



Blockchain Development on Tezos

From constants to contract publishing



Foreword

The printed version of an [online course on blockchain development on Tezos](#) by ForkLog and Tezos Ukraine consists of five lessons and focuses on rookie devs.

We start with the basics: smart contract structure and LIGO syntax. Then you will learn to write contracts and interact with the blockchain. In the end, you will be able to create a test token and NFT on the Tezos network.

IMPORTANT: The community updates the Tezos protocol on a regular basis, while bakers discontinue outdated testnets. If the testnet in the example fails to work, please use a relevant RPC link from the list on [Taquito's website](#) before publishing your smart contract.



Contents

1. What Are Smart Contracts and How They Work	4
Tezos protocol	5
Programming languages in Tezos	6
Examples of smart contracts	6
Online development environment	9
2. Development Environment and Basic Types in LIGO	11
Install and set up VS Code	12
Install the LIGO container	13
Dry-running a smart contract in VS Code	16
Basic syntax in LIGO	18
3. Deploying Smart Contracts on Testnet with Taquito	25
The Taquito library	26
Creating an account on the testnet	33
Publishing the contract with Taquito	37
Interacting with the contract	42
4. Creating an FA 1.2 Token	45
Token standard FA 1.2	46
Implementation and functions of FA 1.2	47
Publishing the token contract	54
Adding the token to the wallet	64
5. Token Metadata and How to Issue NFT on Tezos	69
Contract metadata	70
Adding metadata to the token contract	72
The standard FA 2 for NFT	75
Issuing NFT on Tezos	84



Our partners who supported and consulted us in creating this online course:



Tezos Ukraine

An NPO seeking to popularise Tezos in Ukraine and Eastern Europe



Attic Lab

Creators of the explorer Teztracker and the staking service Everstake.
Digital currency consultants for Ukraine's National Bank



Baking Bad

The developers of the eponymous service for auditing payments for delegating tez, as well as the atomic swap exchange Atomex, blockchain explorers TzKT and Better Call Dev

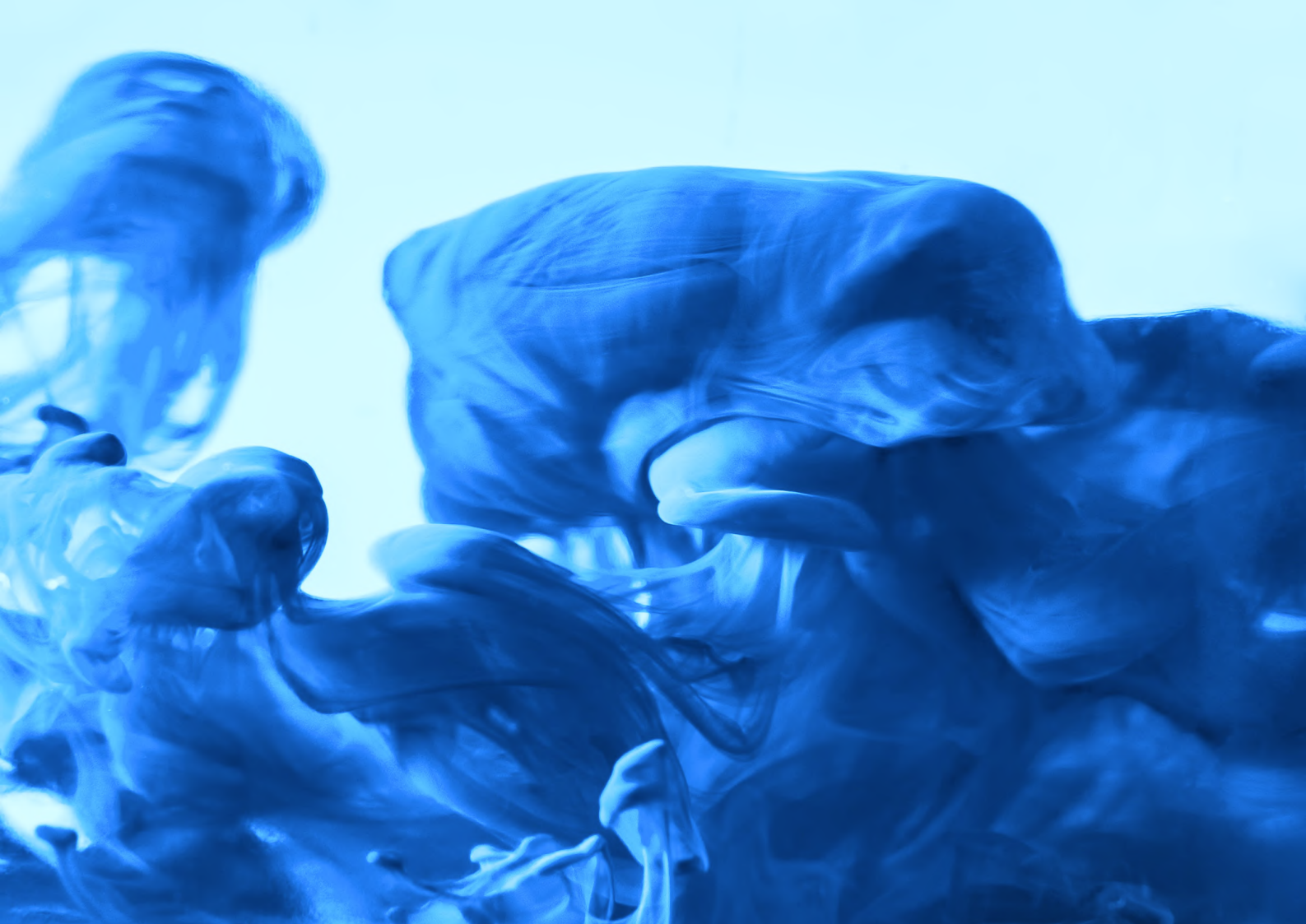


Madfish Solutions

The developers of the web wallet Temple, DEX Quipuswap, and the transpiler Sol2Ligo

Our team who worked on the online course:

Written by	Pavel Skoroplyas
Produced by	Svetlana Lukina
Stylistic framework by	Dmitri Boyko
Visuals by	Krzystof Szpak
Layouts by	Zara Arakelian
Frontend by	Alexander Pupko
Directed by	Vlad Likhuta



1

What are smart contracts and how they work

This article covers smart contracts on Tezos, their programming language, and the work of the virtual machine.

The lesson omits info on syntax as it focuses on the inner workings of blockchain's VM.



What are smart contracts?

A smart contract is a set of functions and data stored on a blockchain. Smart contracts have various purposes, from selling tokens to governing decentralised organisations. Here are some examples:

- Vesting contracts send tokens to users at a specified time. Early adopters of Tezos get their funds from [KT1KCYaULopm8i2CBbGF5EHXeXLUr4R6r2dA](#);
- Oracles question data sources and return averaged prices of assets or other info. This is how [Harbinger](#) gets cryptocurrency prices from trading platforms;
- Algorithmic stablecoins issue tokens and alter the cost of issuance depending on the native token's exchange rate. The [Kolibri](#) smart contract amends the cost of issuing kUSD depending on the tez price.

Smart contracts are executed by a virtual machine (VM). It uses the computational power of blockchain: smart contracts are executed by all nodes in the network yet only the fastest one gets to record the result in the block.

If smart contracts are called too often, the blockchain may get paralysed. To avoid it, protocol devs limit the maximum size of smart contracts in terms of code size and fee amounts.

How do blockchain VMs work?

Virtual machines understand low-level programming languages like byte code. They interpret such code faster compared to interpreting commands in JavaScript or Python. Here's an example of the same thing written differently that shows the difference:

- **low-level language:** $2 + 2 = 4$;
- **high-level language #1:** two plus two equals four;
- **high-level language #2:** If you add two to two You get four;
- **high-level language #3:** as a result of adding two to two we get a four.

When reading, we automatically interpret the latter lines as the simplest understandable variant: $2 + 2 = 4$. The same happens in programming: the compiler translates a high-level language into byte code.

Tezos's VM works with byte code Michelson. It's a low-level programming language with direct access to the stack (a data structure with fast access to data). Code in Michelson looks like this:

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



```
// Operators PUSH add the number 3 to the stack two times. Operator IFCMPEQ
compares the first two elements in the stack and executes the commands { DROP }
and displays the operation results.
```

Seasoned blockchain devs usually write their smart contracts in Michelson. Newbies, however, should better stick to high-level programming languages with libraries and human-readable syntax.

Programming Languages for Tezos Smart Contracts: SmartPy, LIGO and Lorentz

Tezos ecosystem members have designed several high-level programming languages. The most popular of them are listed below.

- **SmartPy** — is an object-oriented Python offshoot supporting the online compiler [SmartPy.io](https://smartpy.io), where one can develop, test, and publish smart contracts.
- **LIGO** — is an imperative programming language with a simple type system and an online compiler available at ide.ligolang.org. There are several dialects using syntaxes from other popular languages: PascaLIGO (Pascal), CameLIGO (OCaml), ReasonLIGO (ReasonML), and jsLIGO (JavaScript).
- **Lorentz** — is an inline object-oriented Haskell offshoot. Using Lorentz, devs can directly access the Michelson stack.

In this course, we'll be using the simplest language of them all, PascaLIGO.

A Simple Smart Contract With a Single Entry Point

A smart contract on LIGO is underpinned by the entry point. It is the main function of the smart contract that accepts the incoming transaction and calls other functions.

```
function function_name (const incoming_transaction_parameter : type; const
storage : type) : result_type is result
```



Input parameters of the main function:

- **incoming transaction parameter.** The function uses it to execute the code;
- **storage.** Information to be stored in the blockchain until the next execution of the smart contract, like the number of tokens or user profiles. The developer sets up the storage initial state when deploying the smart contract. After that, only the smart contract can change the contents of storage.

Output parameters of the main function:

- the list of executed operations, such as transaction details sent by the smart contract;
- results of developer-made functions' execution.

An example of a smart contract with an incrementing function that accepts a number and adds 1 to it:

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

Function code:

- *function main (const number : int; const storage : int)* – calling the main function with input parameters of number and storage. When the user sends a number to the contract, the main function sees it as number and executes the code;
- *: list (operation) * int* defining the type of the result returned by the function. In this case, it's a pair from the list of executed operations and the resulting number;
- *is ((nil : list (operation)), number + 1)* defining the result of function executing:
 - *nil : list (operation)* returns an empty list;
 - *,* divides values of tuple type;
 - *number + 1* a single-string function increasing the number by 1.

This smart contract can be modified: implement a decrement, squaring, or other mathematical operation. It has a single entry point that activates a single string of functions. This smart contract cannot execute several tasks like accepting a deposit, returning the balance, and sending a transaction.

Smart Contracts With Multiple Entry Points

A virtual machine starts executing a smart contract with its main function. It can only execute the functions called by the entry point. Developers increase the number of operations by creating pseudo-entry points within the main function. Here's what it requires:

1. Call the pseudo-entry points and the parameter type they will work with.
2. Describe the functions to be called by each pseudo-point.
3. Use the operator case in the main function. It will point the VM to the function to call if the pseudo-entry point is called.



