



Блокчейн- розробка на Tezos

Від констант до публікації контракту



Передмова

Це друкована версія [онлайн-курсу по блокчейн-розробці на Tezos](#) від ForkLog і Tezos Ukraine. Він складається з п'яти уроків і розрахований на новачків.

Ми почнемо з азів: ви вивчите структуру смарт-контрактів і базовий синтаксис мови LIGO, а потім навчитесь писати контракти і взаємодіяти з блокчейном. В кінці курсу ви створите тестові токен та NFT в мережі Tezos.

Важливо: спільнота регулярно оновлює протокол Tezos, а бейкери відключають застарілі тестнети. Якщо тестнет із прикладів не працює, перед публікацією смарт-контракту використовуйте актуальне RPC-посилання [зі списку](#) на сайті Taquito.



Зміст

1. Що таке смарт-контракти та як вони працюють	4
Протокол Tezos	5
Мови програмування Tezos	6
Приклади смарт-контрактів	6
Онлайн-середовище розробки	9
2. Середовище розробки та базові типи даних LIGO	11
Встановлення і налаштування VS Code	12
Встановлення контейнера LIGO	13
Перевірка смарт-контракту в VS Code	16
Основи синтаксису LIGO	18
3. Розгортання смарт-контракту в тестовій мережі	25
Бібліотека Taquito	26
Створення облікового запису в тестовій мережі	33
Публікація контракту за допомогою Taquito	37
Взаємодія з контрактом	42
4. Створення токена стандарту FA 1.2	45
Стандарт токенів FA 1.2	46
Імплементация та функції FA 1.2	47
Публікація контракту токена	54
Додавання токена в гаманець	65
5. Метадані і випуск NFT	70
Метадані контрактів	71
Додавання метаданих в контракт токена	73
Стандарт FA 2 для NFT	76
Випуск NFT на Tezos	85



Під час роботи над курсом нас консультували і підтримували:



Tezos Ukraine

Некомерційна організація, яка популяризує Tezos в Україні та Східній Європі.



Attic Lab

Розробники оглядача Teztracker і стейкінгового сервісу Everstake. Консультують Нацбанк України по цифрових валютах.



Baking Bad

Розробник однойменного сервісу аудиту виплат за делегування tez, біржі атомарних свопів Atomex, оглядачів TzKT і Better Call Dev.

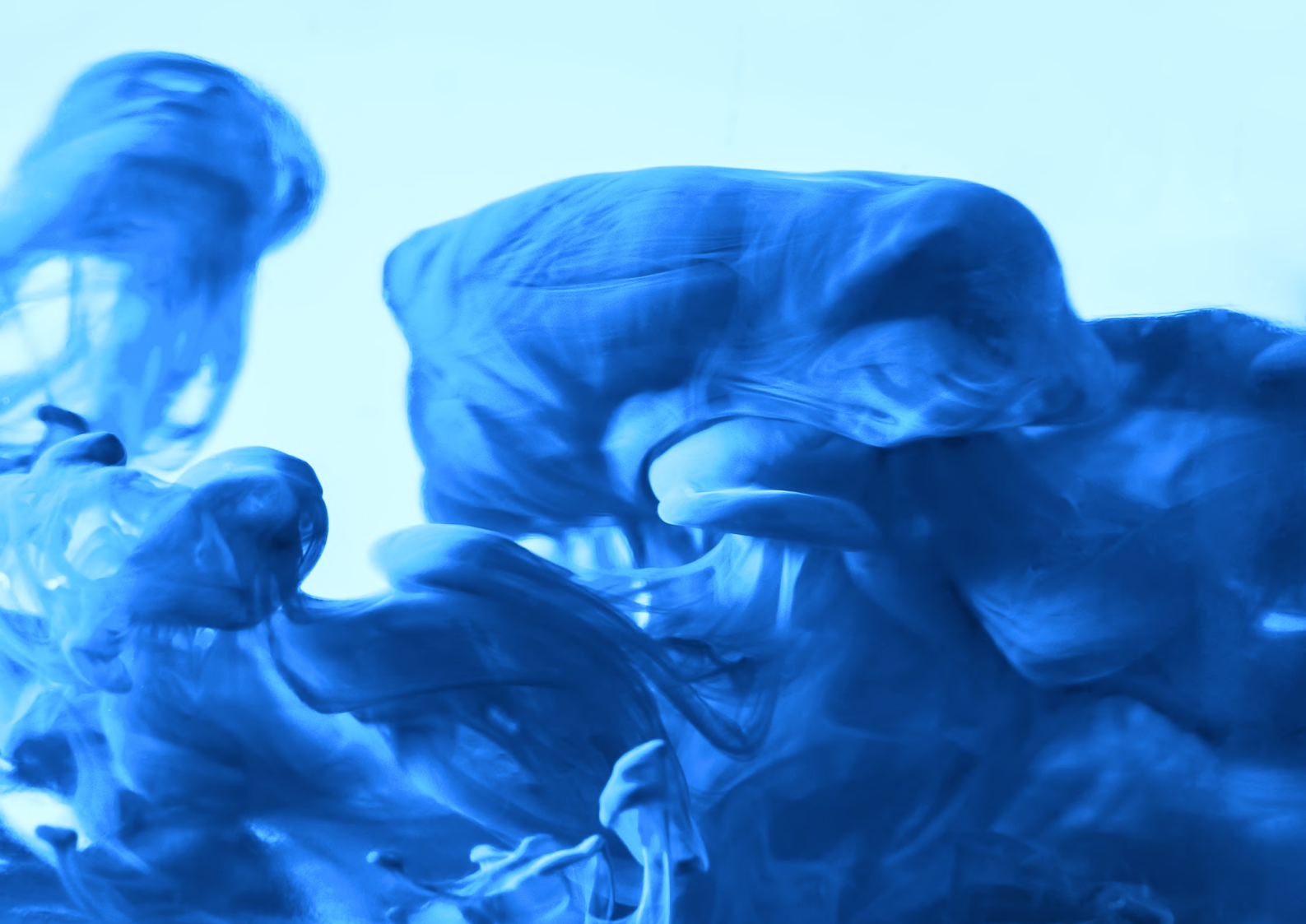


Madfish Solutions

Розробники веб-гаманця Temple, DEX Quipuswap і транспайлера Sol2Ligo.

Над онлайн-курсом працювали:

Автор	Павло Скоропляс
Продюсер	Світлана Лукіна
Стилі	Дмитро Бойко
Ілюстрації	Кшиштоф Шпак
Верстка	Зара Аракелян
Розробка	Олександр Пупко
Керівник	Влад Ліхута



1

Що таке смарт-контракти та як вони працюють

Розповідаємо про смарт-контракти Tezos: якою мовою програмування їх пишуть і як їх виконує віртуальна машина.

В уроці майже немає інформації про синтаксис: на цьому етапі важливіше зрозуміти, як працює віртуальна машина блокчейна.



Що таке смарт-контракти

Смарт-контракт (*Smart contract*) — набір функцій і даних, який зберігається в блокчейне. Смарт-контракти виконують різні завдання — від продажу токенів до управління децентралізованими організаціями. Наприклад:

- вестинг-контракти відправляють користувачам токени в зазначений час. Ранні інвестори Tezos отримують кошти з контракту [KT1KCYaULopm8i2CBbGF5EHXeXLUr4R6r2dA](#);
- оракули опитують джерела даних і повертають усереднені ціни активів або іншу інформацію. Так працює оракул [Harbringer](#), який отримує ціни криптовалюти з торгових площадок;
- алгоритмічні стейблкоїни випускають токени і змінюють вартість їх випуску в залежності від курсу нативного токена. Смарт-контракт [Kolibri](#) коригує ціну випуску kUSD в залежності від ціни tez.

Смарт-контракти виконує віртуальна машина (Virtual Machine або VM). Вона використовує обчислювальну потужність блокчейна: смарт-контракти виконують всі вузли мережі, але тільки найшвидший записує результат в блок.

Часті виклики смарт-контрактів можуть паралізувати блокчейн. Щоб цього уникнути, розробники протоколів обмежують максимальний розмір смарт-контрактів за обсягом коду і розміру комісій.

Як працюють віртуальні машини блокчейнів

Віртуальні машини «розуміють» низькорівневі мови програмування — байт-код. Вони інтерпретують такий код швидше, ніж команди на мовах типу JavaScript або Python. Це можна показати на прикладі однієї фрази, записаної різними словами:

- **низькорівнева мова** — $2 + 2 = 4$;
- **високорівнева мова №1** — два плюс два дорівнює чотири;
- **високорівнева мова №2** — якщо Скласти два і два, Отримаємо чотири;
- **високорівнева мова №3** — в результаті підсумовування двійки і двійки виходить чотири.

При читанні ми автоматично перекладаємо останні рядки в найзручніший для розуміння варіант — $2 + 2 = 4$. У програмуванні так само: компілятор перекладає високорівневу мову в байт-код.

Віртуальна машина Tezos працює з байт-кодом Michelson. Це низькорівнева мова програмування з прямим доступом до стека — структури даних з швидким доступом до інформації. Код на Michelson виглядає так:

```
parameter unit ;
storage unit ;
code { CAR ;
      PUSH int 3 ;
      PUSH int 3 ;
      IFCMPEQ { DROP } { DROP } ;
      UNIT ;
```



```
NIL operation ;  
PAIR }  
  
// Функція if. Оператори PUSH двічі додають число 3 в стек. Оператор IFCMPEQ  
порівнює два перших елемента в стеку, а потім виконує команди {DROP} і виводить  
результат операції.
```

Досвідчені блокчейн-розробники найчастіше пишуть смарт-контракти на Michelson. Новачкам краще використовувати високорівневі мови програмування з набором бібліотек і зрозумілим для людей синтаксисом.

Мови програмування смарт-контрактів на Tezos: SmartPy, LIGO і Lorentz

Учасники екосистеми Tezos розробили кілька високорівневих мов програмування. Найпопулярніші з них:

- **SmartPy** — об'єктно-орієнтована мова на основі Python. Підтримує онлайн-компілятор [SmartPy.io](https://smartpy.io), в якому можна розробляти, тестувати і публікувати смарт-контракти.
- **LIGO** — імперативна мова програмування з простою системою типів і онлайн-компілятором ide.ligolang.org. Існує кілька діалектів с синтаксисом поширених мов: PasaLIGO (Pascal), CamelIGO (OCaml), ReasonLIGO (ReasonML) і jsLIGO (JavaScript).
- **Lorentz** — вбудована предметно-орієнтована мова на базі Haskell. За допомогою Lorentz розробники можуть безпосередньо працювати зі стеком Michelson.

У цьому курсі ми будемо використовувати мову з найпростішим синтаксисом — PasaLIGO.

Простий смарт-контракт з однією точкою входу

Основа смарт-контракту на LIGO — точка входу (*entry point*). Це головна функція смарт-контракту (*main function*), яка приймає вхідну транзакцію і викликає інші функції.

```
function ім'я_функції (const параметр_вхідної_транзакції : тип; const сховище :  
тип) : тип_результата is результат
```

Вхідні параметри *main function*:

- **параметр вхідної транзакції**. Функція використовує його при виконанні коду;
- **сховище (storage)**. Інформація, яку потрібно зберігати в блокчейні до наступного виконання смарт-контракту, наприклад кількість токенів або записи про користувачів. Розробник задає значення *storage* при розгортанні смарт-контракту. Надалі вміст сховища може змінювати тільки смарт-контракт.



Выходные параметры main function:

- список виконаних операцій, наприклад деталі транзакцій, які відправив смарт-контракт;
- результати виконання функцій, які прописав розробник.

Приклад смарт-контракту з функцією-інкрементом, яка приймає число і збільшує його на 1:

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

Код функції:

- *function main (const number : int; const storage : int)* — оголошуємо main function з вхідними параметрами number і storage. Коли користувач відправляє контрактом число, головна функція сприймає його як number і виконує код;
- *: list (operation) * int* — визначаємо тип результату, який повертає функція. В цьому випадку — пара зі списку виконаних операцій і число-результат;
- *is ((nil : list (operation)), number + 1)* — визначаємо результат виконання функції:
 - *nil : list (operation)* — повертає порожній список;
 - *,* — служить роздільником для значень типу tuple;
 - *number + 1* — однорядкова функція, збільшує число на 1.

Смарт-контракт можна модифікувати: реалізувати декремент, зведення в квадрат або іншу математичну операцію. У нього одна точка входу, яка активує єдиний ланцюжок функцій. Такий смарт-контракт не може виконати кілька завдань, наприклад прийняти депозит, повернути баланс і відправити транзакцію.

Смарт-контракт з декількома точками входу

Віртуальна машина починає виконання смарт-контракту з *main function*. Вона може виконати тільки ті функції, які викликає головна точка входу.

Розробники збільшують кількість операцій, створюючи псевдо-точки входу всередині головної функції. Для цього потрібно:

1. Оголосити псевдо-точки входу і тип параметра, з яким вони будуть працювати.
2. Описати функції, які буде викликати кожна псевдо-точка.
3. Використовувати оператор *case* в головній функції. Він вкаже віртуальній машині, яку функцію викликати в разі звернення до псевдо-точки входу.

Приклад смарт-контракту з псевдо-точками — калькулятор, який приймає назву математичної операції і два числа, а повертає результат обчислень. Смарт-контракт не записує значення в *storage*, тому що нам для виконання математичних операцій не потрібно поміщати дані в блокчейн.



```
// оголошення типу numbers, який містить пару з двох чисел (tuple)
type numbers is (int * int)

// оголошення типу action, які містять пари чисел
type action is
| Addition of numbers
| Subtraction of numbers
| Multiplication of numbers
| Division of numbers

// оголошення типів даних в сховищі смарт-контракта
type storage is int

// оголошення математичних функцій

// (const a : int ; const b : int) – параметри функції
// : int – тип результату функції
// is a + b – результат виконання функції

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

// оголошення головної функції
// призначаємо першому параметру тип action, а параметру store – тип storage
// функція поєднує дані типів list(operation) та int – пару зі списку та числа
// після is йде результат виконання функції:
// 1) пустий список nil : list(operation).
// 2) const result : int = – запис результату виконання функції в константу
result.
// 3) case parameter of – результат виконання об'єкта типу action,
// чия назва співпадає з параметром вхідної транзакції.
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;
```

```
// вивід результату виконання головної функції: пустий список операцій та
значення result
} with ((nil : list(operation)), result)
```

Якщо відправити смарт-контракту запит з параметром *Multiplication(3, 9)*, він поверне число 27. Віртуальна машина виконає контракт так:

1. Співставить параметр *Multiplication* з варіантами псевдо-точок під оператором *case*
2. Перейде до функції *multiply*, яку розробник описав на початку контракту.
3. Підставить замість констант *a* і *b* параметри запиту — 3 і 9.
4. Виконає функцію *multiply* і запише її в *store*.
5. Завершить виконання головної функції — поверне порожній список і нове значення *store*.

Для перевірки смарт-контракту скопіюйте код калькулятора в редактор на ide.ligolang.org. Виберіть значення Dry Run зі списку на панелі **Configure**. В поле **Parameters** введіть назву математичної операції і два параметра, а потім натисніть кнопку **Run**.

The screenshot shows the Ligol IDE interface. On the left, there's a sidebar with 'Contract Examples' including 'Increment (PascalIGO)', 'Increment (CameLIGO)', 'Increment (ReasonLIGO)', 'ID (CameLIGO)', 'ID (PascalIGO)', 'ID (ReasonLIGO)', 'Hashlock Contract (CameLIGO)', 'Hashlock Contract (PascalIGO)', and 'Hashlock Contract (ReasonLIGO)'. The main editor displays the code for 'Increment (PascalIGO)' in PascalLIGO. The code defines functions for addition, subtraction, multiplication, and division, and a main function that takes an action and a store as input. The right panel shows the 'Configure' section with 'Dry Run' selected, a 'Run' button, and fields for 'Access function' (set to 'main'), 'Parameters' (set to 'Subtraction(1, 7)'), and 'Storage' (set to '0').

Змініть значення в полях Parameters і Storage і подивіться, як змінюється результат



Підводимо підсумки

Смарт-контракт — це код в блокчейні. Віртуальна машина виконує його, коли отримує транзакцію з потрібними параметрами.

Розробники пишуть смарт-контракти на високорівневих мовах програмування з синтаксисом Python, Pascal, JS або Haskell. Досвідчені розробники часто пишуть байт-код на Michelson.

Смарт-контракти на мові PascalLIGO складаються зі змінних і функцій. Віртуальна машина виконує контракт починаючи з точки входу — головної функції `main`. У неї можна вставити псевдо-точки входу — додаткові функції.

Смарт-контракт завжди повертає результат виконання: список операцій і значення сховища `storage`.

Середовище розробки та базові типи даних LIGO

На минулому уроці ми запустили простий смарт-контракт в онлайн-компіляторі LIGO. У ньому не вийде створити великі проекти: там не можна зберегти код, додати сторонню бібліотеку або викликати інший смарт-контракт.

Програмісти пишуть код в середовищах розробки — текстових редакторах з підтримкою плагінів і бібліотек. На цьому уроці ми встановимо середу розробки VS Code і розповімо про базові типи даних LIGO.

