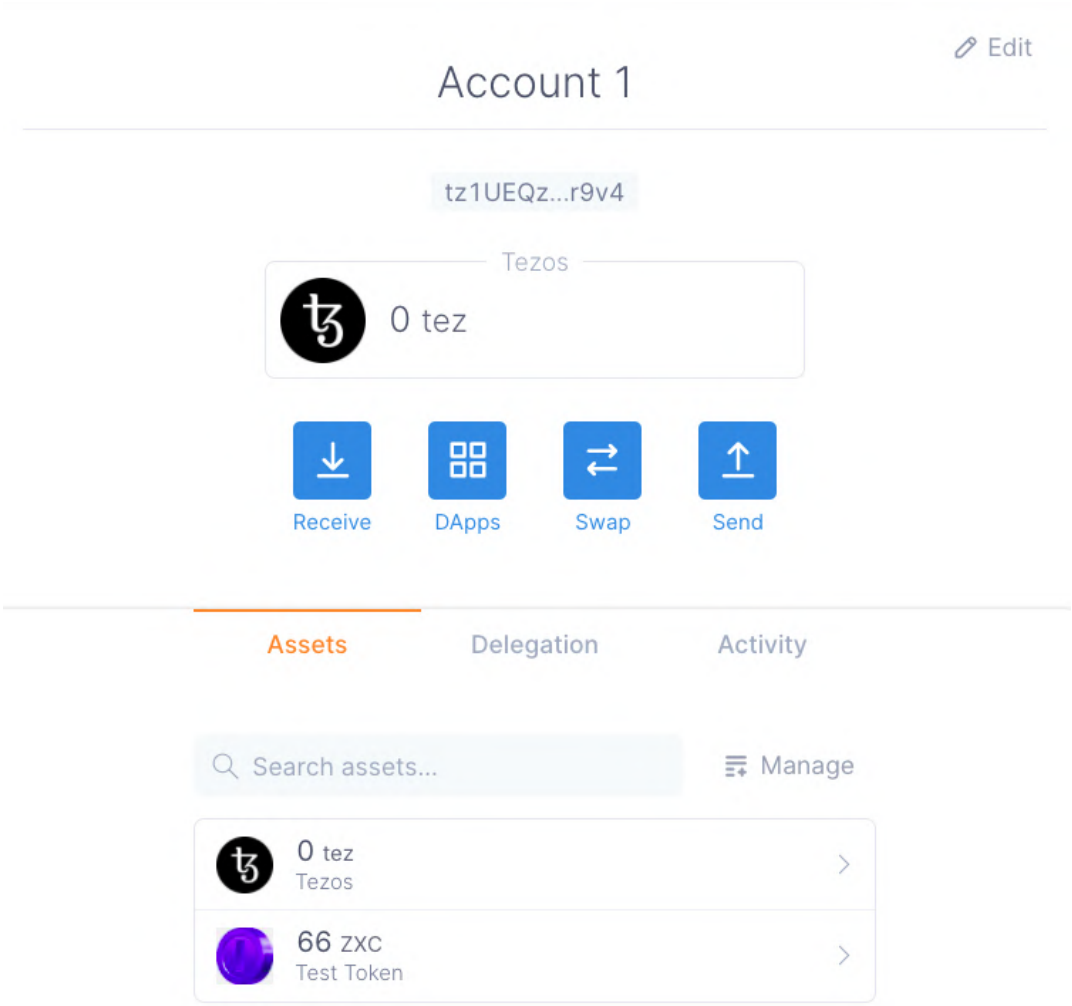
Проверьте, что кошельки видят метаданные — подготовьте и выполните транзакцию. Откройте файл transfer.ts и замените старый адрес токена новым, который вы только что опубликовали.

```
taq-test > TS transfer.ts > [◈] CONTRACT
1    import { token_transfer } from './token-transfer';
2
3    const RPC_URL = "https://florencenet.api.tez.ie"
4    const CONTRACT = "KT1PgHZ6BpU2dhErtffMAcdCrZNkTgRpruwv" //адрес контракта токена
5    const SENDER = "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" //публичный адрес отправителя —
6    const RECEIVER = "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" //публичный адрес получателя —
7    const AMOUNT = 66 // количество токенов для отправки. Можете выбрать другое число
8    new token_transfer(RPC_URL).transfer(CONTRACT, SENDER, RECEIVER, AMOUNT);
```