An example of a smart contract with pseudo-points is a calculator that accepts the name of a mathematical operation and two numbers and returns calculation results. It doesn't record the values in storage because we don't need to put data onto a blockchain to execute mathematical operations.

```
// declaring type of numbers containing a tuple
type numbers is (int * int)

// declaring types of action that contain tuples
type action is
| Addition of numbers
| Subtraction of numbers
| Multiplication of numbers
| Division of numbers

// declaring data type in the smart contract's storage
type storage is int

// declaring mathematical functions

// (const a : int ; const b : int) – function parameters
// : int – function result type
// is a + b – result of function execution

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

// declaring the main function
// assign the type action to the first parameter, storage type to the parameter
store
// the function returns data of list(operation) and int types, a tuple of a list
and a number
// function execution result follows "is":
// 1) empty list nil : list(operation).
// 2) const result : int = recording the function execution result in the
constant "result"
// 3) case parameter of – it's the result of action-type parameter's execution,
```




```
// whose name is the same as the incoming transaction's name.
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;

    // displaying the result of the main function's execution: empty list of
    operations and the value of result
  } with ((nil : list(operation)), result)
```

If you send a request to the smart contract with the parameter `Multiplication(3, 9)` it will return the number 27. The virtual machine will execute the contract as follows:

1. Compare the parameter *Multiplication* with variants of pseudo-points under the operator case.
2. Go to the function *multiply* described by the dev earlier in the contract.
3. Put the request parameters (3 and 9) as *a* and *b*.
4. Execute the function *multiply* and record it in *store*.
5. Finalise the execution of the main function: return the empty list and the new value of *store*.

To inspect the smart contract, copy the calculator's code to the editor form at ide.ligolang.org. Select Dry Run in the drop-down list **Configure**. In the field **Parameters** input the name of the mathematical operation, specify two parameters, and press **Run**.



[Install](#)
[Docs](#)
[Tutorials](#)
[Blog](#)
[Ask Questions](#)
[Cheat Sheet](#)

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)

```

17 //: int - function result type
18 //is a + b - result of function execution
19
20 function add (const a : int ; const b : int) : int is a + b
21
22 function subtract (const a : int ; const b : int) : int is a - b
23
24 function multiply (const a : int ; const b : int) : int is a * b
25
26 function divide (const a : int ; const b : int) : int is a / b
27
28 //declaring the main function
29 //assign the type action to the first parameter, storage type to the parameter store
30 //the function returns data of list(operation) and int types, a tuple of a list and a number
31 //function execution result follows "is":
32 //1) empty list nil : list(operation).
33 //2) const result : int = recording the function execution result in the constant "result"
34 //2) case parameter of - it's the result of action-type parameter's execution,
35 //whose name is the same as the incoming transaction's name.
36 function main (const parameter : action ; const store : storage) :
37   (list(operation) * int) is block {
38     const result : int =
39       case parameter of
40       | Addition(n1, n2) -> add(n1, n2)
41       | Subtraction(n1, n2) -> subtract(n1, n2)
42       | Multiplication(n1, n2) -> multiply(n1, n2)
43       | Division(n1, n2) -> divide(n1, n2)
44     end;
45
46     //displaying the result of the main function's execution: empty list of operations and the value
47   } with ((nil : list(operation)), result)

```

Line 47, Column 43

(LIST_EMPTY() , -6)

Configure

Dry Run Run

Access function
The function name from where your contract will start

main

Parameters

Subtraction(1, 7)

Storage

0

Some conclusions

A smart contract is a code on the blockchain. The virtual machine executes it after receiving a transaction with required parameters.

Developers write smart contracts on high-level programming languages based on Python, Pascal, JS, or Haskell. Seasoned devs often opt to write byte code on Michelson.

Smart contracts written in PascallIGO consist of variables and functions. VM executes the contract starting with the entry point, the main function main. You can insert pseudo-entry points there which are additional functions.

A smart contract always returns the result of execution: a list of operations and the value of storage.

Development environment and basic types in LIGO

Last time, we launched a simple smart contract in the online compiler LIGO. You won't use it for big projects, though: it doesn't store code, add libraries, or call other smart contracts.

Coders use a development environment to write code, which usually are text editor programs that support plugins and libraries. This time, we install VS Code and cover the most fundamental data types in LIGO.

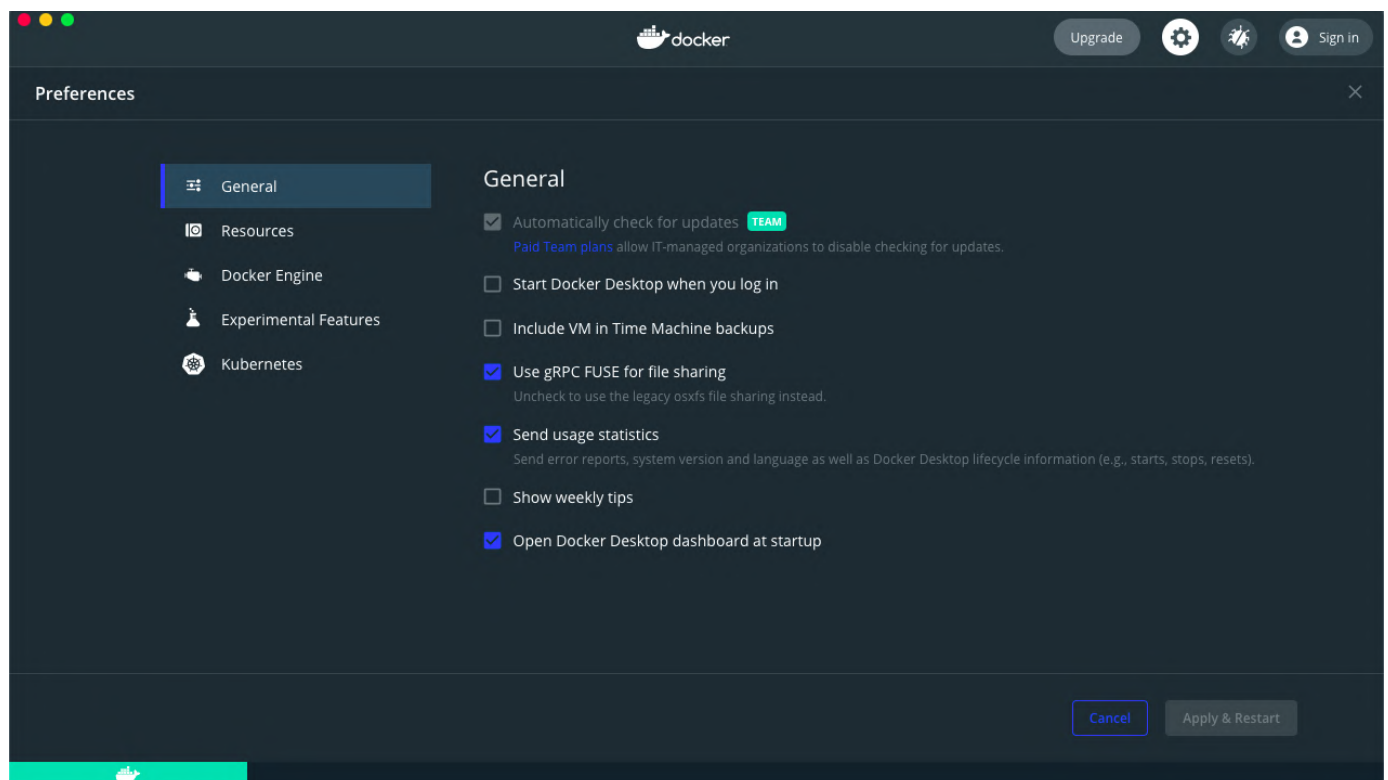


Installing and setting up the environment

We'll be using Visual Studio Code (VS Code) with the plug-in `pascaligo-vscode`.

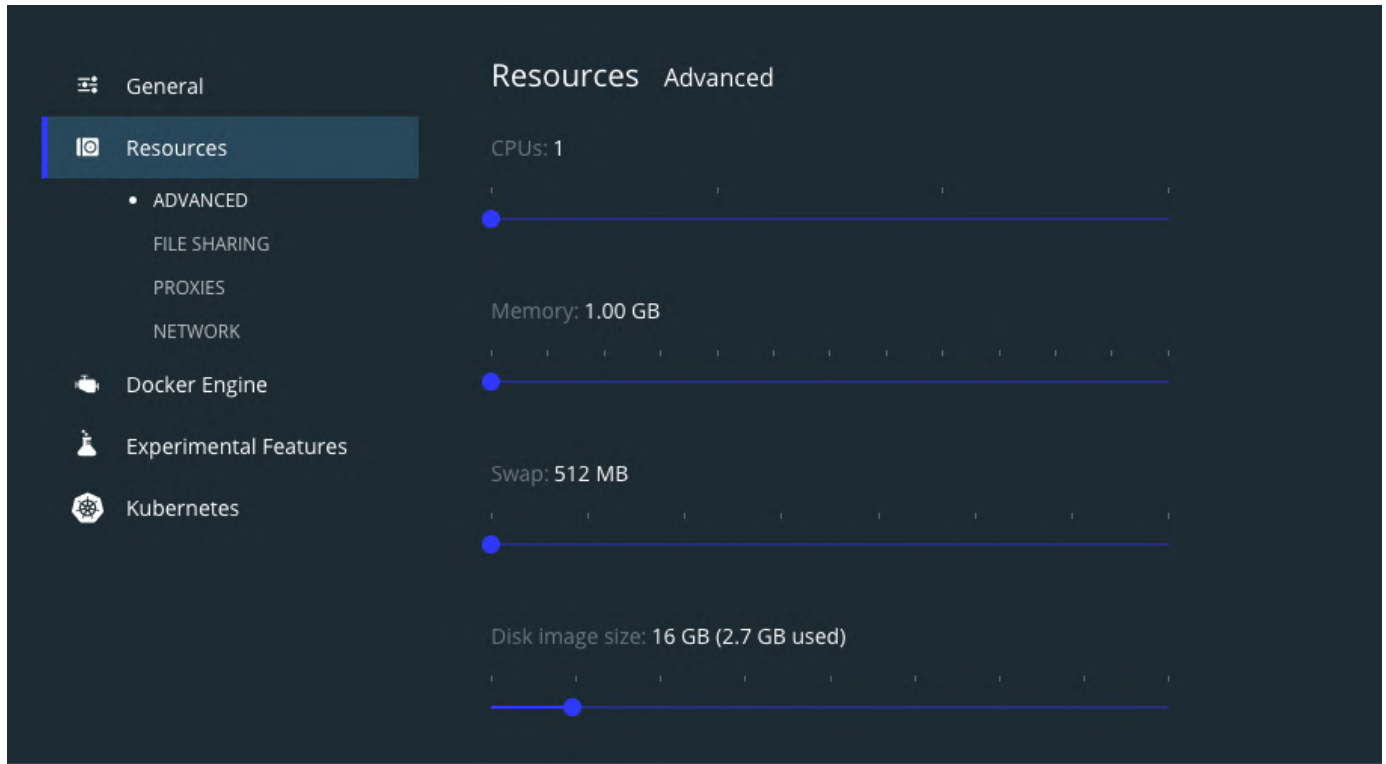
Here's how to install it:

1. Download the software from the official website of [Visual Studio Code](https://code.visualstudio.com/).
2. Install the app and download the plug-in [pascaligo-vscode](https://marketplace.visualstudio.com/items?itemName=pascaligo-vscode).
3. Install [Docker](https://docs.docker.com/desktop/). You will need it to start the LIGO library in the VM. The computer will be able to compile and execute code in PascalLIGO.
4. Start Docker and wait for a green stripe to appear in the lower-left corner of the app's interface.