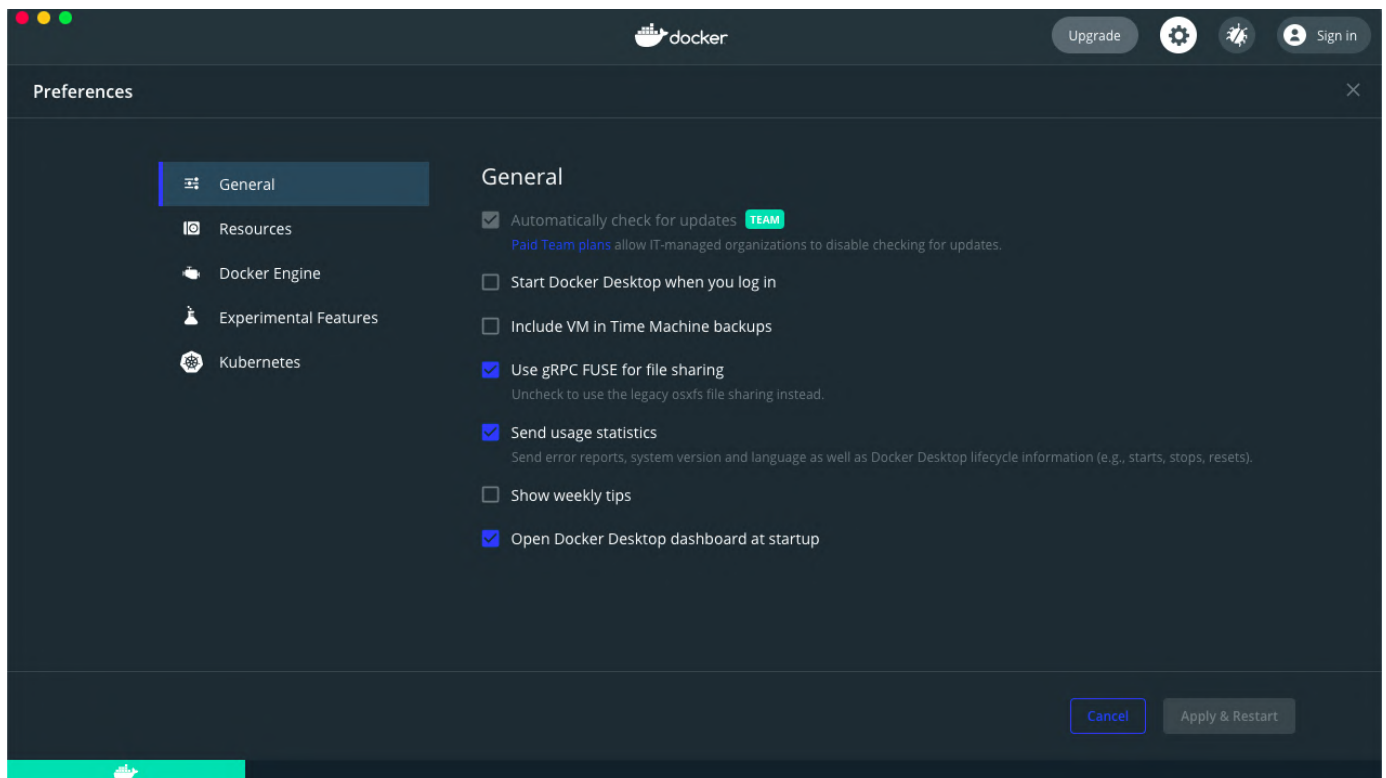


Установка і налаштування середовища розробки

Ми будемо використовувати редактор Visual Studio Code (VS Code) з плагіном `pascaligo-vscode`

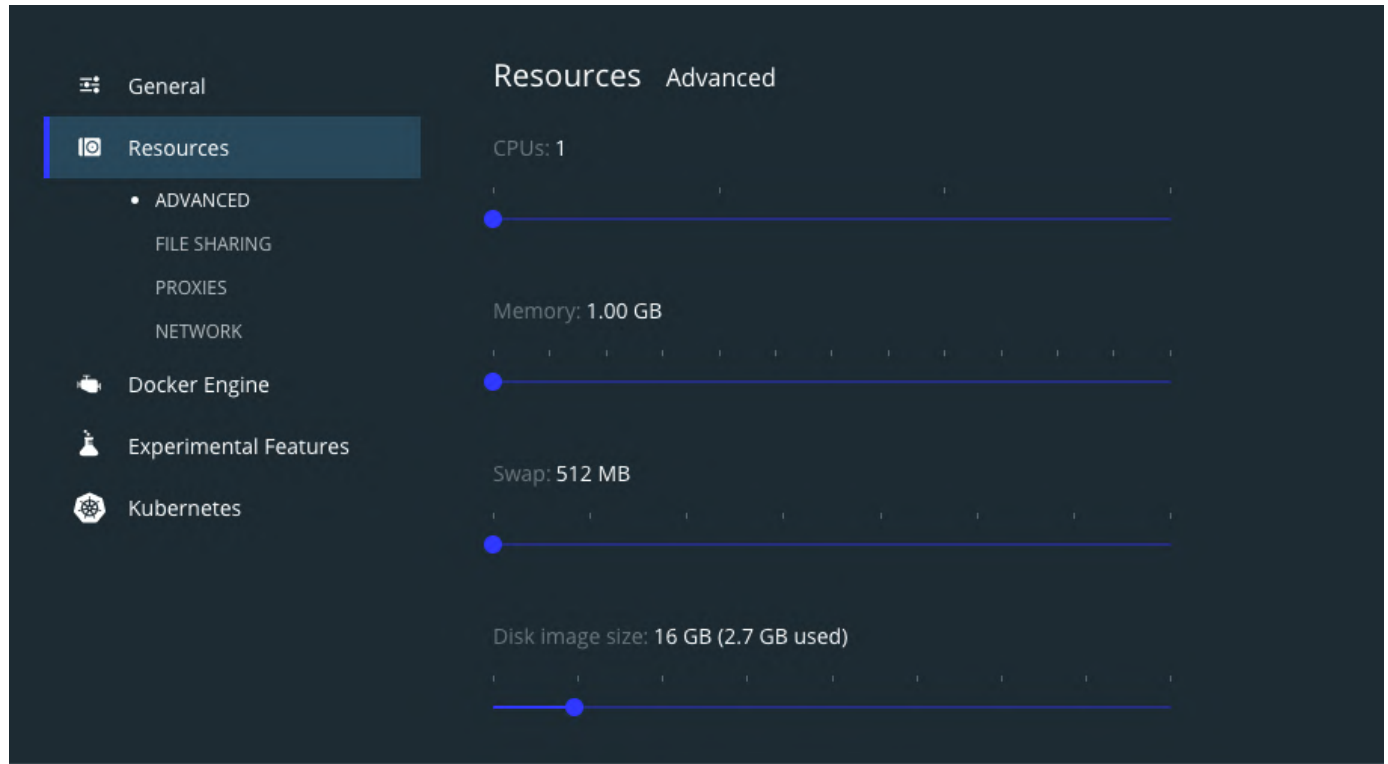
Порядок встановлення середовища розробки:

1. Завантажте редактор з офіційного сайту [Visual Studio Code](#).
2. Встановіть програму та завантажте плагін [pascaligo-vscode](#).
3. Встановіть [Docker](#). Він потрібен для запуску бібліотеки LIGO у віртуальній машині. Так комп'ютер зможе компілювати і виконувати код на PascalLIGO.
4. Запустіть Docker і дочекайтеся, поки в лівому нижньому кутку додатка з'явиться зелена смужка.



Під час установки можуть виникнути такі проблеми:

- **при запуску Docker видає помилку «This computer doesn't have VT-X/AMD-v enabled».** Завантажте BIOS. Увімкніть віртуалізацію VT-X або AMD-V;
- **після запуску Docker комп'ютер повільно працює.** Відкрийте настройки програми, перейдіть у вкладку Resources і зменшіть ресурси комп'ютера, виділені для Docker. Натисніть кнопку **Apply and restart** в нижньому правому куті, щоб застосувати зміни.



Встановіть контейнер LIGO. Для цього відкрийте термінал і виконайте команду:

- для MacOS і 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 завантажить контейнер і видасть вбудовану довідку 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
```

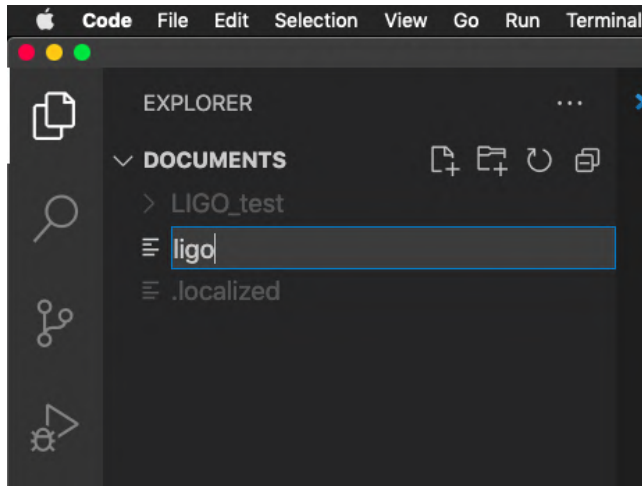



Перевірте, чи все працює. Виконайте в терміналі команду

```
ligo --version
```

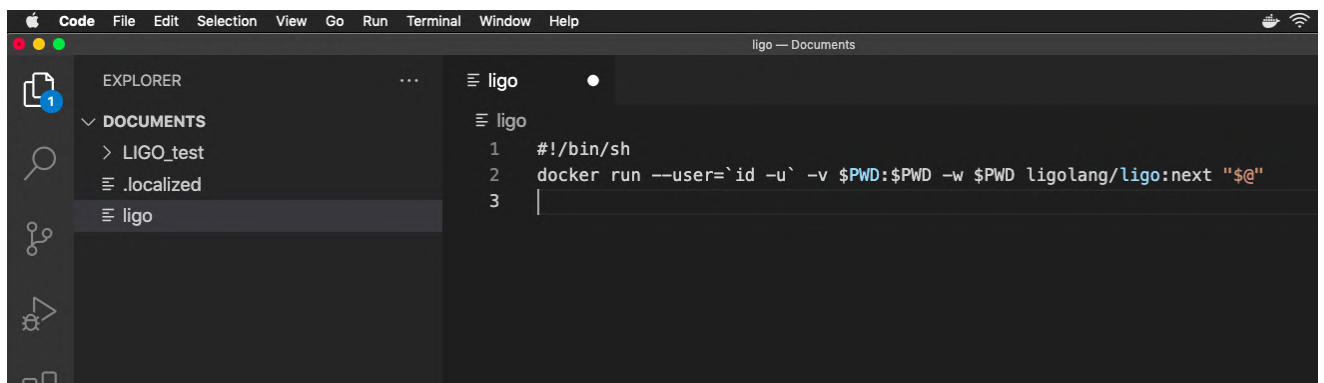
Якщо термінал видає помилку «*ligo: command not found*», значить Docker передчасно закриває контейнер LIGO. Рішення проблеми:

1. Відкрийте VS Code і створіть файл з назвою ligo.



2. Вставте в нього наступний код:

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



3. Збережіть документ і відкрийте термінал. За допомогою команди `cd` перейдіть в папку із файлом ligo:

```
cd ~/Documents
```

4. Якщо у вас Linux або Mac OS, включіть режим суперкористувача:

```
sudo -s
```

5. У терміналі виконайте команди:

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

Вони прив'яжуть запуск контейнера LIGO до команди ligo.



```
Documents — bash — 80x29
Skoroplyass-MacBook-Pro:documents skoroplyas$ sudo -s
Password:
bash-3.2# chmod +x ligo
bash-3.2# ./ligo
NAME
    ligo

SYNOPSIS
    ligo COMMAND ...

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

6. Зробіть файл виконуваним на рівні системи. В Linux і Mac OS відкрийте провідник, і перенесіть файл `ligo` в папку `/usr/local/bin` на системному диску. У Windows відкрийте термінал і виконайте команду:

```
$env:path += ";c:\[шлях до папки з файлом ligo]"
```

7. Перевірте роботу LIGO: знову введіть в терміналі команду `ligo --version`.

```
skoroplyas — bash — 80x24
Last login: Fri Jun 25 00:55:37 on ttys001
Skoroplyass-MacBook-Pro:~ skoroplyas$ sudo -s
Password:
bash-3.2# ligo --version
bash: ligo: command not found
bash-3.2# ligo --version
Rolling release
Commit SHA: c4c8c28a1ae1b21cddd4fb7913bb7480e8c18ee6
Commit Date: 2021-06-24T09:17:47+00:00
bash-3.2#
```

На чорному тлі — помилка при виконанні команди `ligo --version` до перенесення файлу `ligo` в папку `bin`.

На синьому тлі — успішне виконання команди після перенесення файлу.

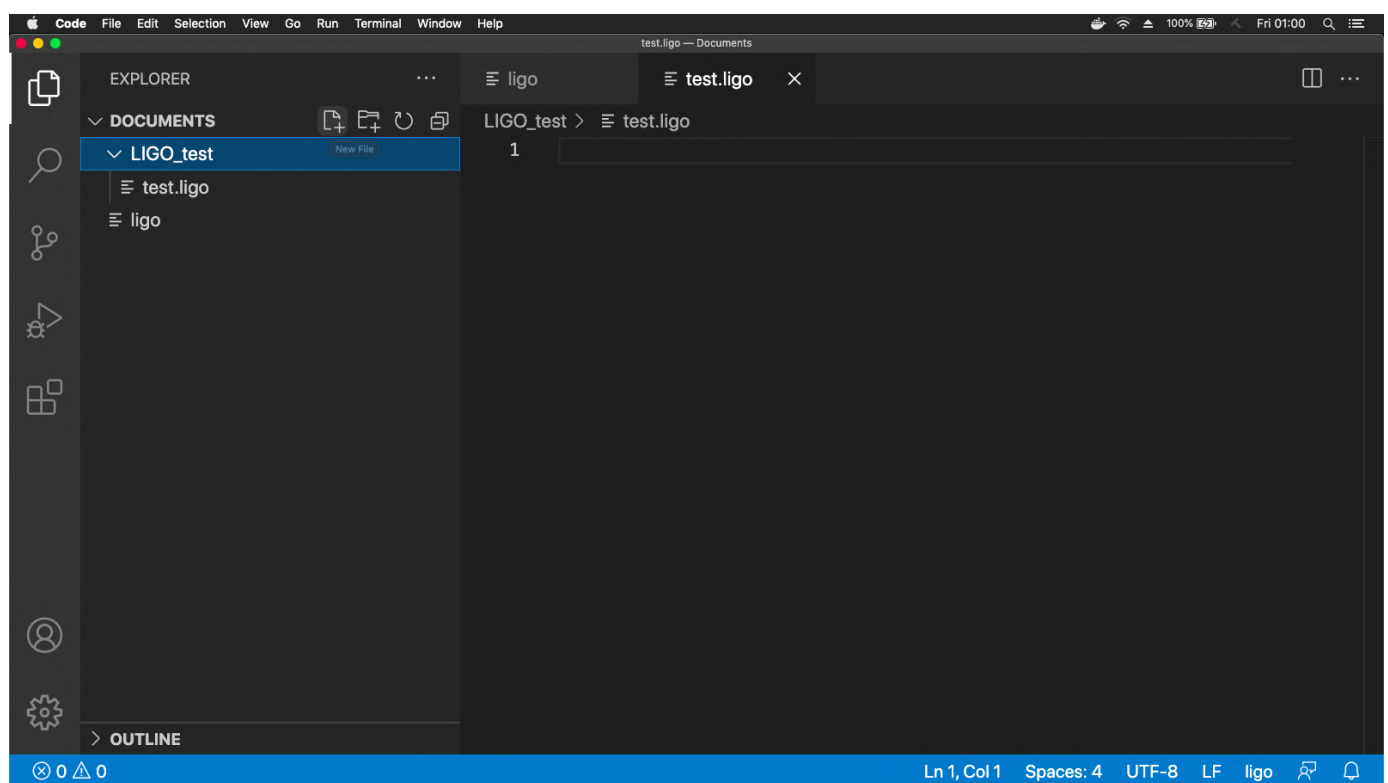


Компіляція пробного контракту

Відкрийте VS Code і створіть новий проект:

1. Натисніть на кнопку **Open....**
2. Виберіть папку, в якій хочете зберегти тестовий проект LIGO.
3. Створіть в ній папку LIGO_test.
4. Відкрийте цю папку.

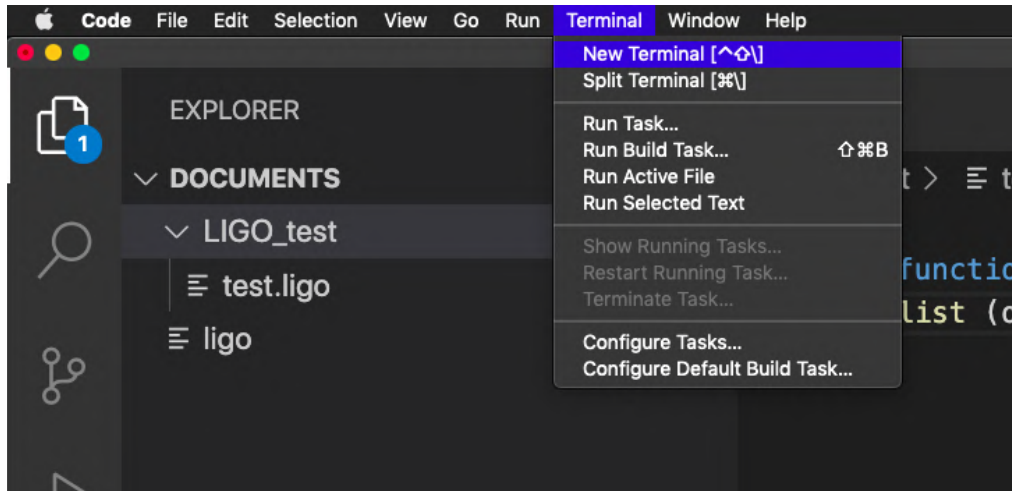
Зліва відображаються папки і файли поточного проекту. Створіть в цьому списку порожній файл з назвою test.ligo. За розширенням .ligo VS Code зрозуміє, що потрібно задіяти плагін для підсвічування синтаксису і використовувати команди PasmaLIGO.



Вставте в файл test.ligo функцію з першого уроку.

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

Спробуйте скомпілювати цей контракт. Відкрийте термінал VS Code.

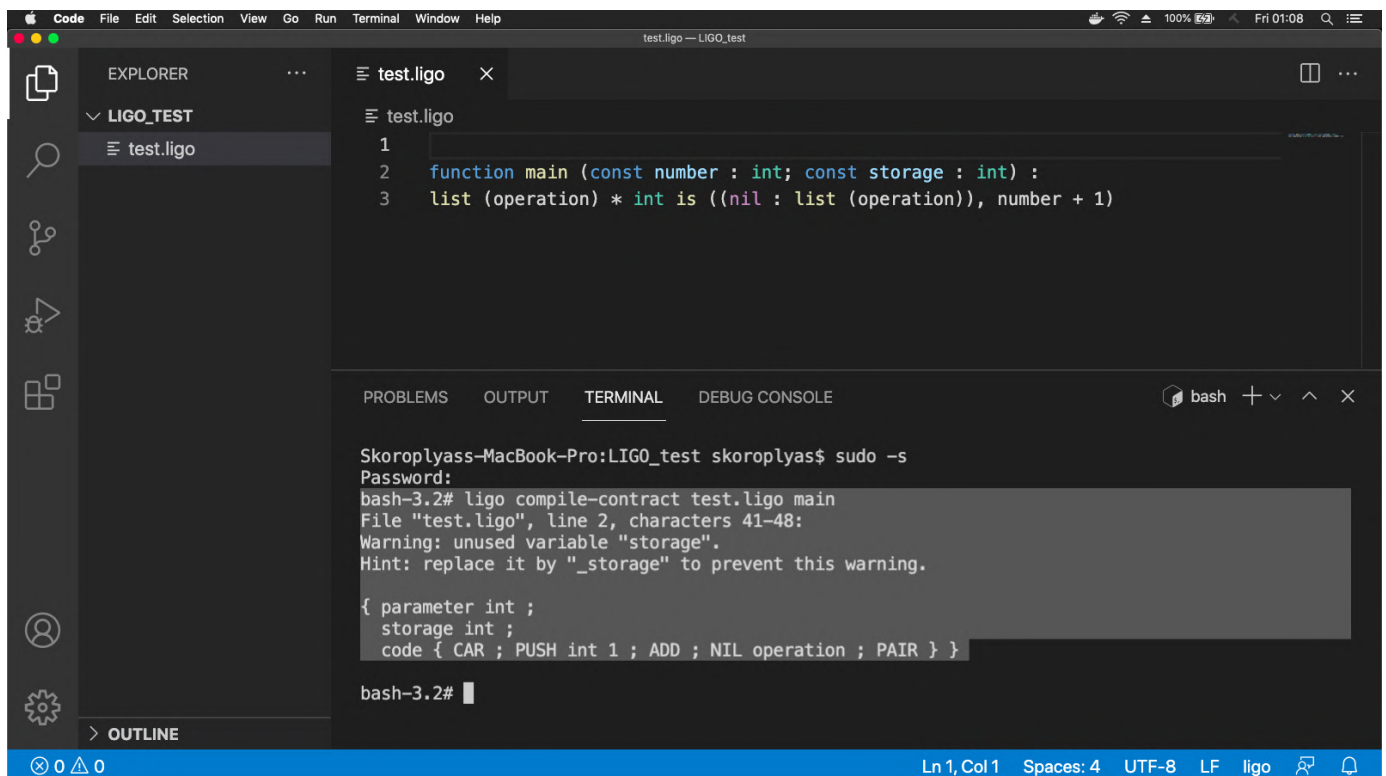


Введіть в ньому таку команду та натисніть Enter:

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

Якщо у вас Linux або Mac OS, термінал може видати помилку «root/ligo». У такому випадку спочатку виконайте команду `sudo -s`, щоб дати терміналу право виконати скрипт ligo в папці bin.