Откройте терминал и выполните команду:

```
npx ts-node transfer.ts
```

Дождитесь подтверждения транзакции и проверьте кошелек. Он должен сразу отобразить токены.



Разбираемся со стандартом FA 2

Уникальность NFT (non-fungible token) обеспечивают два фактора — адрес контракта-эмитента и идентификатор (id). У взаимозаменяемых токенов идентификаторов нет.

NFT можно выпустить с помощью простого контракта из двух функций: transfer для передачи токенов и update для обновления балансов пользователей. Технически, такой токен будет уникальным. Но если не придерживаться стандартов FA 1.2 и FA 2, кошельки не увидят этот NFT. Для передачи токена придется пользоваться клиентом Tezos и терминалом.

Чтобы унифицировать работу с NFT, разработчики Tezos создали стандарт FA 2. Он описывает интерфейс для работы с взаимозаменяемыми, невзаимозаменяемыми, непередаваемыми и другими видами токенов.

Стандарт FA 2 сложнее, чем FA 1.2, но дает разработчикам больше свободы. Например, с помощью FA 2 можно выпустить несколько токенов в одном контракте или группировать транзакции в пакеты (batch), чтобы сэкономить газ.

Код FA 2-токена состоит из четырех частей:

- сообщения об основных ошибках;
- интерфейсы — объявление всех кастомных типов данных;
- функции для работы с операторами — пользователями, которые могут передавать токены;
- ядро — метаданные, хранилище, функции передачи токенов и проверки баланса.