You might encounter problems like these:

- **Docker presents “This computer doesn’t have VT-X/AMD-v enabled” while launching.** Open BIOS and activate the virtualisation of VT-X or AMD-V;
- **The computer noticeably slowed down once Docker started.** Open the software’s settings, go to Resources and assign fewer resources to Docker. To accept the change, click Apply and Restart in the lower right corner.



Install the LIGO container by executing the following commands in the terminal:

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

Docker will download the container and show the help menu for 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
```

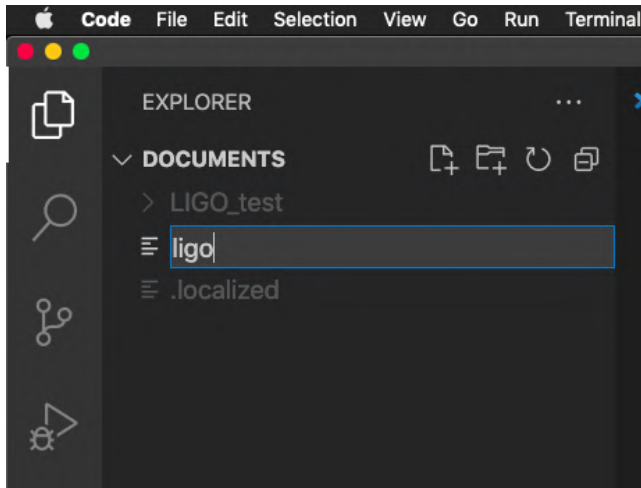



Make sure everything works. Execute the following command:

```
ligo --version
```

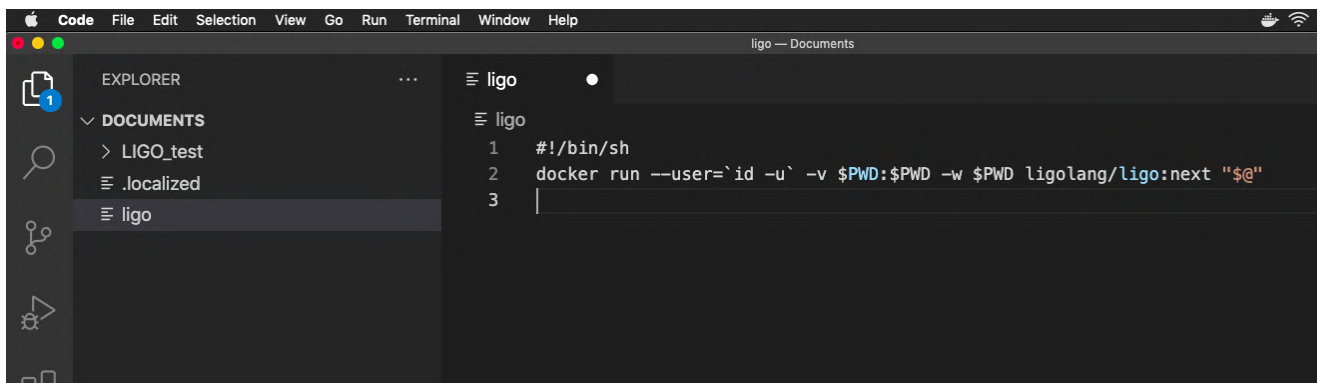
If the terminal returns **"ligo: command not found"**, it means Docker prematurely closes the LIGO container. This is how you fix it:

1. Open VS Code and create a file named "ligo".



2. Paste the following code therein:

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



3. Save the document and open the terminal. Go to the folder containing the file ligo:

```
cd ~/Documents
```

4. If your machine runs on Linux or Mac OS, activate the superuser mode:

```
sudo -s
```

5. Execute the following commands in the terminal:

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

They will link starting LIGO to the command "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. Make the file system-wide executable. In Linux and macOS open Finder and move ligo to /usr/local/bin on the system disc. In Windows, open the terminal and execute the following command:

```
$env:path += ";c:\[path to the folder containing ligo]"
```

7. Check if LIGO works by executing "`ligo --version`" again.

```

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#

```

On the black background, you see an error while executing `ligo --version` before ligo was moved to bin.

On the blue background, you see a successful execution once the file was moved.

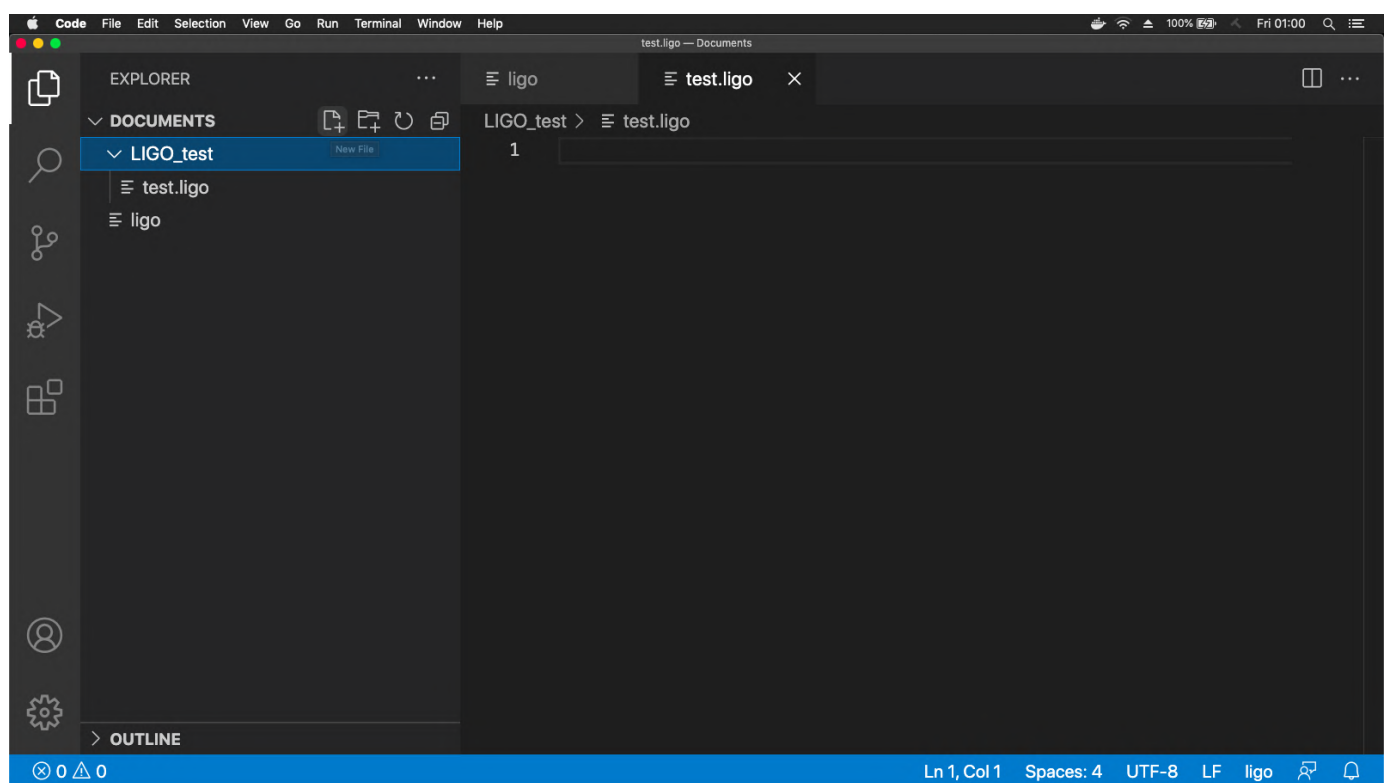


Compiling a test contract

Open VS Code and create a new project:

1. Click Open...
2. Choose the folder to save your text project in.
3. Therein, create a folder named LIGO_test.
4. Open the new folder.

On the left-hand side, you see files and folders in the current project. In the list, create an empty file named test.ligo. The extension .ligo will let VS Code know it has to activate the plugin to highlight the syntax and to use PascaLIGO commands.

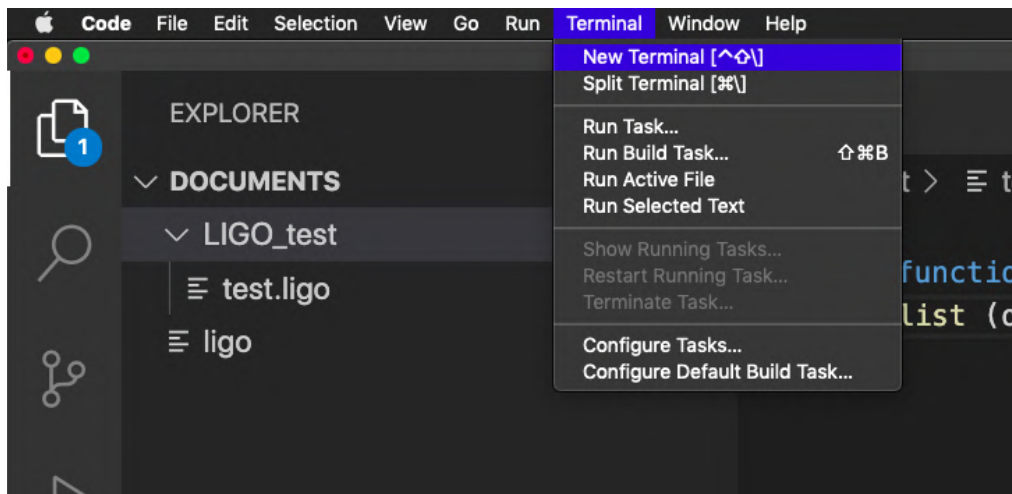


Copy-paste the function from the first lesson into test.ligo.

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



Try to compile the contract. Open the VS Code terminal

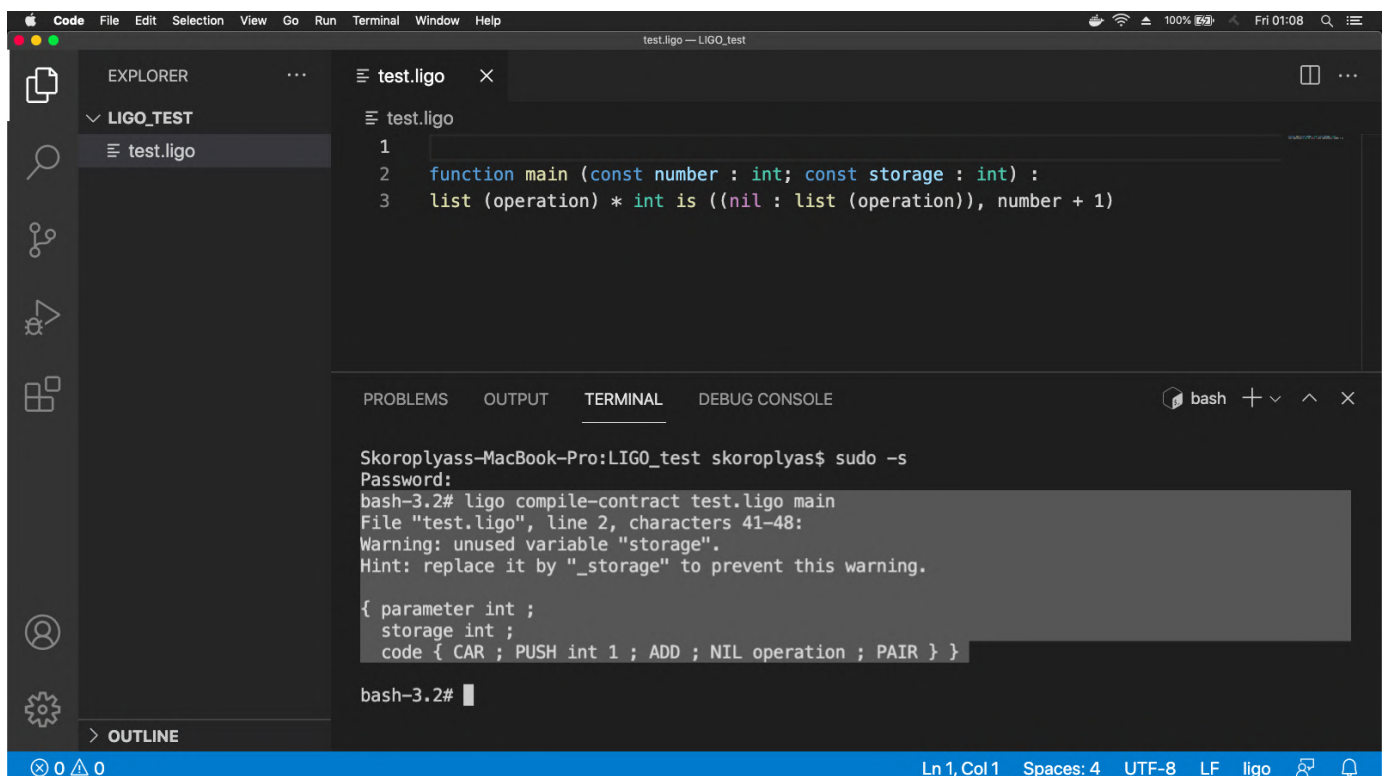


Input the following command and press Enter:

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

If you use Linux or macOS the terminal may run into an error "root/ligo." In this case, execute `sudo -s` to grant the terminal the rights to run the script ligo in the bin folder.

VS Code will translate the PascaLIGO code in test.ligo into Michelson and will check whether the function Main is executable.





Now test the smart contract via the dry run. The dry-run command will execute the contract with pre-set params within the Docker VM. To “rehearse” the test contract, input the following command into the terminal:

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

- *main*: entry point name;
- 2: the first entry param;
- 0: storage state.

You can use any other numbers to dry-run the contract.

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

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#
```