VS Code перекладе код PascaLIGO з test.ligo в код Michelson і перевірить, чи можна виконати функцію `main`.



Тепер перевірте роботу смарт-контракту за допомогою команди `dry-run` (репетиції). Вона виконає контракт із заданими параметрами всередині віртуальної машини Docker.



Для «репетиції» тестового контракту введіть в терміналі команду:

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

- *main* — назва точки входу;
- 2 — перший входить параметр;
- 0 — значення сховища.

Ви можете використовувати інші числа для перевірки цього контракту.

The screenshot shows the VS Code editor with the file `test.ligo` open. The code in the editor is:

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

The terminal output shows the command `ligo dry-run test.ligo main 2 0` being executed. The output includes a warning about an unused variable `storage` and the result of the function execution:

```
Warning: unused variable "storage".
Hint: replace it by "_storage" to prevent this warning.

{ parameter int ;
  storage int ;
  code { CAR ; PUSH int 1 ; ADD ; NIL operation ; PAIR } }

bash-3.2# ligo dry-run test.ligo main 2 0
File "test.ligo", line 2, characters 41-48:
Warning: unused variable "storage".
Hint: replace it by "_storage" to prevent this warning.

( LIST_EMPTY() , 3 )
bash-3.2#
```

При виконанні `dry-run` компілятор видає результат виконання функції `main` з заданими параметрами.

Константи, змінні та базові типи даних LIGO

Для зберігання інформації LIGO використовує:

- **const (константи)**. Смарт-контракт не може змінити значення константи під час виконання функцій. Найчастіше в них записують параметри, які надходять на вхід головної функції;
- **var (змінні)**. Смарт-контракт може змінити значення змінної в будь-який момент. Зазвичай змінні використовують в `loop`-функціях, які контракт викликає кілька разів.

Константам і змінним потрібно присвоїти тип даних, щоб компілятор розумів, як з ними працювати. LIGO використовує такі базові типи даних:

- *int* — необмежене за розміром ціле число, може бути негативним;
- *nat* — необмежене за розміром натуральне число, не може бути негативним;



- *string* — рядок, містить друковані символи: латинські букви, цифри, розділові знаки і деякі спецсимволи на кшталт | чи ~;
- *bool* — логічна змінна *true* або *false*.

Приклад оголошення констант і змінних:

```
// const [назва] : [тип даних] = [значення]
// var [назва] : [тип даних] := [значення]

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

Іноді константі або змінної не можна привласнити значення до виконання смарт-контракту. Приклад: контракт отримує вхідний параметр, а потім використовує його для розрахунків. У такому випадку потрібно оголосити порожню константу або змінну і привласнити їй значення пізніше.

У прикладі нижче ми оголосили константу **number** типу *int*. Під час виконання коду смарт-контракт присвоїть їй значення, яке відправить користувач.

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

Оголошувати константи можна в будь-якому місці смарт-контракту, а змінні — тільки всередині функцій. Наприклад, ми можемо винести число 1 з головної функції в прикладі вище. Для цього потрібно оголосити константу **add** на початку контракту, присвоїти їй значення 1 і замінити одиницю на **add**.

```
const add : int = 1

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

Але якщо зробити **add** змінною, при спробі виконати код компілятор видасть помилку «ILL FORMED CONTRACT» — неправильно сформований контракт.



```

1 //просто заменили const на var и добавили «:=» для присвоения значения
2 var add : int := 3
3
4 function main (const number : int; const storage : int) :
5   list (operation) * int is ((nil : list (operation)), number + add)

```

```

bash-3.2# ligo dry-run test.ligo main 2 0
File "test.ligo", line 3, characters 41-48:
Warning: unused variable "storage".
Hint: replace it by "_storage" to prevent this warning.

( LIST_EMPTY(), 5 )
bash-3.2# ligo dry-run test.ligo main 2 0
File "test.ligo", line 1, characters 0-3:
Ill-formed contract.
At this point, one the following is expected:
* another declaration;
* the end of the file.

```

Просунуті типи і псевдоніми

Крім базових типів даних LIGO підтримує кілька просунутих типів:

- структури даних — tuple, record і map для передачі декількох значень в одному параметрі;
- спеціальні типи — адреси, баланси tez, варіанти, timestamp.

Розробники створюють такі типи за допомогою псевдонімів (alias). Псевдоніми також спрощують читання коду і дозволяють сортувати параметри по способам застосування.

Приклад оголошення псевдонімів:

```

// оголошуємо псевдонім типу int
type numbers is int

// оголошуємо константу типу numbers і присвоюємо їй значення
const a : numbers = -5

// оголошуємо константу admin типу user_id
type user_id is nat
const admin : user_id = 1n

```



Часто використовувані структури даних

Головна функція приймає один параметр, тому базових типів на кшталт `int` і `nat` недостатньо для роботи складних смарт-контрактів. Це обмеження можна обійти за допомогою структур даних, які повертають декілька значень в одному параметрі.

tuple або «кортеж» берігає два і більше значень заданого типу. У прикладі нижче ми задаємо тип `cube`, в який можна записати довжину сторін куба.

```
// оголошуємо псевдонім cube, щоб зберігати в ньому довжину трьох граней куба
type storage is int

type cube is int * int * int

// оголошуємо в функції входить параметр – константу side типу cube
function main (const side : cube ; const _store : storage) :
  (list(operation) * int) is block {

// оголошуємо константу result і присвоюємо їй значення: множення трьох значень,
які зберігаються в кортежі side
  const result : int = side.0 * side.1 * side.2
  } with ((nil : list(operation)), result)
```

record зберігає кілька констант або змінних різних типів. Наприклад, в константу типу `user` можна записати аргументи: `id` типу `int`, `is_admin` типу `bool` і `name` типу `string`:

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

// оголошуємо константу alice і присвоюємо їй значення типу user
const alice : user =
  record [
    id      = 1n;
    is_admin = True;
    name    = "Alice"
  ]
```



map пов'язує дані в набір пар «ключ-значення». Також в LIGO є тип даних **big_map**, який оптимізує завантаження великої кількості «ключів-значень» і економить газ, але не підтримує ітерацію.

```
// оголошення псевдоніму dims типу tuple, потім – cube_dimensions типу map
type dims is int * int * int
type cube_dimension is map (string, dims)

// оголошення map cubes, яка містить два значення: назва куба і значення довжини
його граней
const cubes : cube_dimension =
  map [
    "big cube" -> (123243, 1251, 823);
    "small cube" -> (3, 3, 7)
  ]

// записуємо в константу big типу dims. Для цього отримуємо значення з map за
зверненням до ключу "big cube"
const big : option(dims) = cubes["big cube"]
```

unit — тип даних без значення. Він потрібен, коли за правилами синтаксису функція повинна прийняти параметр, але цей параметр не важливий для роботи коду. Ми використовуємо unit для оголошення типів Labeouf, Nike і Yoda в прикладі нижче.

variant дозволяє використовувати змінні декількох типів в залежності від ситуації і логіки смарт-контракту. У прикладі оголошуємо variant для створення псевдо-точок входу.

```
// оголошуємо псевдонім speach і записуємо кілька варіантів. Для кожного варіанта
вказуємо тип параметра, який він буде приймати. У нашому випадку головна функція
нічого не робить з прийнятим параметром, тому для варіантів вказуємо порожній тип
unit
type speach is
  Labeouf of unit
  |Nike of unit
  |Yoda of unit

// ми хочемо, щоб контракт повертав рядок (string), тому в типі return вказуємо,
що він повертає список операцій і рядок
type return is (list(operation) * string)

// оголошуємо, що функція main повинна прийняти від користувача параметр і
записати його в константу word типу speach
```



```
function main (const word : speach; const _store : string) : return is
  ((nil : list (operation)),
  // "case word of" порівнює прийнятий параметр з варіантами типу speach. Якщо
  // варіант співпадає, контракт поверне зазначений рядок
  case word of
    Labeouf (_n) -> "DO IT!"
  | Nike (_n) -> "Just do it"
  | Yoda (_n) -> "Do it you can"
  end)
```

Спеціальні типи даних

Крім просунутих структур даних, LIGO використовує кілька «блокчейнових» типів даних і функцію *failwith*.

tez — позначає кількість токенів XTZ (tez). При оголошенні, присвоєнні або оперуванні типом tez потрібно додавати до числа суфікс «tz» або «tez».

У LIGO також використовується суфікс «mutez» для позначення однієї мільйонної частини tez. Для зручності читання можна розділяти великі числа на порядки за допомогою нижнього підкреслення. Наприклад, «1_000_000tez» віртуальна машина Tezos сприйме як 1 млн tez.

```
// 1 000 000 mutez = 1 tez
// присвоюємо t значення 1,5003 tez за допомогою суфікса mutez
const t : tez = 1_500_300mutez
```

address — тип даних для зберігання адрес Tezos.

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

failwith — функція обробки помилок і виключень. Наприклад, ми можемо застосувати failwith, щоб калькулятор не ділив числа на 7.

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

Більше прикладів операцій з різними типами даних — [в документації LIGO](#).



Підводимо підсумки

В онлайн-середовищі розробки не можна зберігати код, довантажувати бібліотеки і викликати інші контракти, тому програмісти використовують Visual Studio Code і контейнер LIGO в Docker.

Базові типи LIGO — *int*, *nat*, *string* і *bool*. Спеціальні структури для зберігання даних:

- *tuple* — кілька значень в одній константі або змінній;
- *record* — кілька констант з присвоєними значеннями в одній константі або змінній;
- *map* — кілька констант просунутих типів з прив'язкою до константи-ключу;
- *tez* — баланс XTZ;
- *address* — адреса в блокчейні Tezos;
- *unit* — тип даних без значення.

Розгортання смарт-контракту в тестовій мережі за допомогою Taquito

На минулому уроці ми вивчили базові та просунуті типи даних мови програмування LIGO. Цих знань вистачить, щоб написати код простого смарт-контракту. Але щоб інші користувачі могли взаємодіяти з контрактом, його потрібно розмістити в блокчейні.

Для розгортання смарт-контрактів в блокчейні розробники використовують спеціальні інструменти. У цьому уроці ми встановимо бібліотеку Taquito і з її допомогою опублікуємо контракт в тестовій мережі Tezos.



Встановлення і пробний запуск Taquito

Taquito — це TypeScript-бібліотека, яка спрощує взаємодію з блокчейном Tezos і смарт-контрактами. Наприклад, так виглядає розгортання контракту вручну через консоль клієнта Tezos:

```
tezos-client originate contract <назва контракту>
for <user> transferring <кількість tez> from <from_user>
running <шлях до файлу з кодом>
--init '<стан сховища storage>'
--burn-cap <максимальна комісія>
```

Скрипт Taquito для розгортання контракту виглядає простіше:

```
try {
  const op = await tezos.contract.originate({
    // код смарт-контракту
    code: `{
      `,
    // значення сховища
    init: ``,
  });
```

Також за допомогою Taquito можна відправляти токени, перевіряти баланси адрес і розробити простий веб-інтерфейс для смарт-контракту.

Встановлення Taquito

Для роботи Taquito потрібна платформа nodeJS. Скачайте її з [офіційного сайту](#) і встановіть на свій комп'ютер.

Потім встановіть пакетний менеджер yarn. Він завантажить Taquito і автоматично пропише його в системі.

Для цього відкрийте консоль і виконайте команду:

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



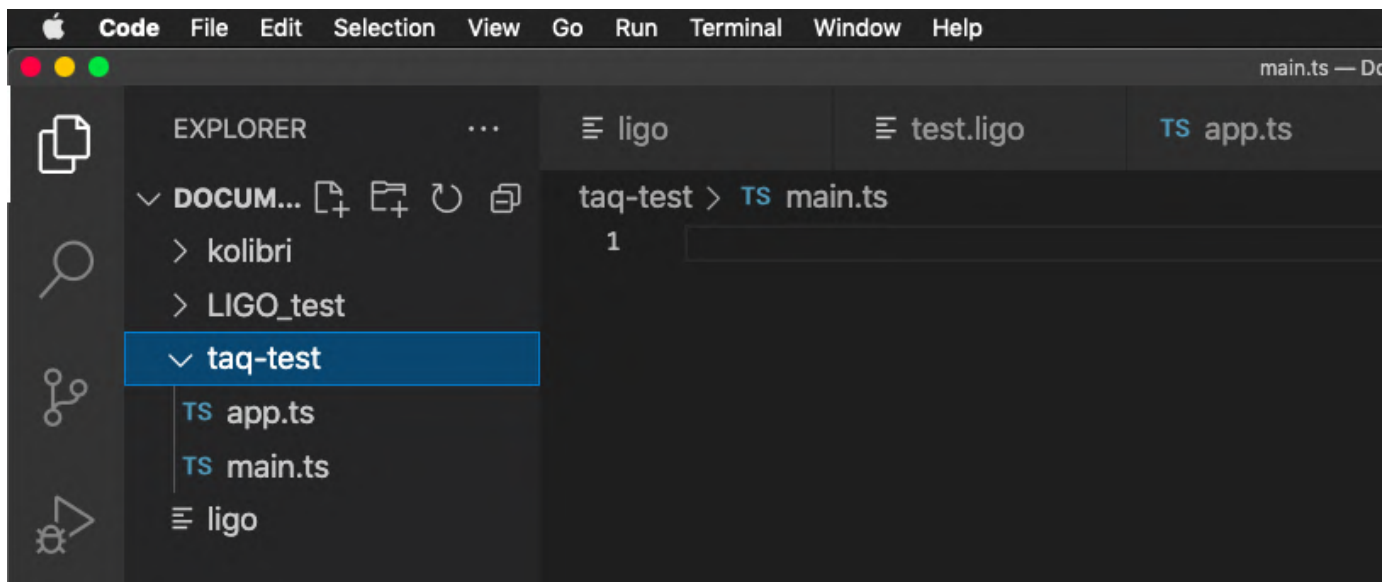
```
Skoroplyass-MacBook-Pro:~ skoroplyas$ sudo npm install --global yarn

> yarn@1.22.10 preinstall /usr/local/lib/node_modules/yarn
> :; (node ./preinstall.js > /dev/null 2>&1 || true)

/usr/local/bin/yarn -> /usr/local/lib/node_modules/yarn/bin/yarn.js
/usr/local/bin/yarnpkg -> /usr/local/lib/node_modules/yarn/bin/yarn.js
+ yarn@1.22.10
added 1 package in 1.304s
Skoroplyass-MacBook-Pro:~ skoroplyas$
```

Засоби для роботи Taquito готові, залишилося протестувати її роботу. Відкрийте Visual Studio Code, створіть папку `taq-test`, а в ній — два файли з назвами `app.ts` і `main.ts`.

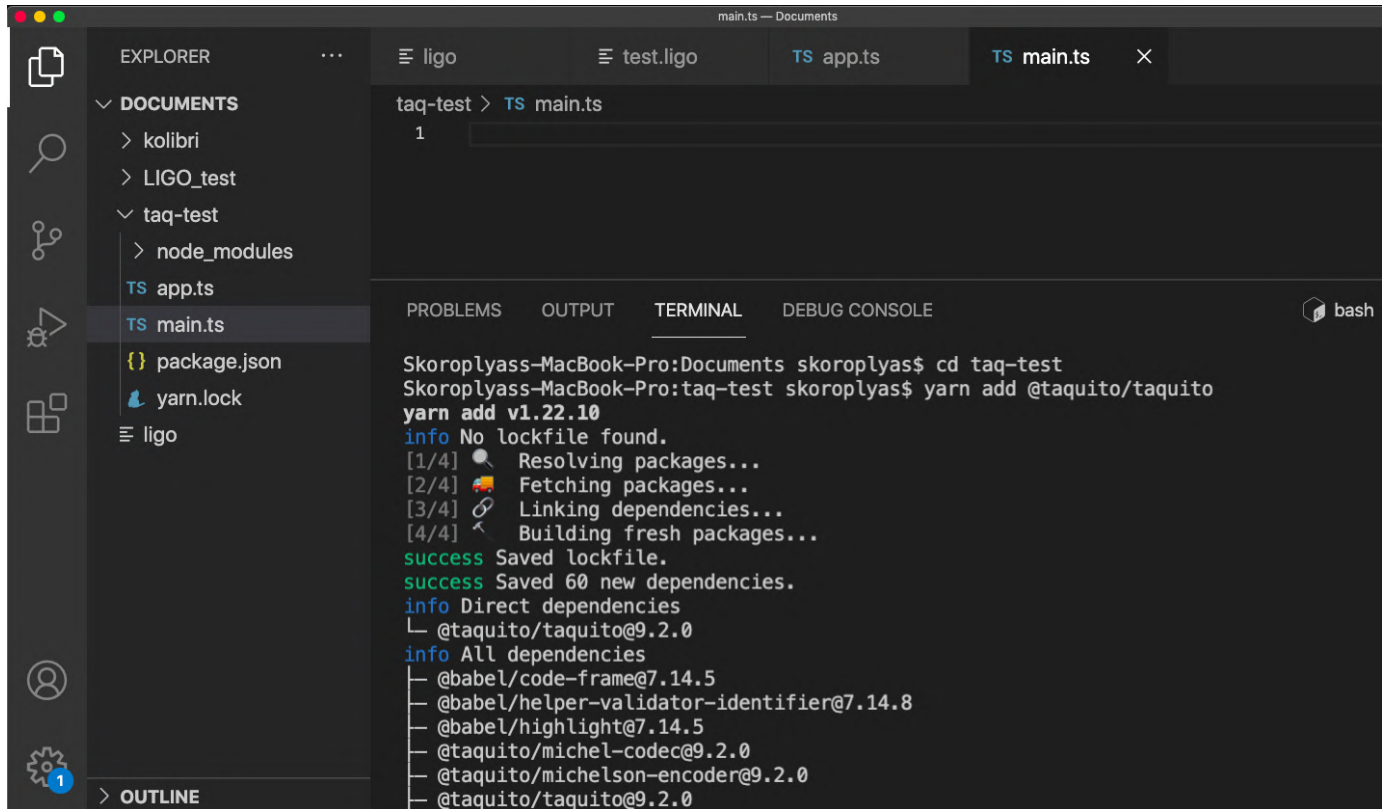
У `app.ts` ми будемо записувати функції Taquito і основний код, а в `main.ts` — параметри і виклики функцій. Такий підхід дозволяє спростити структуру коду і не викликати функції до їх оголошення.



Відкрийте термінал в VS Code і за допомогою команди `cd` перейдіть в папку `taq-test`.

Потім додайте в папку бібліотеку Taquito:

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



Якщо ви правильно встановили NodeJS і yarn, термінал видасть довгий список встановлених файлів.

Відкрийте файл app.ts і додайте в нього код:

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

  public async main() { }

}
```

У файл main.ts додайте:

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

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

Не забудьте зберегти файли за допомогою `ctrl+s` або `⌘+s`, інакше середовище виконання буде вважати їх порожніми.