Имплементация NFT-стандарта FA2 на LIGO выглядит так:

```
// ERRORS

const fa2_token_undefined = "FA2_TOKEN_UNDEFINED"
const fa2_insufficient_balance = "FA2_INSUFFICIENT_BALANCE"
const fa2_tx_denied = "FA2_TX_DENIED"
const fa2_not_owner = "FA2_NOT_OWNER"
const fa2_not_operator = "FA2_NOT_OPERATOR"
const fa2_operators_not_supported = "FA2_OPERATORS_UNSUPPORTED"
const fa2_receiver_hook_failed = "FA2_RECEIVER_HOOK_FAILED"
const fa2_sender_hook_failed = "FA2_SENDER_HOOK_FAILED"
const fa2_receiver_hook_undefined = "FA2_RECEIVER_HOOK_UNDEFINED"
const fa2_sender_hook_undefined = "FA2_SENDER_HOOK_UNDEFINED"

// INTERFACE

// объявляем тип идентификатора токена – натуральное число
type token_id is nat

// объявляем типы входящих параметров, которые принимает функция передачи токена:
   адрес получателя, id и количество токенов. В тип transfer добавляем адрес
   отправителя
type transfer_destination is
[@layout:comb]
record [
 to_ : address;
 token_id : token_id;
 amount : nat;
]

type transfer is
[@layout:comb]
record [
 from_ : address;
```

```
 txs : list(transfer_destination);
]

// объявляем типы для чтения баланса: адрес владельца адреса, id токена,
type balance_of_request is
[@layout:comb]
record [
 owner : address;
 token_id : token_id;
]

type balance_of_response is
[@layout:comb]
record [
 request : balance_of_request;
 balance : nat;
]

type balance_of_param is
[@layout:comb]
record [
 requests : list(balance_of_request);
 callback : contract (list(balance_of_response));
]

// объявляем тип оператора – адреса, который может отправлять токены
type operator_param is
[@layout:comb]
record [
 owner : address;
 operator : address;
 token_id: token_id;
]

// объявляем тип параметров, которые нужны для обновления списка операторов
type update_operator is
[@layout:comb]
 | Add_operator of operator_param
 | Remove_operator of operator_param
```

```
// объявляем тип, который содержит метаданные NFT: ID токена и ссылку
   на json-файл
type token_info is (token_id * map(string, bytes))

type token_metadata is
big_map (token_id, token_info)

// объявляем тип со ссылкой на метаданные смарт-контракта. Эти данные будут
   отображаться в кошельке
type metadata is
big_map(string, bytes)

// объявляем тип, который может хранить записи о нескольких токенах и их
   метаданных в одном контракте
type token_metadata_param is
[@layout:comb]
record [
 token_ids : list(token_id);
 handler : (list(token_metadata)) -> unit;
]

// объявляем псевдо-точки входа: передача токенов, проверка баланса, обновление
   операторов и проверка метаданных
type fa2_entry_points is
 | Transfer of list(transfer)
 | Balance_of of balance_of_param
 | Update_operators of list(update_operator)
 | Token_metadata_registry of contract(address)

type fa2_token_metadata is
 | Token_metadata of token_metadata_param

// объявляем типы данных для изменения разрешений на передачу токенов. Например,
   с их помощью можно сделать токен, который нельзя отправить на другой адрес
type operator_transfer_policy is
 [@layout:comb]
 | No_transfer
 | Owner_transfer
 | Owner_or_operator_transfer

type owner_hook_policy is
```

```
 [@layout:comb]
 | Owner_no_hook
 | Optional_owner_hook
 | Required_owner_hook

type custom_permission_policy is
[@layout:comb]
record [
 tag : string;
 config_api: option(address);
]

type permissions_descriptor is
[@layout:comb]
record [
 operator : operator_transfer_policy;
 receiver : owner_hook_policy;
 sender : owner_hook_policy;
 custom : option(custom_permission_policy);
]

type transfer_destination_descriptor is
[@layout:comb]
record [
 to_ : option(address);
 token_id : token_id;
 amount : nat;
]

type transfer_descriptor is
[@layout:comb]
record [
 from_ : option(address);
 txs : list(transfer_destination_descriptor)
]

type transfer_descriptor_param is
[@layout:comb]
record [
 batch : list(transfer_descriptor);
 operator : address;
```

```
]

// OPERATORS

// объявляем тип, который хранит записи об операторах в одном big_map
type operator_storage is big_map ((address * (address * token_id)), unit)

// объявляем функцию для обновления списка операторов
function update_operators (const update : update_operator; const storage :
operator_storage)
   : operator_storage is
 case update of
 | Add_operator (op) ->
   Big_map.update ((op.owner, (op.operator, op.token_id)), (Some (unit)),
storage)
 | Remove_operator (op) ->
   Big_map.remove ((op.owner, (op.operator, op.token_id)), storage)
 end

// объявляем функцию, которая проверяет, может ли пользователь обновить список
   операторов
function validate_update_operators_by_owner (const update : update_operator;
const updater : address)
   : unit is block {
     const op = case update of
       | Add_operator (op) -> op
       | Remove_operator (op) -> op
     end;
     if (op.owner = updater) then skip else failwith (fa2_not_owner)
   } with unit

// объявляем функцию, которая проверяет, может ли пользователь обновить список
   адресов владельцев токенов, и только в этом случае вызывает функцию обновления
function fa2_update_operators (const updates : list(update_operator); const
storage : operator_storage) : operator_storage is block {
 const updater = Tezos.sender;
 function process_update (const ops : operator_storage; const update :
update_operator) is block {
   const u = validate_update_operators_by_owner (update, updater);
 } with update_operators(update, ops)
} with List.fold(process_update, updates, storage)
```

```
type operator_validator is (address * address * token_id * operator_storage) ->