After that, the compiler will display the results of the main function execution with pre-set params.

Constants, variables, and types in LIGO

To store information, LIGO uses the following:

- **const (constants)**. A smart contract cannot change the value of a constant during function execution. Often, this is used to store params coming to the main function's input.
- **var (variables)**. A smart contract can change the variable's value at any time. Usually, variables are used in loop functions that the contract calls several times.

Constants and variables have to be assigned with a type so that the compiler could know how to treat them. LIGO uses the following types:

- *int*: any integer number, including those negative.
- *nat*: any natural number that cannot be negative.



- *string*: a string containing characters like Latin letters, numbers, punctuation marks and certain special characters like / and ~.
- *bool*: a Boolean variable, i.e. true or false.

Here's an example of declaring variables and constants:

```
//const [name] : [type] = [value]
//var [name] : [type] := [value]

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

Sometimes, you can't assign a value to a param until the smart contract is executed. For instance, a contract gets an entry param and uses it for calculations. In this case, you have to declare an empty constant or variable and assign a value to it later.

In the example below, we declare the constant "**number**" of int type. During the execution, the smart contract will assign the user-sent value to it.

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

You can declare constants anywhere in the contract yet variables have to be declared strictly within functions. For instance, we can take the number 1 from the main function in the example above. To do that, we have to declare the constant "**add**" in the beginning of the contract, assign the value 1 to it, and replace 1 with add.

```
const add : int = 1

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

But if you make **add** a variable, the compiler will run into the error ILL FORMED CONTRACT.



```

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.
bash-3.2#

```

Advanced types and aliases

Aside from basic types, there are several advanced ones in LIGO:

- data types — tuple, record, and map to send several values within one param;
- special types — addresses, tez balances, variants, timestamp.

Developers create such types with aliases. Aliases also simplify code reading and allow one to sort params according to their use cases.

An example of declaring an alias:

```

// declaring an alias of int type
type numbers is int

// declaring a constant of numbers type and assigning a value
const a : numbers = -5

// declaring a constant admin of user_id type
type user_id is nat
const admin : user_id = 1n

```



Frequently used data types

The main function accepts one param, so basic types like `int` or `nat` are not enough for complex smart contracts. We can bypass this, however, with data types that return several values within the same param.

tuple stores one or more values of a given type. In the example below, we declare a type `cube` where one can record the length of cube sides:

```
// declaring the alias cube to store the length of three sides of the cube.
type storage is int

type cube is int * int * int

// declaring the input param in the function, a constant side of cube type
function main (const side : cube ; const _store : storage) :
  (list(operation) * int) is block {

// side declaring the constant result and assigning a value: multiplication of
three values stored in the tuple
  const result : int = side.0 * side.1 * side.2
  } with ((nil : list(operation)), result)
```

record stores several constants/variables of different types. For example, a user type constant can contain id of int type, is_admin of bool type, and name of string type.

```
// declaring the type ser
type user is
  record [
    id      : nat;
    is_admin : bool;
    name     : string
  ]

// declaring the constant alice and assigning it the value of user type
const alice : user =
  record [
    id      = 1n;
    is_admin = True;
    name     = "Alice"
  ]
```



map links data in a set of key-value pairs. LIGO also has a **big_map** data type that optimises uploading a vast array of key-value pairs to save gas. It, however, does not support iteration.

```
// map declaring alias dims of tuple type, then cube dimensions of map type
type dims is int * int * int
type cube_dimension is map (string, dims)

// declaring map cube containing the cube name and its side length values
const cubes : cube_dimension =
  map [
    "big cube" -> (123243, 1251, 823);
    "small cube" -> (3, 3, 7)
  ]
// recording big of dims type to constant. For that, we get values out of map by
// calling "big cube" key
const big : option(dims) = cubes["big cube"]
```

unit is a value-less data type. You need it when syntax rules require the function to accept a parameter that isn't important for the code's operation. We use unit to declare types Labeouf, Nike, and Yoda in the example below.

variant allows one to use variables of different types depending on the situation and the smart contract's logic. In the example, we declare variant to create pseudo-entry points.

```
// unit declaring the alias speach and writing several variants. For each
// variant, we specify the param type it will use. In our case, the main function
// does nothing with the parameter, so we specify the empty type unit for the
// variants
type speach is
  |Labeouf of unit
  |Nike of unit
  |Yoda of unit

// we want the contract to return the string, so in the return type we say it
// returns the list of operations and the string
type return is (list(operation) * string)

// we declare that the function main has to receive a user param and record it in
// the constant word of speach type
function main (const word : speach; const _store : string) : return is
  ((nil : list (operation)),
```



```
// "case word of" compares the param with variants of speech type. If it is the
// same, the contract returns the specified string
case word of
  Labeouf (_n) -> "DO IT!"
| Nike (_n) -> "Just do it"
| Yoda (_n) -> "Do it you can"
end)
```

Special data types

Aside from complex data types, LIGO uses several blockchain-specific data types and the `failwith` function.

tez specifies the number of Tezos tokens XTZ (tez). When declaring, assigning, or operating the `tez` type one has to add `tz` or `tez` to the number.

LIGO also uses the suffix "mutez" for one-millionth of tez. For easier reading, big numbers can be divided by orders with underscoring. Thus, "1_000_000tez" will be read as "one million tez."

```
// 1 000 000 mutez = 1 tez
// assigning t the value of 1.5003 tez with the mutez suffix
const t : tez = 1_500_300mutez
```

address is a data type to store Tezos addresses.

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

failwith is a function for processing errors and exceptions. Thus, we can use `failwith` to prohibit the calculator to divide numbers by 7.

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

You will find more examples of operations with different data types in the [LIGO documentation](#).



Some conclusions

In an online development environment, you can't save code, plug in libraries, and call other contracts so programmers use Visual Studio Code and the LIGO container in Docker.

Basic data types in LIGO are `int`, `nat`, `string`, and `bool`. Special types for data storage include:

- *tuple*: several values within one constant or variable;
- *record*: several constants with assigned values within one constant or variable;
- *map*: several advanced-type constants linked to a key constant;
- *tez*: XTZ balance;
- *address*: address on the Tezos blockchain;
- *unit*: a value-less data type.

Deploying smart contracts on testnet with Taquito

Last time, we looked through basic and complex data types in LIGO. The knowledge is sufficient for a simple smart contract. Yet, for the other users to interact with the contract, it has to be deployed on a blockchain.

Developers use special tools to do that. In this lesson, we install the library Taquito and publish a contract on the Tezos testnet.



Installation and test run of Taquito

Taquito is a TypeScript library that simplifies the interaction with the Tezos blockchain and smart contracts. For example, here's how the manual deployment of a smart contract looks in the Tezos client console:

```
tezos-client originate contract <contract name>
for <user> transferring <number of tez> from <from_user>
running <path to the file with the code>
--init '<state of storage>'
--burn-cap <maximum fee>
```

The deployment script in Taquito looks even simpler:

```
try {
  const op = await tezos.contract.originate({
    // smart contract code
    code: `{
      `,
    // storage state
    init: ``,
  });
}
```

Taquito also enables one to send tokens, check address balances, and develop a simple web interface for a smart contract.

Installing Taquito

To work with Taquito, you will need nodeJS. Download it from the [official website](#) and install it on your machine. Then install the package manager yarn. It will download Taquito and automatically write it in the system.