Розширення `.ts` вказує на те, що в файлах міститься код TypeScript. Для його роботи потрібно встановити бібліотеку TypeScript. Для цього в терміналі VS Code виконайте:

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

```
taq-test > TS main.ts
1  import { App } from './app';
2
3  new App().main();

Skoroplyass-MacBook-Pro:Documents skoroplyas$ cd taq-test
Skoroplyass-MacBook-Pro:taq-test skoroplyas$ sudo npm install typescript --save-dev
Password:
npm WARN taq-test No description
npm WARN taq-test No repository field.
npm WARN taq-test No license field.

+ typescript@4.3.5
added 1 package from 1 contributor and audited 78 packages in 2.297s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

Skoroplyass-MacBook-Pro:taq-test skoroplyas$
```

При успішній установці термінал видасть 3-4 попередження про відсутність ліцензії та опису проекту. Якщо в списку більше пунктів — ви забули додати `sudo` на початку команди або не перебуваєте в папці `taq-test`.

Останній крок — виконайте скрипт `main.ts` в віртуальному вузлі. Для цього введіть у терміналі команду:

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

```
Skoroplyass-MacBook-Pro:taq-test skoroplyas$ npx ts-node main.ts
npx: installed 13 in 2.6s
Skoroplyass-MacBook-Pro:taq-test skoroplyas$
```



Термінал відзвітує про запуск прх одним рядком. Це означає, що прх вдало імпортував в додаток Taquito. Так як головна функція `app.ts` порожня, прх відразу її завершив. Інший результат при виконанні цього кроку означає, що ви допустили помилку під час установки TypeScript або Taquito.

При роботі над смарт-контрактами вам потрібно буде кожен раз встановлювати Taquito і TypeScript в папку проекту. На відміну від JavaScript, їх не можна встановити глобально.

Отримуємо дані з блокчейна Tezos через Taquito

За допомогою Taquito можна зчитувати баланси адрес в мережі Tezos, відправляти токени і публікувати смарт-контракти. Для цього не потрібно піднімати власний вузол Tezos: Taquito під'єднується до публічних вузлів через RPC (Remote Procedure Call або виклик віддалених процедур).

Давайте під'єднаємося до мейннету і запитаємо баланс адреси одного з бейкерів — валідаторів мережі Tezos. Для цього в файлі `app.ts` включимо бібліотеку `TezosToolkit` і підготуємо константу для RPC-посилання публічного вузла Tezos:

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

  // оголошуємо приватний модифікатор tezos типу TezosToolkit
  private tezos: TezosToolkit;

  // оголошуємо конструктор rpcUrl, який буде передавати адресу публічного вузла
  // Tezos в TezosToolkit
  constructor(rpcUrl: string) {
    this.tezos = new TezosToolkit(rpcUrl);
  }

  public async main() { }

}
```

У файлі `main.ts` пропишемо RPC-посилання на публічний вузол Tezos Mainnet і передамо її класу `App`:

```
import { App } from './app';
// оголошуємо константу з адресою вузла
const RPC_URL = "https://mainnet.smartpy.io"
// запускаємо App, який передає головній функції адреса вузла
new App(RPC_URL).main();
```



При виконанні команди `prx ts-node main.ts` Taquito отримає посилання на вузол, але нічого з ним не зробить, тому що в `app.ts` немає методів для взаємодії з мережею. Додамо в `app.ts` метод для отримання балансу адреси:

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

  private tezos: TezosToolkit;

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

  // оголошуємо метод getBalance з вхідним параметром address
  public getBalance(address: string) : void {

    // Taquito відправляє вузлу запит балансу зазначеної адреси. Якщо вузол виконав
    // запит, скрипт виводить отримане значення в консоль. Якщо сталася помилка – видає
    // «Address not found»
    this.tezos.rpc
      .getBalance(address)
      .then(balance => console.log(balance))
      .catch(e => console.log('Address not found'));
  }

  public async main() { }
}
```

Додамо адресу бейкера і викличемо метод `getBalance` в скрипті `main.ts`:

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

const RPC_URL = "https://mainnet.smartpy.io"
// оголошуємо константу з адресою бейкера Everstake
const ADDRESS = "tz1aRoarhSpRYvFdyvgWLL6TGyRoGF51wDjM"
// запускаємо App, передаємо йому посилання на вузол, викликаємо метод getBalance
і передаємо йому адресу
new App(RPC_URL).getBalance(ADDRESS);
```




Запустимо скрипт за допомогою команди `npx ts-node main.ts`.

```

1  import { App } from './app';
2
3  const RPC_URL = "https://mainnet-tezos.giganode.io/"
4
5  const ADDRESS = "tz1aRoarhSpRYvFdyvgWLL6TGyRoGF51wDjM"
6
7  new App(RPC_URL).getBalance(ADDRESS);
8

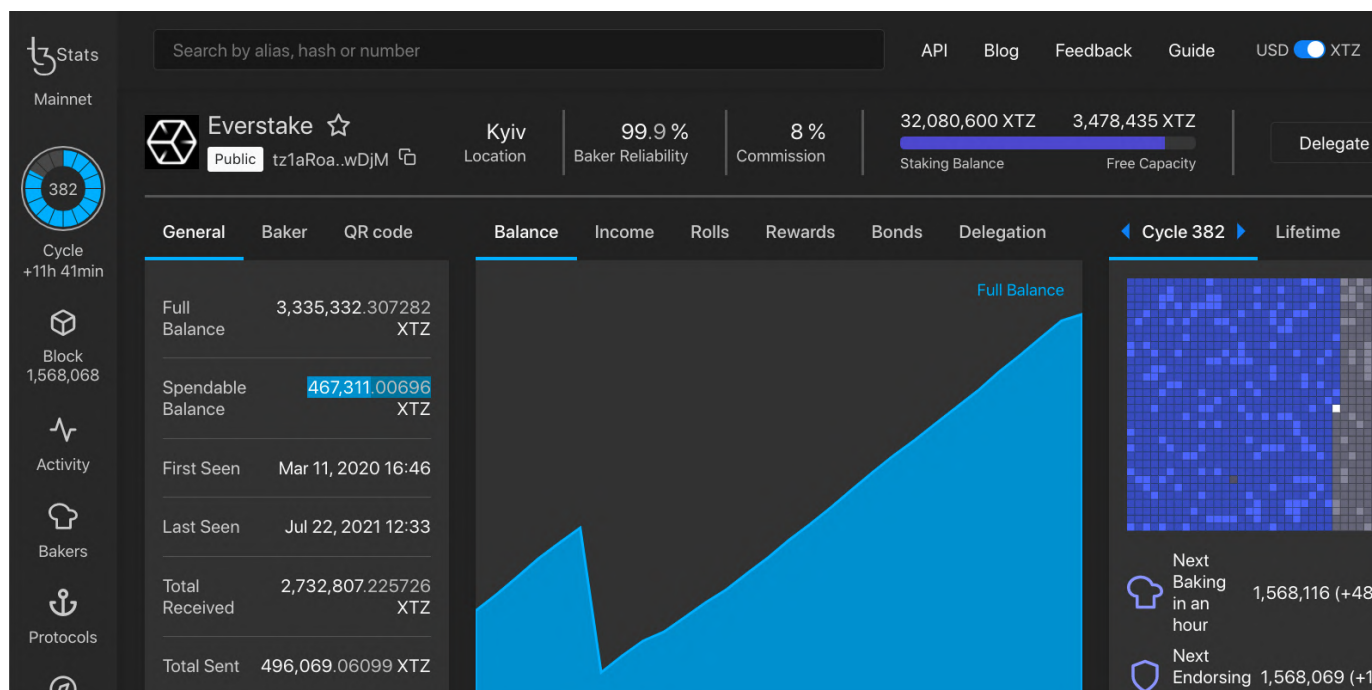
```

```

Skoroplyass-MacBook-Pro:Documents skoroplyas$ cd taq-test
Skoroplyass-MacBook-Pro:taq-test skoroplyas$ npx ts-node main.ts
npx: installed 13 in 2.256s
BigNumber { s: 1, e: 11, c: [ 467311006960 ] }
Skoroplyass-MacBook-Pro:taq-test skoroplyas$

```

Taquito поверне кількість вільних tez на адресі Everstake.



Отриманий баланс збігається з [даними оглядача tzStats](#).



Створюємо аккаунт в тестовій мережі Tezos за допомогою Taquito

Транзакції потрібно підписувати закритим ключем. Для цього Taquito використовує модуль signer, який працює з даними блокчейн-акаунтів.

Щоб не витратити справжні tez, ми підключимося до Tezos Testnet. Спочатку потрібно отримати адресу і тестові tez на крані [tzalpha](#). Перейдіть по посиланню, підтвердіть, що ви не робот, і натисніть кнопку **Get Testnet tez**.

Welcome to the Tezos Faucet

please drink responsibly

Get Testnet ₮

Faucet ₮ can be used on all testnets and zeronet

✓ I'm not a robot



Кран tzalpha створить нову адресу і видасть дані облікового запису: мнемонічну фразу, закритий ключ і публічний адресу.

Welcome to the Tezos Faucet

please drink responsibly

```
{
  "mnemonic": [
    "able",
    "public",
    "usual",
    "hello",
    "october",
    "owner",
    "essence",
    "old",
    "author",
    "original",
    "various",
    "gossip",
    "core",
    "high",
    "hire"
  ],
  "secret": "2bed8dc244ee43a1e737096c4723263c269049d8",
  "amount": "117632913",
  "pkh": "tz1imn4fjJFwmNaiEWnAGdRrRHxzxRdKafZ",
  "password": "ZnnZLS0v6O",
  "email": "hoqfgsoy.qyisbhtk@tezos.example.org"
}
```

Copy



Дані облікового запису потрібно зберегти в окремому файлі. Створіть в папці `taq-test` файл `acc.json` і вставте в нього код з `tzalpha`.

```

1  {
2    "mnemonic": [
3      "able",
4      "public",
5      "usual",
6      "hello",
7      "october",
8      "owner",
9      "essence",
10     "old",
11     "author",
12     "original",
13     "various",
14     "gossip",
15     "core",
16     "high",
17     "hire"
18   ],
19   "secret": "2bed8dc244ee43a1e737096c4723263c269049d8",
20   "amount": "117632913",
21   "pkh": "tz1imn4fjJFwmNaiEWnAGdRrRHxzxRbDkafZ",
22   "password": "ZnnZLS0v60",
23   "email": "hoqfgsoy.qyisbhtk@tezos.example.org"
24 }

```

Тепер можна додати в проект модуль `signer`. В VS Code відкрийте термінал, перейдіть в папку `taq-test` і виконайте команду:

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

Як і при установці Taquito, термінал видасть попередження про відсутніх ліцензіях і список встановлених файлів.

```

Skoroplyass-MacBook-Pro:Documents skoroplyas$ cd taq-test
Skoroplyass-MacBook-Pro:taq-test skoroplyas$ yarn add @taquito/signer
yarn add v1.22.10
warning package.json: No license field
warning package-lock.json found. Your project contains lock files generated by tools other than Yarn. It is advised not to mix package managers in order to avoid resolution inconsistencies caused by unsynchronized lock files. To clear this warning, remove package-lock.json.
warning No license field
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...

```



Створимо файл tx.ts і додамо в нього модуль signer:

```
import { TezosToolkit } from '@taquito/taquito';
// імпортуємо inMemorySigner. Він збереже приватний ключ в оперативній пам'яті і
// буде підписувати ним транзакції
import { InMemorySigner } from '@taquito/signer';
// оголошуємо константу ас, яка направить скрипт до файлу ас.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

    // оголошуємо параметри за допомогою методу fromFundraiser: пошту, пароль і
    // мнемонічну фразу, з якої можна отримати приватний ключ
    this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
    acc.password, acc.mnemonic.join(' ')))
  }
  // отримуємо публічний і приватний ключі і активуємо 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() { }
}
```

У файлі main.ts викликаємо метод `activateAccount()`:

```
import { App } from './app';
// імпортуємо Tx.ts
import { Tx } from './tx';
// міняємо RPC-посилання з мейннета на тестову мережу. Не лякайтеся smartpy в
```



посиланні – це просто адреса сервера

```
const RPC_URL = "https://florencenet.smartpy.io/"
const ADDRESS = "tz1aRoarhSpRYvFdyvgWLL6TGyRoGF51wDjM"
// викликаємо функцію Tx, передаємо їй посилання на тестову мережу і просимо
активувати обліковий запис
new Tx(RPC_URL).activateAccount();
```

Залишилося виконати main.ts в терміналі:

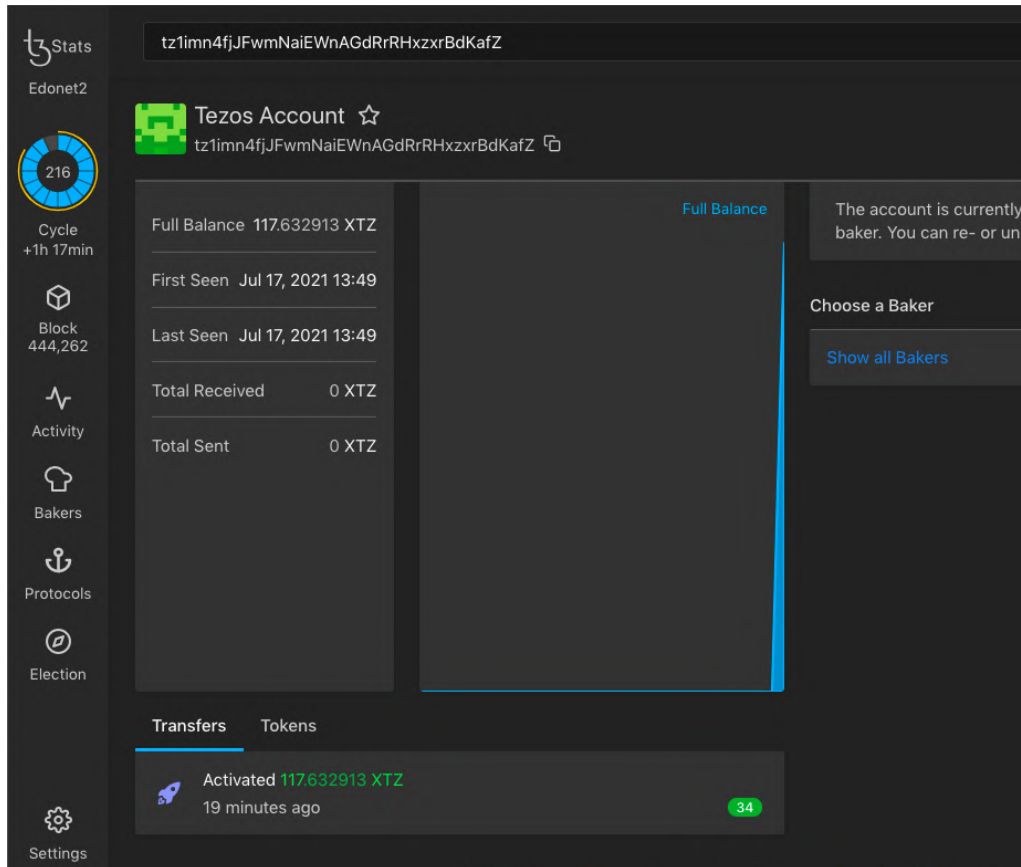
```
npx ts-node main.ts
```

```
taq-test > TS main.ts > ...
1 import { App } from './app';
2 import { Tx } from './tx';
3
4 const RPC_URL = "https://edonet.smartpy.io/"
5
6 const ADDRESS = "tz1aRoarhSpRYvFdyvgWLL6TGyRoGF51wDjM"
7
8 new Tx(RPC_URL).activateAccount();
```

```
Skoroplyass-MacBook-Pro:taq-test skoroplyas$ npx ts-node main.ts
npx: installed 13 in 3.25s
Skoroplyass-MacBook-Pro:taq-test skoroplyas$
```

Якщо тестова мережа активує аккаунт, консоль відзвітує про запуск прх без повідомлень про помилки. Тепер можете перевірити активацію в браузері тестових мереж.

Відкрийте acc.json і скопіюйте публічний ключ з поля «pkh». Перейдіть на florence.tzstats і знайдіть свій аккаунт з публічного ключу. На балансі повинно бути близько сотні тестових tez, а в історії транзакцій – запис про активацію.



Ми налаштували Taquito і активували тестовий аккаунт. Для роботи в мейннеті потрібно буде зробити те ж саме: створити файл .json з даними облікового запису і під'єднати його за допомогою signer.

Публікуємо контракт в тестнеті

На минулому уроці ми запускали смарт-контракт з допомогою контейнера Docker і команди *dry-run*. Віртуальна машина з контрактом відразу закривалася, і при цьому не використовувала сховище storage.

Справжні смарт-контракти токенів і децентралізованих додатків постійно використовують storage для зберігання даних користувачів. Ми докладніше розповімо про це в наступному уроці. А поки опублікуємо контракт в тестовій мережі і подивимось, як storage зберігає інформацію.

Перейдіть в VS Code і створіть файл `deploy.js`. Вставте в нього наступний 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,
  "", // пошта
  "", // пароль
  [
    "", // мнемоніка
  ].join(' '),
  "" // приватний ключ
);

deploy();
```

Впишіть в порожні місця дані облікового запису: пошту, пароль, слова мнемонічної фрази і приватний ключ з файлу acc.json.

```
TS deploy.ts x
TS deploy.ts > deploy
8  const tezos = new TezosToolkit(provider);
9  await importKey(
10   tezos,
11   "hoqfgsoy.qyisbhtk@tezos.example.org", //пошта
12   "ZnnZLS0v60", //пароль
13   [
14     "able", //мнемонічна фраза
15     "public",
16     "usual",
17     "hello",
18     "october",
19     "owner",
20     "essence",
21     "old",
22     "author",
23     "original",
24     "various",
25     "gossip",
26     "core",
27     "high",
28     "hire"
29   ].join(' '),
30   "2bed8dc244ee43a1e737096c4723263c269049d8" //приватний ключ
```

Після даних аккаунта додайте код контракту. Ми будемо використовувати модифікований контракт з першого уроку:



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

Цей смарт-контракт отримує від користувача параметр `num` типу `int` і підсумовує його з числом в сховищі.

Скомпілюйте код контракту на мову Michelson. Щоб не запускати Docker зайвий раз, скористайтесь онлайн середовищем розробки.

Вставте код в редактор, в випадаючому списку виберіть пункт **Compile Contract** і натисніть кнопку **Run**. Компілятор видасть код контракту на мові Michelson.