unit

// объявляем функцию, которая проверяет разрешения на передачу токенов. Если
   пользователь не может передать токен, функция прекращает выполнение контракта
function make_operator_validator (const tx_policy : operator_transfer_policy) :
operator_validator is block {
 const x = case tx_policy of
 | No_transfer -> (failwith (fa2_tx_denied) : bool * bool)
 | Owner_transfer -> (True, False)
 | Owner_or_operator_transfer -> (True, True)
 end;
 const can_owner_tx = x.0;
 const can_operator_tx = x.1;
 const inner = function (const owner : address; const operator : address; const
token_id : token_id; const ops_storage : operator_storage):unit is
   if (can_owner_tx and owner = operator)
   then unit
   else if not (can_operator_tx)
   then failwith (fa2_not_owner)
   else if (Big_map.mem  ((owner, (operator, token_id)), ops_storage))
   then unit
   else failwith (fa2_not_operator)
} with inner

// объявляем функцию для передачи токена владельцем
function default_operator_validator (const owner : address; const operator :
address; const token_id : token_id; const ops_storage : operator_storage) : unit
is
 if (owner = operator)
 then unit
 else if Big_map.mem ((owner, (operator, token_id)), ops_storage)
 then unit
 else failwith (fa2_not_operator)

// объявляем функцию, которая собирает все транзакции одного токена в пакет
(batch)
function validate_operator (const tx_policy : operator_transfer_policy; const txs
: list(transfer); const ops_storage : operator_storage) : unit is block {
 const validator = make_operator_validator (tx_policy);
 List.iter (function (const tx : transfer) is
```

```
    List.iter (function (const dst : transfer_destination) is
      validator (tx.from_, Tezos.sender, dst.token_id ,ops_storage),
      tx.txs),
    txs)
} with unit

//MAIN


// объявляем тип данных для хранения записей о том, на каком адресе хранится
    токен с заданным id
type ledger is big_map (token_id, address)

// объявляем хранилище контракта: метаданные TZIP-16, реестр адресов и токенов,
    список операторов и ончейн-метаданные
type collection_storage is record [
 metadata : big_map (string, bytes);
 ledger : ledger;
 operators : operator_storage;
 token_metadata : token_metadata;
]


// объявляем функцию передачи токена. Она получает id токена, адрес отправителя
    и получателя, а затем проверяет, есть ли у отправителя право передать токен
function transfer (
 const txs : list(transfer);
 const validate : operator_validator;
 const ops_storage : operator_storage;
 const ledger : ledger) : ledger is block {
    // проверка права отправителя передать токен
    function make_transfer (const l : ledger; const tx : transfer) is
      List.fold (
        function (const ll : ledger; const dst : transfer_destination) is block {
          const u = validate (tx.from_, Tezos.sender, dst.token_id, ops_storage);
        } with
          // проверка количества передаваемых NFT. Подразумеваем, что контракт
выпустил только 1 токен с этим id
          // Если пользователь хочет передать 0, 0.5, 2 или другое количество
токенов, функция прерывает выполнение контракта
          if (dst.amount = 0n) then
```

```
  ll
        else if (dst.amount =/= 1n)
        then (failwith(fa2_insufficient_balance): ledger)
        else block {
          const owner = Big_map.find_opt(dst.token_id, ll);
        } with
          case owner of
            Some (o) ->
            // проверка, есть ли у отправителя токен
            if (o =/= tx.from_)
            then (failwith(fa2_insufficient_balance) : ledger)
            else Big_map.update(dst.token_id, Some(dst.to_), ll)
          | None -> (failwith(fa2_token_undefined) : ledger)
          end
      ,
      tx.txs,
      l
    )
} with List.fold(make_transfer, txs, ledger)

// объявляем функцию, которая вернет баланс отправителя
function get_balance (const p : balance_of_param; const ledger : ledger) :
operation is block {
 function to_balance (const r : balance_of_request) is block {
   const owner = Big_map.find_opt(r.token_id, ledger);
 }
 with
   case owner of
     None -> (failwith (fa2_token_undefined): record[balance: nat; request:
record[owner: address ; token_id : nat]])
   | Some (o) -> block {
     const bal = if o = r.owner then 1n else 0n;
   } with record [request = r; balance = bal]
   end;
 const responses = List.map (to_balance, p.requests);
} with Tezos.transaction(responses, 0mutez, p.callback)

// объявляем главную функцию с псевдо-точками входа. Эти псевдо-точки — основа
стандарта FA2
function main (const param : fa2_entry_points; const storage :
collection_storage) : (list (operation) * collection_storage) is
```

```
 case param of
   | Transfer (txs) -> block {
     const new_ledger = transfer (txs, default_operator_validator,
storage.operators, storage.ledger);
     const new_storage = storage with record [ ledger = new_ledger ]
   } with ((list [] : list(operation)), new_storage)
   | Balance_of (p) -> block {
     const op = get_balance (p, storage.ledger);
   } with (list [op], storage)
   | Update_operators (updates) -> block {
     const new_operators = fa2_update_operators(updates, storage.operators);
     const new_storage = storage with record [ operators = new_operators ];
   } with ((list [] : list(operation)), new_storage)
   | Token_metadata_registry (callback) -> block {
     const callback_op = Tezos.transaction(Tezos.self_address, 0mutez, callback);
   } with (list [callback_op], storage)
 end
```