To do so, open the console and execute the following command:

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



```

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

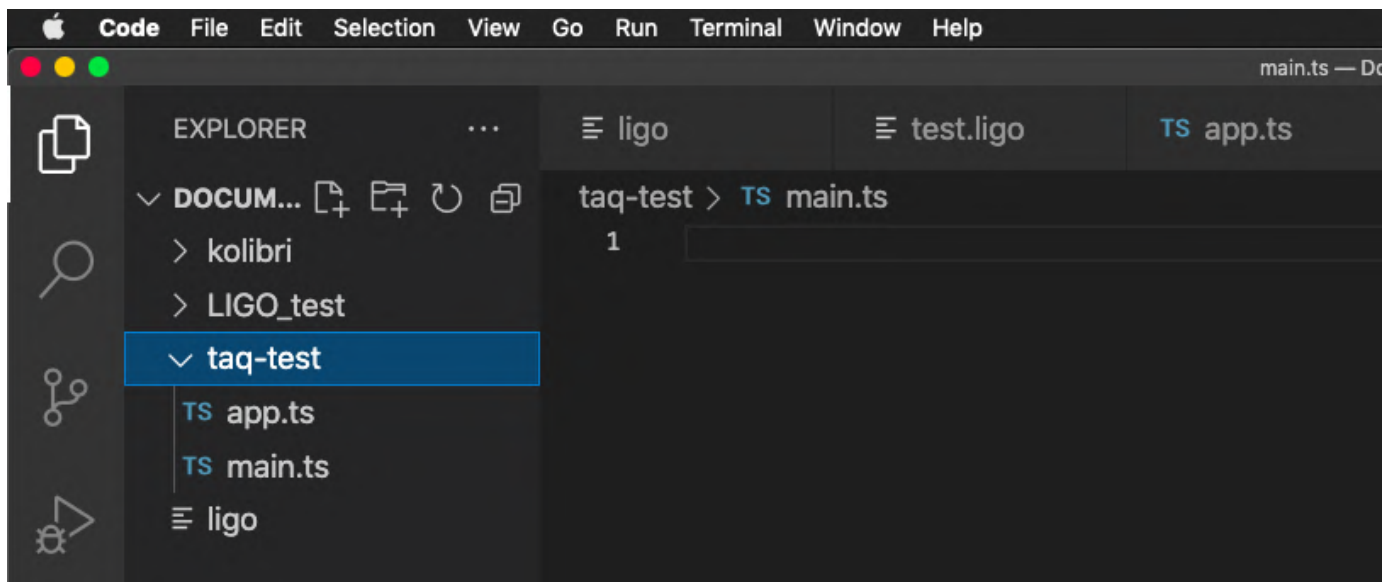
> yarn@1.22.10 preinstall /usr/local/lib/node_modules/yarn
> :; (node ./preinstall.js > /dev/null 2>&1 || true)

/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$

```

Taquito tools are now ready, so all that's left is to test its operation. Open Visual Studio Code, create a new folder called `taq-test`, and create two files in it named `app.ts` and `main.ts`.

`App.ts` will record Taquito functions and the basic code, and params and function calls in `main.ts`. This would simplify the code structure and help us avoid calling functions before declaring them.

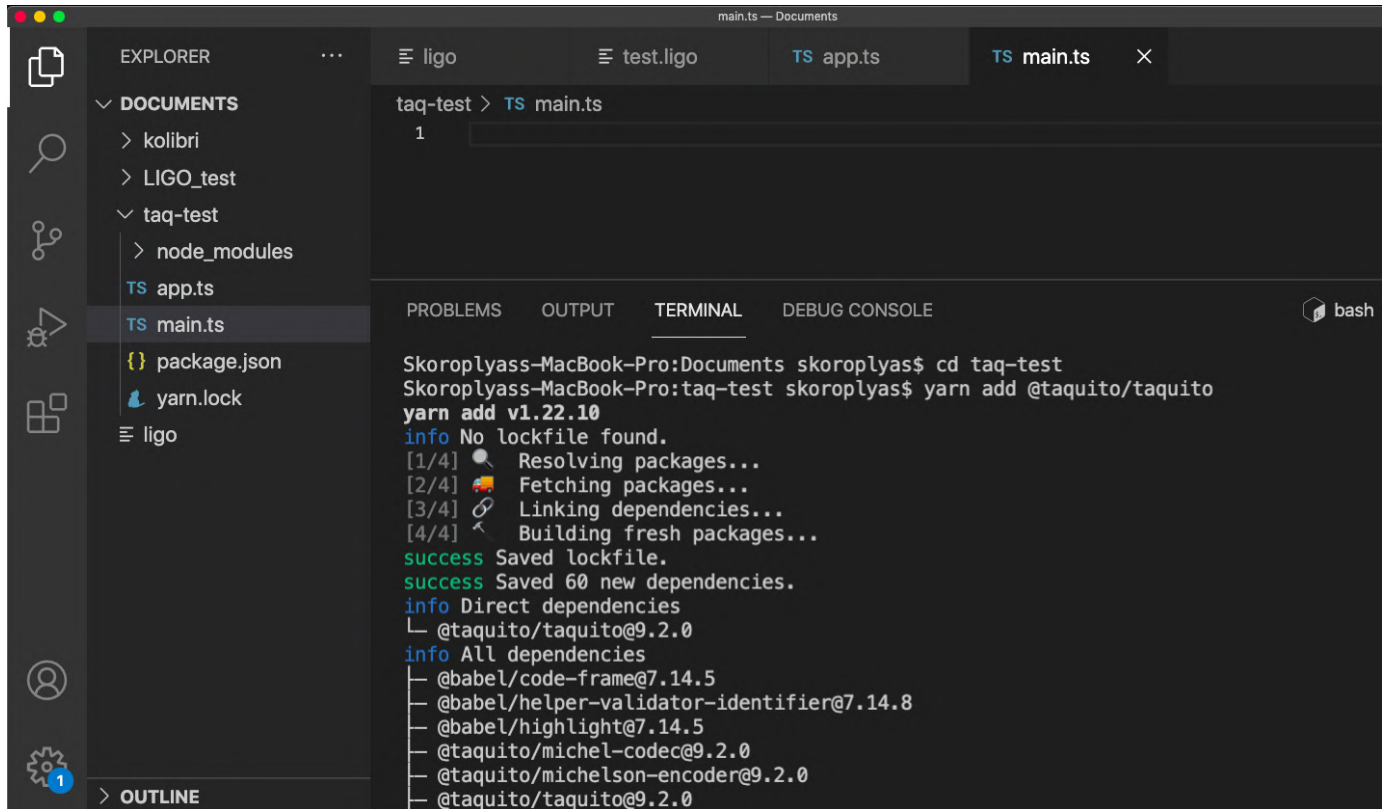


Open the terminal in VS Code and go to `taq-test` with the command `cd`. Then add the Taquito library to the folder:

```

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

```



If NodeJS and yarn were installed correctly, the terminal will display a long list of installed files.

Open `app.ts` and add the following code:

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

  public async main() { }

}
```

Add the following in main.ts:

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

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

Don't forget to save the files by pressing `ctrl+s` or `⌘+s`, otherwise, the execution environment will perceive them as empty.



The extension .ts means that the files contain TypeScript code. You will need the TypeScript library to make it work, so execute the following in the VS Code terminal:

```
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$

```

In case of successful installation, the terminal will show 3 or 4 warnings about licensing and the lack of project description. If there are more warnings, it means you forgot sudo at the start of the command or do it in a folder other than taq-test.

The last step is to execute the script main.ts in the virtual node. To do that, type the following in the terminal:

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

```

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

```




The terminal will report the launch of npx with a single string. It means that npx was successfully imported into the Taquito app. As the main function of app.ts is empty, npx has closed it at once. Any other result than that means that you have made a mistake while installing TypeScript or Taquito.

Please note that in order to work with smart contracts, you will have to install Taquito and TypeScript in the project folder every single time. Unlike JavaScript, they are uninstallable globally.

Getting data from the Tezos blockchain through Taquito

Taquito allows one to read address balances in the Tezos network, send tokens, and publish smart contracts. It doesn't require you to run your own Tezos node: Taquito connects to public nodes via RPC (remote procedure call).

So let's connect to the mainnet and request the balance from a baker's address. To do that, we should activate the TezosToolkit library in app.ts and prepare a constant for an RPC link of a Tezos public node:

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

  // declaring private tezoz modifier of TezosToolkit type
  private tezoz: TezosToolkit;

  // declaring the constructor rpcUrl that will broadcast the Tezos public node
  address to TezosToolkit
  constructor(rpcUrl: string) {
    this.tezoz = new TezosToolkit(rpcUrl);
  }

  public async main() { }

}
```

Write the RPC link to the Tezos Mainnet public node in main.ts and transfer it to the App class:

```
import { App } from './app';
// declaring the constant with the node's address
const RPC_URL = "https://mainnet.smartpy.io"
// launch App that will broadcast the node's address to the main function
new App(RPC_URL).main();
```



When executing `npx ts-node main.ts` Taquito will get a link to a node but won't do anything as `app.ts` contains no methods for interacting with the network. So what we do now is adding a method to get an address balance to `app.ts`:

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

  private tezos: TezosToolkit;

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

  // declaring the method getBalance with input param address
  public getBalance(address: string) : void {

    // Taquito sends a request for balance to the node. If the node executed the
    // request, the script displays the value in the console, otherwise it says "Address
    // not found"
    this.tezos.rpc
      .getBalance(address)
      .then(balance => console.log(balance))
      .catch(e => console.log('Address not found'));
  }

  public async main() { }
}
```

Add the baker's address and call the method `getBalance` in the `main.ts` script:

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

const RPC_URL = "https://mainnet.smartpy.io"
// declare the constant with the Everstake baker's address
const ADDRESS = "tz1aRoarhSpRYvFdyvgWLL6TGyRoGF51wDjM"
// launching App, sending a link to the node, calling getBalance and sending it
// the address
new App(RPC_URL).getBalance(ADDRESS);
```



Launch the script with `npx ts-node main.ts`.

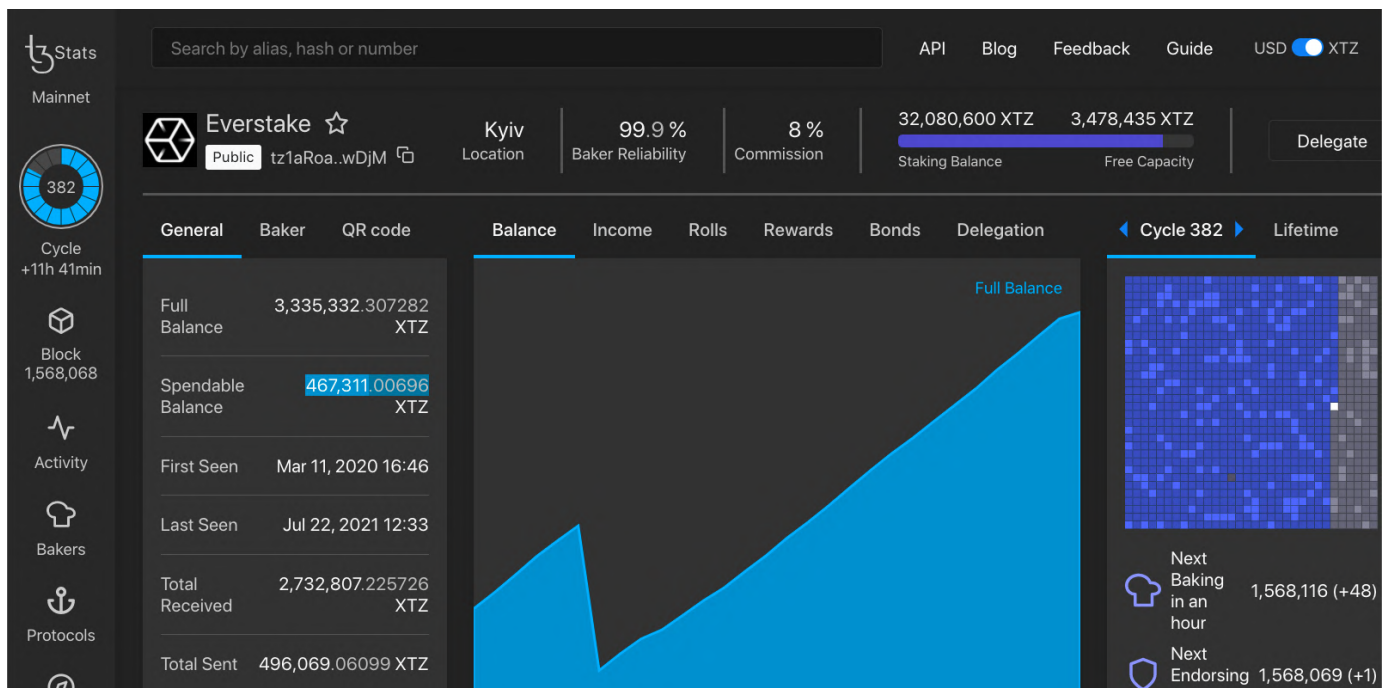
The screenshot shows a VS Code editor with a file explorer on the left, a code editor in the center, and a terminal at the bottom. The file explorer shows a project structure with files like `package-lock.json`, `package.json`, `yarn.lock`, and `main.ts`. The code editor shows the following TypeScript code:

```
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
```

The terminal shows the command `npx ts-node main.ts` being executed, resulting in the output:

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

Taquito will return the number of free tez at Everstake's address.