Поверніться до файлу `deploy.ts` і після даних аккаунта вставте наступний код:

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



Тепер опишіть процес розгортання контракту. Фінальний код буде виглядати 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", // пошта
    "ZnnZLS0v60", // пароль
    [
      "able", // мнемоніка
      "public",
      "usual",
      "hello",
      "october",
      "owner",
      "essence",
      "old",
      "author",
      "original",
      "various",
      "gossip",
      "core",
      "high",
      "hire"
    ].join(' '),
    "2bed8dc244ee43a1e737096c4723263c269049d8" // приватний ключ
  );

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



```
// значення сховища
init: `0`,
});

// початок розгортання
console.log('Awaiting confirmation...');
const contract = await op.contract();
// звіт про розгортання: кількість використаного газу, значення сховища
console.log('Gas Used', op.consumedGas);
console.log('Storage', await contract.storage());
// хеш операції, за яким можна знайти контракт в блокчейн-браузері
console.log('Operation hash:', op.hash);
} catch (ex) {
  console.error(ex);
}
}

deploy();
```

Відкрийте консоль і виконайте команду `npx ts-node deploy.ts`. Термінал видасть звіт про публікації контракту.

```
TS deploy.ts x
TS deploy.ts > ...
42   });
43
44   //початок розгортання
45   console.log('Awaiting confirmation...');
46   const contract = await op.contract();
47   // звіт про розгортання: кількість використаного газу, значення сховища
48   console.log ( 'Gas Used', op.consumedGas);
49   console.log ( 'Storage', await contract.storage ());
50   // хеш операції, за яким можна знайти контракт в блокчейн-браузері
51   console.log ( 'Operation hash:', op.hash);
52 } catch (ex) {
53   console.error (ex);
54 }

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
skoroplyas@MacBook-Air taq-test % npx ts-node deploy.ts
npx: installed 14 in 3.407s
Awaiting confirmation...
Gas Used 1577
Storage BigNumber { s: 1, e: 0, c: [ 0 ] }
Operation hash: opHAY8fxXJLzEiNapn5guRYzH5rrVXz7wkPKx8muYQFcnSfbKnX
skoroplyas@MacBook-Air taq-test %
```



Скопіюйте хеш операції і знайдіть транзакцію розгортання контракту в оглядачі [florence.tzstats](https://florence.tzstats.com/). У ній буде вказана адреса смарт-контракту в тестовій мережі.

Спробуйте викликати опублікований контракт. Для цього створіть скрипт, який викличе головну функцію і передасть їй число.

Створіть файл `call.ts` і вставте в нього 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

    // оголошуємо параметри за допомогою методу fromFundraiser: пошту, пароль і
    // мнемонічну фразу, з якої можна отримати приватний ключ
    this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
    acc.password, acc.mnemonic.join(' ')))
  }

  public add(add: number, contract: string) {
    this.tezos.contract
      .at(contract) // звертаємося до контракту, щоб отримати його точки
    входу
      .then((contract) => {
        console.log(`Adding ${add} to storage...`);
        // звертаємося до головної функції. На відміну від синтаксису
    ligo, головна точка входу називається 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); // чекаємо одне
    підтвердження мережі, щоб швидше отримати результат
      })
  }
}
```



```
.then((hash) => console.log(`Call done`)) // успішний виклик
.catch((error) => console.log(`Error: ${JSON.stringify(error, null,
2)}}`));
}
}
```

Тепер зробіть окремий скрипт для виклику call.ts, передачі адреси контракту і значення аргументу. Для цього створіть файл maincall.ts і скопіюйте в нього код:

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

const RPC_URL = "https://florencenet.smartpy.io/"
const CONTRACT = "" // адреса опублікованого контракту
const ADD = 5 // число, яке отримає головна функція. Можете змінити його на інше
new Call(RPC_URL).add(ADD, CONTRACT);
```

Присвойте константі CONTRACT адресу опублікованого тестового контракту. Відкрийте консоль і виконайте команду:

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

Коли термінал видасть повідомлення про завершення операції, перейдіть за посиланням або знайдіть свій контракт за адресою. Як бачите, значення сховища змінилося

The screenshot shows the 'Tezos Contract' page for the address KT1Qc2QyRgRzfczWH3Sx4aw1JzC7DMNBrihC. The left sidebar displays contract statistics: Full Balance (0 XTZ), First Seen (Jul 23, 2021 10:46), Last Seen (Jul 23, 2021 11:43), Total Received (0 XTZ), and Total Sent (0 XTZ). The main area shows the 'Calls' tab with a blue bar chart. At the bottom, the 'Storage' tab is active, showing a table with one entry: 'storage: int' with a value of '130'.

#	Key & Type	Content
1	storage: int	130



Підбиваємо підсумки

Taquito — це інструмент для роботи з блокчейном Tezos через код на JavaScript і TypeScript. За допомогою Taquito розробники звертаються до точок входу смарт-контрактів і до мережі Tezos, як до звичайних JS-методів. Також Taquito дозволяє зберігати дані декількох акаунтів в різних файлах і виносити часто використовуваний код в окремі модулі.

Бібліотека Taquito потрібна для розробки DApp на Tezos. Розробникам простіше інтегрувати в веб-додаток JavaScript, ніж перекладати запити користувачів в консольні команди клієнта Tezos. У наступних уроках ми також будемо використовувати Taquito для створення взаємозамінного токена і NFT.

Створення токена стандарту FA 1.2

На минулих уроках ми вивчили синтаксис мови програмування LIGO і опублікували найпростіший контракт в тестовій мережі Tezos.

Тепер ми вивчимо механізм роботи цифрових активів, розберемося зі стандартом FA 1.2, і випустимо токен в тестовій мережі.



Види і стандарти токенів

Для початку визначимося з термінами. Токенами називають два типи одиниць обліку: нативні токени і токени, які випускають смарт-контракти. Логіка роботи перших описана в коді протоколу, а других — в коді смарт-контрактів (СК).

Нативні токени також називають криптовалютою. До них відносяться біткоїни, Ether (ETH), Tezos (XTZ, tez), Litecoin та інші.

Токени смарт-контрактів — цифрові активи, створені користувачами блокчейна, наприклад USDT і DOGE на блокчейні Ethereum, kUSD — на Tezos.

В цьому і наступному уроках ми будемо говорити про токени смарт-контрактів, але для простоти називати їх токенами.

Ми з'ясували, що логіка роботи токенів не описана в протоколі. Розробники створюють її в коді смарт-контрактів: додають базові функції на кшталт зберігання балансів і передачі монет.

Так, 100 kUSD на гаманці користувача — це запис в сховищі СК виду «Адреса TZ1 може відправити 100 токенів цього контракту». Іншими словами, токени — це реєстри всередині блокчейна.

Щоб розробники випускали робочі і стійкі до атак смарт-контракти, спільнота приймає стандарти токенів. Вони містять списки функцій, необхідних для взаємодії смарт-контракту з гаманцями і блокчейн-додатками.

Розробники можуть прописувати в смарт-контрактах додаткові функції, наприклад спалювання токенів. Але базові функції стандарту обов'язкові.

Базові функції стандарту FA 1.2

Стандарти Tezos називаються Financial Application (FA). Два найпопулярніших з них:

- **FA 1.2** для взаємозамінних токенів;
- **FA 2** для взаємозамінних токенів і NFT.

На цьому уроці ми розглянемо більш простий стандарт FA 1.2, а на наступному — розберемося з FA 2.

Стандарт FA 1.2 включає функції:

- **передача (transfer)** — проведення транзакцій між користувачами;
- **схвалення (approve)** — дозвіл Алісі відправити токени з адреси Боба. Дозволяє контракту відправляти користувачам запити на транзакції;
- **схвалене кількість (getAllowance)** — перегляд кількості токенів, які Аліса може відправити з адреси контракту;
- **баланс користувача (getBallance)** — перегляд балансу користувача;
- **вільна емісія (getTotalSupply)** — перегляд кількості токенів на адресах всіх користувачів.



Крім того, контракт токена повинен зберігати в `storage` інформацію про повну емісію та баланси користувачів, а в коді — повідомлення про стандартні помилки на кшталт браку коштів.

Розробник може включити в сховище метадані з назвою контракту і токена, а також посилання на його логотип.

Що таке імплементація FA 1.2

Імплементація — це робочий продукт на основі будь-якої ідеї. У випадку FA 1.2 — шаблон смарт-контракту для операцій з токенами.

Розробник може скопіювати імплементацію, змінити її під свої завдання, опублікувати в блокчейні і отримати робочий токен.

Імплементація токена FA 1.2 складається приблизно з сотні рядків. Код можна розділити на вісім частин:

- псевдоніми, типи даних і псевдо-точки входу;
- функція `getAccount`, яка отримує дані користувача;
- функція `getAllowance`, яка отримує кількість tokenів, доступних для транзакції;
- функція `transfer` для переказу tokenів з однієї адреси на інший;
- функція `approve`, яка підтверджує право користувача на переказ tokenів;
- функція `getBalance`, яка повертає баланс користувача;
- функція `getTotalSupply`, яка повертає кількість вільних tokenів;
- функція `main`, яка приймає вхідні параметри і передає їх одній з попередніх функцій

Нижче докладніше розповімо про кожну з цих частин.

Оголошення псевдонімів і типів

На початку контракту потрібно оголосити всі типи і псевдоніми, які він буде використовувати у функціях:

```
// оголошуємо псевдонім trusted типу address. Ми будемо використовувати його для
// позначення адрес, у яких є право відправляти токени
type trusted is address;

// оголошуємо псевдонім amt (amount) типу nat для зберігання балансів
type amt is nat;

(* оголошуємо псевдонім account типу record. У ньому будемо зберігати дані
користувачів, яким можна передавати токени *)
type account is
  record [
```



```

    balance      : amt;
    allowances    : map (trusted, amt);
  ]

```

(* оголошуємо тип сховища смарт-контракту. В ньому зберігається загальна кількість токенів, а також структура даних `big_map`, яка пов'язує публічні адреси і баланси користувачів *)

```

type storage is
  record [
    totalSupply      : amt;
    ledger           : big_map (address, account);
  ]

```

(* оголошуємо псевдонім для методу `return`, який будемо використовувати для повернення операцій. У коротких контрактах можна обійтися без нього, але в контрактах з декількома псевдо-точками входу простіше один раз прописати тип повернення і використовувати його в кожній функції *)

```

type return is list (operation) * storage

```

(* оголошуємо порожній список `noOperations`. Його повертатимуть функції `transfer` і `approve` *)

```

const noOperations : list (operation) = nil;

```

Компілятор `Michelson` автоматично сортує вміст просунутих структур даних в алфавітному порядку. Сортуння може випадково зламати контракт. Наприклад, якщо перший параметр — адреса відправника — починається з `TZ19`, а другий — адреса одержувача — починається з `TZ11`, компілятор поміняє їх місцями. В такому випадку контракт спробує відправити токени з не того аккаунта.

Щоб зберегти потрібний порядок параметрів, розробники записують важливі структури даних в типі `michelson_pair`. Така структура включає два аргументи і їх назви, наприклад, дві адреси: `sender` і `receiver`. Компілятор переносить в код на `Michelson` значення `michelson_pair` без сортування.



```
(* оголошуємо псевдоніми вхідних параметрів для кожної базової функції FA 1.2 *)

// функція transfer отримує на вхід адресу відправника, адресу одержувача і суму
транзакції
type transferParams is michelson_pair(address, "from", michelson_pair(address,
"to", amt, "value"), "")
// approve отримує адресу користувача і кількість токенів, які він може
відправити з балансу смарт-контракту
type approveParams is michelson_pair(trusted, "spender", amt, "value")
// getBallance отримує адресу користувача і проксі-контракту, яким вона
відправить дані про баланс
type balanceParams is michelson_pair(address, "owner", contract(amt), "")
// getAllowance отримує адресу користувача, дані його аккаунта в смарт-контракті
і проксі-контракт
type allowanceParams is michelson_pair(michelson_pair(address, "owner", trusted,
"spender"), "", contract(amt), "")
// totalSupply не використовує michelson_pair, тому що перший вхідний параметр –
пусте значення unit – і так виявиться першим після сортування компілятора
Michelson
type totalSupplyParams is (unit * contract(amt))

(* оголошуємо псевдо-точки входу: даємо назву і присвоюємо їм тип параметрів, які
описали вище *)
type entryAction is
  | Transfer of transferParams
  | Approve of approveParams
  | GetBalance of balanceParams
  | GetAllowance of allowanceParams
  | GetTotalSupply of totalSupplyParams
```

Функція `getAccount` отримує вхідний параметр типу `address` і значення `storage` з смарт-контракту:

```
function getAccount (const addr : address; const s : storage) : account is
  block {
    // присвоюємо змінної acct значення типу account: нульовий баланс і
    порожній запис allowances
    var acct : account :=
      record [
        balance      = 0n;
        allowances = (map [] : map (address, amt));
```



```
];

(* перевіряємо, чи є в сховищі аккаунт користувача. Якщо немає – залишаємо в
acst пусте значення з попереднього блоку. Якщо є – присвоюємо змінній acst
значення зі сховища. Функція повертає значення acst *)
case s.ledger[addr] of
  None -> skip
| Some(instance) -> acst := instance
end;
} with acst
```

Функція *getAllowance* запитує у користувача, скільки токенів він дозволяє перевести на іншу адресу. Вона отримує адресу користувача, адреса контракту (*spender*) і стан сховища, а повертає аргумент **amt** – кількість токенів дозволених для витрачання:

```
function getAllowance (const ownerAccount : account; const spender : address;
const s : storage) : amt is
  (* якщо користувач дозволив відправити кілька токенів, функція присвоює це
кількість змінної amt. Якщо не дозволив – кількість токенів дорівнює нулю *)
  case ownerAccount.allowances[spender] of
    Some (amt) -> amt
  | None -> 0n
  end;
```

Функція *transfer* отримує від користувача адреси відправника і одержувача, кількість токенів для переказу і стан сховища:

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

    (* викликаємо функцію getAccount, щоб привласнити змінній senderAccount дані
облікового запису користувача. Потім ми використовуємо senderAccount, щоб
зчитувати баланс користувача і дозволи *)
    var senderAccount : account := getAccount(from_, s);
```




```
(* перевіряємо, чи достатньо у користувача засобів для переказу. Якщо ні –  
віртуальна машина перериває виконання контракту, якщо достатньо – продовжує  
виконувати контракт *)  
if senderAccount.balance < value then  
    failwith("NotEnoughBalance")  
else skip;  
  
(* перевіряємо, чи може адрес-ініціатор транзакції відправити токени. Якщо  
він запитує переказ з чужої адреси, функція запитує дозвіл у справжнього  
власника. Якщо ініціатор і відправник – одна адреса, віртуальна машина продовжує  
виконувати контракт *)  
if from_ /= Tezos.sender then block {  
    (* викликаємо функцію getAllowance, щоб власник адреси-відправника вказав,  
    скільки tokenів він дозволяє відправити. Надаємо це значення константі  
    spenderAllowance *)  
    const spenderAllowance : amt = getAllowance(senderAccount, Tezos.sender, s);  
  
    (* якщо власник дозволив відправити менше tokenів, ніж зазначено у вхідному  
    параметрі, віртуальна машина припинить виконувати контракт *)  
    if spenderAllowance < value then  
        failwith("NotEnoughAllowance")  
    else skip;  
  
    (* віднімаємо від дозволеної для переказу кількості tokenів суму  
    транзакції *)  
    senderAccount.allowances[Tezos.sender] := abs(spenderAllowance - value);  
} else skip;  
  
(* віднімаємо від балансу адреси-відправника кількість відправлених  
tokenів *)  
senderAccount.balance := abs(senderAccount.balance - value);  
  
(* оновлюємо запис про баланс відправника в storage *)  
s.ledger[from_] := senderAccount;  
  
(* ще раз викликаємо функцію getAccount, щоб отримати або створити запис  
аккаунта для адреси-одержувача *)  
var destAccount : account := getAccount(to_, s);  
  
(* додаємо до балансу одержувача кількість відправлених tokenів *)  
destAccount.balance := destAccount.balance + value;
```



```
(* оновлюємо запис про баланс одержувача в storage *)
s.ledger[to_] := destAccount;

}
// повертаємо порожній список операцій і стан storage після виконання функції
with (noOperations, s)
```

Функція `approve` запитує підтвердження кількості токенів, яку `spender`-адреса може відправити з `sender`-адреси. Приклад використання: блокчейн-додаток (`spender`) запитує у користувача (`sender`) дозвіл на переказ токенів.

Ця функція вразлива до атаки випереджаючого витрачання. Наприклад, `sender` знижує кількість дозволених для переказу токенів з 20 до 10. `Spender` дізнається про це і створює транзакцію з витрачанням 20 токенів. Він платить підвищену комісію, щоб транзакція з тратою потрапила в блок раніше, ніж транзакція зі зміною дозволів.

`Spender` отримує 20 токенів і чекає зміни дозволу. Після цього він створює ще одну транзакцію — на відправку 10 токенів. Функція дозволяє переказ. В результаті `spender` відправляє з `sender`-адреси 30 токенів замість дозволених 10.

Щоб уникнути подібної ситуації, розробники реалізують в `approve` затримку зміни дозволу. Якщо дозволена кількість токенів більше нуля, її можна змінити тільки на нуль, а якщо дорівнює нулю — її можна змінити на натуральне число. В такому випадку `spender`-адреса не може витратити токени двічі.

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

    (* отримуємо дані облікового запису користувача *)
    var senderAccount : account := getAccount(Tezos.sender, s);

    (* отримуємо поточну кількість токенів, яку користувач дозволив відправити *)
    const spenderAllowance : amt = getAllowance(senderAccount, spender, s);

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

    (* вносимо в дані облікового запису нову дозволену кількість токенів для
    витрачання *)
    senderAccount.allowances[spender] := value;
```



```
(* оновлюємо сховище смарт-контракту *)
s.ledger[Tezos.sender] := senderAccount;

} with (noOperations, s)
```

Функції *getBalance*, *getAllowance* і *getTotalSupply* відносяться до оглядових (view). Вони повертають запитовані значення не користувачеві, а спеціальному проміжковому контракту (proxy-contract). Останній дозволяє додаткам отримувати дані від користувача контрактів і відображати їх в інтерфейсі.

```
function getBalance (const owner : address; const contr : contract(amt); var s :
storage) : return is
  block {
    // присвоюємо константі ownerAccount дані облікового запису
    const ownerAccount : account = getAccount(owner, s);
  }
  // повертаємо проміжковому контракту баланс аккаунта
  with (list [transaction(ownerAccount.balance, 0tz, contr)], s)
```

Функція *getAllowance* повертає кількість дозволених для витрачання токенів запитаного аккаунта:

```
function getAllowance (const owner : address; const spender : address; const
contr : contract(amt); var s : storage) : return is
  block {
    // отримуємо дані облікового запису, а з них – кількість дозволених для
    витрачання токенів
    const ownerAccount : account = getAccount(owner, s);
    const spenderAllowance : amt = getAllowance(ownerAccount, spender, s);
  } with (list [transaction(spenderAllowance, 0tz, contr)], s)
```

Функція *getTotalSupply* повертає кількість токенів на балансах всіх користувачів:



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

Головна функція приймає назву псевдо-точки входу і її параметри:

```
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;
```

Готуємо смарт-контракт токена до публікації

Щоб не витратити час на додавання Taquito і створення файлів проекту, скористаємося папкою taq-test з минулого уроку.