Выпускаем NFT на FA 2

Откройте VS Code и создайте файл nft.ligo в папке taq-test. Вставьте в него код из окна выше или скопируйте его с Gist.

Скомпилируйте код на LIGO в JSON. Скопируйте содержимое файла окна и перейдите на сайт ide.ligolang.org. Выберите пункт **Compile Contract** в выпадающем списке и поставьте галочку в поле **Output Michelson in JSON format**. Нажмите кнопку **Run**. Компилятор выдаст готовый код под полем редактора. Создайте в VS Code файл с названием nft.json и скопируйте туда скомпилированный код.

Подготовьте два JSON-файла с метаданными — для и NFT и смарт-контракта. Перейдите в Gist и нажмите на **+** в правом верхнем углу, чтобы создать новый файл.

Назовите файл nft_meta.json и вставьте в него шаблон:

```
{"symbol" : "тикер",
 "name" : "название NFT",
 "description" : "Описание",
 "decimals" : "0",
 "isBooleanAmount" : true,
 "artifactUri" : "ссылка на токенизируемый объект",
 "thumbnailUri" : "логотип NFT",

 "minter" : "имя эмитента токена",
 "interfaces" : ["TZIP-007-2021-04-17","TZIP-016-2021-04-17", "TZIP-21"]}
```

Заполните все поля, кроме interfaces. Укажите ссылки на токенизируемый объект и логотип токена в виде «https://...», а не байтовом формате.

Откройте выпадающий список на зеленой кнопке и выберите пункт **Create public gist**. После этого нажмите саму кнопку. Так вы сохраните файл и сделаете его доступным для всех пользователей.

После обновления страницы нажмите кнопку **Raw** в правом верхнем углу окна, чтобы открыть файл по прямой ссылке. Не закрывайте вкладку.

Теперь создайте в Gist файл contract_meta.json. Вставьте в него шаблон:

```
{"name" : "название контракта",
 "description" : "описание контракта",

  "interfaces": ["TZIP-012-2020-11-17"]
}
```

Заполните метаданные контракта, нажмите кнопку **Create Public Gist**, а затем — кнопку **Raw**. Не закрывайте вкладку, она пригодится через несколько минут.

Подготовьте скрипт для публикации NFT. Создайте файл nft-deploy.ts и скопируйте в него код из token-deploy.ts.