The balance we get is the same as in [tzStats](https://tzstats.com).



Creating a testnet account in the Tezos network with Taquito


Transactions have to be signed with a secret key. For those purposes, Taquito uses the module signer that works with blockchain account data. To avoid spending actual tez, we connect to the Tezos testnet. First, we get the address and test tez at the faucet [tzalpha](#). Follow the link, convince them you're not a robot, and click **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



The faucet will create a new address and issue account data: a passphrase, a secret key, and a public address.

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": "tz1imn4fjJFwmNaiEWnAGdRrRHxzxRbDkafZ",
  "password": "ZnnZLS0v6O",
  "email": "hoqfgsoy.qyisbhtk@tezos.example.org"
}
```

Copy



Save the account data in a separate file `acc.json` in the `taq-test` folder.

The screenshot shows the VS Code interface with the Explorer sidebar on the left. The 'taq-test' folder is expanded, showing files: `node_modules`, `acc.json`, `app.ts`, `main.ts`, `package-lock.json`, `package.json`, and `yarn.lock`. The `acc.json` file is selected and its content is displayed in the editor:

```

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 }
```

Now we can add the signer module to the project. In VS Code, open the terminal, go to `taq-test` and execute the following:

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

Just like with installing Taquito, the terminal will warn you about missing licenses and display a list of installed files:

The screenshot shows the VS Code interface with the terminal panel at the bottom. The Explorer sidebar shows the same file structure as before. The terminal output is as follows:

```

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...
```



Create a new file `tx.ts` and add signer into it:

```
import { TezosToolkit } from '@taquito/taquito';
// import inMemorySigner. It will save the private key in the memory and use it
// to sign transactions
import { InMemorySigner } from '@taquito/signer';
// declare the constant acc directing the script to acc.json
const acc = require('./acc.json');
export class Tx {
  private tezos: TezosToolkit;
  rpcUrl: string;
  constructor(rpcUrl: string) {
    this.tezos = new TezosToolkit(rpcUrl);
    this.rpcUrl = rpcUrl

    // declare params with the method fromFundraiser: mail, password, and passphrase
    // that allows one to get the private key
    this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
    acc.password, acc.mnemonic.join(' ')))
  }
  // get the public and the private keys and activate the account
  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() { }
}
```

In `main.ts`, call the method `activateAccount()`:

```
import { App } from './app';
// import Tx.ts
import { Tx } from './tx';
// change the RPC link from the mainnet to testnet. Don't you fear smartpy
```




```

in the link: it's merely the server address.
const RPC_URL = "https://florencenet.smartpy.io/"
const ADDRESS = "tz1aRoaRhSpRYvFdyvgWLL6TGyRoGF51wDjM"
// call the function Tx, send it the testnet link, and ask to activate
the account
new Tx(RPC_URL).activateAccount();

```

All that's left is to execute main.ts in the terminal:

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

The screenshot shows the VS Code interface with the Explorer sidebar on the left displaying the project structure. The main editor area shows the code for main.ts, which imports App and Tx from local modules and calls activateAccount on a Tx object with a specific RPC URL and address. The terminal at the bottom shows the command 'npx ts-node main.ts' being executed successfully without errors.

```

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();

```

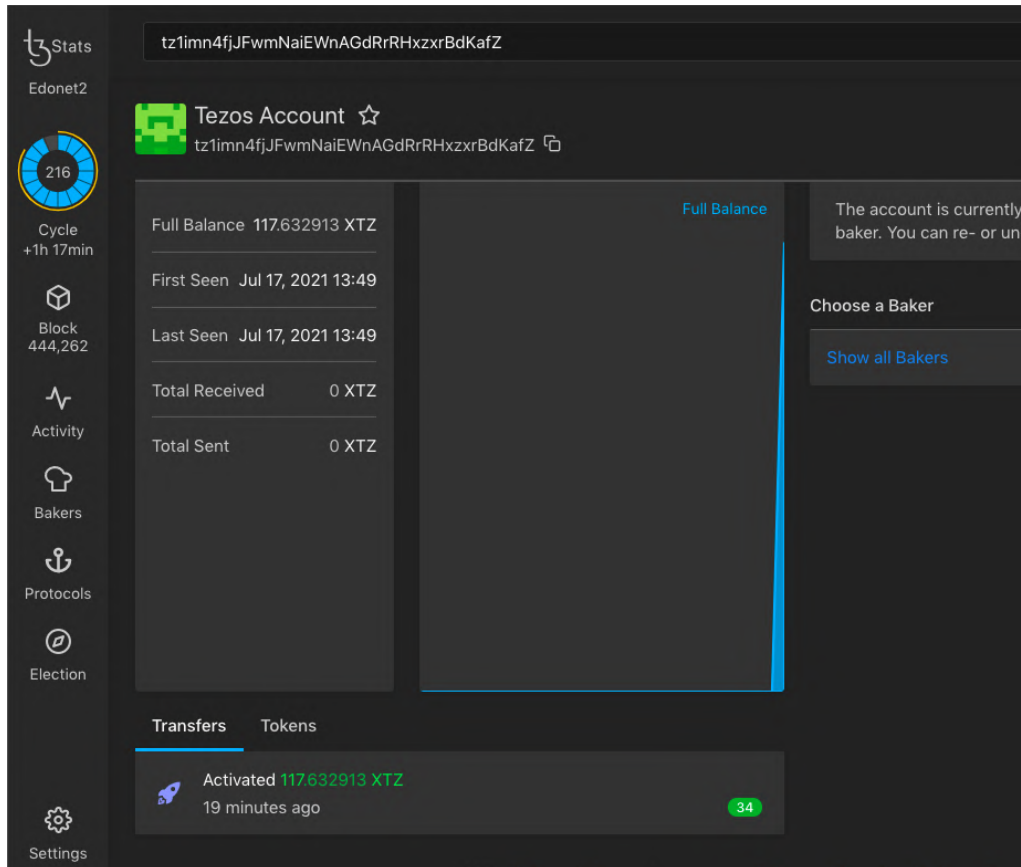
```

Skorplyass-MacBook-Pro:taq-test skorplyas$ npx ts-node main.ts
npx: installed 13 in 3.25s
Skorplyass-MacBook-Pro:taq-test skorplyas$

```

If the testnet has activated the account, the console will report the launch of npx without errors. Now you can confirm the activation in the testnet browser.

Open acc.json and copy the public key from "pkh." Go to florence.tzstats and find your account using the key. The balance should have nearly a hundred test tez, and a record on activation in the history of transactions.



Now we have set up Taquito and activated the test account. To work in the mainnet, do the same: create a JSON file with account details and connect it using signer.

Publish the contract on the testnet

Last time, we launched a smart contract using Docker and dry-run. VM with the contract closed at once and didn't use storage.

Actual smart contracts for tokens and DApps use storage to store user data all the time. We'll cover that in greater detail next time. Now we should focus on publishing the contract in the testnet to see how storage saves data.

Go to VS Code and create `deploy.js`. Paste the following code in the file:

```
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,
  "", // mail
  "", // password
  [
    "", // passphrase
  ].join(' '),
  "" // private key
);

deploy();
```

Input the account data in the empty spaces: mail, password, passphrase, and private key found in acc.json.

```
TS deploy.ts x
TS deploy.ts > ...
1 import { TezosToolkit } from '@taquito/taquito';
2 import { importKey } from '@taquito/signer';
3
4 const provider = 'https://florencenet.smartpy.io/';
5
6 async function deploy() {
7   const tezos = new TezosToolkit(provider);
8   await importKey(
9     tezos,
10    "hoqfgsoy.qyisbhtk@tezos.example.org", //email
11    "ZnnZLS0v60", //password
12    [
13      "able", //mnemonics
14      "public",
15      "usual",
16      "hello",
17      "october",
18      "owner",
19      "essence",
20      "old",
21      "author",
22      "original",
23      "various",
24      "gossip",
25      "core",
26      "high",
27      "hire"
28    ].join(' '),
29    "2bed8dc244ee43a1e737096c4723263c269049d8" //secret key
30  );
```

After the account details, add the contract code. We will use a modified version from the one we saw in the first lesson:

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



This smart contract receives an int-type param num from the user and sums it up with the number kept in storage.

Compile the contract code in Michelson. To avoid launching Docker yet another time, use the online development environment.

Paste the code in the editor, select **Compile Contract** in the drop-down list, and click **Run**. The compiler will produce the contract code in Michelson.

LIGO Install Docs Tutorials Blog Ask Questions Cheat Sheet

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) PascalLIGO

```

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

```

[View in Try-Michelson IDE](#)

```

{ parameter int ;
  storage int ;
  code { UNPAIR ; ADD ; NIL operation ; PAIR } }

```

Configure

Compile Contract Run

Access function
The function name from where your contract will start

main

☐ Output michelson in JSON format

Return to `deploy.ts` and paste the following after the account details:

```

try {
  const op = await tezos.contract.originate({
    // smart contract code
    code: `{ parameter int ;
      storage int ;
      code { UNPAIR ; ADD ; NIL operation ; PAIR } }`,
    // storage state
    init: `0`,
  });
}

```



Now describe the contract deployment process. The final code will look like this:

```
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", // mail
    "ZnnZLS0v60", // password
    [
      "able", // passphrase
      "public",
      "usual",
      "hello",
      "october",
      "owner",
      "essence",
      "old",
      "author",
      "original",
      "various",
      "gossip",
      "core",
      "high",
      "hire"
    ].join(' '),
    "2bed8dc244ee43a1e737096c4723263c269049d8" // private key
  );

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



```

    // storage state
    init: `0`,
  });

  // beginning to deploy
  console.log('Awaiting confirmation...');
  const contract = await op.contract();
  // deployment report: amount of used gas, storage state
  console.log('Gas Used', op.consumedGas);
  console.log('Storage', await contract.storage());
  // operation hash one can use to find the contract in the explorer
  console.log('Operation hash:', op.hash);
} catch (ex) {
  console.error(ex);
}
}

deploy();

```

Open the console and execute `npx ts-node deploy.ts`. The terminal will produce a report on contract publishing.

```

TS deploy.ts x
TS deploy.ts > ...
42  });
43
44  //beginning to deploy
45  console.log('Awaiting confirmation...');
46  const contract = await op.contract();
47  //deployment report: amount of used gas, storage state
48  console.log('Gas Used', op.consumedGas);
49  console.log('Storage', await contract.storage());
50  //operation hash one can use to find the contract in the explorer
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 4.71s
Awaiting confirmation...
Gas Used 1577
Storage BigNumber { s: 1, e: 0, c: [ 0 ] }
Operation hash: ooHxYT2hXttYHPFNyHUxa5AFZgf9ndVbKti3RxYS3uFehjEv4sx
skoroplyas@MacBook-Air taq-test %

```




Copy the hash and find the contract deployment transaction in [florence.tzstats](https://florence.tzstats.com). It will specify the smart contract's address on the testnet.