Запустіть редактор VS Code. Створіть в папці taq-test папку token, а в ній — файл token.ligo. Скопіюйте в файл код токена.

```
// оголошуємо псевдонім trusted типу address. Ми будемо використовувати його для
позначення адрес, у яких є право відправляти токени
type trusted is address;

// оголошуємо псевдонім amt (amount) типу nat для зберігання балансів
type amt is nat;

(* оголошуємо псевдонім account типу record. У ньому будемо зберігати дані
користувачів, яким можна передавати токени *)

type account is
  record [
    balance          : amt;
```



```

    allowances      : map (trusted, amt);
  ]

(* оголошуємо тип сховища смарт-контракту. В ньому зберігається загальна
кількість токенів, а також структура даних big_map, яка пов'язує публічні адреси
і баланси користувачів *)
type storage is
  record [
    totalSupply      : amt;
    ledger           : big_map (address, account);
  ]

(* оголошуємо псевдонім для методу return, який будемо використовувати для
повернення операцій. У коротких контрактах можна обійтися без нього, але в
контрактах з декількома псевдо-точками входу простіше один раз прописати тип
повернення і використовувати його в кожній функції *)
type return is list (operation) * storage

(* оголошуємо порожній список noOperations. Його повертатимуть функції transfer і
approve *)
const noOperations : list (operation) = nil;

(* оголошуємо псевдоніми вхідних параметрів для кожної базової функції FA 1.2 *)
// функція transfer отримує на вхід адресу відправника, адресу одержувача і суму
транзакції
type transferParams is michelson_pair(address, "from", michelson_pair(address,
"to", amt, "value"), "")
// approve отримує адресу користувача і кількість токенів, які він може
відправити з балансу смарт-контракту
type approveParams is michelson_pair(trusted, "spender", amt, "value")
// getBallance отримує адресу користувача і проксі-контракту, яким вона
відправить дані про баланс
type balanceParams is michelson_pair(address, "owner", contract(amt), "")
// getAllowance отримує адресу користувача, дані його аккаунта в смарт-контракті
і проксі-контракт
type allowanceParams is michelson_pair(michelson_pair(address, "owner", trusted,
"spender"), "", contract(amt), "")
// totalSupply не використовує michelson_pair, тому що перший вхідний параметр –
пусте значення unit – і так виявиться першим після сортування компілятора
Michelson
type totalSupplyParams is (unit * contract(amt))

```



```
(* голошуємо псевдо-точки входу: даємо назву і присвоюємо їм тип параметрів, які описали вище *)
type entryAction is
  | Transfer of transferParams
  | Approve of approveParams
  | GetBalance of balanceParams
  | GetAllowance of allowanceParams
  | GetTotalSupply of totalSupplyParams

function getAccount (const addr : address; const s : storage) : account is
  block {
    // присвоюємо змінної acct значення типу account: нульовий баланс і порожній запис allowances
    var acct : account :=
      record [
        balance      = 0n;
        allowances = (map [] : map (address, amt));
      ];

    (* перевіряємо, чи є в сховищі аккаунт користувача. Якщо немає – залишаємо в acct пусте значення з попереднього блоку. Якщо є – присвоюємо змінній acct значення зі сховища. Функція повертає значення acct *)
    case s.ledger[addr] of
      None -> skip
    | Some(instance) -> acct := instance
    end;
  } with acct

function getAllowance (const ownerAccount : account; const spender : address;
const _s : storage) : amt is
  (* якщо користувач дозволив відправити кілька токенів, функція присвоює це кількість змінної amt. Якщо не дозволив – кількість токенів дорівнює нулю *)
  case ownerAccount.allowances[spender] of
    Some (amt) -> amt
  | None -> 0n
  end;

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



```
(* викликаємо функцію getAccount, щоб привласнити змінній senderAccount дані
облікового запису користувача. Потім ми використовуємо senderAccount, щоб
зчитувати баланс користувача і дозволи *)
var senderAccount : account := getAccount(from_, s);

(* перевіряємо, чи достатньо у користувача засобів для переказу. Якщо ні –
віртуальна машина перериває виконання контракту, якщо достатньо – продовжує
виконувати контракт *)
if senderAccount.balance < value then
  failwith("NotEnoughBalance")
else skip;

(* перевіряємо, чи може адрес-ініціатор транзакції відправити токени. Якщо він
запитує переказ з чужої адреси, функція запитує дозвіл у справжнього власника.
Якщо ініціатор і відправник – одна адреса, віртуальна машина продовжує виконувати
контракт *)
if from_ /= Tezos.sender then block {
  (* викликаємо функцію getAllowance, щоб власник адреси-відправника вказав,
скільки tokenів він дозволяє відправити. Надаємо це значення константі
spenderAllowance *)
  const spenderAllowance : amt = getAllowance(senderAccount, Tezos.sender, s);

  (* якщо власник дозволив відправити менше tokenів, ніж зазначено у вхідному
параметрі, віртуальна машина припинить виконувати контракт *)
  if spenderAllowance < value then
    failwith("NotEnoughAllowance")
  else skip;

  (* віднімаємо від дозволеної для переказу кількості tokenів суму
транзакції *)
  senderAccount.allowances[Tezos.sender] := abs(spenderAllowance - value);
} else skip;

(* віднімаємо від балансу адреси-відправника кількість відправлених tokenів *)
senderAccount.balance := abs(senderAccount.balance - value);

(* оновлюємо запис про баланс відправника в storage *)
s.ledger[from_] := senderAccount;

(* ще раз викликаємо функцію getAccount, щоб отримати або створити запис
аккаунта для адреси-одержувача *)
```




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

(* додаємо до балансу одержувача кількість відправлених токенів *)
destAccount.balance := destAccount.balance + value;

(* оновлюємо запис про баланс одержувача в storage *)
s.ledger[to_] := destAccount;

}
// повертаємо порожній список операцій і стан storage після виконання функції
with (noOperations, s)

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

    (* отримуємо дані облікового запису користувача *)
    var senderAccount : account := getAccount(Tezos.sender, s);

    (* отримуємо поточну кількість токенів, яку користувач дозволив відправити *)
    const spenderAllowance : amt = getAllowance(senderAccount, spender, s);
    if spenderAllowance > 0n and value > 0n then
        failwith("UnsafeAllowanceChange")
    else skip;

    (* вносимо в дані облікового запису нову дозволену кількість токенів для
витрачання *)
    senderAccount.allowances[spender] := value;

    (* оновлюємо сховище смарт-контракту *)
    s.ledger[Tezos.sender] := senderAccount;

} with (noOperations, s)

function getBalance (const owner : address; const contr : contract(amt); var s :
storage) : return is
block {

    // присвоюємо константі ownerAccount дані облікового запису
    const ownerAccount : account = getAccount(owner, s);
```



```

}
// повертаємо проміжковому контракту баланс аккаунта
with (list [transaction(ownerAccount.balance, 0tz, contr)], s)

function getAllowance (const owner : address; const spender : address; const
contr : contract(amt); var s : storage) : return is
  block {
    // отримуємо дані облікового запису, а з них – кількість дозволених для
    витрачання токенів
    const ownerAccount : account = getAccount(owner, s);
    const spenderAllowance : amt = getAllowance(ownerAccount, spender, s);
  } with (list [transaction(spenderAllowance, 0tz, contr)], s)

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 потрібно скомпілювати в Michelson, щоб віртуальна машина Tezos могла його виконати. На минулому уроці ми вставили код контракту з двох рядків безпосередньо в скрипт розгортання. Код токена включає 200 рядків, тому краще зберегти його окремим файлом і імпортувати в скрипт за допомогою команди `import`.

Відкрийте онлайн-середовище LIGO і вставте код токена в поле редактора. У випадяючому списку виберіть пункт `Compile Contract` і поставте галочку в поле `Output Michelson in JSON format`. Натисніть кнопку `Run`. Компілятор видасть готовий код під полем редактора.



Contract Examples
 Increment (PascaliGO)
 Increment (CameLIGO)
 Increment (ReasonLIGO)
 ID (CameLIGO)
 ID (PascaliGO)
 ID (ReasonLIGO)
 Hashlock Contract (CameLIGO)
 Hashlock Contract (PascaliGO)
 Hashlock Contract (ReasonLIGO)

Increment (PascaliGO)

```

106   else skip;
107   (* Вносимо в дані облікового запису нову дозволену кількість токенів для витрачання *)
108   senderAccount.allowances [spender] := value;
109   (* Оновлюємо сховище смарт-контракту *)
110   s.ledger [Tezos.sender] := senderAccount;
111   } with (noOperations, s)
112
113   function getBalance (const owner: address; const contr: contract (amt); var s: storage): return is
114   block {
115     // присвоюємо константі ownerAccount дані облікового запису
116     const ownerAccount: account = getAccount (owner, s);
117   }
118   // повертаємо проміжковому контракту баланс аккаунта
119   with (list [transaction (ownerAccount.balance, 0tz, contr)], s)
120
121
122   function getAllowance (const owner: address; const spender: address; const contr: contract (amt); var s:
123   block {
124     // отримуємо дані облікового запису, а з них — кількість дозволених для витрачання токенів
125     const ownerAccount: account = getAccount (owner, s);
126     const spenderAllowance: amt = getAllowance (ownerAccount, spender, s);
127   } with (list [transaction (spenderAllowance, 0tz, contr)], s)
128
129
130   function getTotalSupply (const contr: contract (amt); var s: storage): return is
131   block {
132     skip
133   } with (list [transaction (s.getTotalSupply, 0tz, contr)], s)
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
264
```



Створіть в папці `taq-test` файл `token-deploy.ts` і вставте в нього код з `deploy.ts`. Потім його потрібно доопрацювати: додати метод `file sync` для читання інших файлів, імпортувати код контракту і вказати початковий стан сховища.

Після методів `import` додайте метод `file sync` і константу `Tezos` для виклику методів `Taquito`:

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

Замініть метод `try`. Після цього задайте початковий стан сховища — загальну кількість токенів і баланс користувача:

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

    // зчитуємо код з файлу token.json
    code: JSON.parse(fs.readFileSync("./token.json").toString()),
    // задаємо стан сховища на мові Michelson. Замініть обидві адреси на
    // адресу свого облікового запису в тестовій мережі,
    // а числа — на кількість токенів, яку ви хочете випустити
    init:
      '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" (Pair { Elt
"tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" 1000 } 1000) } 1000)',
  })
}
```

Готовий код виглядає 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 tezoz = new TezosToolkit(provider);
  await importKey(
    tezoz,
    "hoqfgsoy.qyisbhtk@tezoz.example.org", // пошта
```



```
"ZnnZLS0v60", // пароль
[
  "able", // мнемоніка
  "public",
  "usual",
  "hello",
  "october",
  "owner",
  "essence",
  "old",
  "author",
  "original",
  "various",
  "gossip",
  "core",
  "high",
  "hire"
].join(' '),
"2bed8dc244ee43a1e737096c4723263c269049d8" // приватний ключ
);

try {
  const op = await tezos.contract.originate({
    // зчитуємо код з файлу token.json
    code: JSON.parse(fs.readFileSync("./token.json").toString()),
    // задаємо стан сховища на мові Michelson. Замініть обидві адреси на
    адресу свого облікового запису в тестовій мережі,
    // а числа – на кількість токенів, яку ви хочете випустити
    init:
      '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" (Pair { Elt
"tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" 1000 } 1000) } 1000)',
      })

    // початок розгортання
    console.log('Awaiting confirmation...');
    const contract = await op.contract();

    // звіт про розгортання: кількість використаного газу, значення сховища
    console.log('Gas Used', op.consumedGas);
    console.log('Storage', await contract.storage());
```



```
// хеш операції, за яким можна знайти контракт в блокчейн-оглядачі
console.log('Operation hash:', op.hash);
} catch (ex) {
  console.error(ex);
}
}

deploy();
```

Відкрийте термінал і виконайте команду `prx ts-node token-deploy.ts`. Через кілька хвилин Taquito опублікує контракт токена в тестовій мережі і видасть хеш операції.

Знайдіть її в [florence.tzstats](https://florence.tzstats.com) і перевірте стан сховища контракту. У нас в ньому записана тисяча tokenів, у вас — ваша кількість.

Tezos Contract ☆
KT1DUDLarkFG9tmA4uZCgbiHv5SJA9oUBw8G

General | Contract | QR code | Calls | Gas Used

Key	Value
Full Balance	0 XTZ
First Seen	Aug 5, 2021 16:52
Last Seen	Aug 5, 2021 16:52
Total Received	0 XTZ
Total Sent	0 XTZ

Storage | Bigmap #129,708

#	Key & Type	Content
1	<code>storage: struct</code>	<code>2 entries</code>
2	<code>ledger: big_map { address => struct }</code>	<code>[object Object]</code>
3	<code>totalSupply: nat</code>	<code>1000</code>



В полі Bigmap знаходяться записи про адреси власників токенів. При публікації контракту ми випустили всі токени на тестову адресу.

#	Key & Type	Content
1	<code>tz1imn4..KafZ</code>	2 entries
2	<code>allowances: map { address => nat }</code>	1 entry
3	<code>tz1imn4..KafZ</code>	1000
4	<code>balance: nat</code>	1000

Відправляємо токени за допомогою Taquito

Пам'ятайте, як ми додали число в сховище контракту за допомогою Taquito? Тепер ускладнюємо завдання: викличемо функцію `transfer` і передамо токени іншому користувачеві.

Встановіть гаманець Tezos і створіть акаунт, якщо не зробили це на минулому уроці. Ми рекомендуємо [Temple Wallet](#), тому що він з коробки підтримує тестові мережі Tezos.

Відкрийте VS Code і створіть файл `token-transfer.ts`. Вставте в нього код:

```
// імпортуємо методи Taquito і файл з даними тестового аккаунта acc.json

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

  // налаштовуємо посилання на публічний вузол тестової мережі
  private tezoz: TezosToolkit;
  rpcUrl: string;
```




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

    // зчитуємо пошту, пароль і мнемонічну фразу, з якої можна отримати приватний
    // ключ
    this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
    acc.password, acc.mnemonic.join(' ')))
}

// оголошуємо метод transfer, який приймає параметри:
//
// 1) contract – адреса контракту;
// 2) sender – адреса відправника;
// 3) receiver – адреса одержувача;
// 4) amount – кількість токенів для відправки

public transfer(contract: string, sender: string, receiver: string, amount:
number) {
    this.tezos.contract
        .at(contract) // звертаємося до контракту за адресою
        .then((contract) => {
            console.log(`Sending ${amount} from ${sender} to ${receiver}...`);
            // звертаємося до точки входу transfer, передаємо їй адреси
            // відправника і одержувача, а також кількість токенів для відправки
            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); // чекаємо одне
            // підтвердження мережі
        })
        .then((hash) => console.log(`Hash: https://florence.tzstats.com/
        ${hash}`)) // отримуємо хеш операції
        .catch((error) => console.log(`Error: ${JSON.stringify(error, null,
        2)}`));
}
}
```



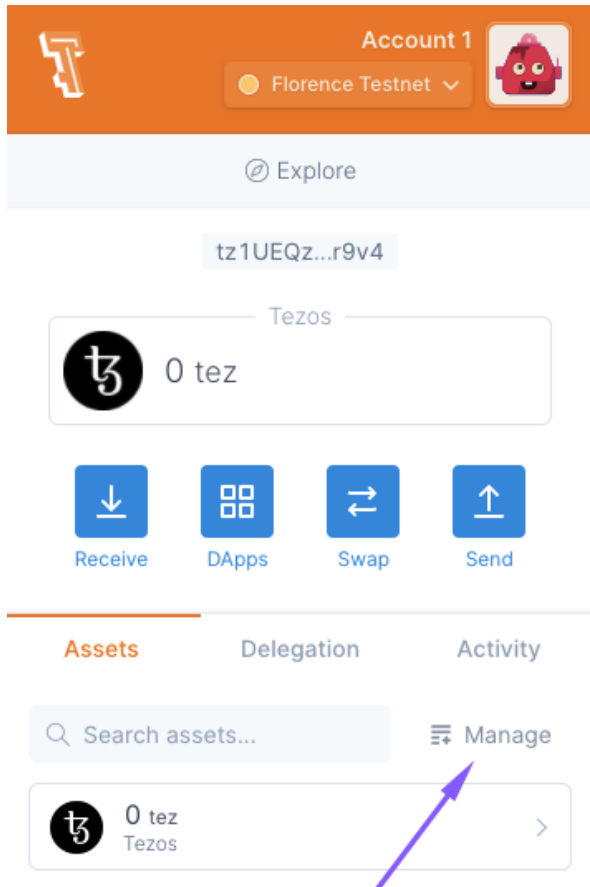
Тепер створіть файл `transfer.ts` і вставте в нього код:

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

const RPC_URL = "https://florencenet.api.tez.ie"
const CONTRACT = "KT1DUdLarkFG9tmA4uZCgbiHv5SJA9oUBw8G" // адреса опублікованого
контракту
const SENDER = "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" // публічна адреса
відправника – візьміть її з acc.json
const RECEIVER = "tz1UEQzJbuaGJgwwkekk6HwGwaKvjZ7rr9v4" /// публічна адреса
одержувача – візьміть її з гаманця Tezos, який ви створили
const AMOUNT = 3 // кількість токенів для переказу. Можете ввести інше число
new token_transfer(RPC_URL).transfer(CONTRACT, SENDER, RECEIVER, AMOUNT);
```

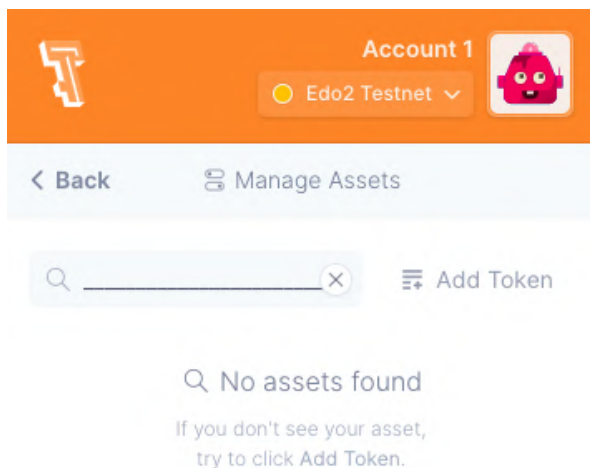
Відкрийте консоль і виконайте команду `npm ts-node transfer.ts`. Зачекайте, поки консоль поверне хеш операції.

Тепер відкрийте гаманець Temple Wallet. Натисніть на кнопку **Tezos Mainnet** у верхній частині вікна і виберіть пункт **Florence Testnet** в випадаючому списку.



Гаманець виглядає порожнім, але насправді токени вже прийшли на вашу адресу. Temple не бачить токени, тому що в коді контракту немає метаданих з назвою та іншими параметрами. Метадані — тема наступного уроку. Але зараз ми можемо додати їх вручну і побачити токени на гаманці.

Скопіюйте адресу контракту токена з файлу transfer.ts. Відкрийте Temple Wallet і натисніть кнопку **Manage** праворуч від поля **Search Assets**, а потім — кнопку **Add Token**.



Temple відкриє браузер з вкладкою, на якій можна додати токен вручну. Виберіть стандарт FA 1.2 в випадаючому списку **Token type**, а потім вставте адресу смарт-контракту в поле **Address**.



Temple Account 1 Florence Testnet

< Back + Add Token

Token type

FA 1.2
FA 2

Address
Address of a deployed token contract.

KT10uvjm34tv34uncq09c23

Заповніть дані про токені: тікер, опис, число знаків після коми і посилання на логотип. Пишіть, що подобається, але залиште значення «0» в полі **Decimals**. Натисніть кнопку **Add Token**, щоб додати в гаманець токен з заданими настройками.

Symbol
Token symbol, like 'USD' for United States Dollar.

TST

Name
Token name, like 'Bitcoin' for BTC asset.

Simple test token

Decimals
A number of decimal places after point. For example: 8 for BTC, 2 for USD.

0

Icon URL (optional)
Image URL for token logo.

<https://www.pngitem.com/pimgs/m/79-7!>

Add Token

Готово! Тепер Temple Wallet відображає ваш токен. Він підхопив точки входу FA 1.2, а значить ви можете переглядати баланс і відправляти токени з інтерфейсу гаманця. Однак одержувачу доведеться вручну налаштувати метадані, щоб побачити переказ.

Assets Delegation Activity

Search assets... Manage

	0 tez Tezos	>
	69 TST Simple test token	>

На наступному уроці ми розберемося з метаданими і опублікуємо повноцінний токен, який гаманці будуть бачити відразу. А ще випустимо NFT стандарту FA 2.



Підбиваємо підсумки

Токени — це записи в сховищі смарт-контрактів на кшталт «Адресу TZ1 належить тисяча монет». Переказ токенів відбувається так:

1. Користувач або додаток викликає смарт-контракт.
2. Смарт-контракт перевіряє, чи може користувач відправити токени.
3. Смарт-контракт оновлює записи про баланси користувачів: зменшує кількість токенів відправника на суму переказу, а потім додає її до балансу одержувача.

Розробники протоколу створюють стандарти, які описують основні функції токенів: зберігання записів про користувачів, перевірку балансів і проведення транзакцій. У Tezos два популярних стандарта: FA 1.2 для взаємозамінних активів і FA 2 для NFT.

Розробники можуть писати смарт-контракти токенів з нуля або використовувати імплементації — шаблони з базовими функціями. Просунуті механізми на кшталт випуску і спалювання токенів потрібно створювати самостійно.

Метадані токенів та випуск NFT на Tezos

На минулому уроці ми вивчили стандарт FA 1.2 і опублікували смарт-контракт токена в тестовій мережі Tezos. Але ми не використали метадані, тому токен не відображався в гаманці автоматично.

На цьому уроці ми додамо в смарт-контракт метадані, щоб його бачили гаманці і блокчейн-оглядачі. Потім вивчимо стандарт NFT в мережі Tezos і створимо невзаємозамінний токен.



Як додатки бачать токени

Нагадаємо, токени — це не монетки і не окремі файли, а записи в сховищі смарт-контрактів. Якщо у вашому гаманці є 100 kUSD, то в сховищі контракту записано «Адресу TZ1 належить 100 tokenів».

Крім балансів користувачів в сховищі знаходяться метадані — інформація про смарт-контракти: назва та короткий опис токена, тікер, число знаків після коми, ідентифікатор (id), посилання на логотип, адреса власника, псевдо-точки входу і інші параметри.

Метадані потрібні, щоб додатки бачили токени і правильно відображали інформацію про них. Якщо метадані відсутні, додаток не побачить токен, а якщо заповнені неправильно — відобразить його з помилками.

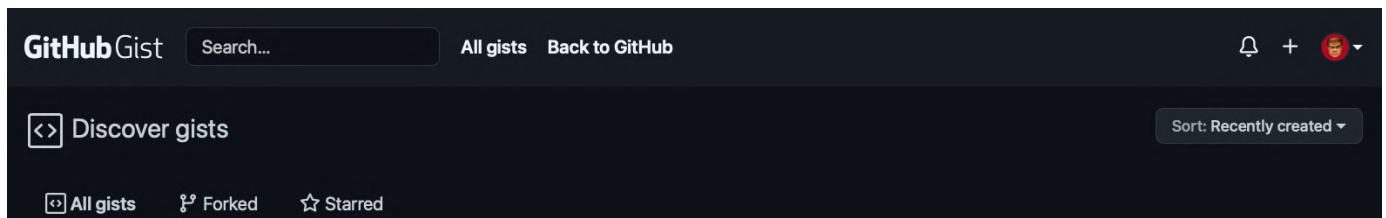
Створюємо JSON-файл з метаданими

З 2020 року більшість розробників Tezos використовує стандарт [TZIP-16](#), згідно з яким метадані зберігаються в JSON-файлі.

Розробник повинен завантажити файл на будь-який публічний сервер — власний сайт, IPFS або Github, а потім розмістити посилання на нього в сховищі смарт-контракту.

Ми додамо метадані в код токена token.ligo з минулого уроку. Для цього скористаємося сервісом [Gist](#) від Github: створимо JSON-файл і отримаємо пряме посилання на нього (Uniform Resource Identifier, URI).

Зареєструйтеся або авторизуйтеся на Github, а потім перейдіть в [Gist](#). Натисніть на + в правому верхньому куті вікна, щоб створити новий файл.



Назвіть файл fa12-metadata.json і вставте в нього шаблон:

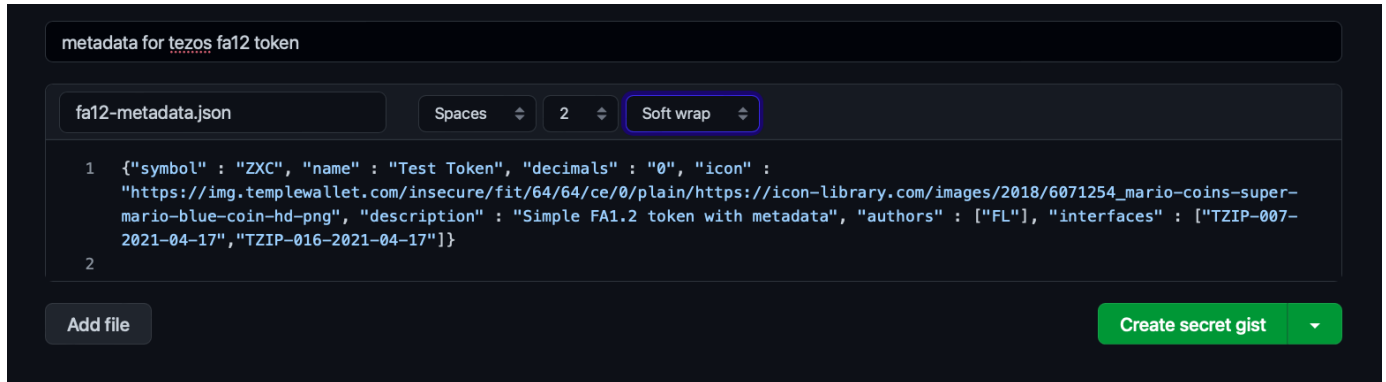
```
{ "Symbol": "Коротка назва, тікер", "name": "Повна назва", "decimals": "0",  
  "icon": "посилання на зображення-логотип. Воно повинне закінчуватися на .jpg,  
  .png або .svg ", "description": "Опис токена ", "authors": "автор ",  
  "interfaces": [ " TZIP-007-2021-04-17 ", " TZIP-016-2021-04-17 " ] }
```

Заповніть поля: введіть тікер, назву токена і додайте посилання на логотип. Поля **decimals** і **interfaces** залиште як є:

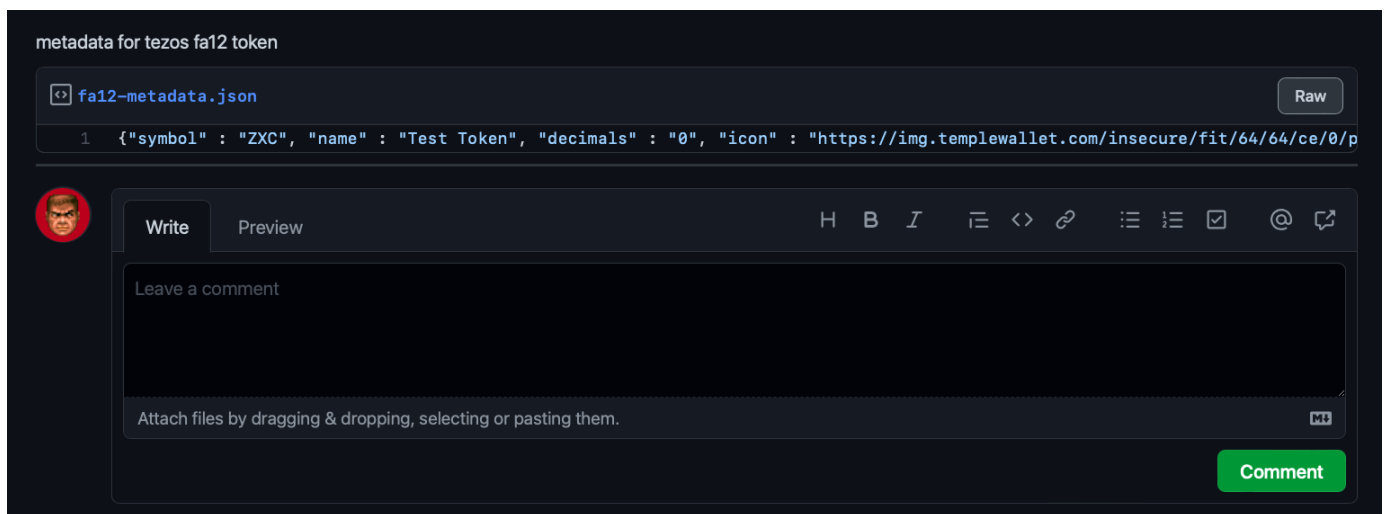


- **decimals** — кількість знаків після коми. Якщо встановити його більше 0, наприклад 3, гаманці відобразять токен у вигляді 0,001. Ми залишаємо значення за замовчуванням, щоб не ускладнювати приклад;
- **interfaces** — правила відображення метаданих і список стандартних точок входу, які використовує контракт. Зміна цього поля може привести до проблем з відображенням токена в деяких гаманцях.

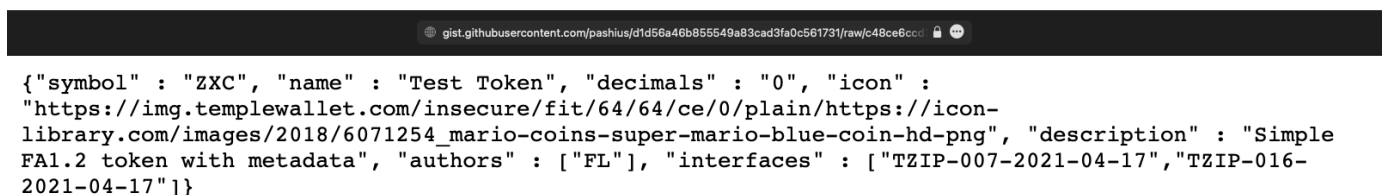
Відкрийте список, що випадає на зеленій кнопці, і виберіть пункт **Create public gist**. Після цього натисніть саму кнопку. Так ви збережете файл і зробите його доступним для всіх користувачів.



Після оновлення сторінки натисніть кнопку **Raw** в правому верхньому куті вікна, щоб відкрити файл за прямим посиланням.



Не закривайте вкладку з файлом, вона знадобиться через кілька хвилин.





Додаємо метадані в смарт-контракт

URI-посилання на метадані повинна знаходитися в сховищі контракту в спеціальному форматі — *metadata: big_map (string, bytes)*. Додамо його в storage.

Запустіть VS Code і відкрийте файл token.ligo. На початку контракту знайдіть оголошення типу storage і додайте в нього формат метаданих.

```
(* Оголошуємо тип сховища смарт-контракту. В ньому зберігається загальна кількість токенів)
type storage is
record [
  metadata: big_map (string, bytes)
  totalSupply: amt;
  ledger: big_map (address, account);
]

(* Оголошуємо псевдонім для методу return, який будемо використовувати для повернення операцій)
type return is list (operation) * storage

(* Оголошуємо порожній список операцій. Його повертатимуть функції transfer і approve
```

Ми оновили контракт на LIGO. Тепер його потрібно скомпілювати в формат JSON для публікації в тестнеті.

Скопіюйте вміст файлу token.ligo і перейдіть на сайт ide.ligolang.org. Виберіть пункт **Compile Contract** в випадаючому списку і поставте галочку в поле **Output Michelson in JSON format**. Натисніть кнопку **Run**. Компілятор видасть готовий код під полем редактора.

Contract Examples

- Increment (PascalLIGO)
- Increment (CameLIGO)
- Increment (ReasonLIGO)
- ID (CameLIGO)
- ID (PascalLIGO)
- ID (ReasonLIGO)
- Hashlock Contract (CameLIGO)
- Hashlock Contract (PascalLIGO)
- Hashlock Contract (ReasonLIGO)

Increment (PascalLIGO)

```
56 } with acct
57
58 function getAllowance (const ownerAccount: account; const spender: address; const s: storage): amt is
59 (* Якщо користувач дозволив відправити кілька токенів, функція присвоєє це кількість змінної amt. Якщо
60 case ownerAccount.allowances [spender] of
61   Some (amt) => amt
62   | None => 0n
63 end;
64
65 function transfer (const from_: address; const to_: address; const value: amt; var s: storage): return is
66 block {
67   (* Викликаємо функцію getAccount, щоб привласнити змінній senderAccount дані облікового запису корист
68   var senderAccount: account := getAccount (from_, s);
69   (* Перевіряємо, чи достатньо у користувача засобів для переказу. Якщо ні – віртуальна машина перериває
70   if senderAccount.balance < value then
71     failwith ( "NotEnoughBalance")
72   else skip;
73
74   (* Перевіряємо, чи може адрес-ініціатор транзакції відправити токени. Якщо він запитує переказ з чужої
75   if from_ != Tezos.sender then block {
76     (* Викликаємо функцію getAllowance, щоб власник адреси-відправника вказав, скільки токенів він дозво
77     const spenderAllowance: amt = getAllowance (senderAccount, Tezos.sender, s);
78     (* Якщо власник дозволив відправити менше токенів, ніж зазначено у вхідному параметрі, віртуальна маш
79     if spenderAllowance < value then
80       failwith ( "NotEnoughAllowance")
81     else skip;
82     (* Віднімаємо від дозволеної для переказу кількості токенів суму транзакції *)
```

Configure

Compile Contract **Run**

Access function

The function name from where your contract will start

main

☒ Output michelson in JSON format

Line 73, Column 4

View in Try-Michelson IDE

```
[ { "prim": "parameter",
  "args":
    [ { "prim": "or",
      "args":
        [ { "prim": "or",
```

Перейдіть в VS Code, відкрийте файл token.json. Замініть старий код новим і збережіть файл.

Тепер в **storage** є поле **metadata**: гаманці і блокчейн-оглядачі будуть шукати в ньому посилання на JSON-файл з метаданими. Поки в цьому полі порожньо: ми вкажемо URI файлу в сховищі контракта перед розгортанням.



Щоб віртуальна машина Michelson могла прочитати посилання, його потрібно перекласти в байтовий формат. Поверніться на вкладку з URI на meta.json і скопіюйте вміст адресного рядка.

```
gist.githubusercontent.com/pashius/d1d56a46b85549a83cad3fa0c561731/raw/c48ce8ccc

{"symbol": "ZXC", "name": "Test Token", "decimals": "0", "icon": "https://img.templewallet.com/insecure/fit/64/64/ce/0/plain/https://icon-library.com/images/2018/6071254_mario-coins-super-mario-blue-coin-hd-png", "description": "Simple FA1.2 token with metadata", "authors": ["FL"], "interfaces": ["TZIP-007-2021-04-17", "TZIP-016-2021-04-17"]}
```

Відкрийте онлайн-конвертер на кшталт [Onlinestringtools](https://www.onlinestringtools.com/). Вставте текст посилання в поле **String** і зніміть галочку з **Add Whitespaces**, щоб прибрати пропуски між байтами.

Конвертер видасть довге число в поле **Bytes** — це і є посилання в байтовому форматі. Ми будемо використовувати її пізніше, а поки не закривайте вкладку.

string

https://raw.githubusercontent.com/pashius/token-metadata/main/meta.json

Import from file

Save as...

Copy to clipboard

bytes

68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f706173686975732f746f6b656e2d6d657461646174612f6d61696e2f6d6574612e6a736f6e

Chain with...

Save as...

Copy to clipboard

string to bytes converter options

Full Bytes

☒ Use Full Bytes
If a byte is less than 0xf, make it 0x0f.

Delimiter

☐ Add a Whitespace
Select this option to add a whitespace char after each byte.

Перейдіть на VS Code і створіть файл storage.tz. Вставте в нього шаблон сховища:

```
'(Pair (Pair {Elt "публічний адресу з файлу асс.json" (Pair {Elt "публічний адресу з файлу асс.json" кількість токенів} кількість токенів)} {Elt "" 0хпосилання на meta.json в байтовому форматі})) кількість токенів) '
```

Заповніть поля шаблону. Важливо: перед посиланням на meta.json потрібно обов'язково додати порожній рядок — "". Це ключ, за яким віртуальна машина Tezos розуміє, що далі буде посилання на метадані.



Публічна адреса обов'язково повинна бути в лапках, а кількість токенів і посилання — без. Перед посиланням в байтовому форматі обов'язково допишіть 0x, інакше компілятор її не прочитає.

```
taq-test > stuff > ≡ storage.tz
1  '(Pair (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
2      (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" 1000 } 1000) }
3      { Elt ""
4          0x68747470733a2f2f636c6f75646666c6172652d697066732e636f6d2f697066732f62616666b
5          1000) )'
6
```

Скопіюйте код з storage.js і переключіться на token-deploy.ts. Замініть старий запис стану сховища на новий.