Найдите метод code: *`JSON.parse()`* и замените в нем название файла для считывания:

```
code: JSON.parse(fs.readFileSync("./token.json").toString()),
  ↓↓↓
code: JSON.parse(fs.readFileSync("./nft.json").toString()),
```

```
try {
  const op = await tezos.contract.originate({
      // считываем код из файла nft.json
      code: JSON.parse(fs.readFileSync("./nft.json").toString()),
      // задаем состояние хранилища на языке Michelson.
      init:
      '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt
  })
```

Для публикации смарт-контракта нужно задать состояние хранилища. Для этого создайте файл nft-storage без расширения и вставьте в него шаблон:

```
'(Pair (Pair { Elt id "адрес вашего кошелька" } { Elt "" 0x ссылка на метаданные
контракта в байтовом формате}) { Elt (Pair "адрес вашего
кошелька" id) Unit } { Elt 0 (Pair id { Elt "" 0x контракта ссылка на метаданные
токена в байтовом формате }) })'
```

Заполните поля в шаблоне. Этот код должен быть одной строкой без переносов.

Указывайте идентификатор (id) токена без кавычек. Не используйте кириллицу в названии токена: Michelson работает только с Unicode, то есть латиницей.

Мы выпустим токен с id=5:

```
taq-test > nft-storage.tz
1   '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt ""
    0x68747470733a2f2f676973742e676974687562757365726572636f6e74656e742e636f6d2f706173686975732f623931
    343134316436633437396565313030343265323762656138313462653432f7261772f336461356234376462303162
    386330393534633837353930663636656432323032323536436438322f636f6e74726163745f6d6574612e6a736f6e }
    ) { Elt (Pair "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
    5) Unit } { Elt 0 (Pair 5 { Elt ""
    0x68747470733a2f2f676973742e676974687562757365726572636f6e74656e742e636f6d2f706173686975732f626630
    636533373337383134316362303930333965643234613631343161663372f7261772f656636373766643462623639
    643631643962643866636161623436626366383030626337366462 2f6e66746d6574322e6a736f6e }) })' \
    --burn-cap 0.6215
```

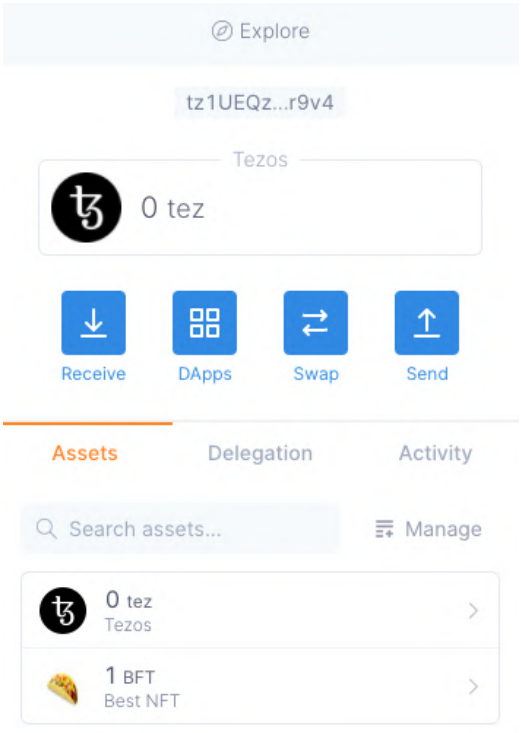
Вставьте код состояния хранилища в файл nft-deploy.ts в поле init. Не забудьте выделить его апострофами и поставить запятую. Наш скрипт выглядит так:

```
try {
  const op = await tezos.contract.originate({
    // считываем код из файла nft.json
    code: JSON.parse(fs.readFileSync("./nft.json").toString()),
    // задаем состояние хранилища на языке Michelson.
    init:
    '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt ""
    0x68747470733a2f2f676973742e676974687562757365726572636f6e74656e742e636f6d2f70617368697573
    2f623931343134316436633437396565313030343265323762656138313462653432f7261772f3364613562
    343764616230316238633039353463383735393066363665643232303232356364383322f636f6e74726163
    745f6d6574612e6a736f6e }) { Elt (Pair "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
    "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" 5) Unit } { Elt 0 (Pair 5 { Elt ""
    0x68747470733a2f2f676973742e676974687562757365726572636f6e74656e742e636f6d2f70617368697573
    2f626630636533373337383134316362303930333965643234613631343161663372f7261772f6566363737
    666434626236396564363164396264386663616162343662636638303062633736646 22f6e66746d657432
    2e6a736f6e }) })',
})

//начало развертывания
console.log('Awaiting confirmation...');
const contract = await op.contract();
//отчет о развертывании: количество использованного газа, значение хранилища
console.log('Gas Used', op.consumedGas);
```