Now try to call the published contract. To do that, create a script that would call the main function and send it a number. Create a new file called `call.ts` and paste in it the following code:

```
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

    // declaring the parameters using fromFundraiser: mail, password, and
    // passphrase from which one can extract the private key
    this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
    acc.password, acc.mnemonic.join(' ')))
  }

  public add(add: number, contract: string) {
    this.tezos.contract
      .at(contract) // call the contract to get its entry points
      .then((contract) => {
        console.log(`Adding ${add} to storage...`);
        // calling the main function. Unlike ligo syntax, the main entry
        // point is referred to as 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); // waiting for 1
        // confirmation to get the results faster
      })
      .then((hash) => console.log(`Call done`)) // call is successful
  }
}
```



```

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

```

Now make up a separate script to call `call.ts`, send contract address, and argument value. To that end, create a file called `maincall.ts` and paste the following code in it:

```

import { Call } from './call';

const RPC_URL = "https://florencenet.smartpy.io/"
const CONTRACT = "" // published contract address
const ADD = 5 // number to be received by the main function. You can change it
new Call(RPC_URL).add(ADD, CONTRACT);

```

Assign the address of the published test contract to the constant `CONTRACT`. Open the console and execute the following command:

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

When the terminal finishes the operation, follow the link or find your contract by the address. As you see, the storage state has changed.

The screenshot shows the Tezos Contract interface for a contract with address `KT1Qc2QyRgRzfczWH3Sx4aw1JzC7DMNBrihC`. The interface includes a sidebar with contract statistics and a main area with tabs for Calls, Entrypoints, Contract Code, and Storage. The Storage tab is active, showing a table with one entry.

#	Key & Type	Content
1	storage: int	130



Some conclusions

Taquito is a tool for working with the Tezos blockchain using JavaScript and TypeScript. With its help, developers can call smart contract entry points and the Tezos network similar to regular JS methods. Taquito also allows one to store data from several accounts in different files and place frequently used code in separate modules.

The Taquito library is required for the development of DApp on Tezos. Developers will find it easier to integrate JavaScript in a web application rather than to translate user requests into console commands of the Tezos client.

We will keep using Taquito to create fungible and non-fungible tokens (NFT) on Tezos.

Creating an FA 1.2 token

In our past lessons, we studied the syntax of LIGO and published a simple contract on the Tezos testnet. This time, we'll delve into the mechanisms behind digital assets, take a closer look at the standard FA 1.2, and release a token on the testnet.



Token standards

Let's settle the matter of terms and definitions first. The word token refers to two kinds of units: native tokens and smart-contract issued tokens. The logic behind the former is described in the protocol code, and the latter, in the code of smart contracts (SC).

Native tokens are also known as cryptocurrencies. They include Bitcoin, Ether, Tezos, Litecoin, and so forth.

Smart-contract tokens are also known as digital assets created by blockchain users. These are USDT or DOGE based on Ethereum, kUSD based on Tezos. For the purposes of our lessons, referring to tokens we'll mean smart-contract tokens.

So far, we know the token logic is not described in their protocol. Devs create it in the smart contract code: they add basic functions like balance storage or transfer. Thus, 100 kUSD on the user's wallet is a record in the SC's storage saying something like "address TZ1 can send 100 tokens of this contract." In other words, tokens are ledgers within the blockchain.

For the developers to release operational and attack-resilient SCs, the community approves token standards. They include lists of functions required for the smart contract to interact with wallets or blockchain apps. Devs can also write additional functions into the contracts, such as token burning. Anyway, the main standard functions are mandatory.

Main functions of the FA 1.2 standard

Tezos standards are called Financial Applications, hence FA. The most popular of them are the following two:

- **FA 1.2** for fungible tokens;
- **FA 2** for both fungible and non-fungible tokens (NFTs).

This time we'll take a look at the simpler standard FA1.2, and cover FA2 next time.

The FA 1.2 standard includes the following functions:

- **transfer**: transacting between users;
- **approve**: allowing Alice to send tokens from Bob's address. This allows the contract to send transaction requests to users;
- **allowance (getAllowance)**: browsing the amount of tokens available for Alice to sending from the contract's address;
- **user balance (getBalance)**: browsing user balance;
- **current supply (getTotalSupply)**: browsing the number of tokens on all user addresses.

Aside from that, a token contract has to keep data on the total supply and user balances in storage, and messages on standard errors like insufficient funds in the code. Also, a developer can incorporate metadata with the contract's and the token's names as well as a link to the logo in the storage.



What is FA 1.2 implementation?

Implementation is a working product based on an idea. In the case of FA1.2, it's a template smart contract for operations with tokens.

A developer can copy the implementation, change it for his or her current purposes, publish it on the blockchain, and thus get a working token. An FA1.2 implementation includes nearly a hundred strings. The code can be conventionally divided into eight parts:

- aliases, data types, and pseudo-entry points;
- function *getAccount*, which gets user data;
- function *getAllowance*, which gets the number of tokens available for transacting;
- function *transfer* for transferring tokens between addresses;
- function *approve*, which confirms the user's right to transfer tokens;
- function *getBalance*, which returns the user's balance;
- function *getTotalSupply*, which returns the number of free tokens;
- function *main*, which accepts entry parameters and sends them to one of the previous functions.

Below we'll cover each of those parts in greater detail.

Declaring aliases and types

First of all, you have to declare all types and aliases the smart contract will use in functions:

```
// declaring the alias trusted of address type. We will use it for addresses
entitled to send tokens
type trusted is address;

// declaring the alias amt (amount) of nat type to store balances
type amt is nat;

(* announcing the alias account of record type. It will store data on users
entitled to receive tokens *)
type account is
  record [
    balance      : amt;
    allowances    : map (trusted, amt);
  ]

(* declaring the type of SC storage. It keeps the overall quantity of tokens and
big_map data structure that ensures the connection between user balances and
public addresses *)
```




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

(* declaring the alias for the return method that will return operations. In
short contracts one can do without it, yet it is easier to describe the return
type just once in contracts with several pseudo-entry points and then use it in
every function *)
type return is list (operation) * storage

(* declare the empty list noOperations. It will be returned by transfer and
approve *)
const noOperations : list (operation) = nil;
```

The Michelson compiler automatically sorts the contents of complex data in alphabetical order but sorting can accidentally break the contract. Thus, if the first param (sender's address) starts with TZ19, and the second (recipient's address) starts with TZ11, the compiler will switch their places and the contract will attempt to send tokens from the wrong contract.

To retain the correct order of params, devs record important data types in `michelson_pair`. The structure includes two arguments and their names (e.g., two addresses): sender and receiver. The compiler transfers the `michelson_pair` values to the Michelson code without sorting.



```
(* declaring aliases of input params for each main function of FA 1.2. *)

// transfer function gets the sender's address, the recipient's address, and the
// transaction amount to the input
type transferParams is michelson_pair(address, "from", michelson_pair(address,
"to", amt, "value"), "")
// approve gets user address and number of tokens they can send from the SC's
// balance
type approveParams is michelson_pair(trusted, "spender", amt, "value")
// getBalance gets the addresses of the user and the proxy contract where it
// sends balance data
type balanceParams is michelson_pair(address, "owner", contract(amt), "")
// getAllowance gets the user's address, their SC account data abd the proxy
// contract
type allowanceParams is michelson_pair(michelson_pair(address, "owner", trusted,
"spender"), "", contract(amt), "")
// totalSupply doesn't use michelson_pair as the first input param is the empty
// value of unit will be the first anyway after being sorted by Michelson compiler
Michelson
type totalSupplyParams is (unit * contract(amt))

(* declaring pseudo-entry points: give them a name and assign the type of params
described above *)
type entryAction is
  | Transfer of transferParams
  | Approve of approveParams
  | GetBalance of balanceParams
  | GetAllowance of allowanceParams
  | GetTotalSupply of totalSupplyParams
```

The function `getAccount` gets the input parameter of address type and the value of storage from the smart contract:

```
function getAccount (const addr : address; const s : storage) : account is
  block {
    // allowances assigning the variable acct the value of account type: zero
    // balance and empty entry
    var acct : account :=
      record [
```



```

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

```

(* checking if the storage has the user account: if no, leave acct empty with the previous block's value; if yes, assign the value from the storage to acct. The function returns the acct value *)

```

case s.ledger[addr] of
  None -> skip
| Some(instance) -> acct := instance
end;
} with acct

```

The function *getAllowance* asks the user how many tokens they allow it to transfer to another address. It gets the user's address, the contract address (spender), and the storage state, and returns the argument **amt**, which is the number of tokens subject to spending.