```
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
```

Відкрийте термінал в VS Code. Переконайтеся, що знаходитесь в папці taq-test, і виконайте команду:
`npx ts-node token-deploy-ts`

Taquito видасть посилання на хеш операції. Перейдіть на сайт tzstats.io, знайдіть по хешу адреса контракту і скопіюйте його.

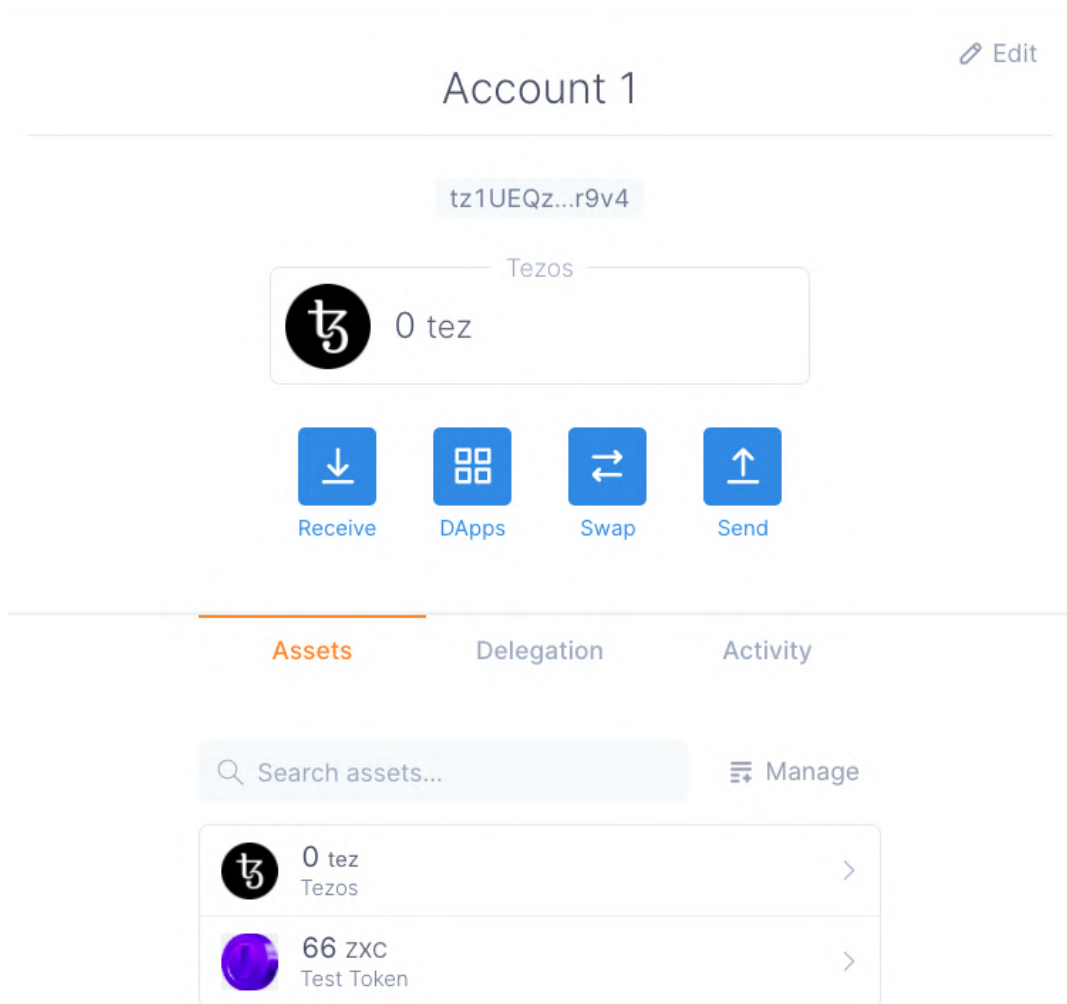
Перевірте, що гаманці бачать метадані — підготуйте і виконайте транзакцію. Відкрийте файл transfer.ts і замініть стару адресу токена новою, яку ви тільки що опублікували.

```
taq-test > TS transfer.ts > [E] CONTRACT
1  import { token_transfer } from './token-transfer';
2
3  const RPC_URL = "https://florenenet.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);
```

Відкрийте термінал і виконайте команду:
`npx ts-node transfer.ts`



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



Разбираемся со стандартом FA 2

Унікальність NFT (non-fungible token) забезпечують два фактори — адреса контракту-емітента і ідентифікатор (id). У взаємозамінних токенів ідентифікаторів немає.

NFT можна випустити з допомогою простого контракту з двох функцій: transfer для передачі токенів і update для поновлення балансів користувачів. Технічно, такий токен буде унікальним. Але якщо не дотримуватися стандартів FA 1.2 і FA 2, гаманці не побачать цей NFT. Для передачі токена доведеться користуватися клієнтом Tezos і терміналом.

Щоб уніфікувати роботу з NFT, розробники Tezos створили стандарт FA 2. Він описує інтерфейс для роботи з взаємозамінними, невзаємозамінними, непередаваними і іншими видами токенів.

Стандарт FA 2 складніший, ніж FA 1.2, але дає розробникам більше свободи. Наприклад, за допомогою FA 2 можна випустити кілька токенів в одному контракті або групувати транзакції в пакети (batch), щоб заощадити газ.



Код FA 2-токена складається з чотирьох частин:

- повідомлення про основні помилки;
- інтерфейси — оголошення всіх кастомних типів даних;
- функції для роботи з операторами — користувачами, які можуть передавати токени;
- ядро — метадані, сховище, функції передачі tokenів і перевірки балансу.

Імплементація NFT-стандарту FA2 на LIGO виглядає 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

// оголошуємо тип ідентифікатора токена — натуральне число
type token_id is nat

// оголошуємо типи вхідних параметрів, які приймає функція переказу токена:
// адресу одержувача, id і кількість tokenів. В тип transfer додаємо адресу
// відправника
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);
]

// оголошуємо типи для читання балансу: адреса власника, 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));
]

// оголошуємо тип оператора – адреси, яка може відправляти токени
type operator_param is
[@layout:comb]
record [
  owner : address;
  operator : address;
  token_id: token_id;
]

// оголошуємо тип параметрів, які потрібні для оновлення списку операторів
type update_operator is
[@layout:comb]
| Add_operator of operator_param
| Remove_operator of operator_param
```




```
// оголошуємо тип, який містить метадані NFT: ID токена і посилання на json-файл
type token_info is (token_id * map(string, bytes))

type token_metadata is
big_map (token_id, token_info)

// оголошуємо тип з посиланням на метадані смарт-контракту. Ці дані будуть
відображатися в гаманці
type metadata is
big_map(string, bytes)

// оголошуємо тип, який може зберігати записи про кілька токенів і їх метадані в
одному контракті
type token_metadata_param is
[@layout:comb]
record [
  token_ids : list(token_id);
  handler : (list(token_metadata)) -> unit;
]

// оголошуємо псевдо-точки входу: передачу токенів, перевірку балансу, оновлення
операторів і перевірку метаданих
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

// оголошуємо типи даних для зміни дозволів на передачу токенів. Наприклад, з їх
допомогою можна зробити токен, який не можна відправити на іншу адресу
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

// оголошуємо тип, який зберігає записи про операторів в одному big_map
type operator_storage is big_map ((address * (address * token_id)), unit)

// оголошуємо функцію для оновлення списку операторів
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

// оголошуємо функцію, яка перевіряє, чи може користувач оновити список
операторів
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

// оголошуємо функцію, яка перевіряє, чи може користувач оновити список адрес
власників токенів, і тільки в цьому випадку викликає функцію оновлення
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

// оголошуємо функцію, яка перевіряє дозволи на передачу токенів. Якщо користувач
не може передати токен, функція припиняє виконання контракту
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

// оголошуємо функцію для передачі токена власником
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)

// оголошуємо функцію, яка збирає всі транзакції одного токена в пакет (batch)
(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

// оголошуємо тип даних для зберігання записів про те, на якій адресі знаходяться
токени з заданим id
type ledger is big_map (token_id, address)

// оголошуємо сховище контракту: метадані TZIP-16, реєстр адрес і токенів, список
операторів і ончейн-метадані
type collection_storage is record [
    metadata : big_map (string, bytes);
    ledger : ledger;
    operators : operator_storage;
    token_metadata : token_metadata;
]

// оголошуємо функцію передачі токена. Вона отримує id токена, адреси відправника
та одержувача, а потім перевіряє, чи є у відправника право передати токен
function transfer (
    const txs : list(transfer);
    const validate : operator_validator;
    const ops_storage : operator_storage;
    const ledger : ledger) : ledger is block {
    // перевірка права відправника передати 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
                // перевірка кількості переданих NFT. Маємо на увазі, що контракт
                випустив тільки один токен з цим id
                // Якщо користувач хоче передати 0, 0.5, 2 або іншу кількість токенів,
                функція перериває виконання контракту
                if (dst.amount = 0n) then

```



```

11
    else if (dst.amount /= 1n)
    then (failwith(fa2_insufficient_balance): ledger)
    else block {
        const owner = Big_map.find_opt(dst.token_id, 11);
    } with
        case owner of
            Some (o) ->
                // перевірка, чи є у відправника токен
                if (o /= tx.from_)
                then (failwith(fa2_insufficient_balance) : ledger)
                else Big_map.update(dst.token_id, Some(dst.to_), 11)
            | None -> (failwith(fa2_token_undefined) : ledger)
        end

    ,
    tx.txs,
    1
)
} with List.fold(make_transfer, txs, ledger)

// оголошуємо функцію, яка поверне баланс відправника
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)

// оголошуємо головну функцію з псевдо-точками входу. Ці псевдо-точки – основа
стандарту 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

```

Випускаємо NFT на FA 2

Відкрийте VS Code і створіть файл `nft.ligo` в папці `taq-test`. Вставте в нього вміст вікна вище або скопіюйте код з [Gist](#).

Скомпільуйте код на LIGO в JSON. Скопіюйте вміст файлу вікна і перейдіть на сайт ide.ligolang.org. Виберіть пункт **Compile Contract** в випадаючому списку і поставте галочку в полі **Output Michelson in JSON format**. Натисніть кнопку **Run**. Компілятор видасть готовий код під полем редактора. Створіть в VS Code файл з назвою `nft.json` і скопіюйте туди скомпільований код.





Підготуйте два JSON-файлу з метаданими — для і NFT і смарт-контракту. Перейдіть в [Gist](#) і натисніть на + в правому верхньому куті, щоб створити новий файл.

Назвіть файл `nft_meta.json` і вставте в нього шаблон:

```
{ "symbol": "тікер",
  "name": "назва NFT",
  "description": "опис",
  "decimals": "0",
  "isBooleanAmount": true,
  "artifactUri": "посилання на токенизований об'єкт",
  "thumbnailUri": "логотип NFT",

  "Minter": "ім'я емітента токена",
  "Interfaces": [ "TZIP-007-2021-04-17", "TZIP-016-2021-04-17", "TZIP-21" ] }
```

Заповніть всі поля, крім `interfaces`. Вкажіть посилання на токенизований об'єкт і логотип токена у вигляді «`https:// ...`», а не у байтовому форматі.

Відкрийте список, що випадає на зеленій кнопці, і виберіть пункт **Create public gist**. Після цього натисніть саму кнопку. Так ви збережете файл і зробите його доступним для всіх користувачів.

Після оновлення сторінки натисніть кнопку **Raw** в правому верхньому куті вікна, щоб відкрити файл за прямим посиланням. Чи не закривайте вкладку.

Тепер створіть в Gist файл `contract_meta.json`. Вставте в нього шаблон:

```
{ "name": "назва контракту",
  "description": "опис контракту",

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

Заповніть метадані контракту, натисніть кнопку **Create Public Gist**, а потім — кнопку **Raw**. Не закривайте вкладку, вона стане в нагоді через кілька хвилин.

Підготуйте скрипт для публікації NFT. Створіть файл `nft-deploy.ts` і скопіюйте в нього код з `token-deploy.ts`.

Знайдіть метод `code: JSON.parse()` і замініть в ньому назву файлу для зчитування:



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

```
try {
  const op = await tezos.contract.originate({
    // зчитуємо файла nft.json
    code: JSON.parse(fs.readFileSync("./nft.json").toString()),
    // задаємо стан сховища на мові Michelson.
    init:
      '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt "" 0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f706173686975732f62393134313431643663343739656531303034326532376265613831346265342f7261772f336461356234376461623031623863303935346338373539306636366564323230323235636438322f636f6e74726163745f6d6574612e6a736f6e } ) { Elt (Pair "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" 5) Unit } { Elt 0 (Pair 5 { Elt "" 0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f706173686975732f62663063653337333738313431636230393033396564323461363134316166372f7261772f656636373766643462623639656436316439626438666361616234366263663830306263373664622f6e66746d6574322e6a736f6e } ) } )' \
    --burn-cap 0.6215
  })
}
```

Для публікації смарт-контракту потрібно задати стан сховища. Для цього створіть файл nft-storage без розширення і вставте в нього шаблон:

```
'(Pair (Pair { Elt id "адреса вашого гаманця" } { Elt "" 0x посилання на метадані
контракта в байтовому форматі}) { Elt (Pair "адреса вашого гаманця" "адрес вашого
гаманця" id) Unit } { Elt 0 (Pair id { Elt "" 0x посилання на метадані токена в
байтовому форматі}) })')
```

Заповніть поля в шаблоні. Цей код повинен бути одним рядком без переносів.

Вказуйте ідентифікатор (id) токена без лапок. Не використовуйте кирилицю в назві токена: Michelson працює тільки з Unicode, тобто латиницею.

Ми випустимо токен з id = 5:

```
taq-test > nft-storage.tz
1 '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt ""
0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f706173686975732f623931
34313431643663343739656531303034326532376265613831346265342f7261772f33646135623437646162303162
3863303935346338373539306636366564323230323235636438322f636f6e74726163745f6d6574612e6a736f6e }
) { Elt (Pair "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
5) Unit } { Elt 0 (Pair 5 { Elt ""
0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f706173686975732f626630
63653337333738313431636230393033396564323461363134316166372f7261772f65663637376664346262363965
6436316439626438666361616234366263663830306263373664622f6e66746d6574322e6a736f6e } ) } )' \
--burn-cap 0.6215
```

Вставте код стану сховища в файл nft-deploy.ts в поле init. Не забудьте виділити його апострофами і поставити кому. Наш скрипт виглядає так:



```

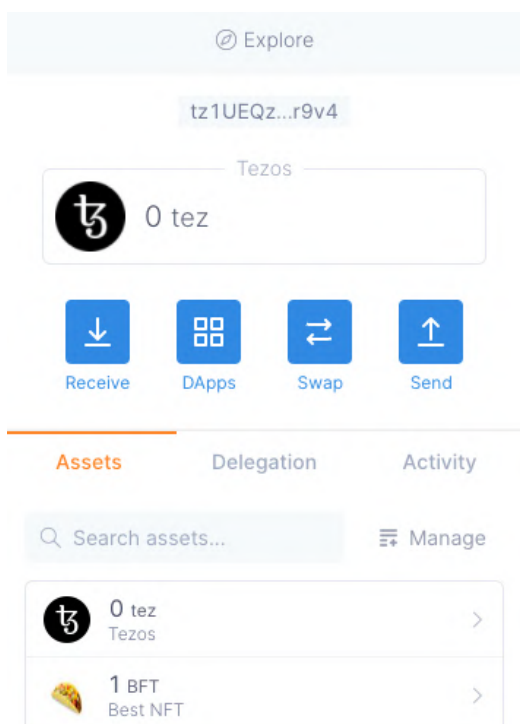
TS nft-deploy.ts > deploy
32   "2bed8dc244ee43a1e737096c4723263c269049d8" //приватний ключ
33   };
34
35   try {
36     const op = await tezos.contract.originate({
37       // зчитуємо файла nft.json
38       code: JSON.parse(fs.readFileSync("./nft.json").toString()),
39       // задаємо стан сховища на мові Michelson.
40       init:
41         '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt ""
42           0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f7061736
43           86975732f62393134313431643663343739656531303034326532376265613831346265342f7261
44           772f64633735306666653865643031393437396463326663376533333383330323937623830623
45           636382f636f6e74726163745f6d6574612e6a736f6e }) { Elt (Pair
46             "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
47             5) Unit } { Elt 5 (Pair 5 { Elt ""
48               0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f7061736
49               86975732f62663063653337333738313431636230393033396564323461363134316166372f7261
50               772f656636373766643462623639656436316439626438666361616234366263663830306263373
51               664622f6e66746d6574322e6a736f6e }) })',
52     });
53
54     //початок розгортання
55     console.log('Awaiting confirmation...');

```

Відкрийте термінал і виконайте команду:

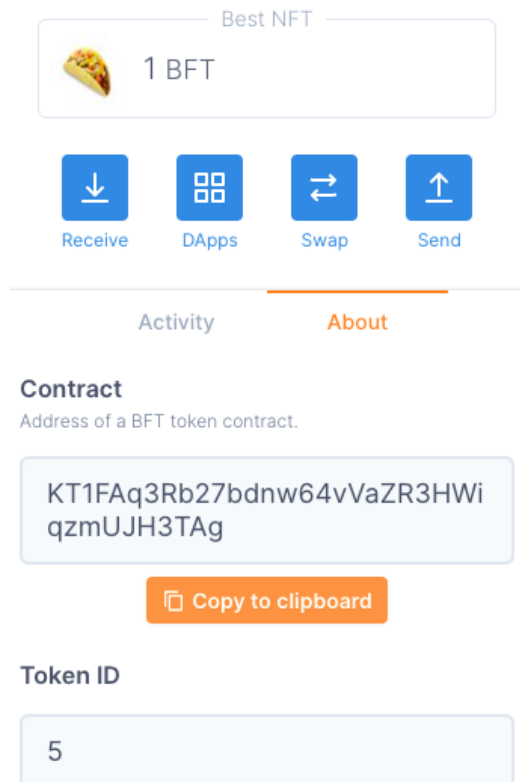
```
npx ts-node nft-deploy.js
```

Дочекайтеся завершення розгортання і перевірте свій гаманець. NFT з'явиться в ньому протягом хвилини.



Якщо розгортання пройшло без помилок, але гаманець не показує токен, перевірте мережу. Можливо, гаманець під'єднаний до мейннету, а не тестової мережі. В крайньому випадку додайте токен вручну за допомогою кнопки **Manage**. Іноді при створенні нових токенів з випуском на адресу користувача гаманець може їх не відображати до першої транзакції.

Ви побачите ідентифікатор токена і наявність NFT на балансі, але не сам токенізований контент. Гаманці не можуть його відобразити: у них немає інтерфейсів для читання поля **artifactUri**, як у NFT-маркетплейсов.



Перейдіть на блокчейн-оглядач [BetterCallDev](#) і знайдіть свій контракт за хешем операції або адресою.

Перейдіть на вкладку **Tokens**. Посилання на токенизований контент знаходиться в полі **artifact_uri**.



OPERATIONS (1)
STORAGE
CODE
INTERACT
TOKENS (1)
METADATA

TOKEN ID

METADATA URI

{ } RAW JSON

5

https://gist.githubusercontent.co...

Best N

TOKEN INFO

symbol: **BFT**
name: **Best NFT**
decimals: **0**
description: **Best NFT EVER**
artifact_uri: **https://e7.pngegg.com/pngimages/108/458/png-clipart**
display_uri: **https://e7.pngegg.com/pngimages/855/226/png-clipart**
thumbnail_uri: **https://dictionary.cambridge.org/ru/images/thumb/1**
is_transferable: **true**
is_boolean_amount: **true**
should_prefer_symbol: **false**

Top holders

HOLDER 1

tz1UEQz...r9v4

1 BFT

Для випуску і спалювання NFT потрібно додати відповідні функції в код контракту. Ми цього не робили, щоб не ускладнювати знайомство з FA 2.

Підведемо підсумки

Метадані — стандартизований опис смарт-контракту або токена, в який входить назва, тикер, опис, кількість знаків після коми, посилання на логотип, підтримувані інтерфейси та інші дані.

Найзручніший спосіб додати метадані — скористатися стандартом TZIP-16. Згідно з ним, інформація про токени та смарт-контракти зберігається в JSON-файлах. Їх потрібно розмістити на публічному сервері, наприклад на Github, і додати посилання в сховище. Плюс такого підходу: файли можна виправити, якщо ви допустили помилку.

NFT існують завдяки метаданим: розробник записує в них ідентифікатор токена і посилання на токенизований контент. Зв'язка ідентифікатора і адреси контракту унікальна, тому NFT з однаковим контентом і метаданими будуть відрізнятися один від одного.