Откройте терминал и выполните команду:

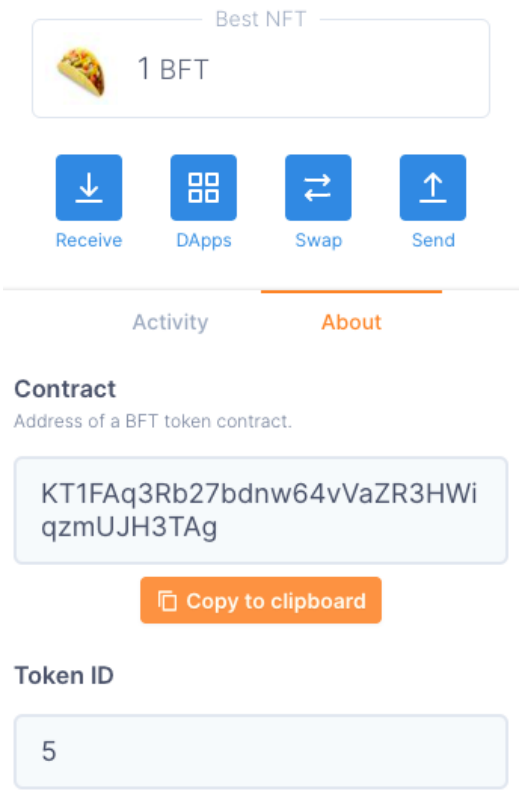
```
npx ts-node nft-deploy.js
```

Дождитесь завершения развертывания и проверьте свой кошелек. NFT появится в нем в течение минуты.



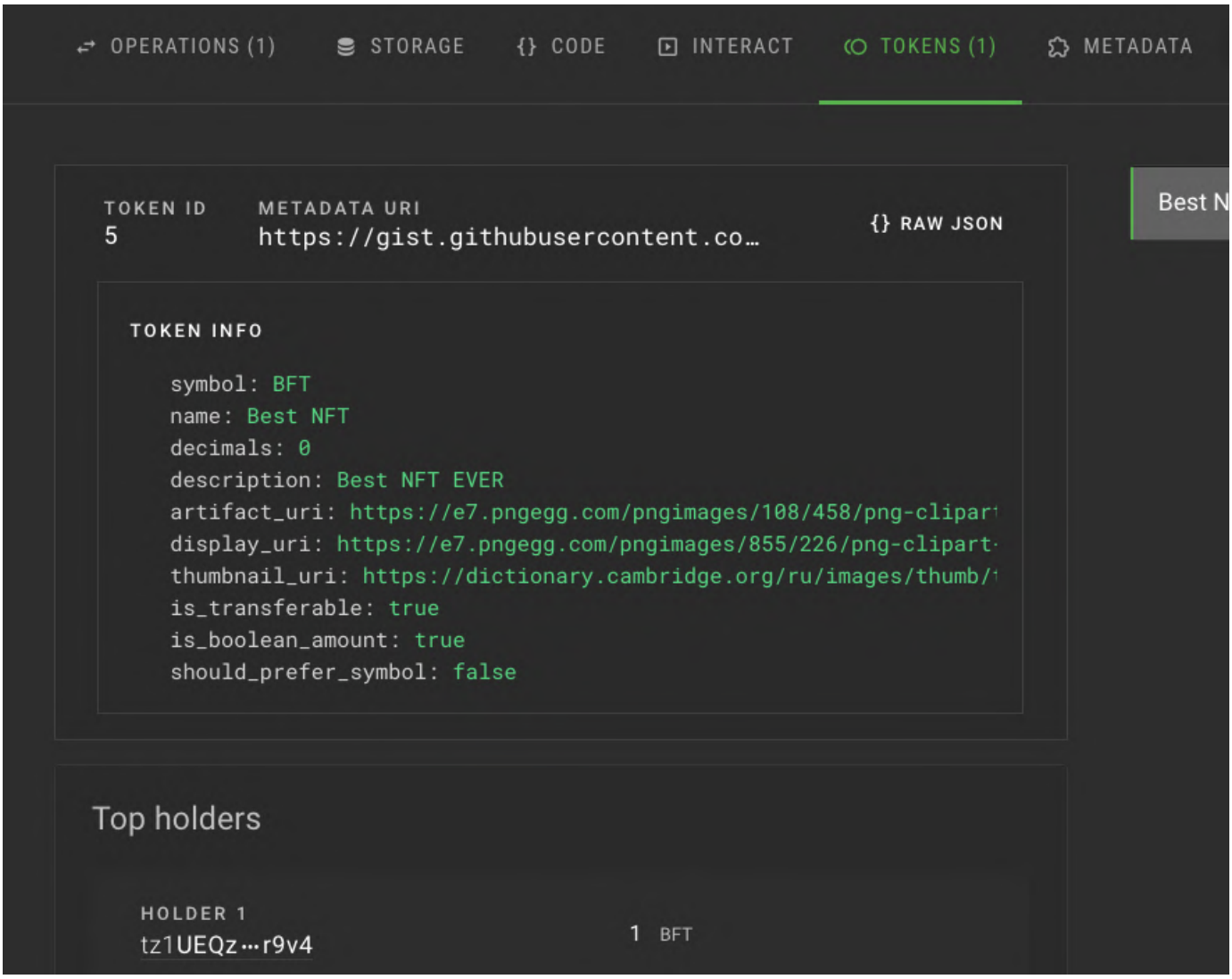
Если развертывание прошло без ошибок, но кошелек не показывает токен, проверьте сеть. Возможно, кошелек подключен к мейннету, а не тестовой сети. В крайнем случае добавьте токен вручную с помощью кнопки **Manage**. Иногда при создании новых токенов с выпуском на адрес пользователя кошелек может их не отображать до первой транзакции.

Вы увидите идентификатор токена и наличие NFT на балансе, но не сам токенизированнвй контент. Кошельки не могут его отобразить: у них нет интерфейсов для чтения поля **artifactUri**, как у NFT-маркетплейсов.



Перейдите на блокчейн-обозреватель BetterCallDev и найдите свой контракт по хешу операции или адресу.

Перейдите на вкладку **Tokens**. Ссылка на токенизированный контент находится в поле **artifact_uri**.



Для выпуска и сжигания NFT нужно добавить соответствующие функции в код контракта. Мы не сделали это, чтобы не переусложнить знакомство с FA 2.

Подбиваем итоги

Метаданные — стандартизированное описание смарт-контракта или токена, которое включает название, тикер, описание, количество знаков после запятой, ссылку на иконку, поддерживаемые интерфейсы и другие данные.

Самый удобный способ добавить метаданные — воспользоваться стандартом TZIP-16. Согласно ему, информация о токене и смарт-контракте хранится в JSON-файлах. Их нужно разместить на публичном сервере вроде Github и добавить ссылки в хранилище. Плюс такого подхода: файлы можно исправить, если вы допустили ошибку.

NFT существуют благодаря метаданным: разработчик записывает в них идентификатор токена и ссылку на токенизированный контент. Связка идентификатора и адреса контракта уникальна, поэтому NFT с одинаковым контентом и метаданными будут отличаться друг от друга.