```

function getAllowance (const ownerAccount : account; const spender : address;
const s : storage) : amt is
  (* if the user has allowed to send a certain amount of tokens, the function
  assigns the amount to amt. If not, the number of tokens equals zero *)
  case ownerAccount.allowances[spender] of
    Some (amt) -> amt
  | None -> 0n
  end;

```

The function *transfer* gets sender and recipient addresses from the user along with the number of tokens subject to sending and the storage state:

```

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

    (* call getAccount to assign user account data to senderAccount. Then we use
    senderAccount to read the user's balance and permissions *)
    var senderAccount : account := getAccount(from_, s);

    (* checking whether the user has sufficient funds. If not, the VM terminates the
    contract execution, if yes, it carries on with executing the contract *)

```



```

if senderAccount.balance < value then
  failwith("NotEnoughBalance")
else skip;

(* checking if the initiating address can send tokens. If it requests a
transfer from someone else's address, the function asks the real owner for
permission. If the initiator and the sender are the same address, the VM carries
on with executing the contract *)
if from_ /= Tezos.sender then block {
  (* calling the function getAllowance so that the owner would specify how many
tokens they allow to be sent, and assign the value to the constant
spenderAllowance *)
  const spenderAllowance : amt = getAllowance(senderAccount, Tezos.sender,s);

  (* if the owner has allowed sending less tokens than specified in the input
param, the VM will terminate the contract execution *)
  if spenderAllowance < value then
    failwith("NotEnoughAllowance")
  else skip;

  (* subtracting the transaction amount from the allowed amount *)
  senderAccount.allowances[Tezos.sender] := abs(spenderAllowance - value);
} else skip;

(* subtracting the sent tokens from the balance of the sender address *)
senderAccount.balance := abs(senderAccount.balance - value);

(* updating the balance record in the sender's storage *)
s.ledger[from_] := senderAccount;

(* once again call getAccount to get or create an account record for the
recipient address *)
var destAccount : account := getAccount(to_, s);

(* adding the amount of sent tokens to the recipient's balance *)
destAccount.balance := destAccount.balance + value;

(* updating the balance record in the recipient's storage *)
s.ledger[to_] := destAccount;
}

```



```
// returning the empty list of operations and the storage state after the
function's execution
with (noOperations, s)
```

The function `approve` requests the confirmation of the number of tokens the spender address can send from the sender address. For example, a blockchain app (spender) requests the user (sender) for permission to send tokens.

This function can be subject to a double-spending attack. For example, the sender reduces the number of tokens allowed for spending from 20 to 10. The spender somehow finds out about it and creates a transaction that spends 20 tokens. They pay an increased fee to make the transaction get into the block before the transaction with the change of permissions. Then the spender gets 20 and waits until the permission changes. After that, they create yet another transaction, this time to send 10 tokens. The function allows them to send it. As a result, the spender sends 30 tokens instead of 10 from the sender address.

To avoid this, developers implement a delay in permission change in `approve`. If the allowed quantity of tokens is more than null, it can be only changed to null, and if it equals zero in the first place, it can be changed into a natural number. In this case, the spender address cannot spend tokens twice.

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

    (* getting the user account data *)
    var senderAccount : account := getAccount(Tezos.sender, s);

    (* getting the current amount of tokens the user opted to sen *)
    const spenderAllowance : amt = getAllowance(senderAccount, spender, s);

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

    (* introducing the number of tokens newly permitted for spending in the
    account data *)
    senderAccount.allowances[spender] := value;

    (* updating the SC storage *)
    s.ledger[Tezos.sender] := senderAccount;

  } with (noOperations, s)
```



The functions *getBalance*, *getAllowance* and *getTotalSupply* are so-called view functions. They return the requested number not to the sender but to a proxy contract. The latter allows apps to get data from user contracts and show them in the interface.

The function *getBalance* returns the balance of the specified address:

```
function getBalance (const owner : address; const contr : contract(amt); var s :
storage) : return is
  block {
    // assigning account data to the constant ownerAccount
    const ownerAccount : account = getAccount(owner, s);
  }
  // returning the account balance to the proxy contract
  with (list [transaction(ownerAccount.balance, 0tz, contr)], s)
```

The function *getAllowance* returns the amount of tokens allowed to be spent by the account in question:

```
function getAllowance (const owner : address; const spender : address; const
contr : contract(amt); var s : storage) : return is
  block {
    // getting account data and retrieve the number of tokens allowed for
    spending therefrom
    const ownerAccount : account = getAccount(owner, s);
    const spenderAllowance : amt = getAllowance(ownerAccount, spender, s);
  } with (list [transaction(spenderAllowance, 0tz, contr)], s)
```

The function *getTotalSupply* returns the number of tokens on all users' balances:

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



The main function assumes the name of the pseudo-entry point and its params:

```
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;
```

Preparing the token's smart contract for publication

To save time, we'll use the `taq-test` folder from the previous lesson. Launch the VS Code editor. Create the folder `Token` in `taq-test`, and `token.ligo` within the former. Copy the token code below.

```
// declaring the alias trusted of address type. We will use it for addresses
entitled to send tokens
type trusted is address;

// declaring the alias amt (amount) of nat type to store balances
type amt is nat;

(* announcing the alias account of record type. It will store data on users
entitled to receive tokens
*)
type account is
  record [
    balance          : amt;
    allowances       : map (trusted, amt);
  ]

(* declaring the type of SC storage. It keeps the overall quantity of tokens and
big_map data structure that ensures the connection between user balances and
public addresses *)
type storage is
  record [
    totalSupply      : amt;
```




```

    ledger          : big_map (address, account);
]

(* declaring the alias for the return method that will return operations. In
short contracts one can do without it, yet it is easier to describe the return
type just once in contracts with several pseudo-entry points and then use it in
every function *)
type return is list (operation) * storage

(* declare the empty list noOperations. It will be returned by transfer and
approve *)
const noOperations : list (operation) = nil;
(* declaring aliases of input params for each main function of FA 1.2. *)

type transferParams is michelson_pair(address, "from", michelson_pair(address,
"to", amt, "value"), "")
type approveParams is michelson_pair(trusted, "spender", amt, "value")
type balanceParams is michelson_pair(address, "owner", contract(amt), "")
type allowanceParams is michelson_pair(michelson_pair(address, "owner", trusted,
"spender"), "", contract(amt), "")
type totalSupplyParams is (unit * contract(amt))

(* declaring pseudo-entry points: give them a name and assign the type of params
described above *)
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 {
    // allowances assigning the variable acct the value of account type: zero
    balance and empty entry
    var acct : account :=
        record [
            balance      = 0n;
            allowances = (map [] : map (address, amt));
        ];

```



```

    (* checking if the storage has the user account: if no, leave acct empty with
    the previous block's value; if yes, assign the value from the storage to acct.
    The function returns the acct value *)
    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
  (* if the user has allowed to send a certain amount of tokens, the function
  assigns the amount to amt. If not, the number of tokens equals zero *)
  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 {

    (* call getAccount to assign user account data to senderAccount. Then we use
    senderAccount to read the user's balance and permissions *)
    var senderAccount : account := getAccount(from_, s);

    (* checking whether the user has sufficient funds. If not, the VM terminates
    the contract execution, if yes, it carries on with executing the contract *)
    if senderAccount.balance < value then
      failwith("NotEnoughBalance")
    else skip;

    (* checking if the initiating address can send tokens. If it requests a
    transfer from someone else's address, the function asks the real owner for
    permission. If the initiator and the sender are the same address, the VM carries
    on with executing the contract *)
    if from_ /= Tezos.sender then block {
      (* calling the function getAllowance so that the owner would specify how many
      tokens they allow to be sent, and assign the value to the constant
      spenderAllowance *)

```



```

    const spenderAllowance : amt = getAllowance(senderAccount, Tezos.sender, s);
    (* if the owner has allowed sending less tokens than specified in the input
    param, the VM will terminate the contract execution *)
    if spenderAllowance < value then
        failwith("NotEnoughAllowance")
    else skip;

    (* subtracting the transaction amount from the allowed amount *)
    senderAccount.allowances[Tezos.sender] := abs(spenderAllowance - value);
} else skip;

(* subtracting the sent tokens from the balance of the sender address *)
senderAccount.balance := abs(senderAccount.balance - value);

(* updating the balance record in the sender's storage *)
s.ledger[from_] := senderAccount;

(* once again call getAccount to get or create an account record for the
recipient address *)
var destAccount : account := getAccount(to_, s);

(* adding the amount of sent tokens to the recipient's balance *)
destAccount.balance := destAccount.balance + value;

(* updating the balance record in the recipient's storage *)
s.ledger[to_] := destAccount;

}
// returning the empty list of operations and the storage state after the
function's execution
with (noOperations, s)
function approve (const spender : address; const value : amt; var s : storage) :
return is
block {

    (* getting the user account data *)
    var senderAccount : account := getAccount(Tezos.sender, s);

    (* getting the current amount of tokens the user opted to send *)
    const spenderAllowance : amt = getAllowance(senderAccount, spender, s);

```



```

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

    (* introducing the number of tokens newly permitted for spending in the
    account data *)
    senderAccount.allowances[spender] := value;

    (* updating the SC storage *)
    s.ledger[Tezos.sender] := senderAccount;

  } with (noOperations, s)

function getBalance (const owner : address; const contr : contract(amt); var s :
storage) : return is
  block {
    // assigning account data to the constant ownerAccount
    const ownerAccount : account = getAccount(owner, s);
  }
  // returning the account balance to the proxy contract
  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 {
    // getting account data and retrieve the number of tokens allowed for
    spending therefrom
    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)

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;

```

The LIGO code has to be compiled in Michelson for the Tezos VM to execute it. Last time, we pasted a two-string contract code directly in the deployment script. This time, the code contains 200 strings so it would be wiser to store it in a separate file and import it in the script through the import command. Open the online environment LIGO and paste the token code in the editor. In the drop-down list choose Compile Contract and put a flag in Output Michelson in JSON format. Then click Run. The compiler will produce a ready-made code underneath the editor field.

The screenshot shows the LIGO online environment. On the left, there's a 'Contract Examples' list. The main editor displays the LIGO code for an 'Increment' contract. The code is as follows:

```

179
180 (* The main function assumes the name of the pseudo-entry point and its params:*)
181 function main (const action : entryAction; var s : storage) : return is
182 block {
183 skip
184 } with case action of
185 | Transfer(params) -> transfer(params.0, params.1.0, params.1.1, s)
186 | Approve(params) -> approve(params.0, params.1, s)
187 | GetBalance(params) -> getBalance(params.0, params.1, s)
188 | GetAllowance(params) -> getAllowance(params.0.0, params.0.1, params.1, s)
189 | GetTotalSupply(params) -> getTotalSupply(params.1, s)
190 end;
191
192
193

```

Below the editor, the compiled Michelson code is shown in JSON format:

```

[ { "prim": "parameter",
  "args":
    [ { "prim": "or",
      "args":
        [ { "prim": "or",
          "args":
            [ { "prim": "or",
              "args":
                [ { "prim": "pair",
                  "args":
                    [ { "prim": "address",
                      "annots": [ "%spender" ] },
                    { "prim": "nat", "annots": [ "%value" ] } ],
                  "annots": [ "%approve" ] },
                f "main" "main"
            ]
          ]
        ]
      ]
    ]
  ]
}

```

On the right, the 'Configure' section shows 'Compile Contract' selected and 'Run' button. The 'Access function' dropdown is set to 'main'. The checkbox 'Output michelson in JSON format' is checked.



In the token folder, create token.json and paste the JSON code therein.

```

1  [ { "prim": "parameter",
2      "args":
3      [ { "prim": "or",
4          "args":
5          [ { "prim": "or",
6              "args":
7              [ { "prim": "or",
8                  "args":
9                  [ { "prim": "pair",
10                     "args":
11                     [ { "prim": "address",
12                         "annots": [ "%spender" ] },
13                         { "prim": "nat", "annots": [ "%value" ] } ],
14                     "annots": [ "%approve" ] },
15                     { "prim": "pair",
16                         "args":
17                         [ { "prim": "pair",
18                             "args":
19                             [ { "prim": "address",
20                                 "annots": [ "%owner" ] },
21                                 { "prim": "address",
22                                     "annots": [ "%spender" ] } ] },
23                             { "prim": "contract",
24                                 "args": [ { "prim": "nat" } ] },
25                             "annots": [ "%getAllowance" ] } ] },
26                     { "prim": "or",
27                         "args":
28                         [ { "prim": "pair",

```

Last time, we put the test account data into the deploy.ts script and set up an RPC link of the public node on the Tezos testnet. We can use this code to publish our token.

Create the file token-deploy.tx in taq-test and paste the code from deploy.ts therein. Then we have to update it: add the method file sync to read other files, import the contract code, and specify the original storage state.

After import methods, add the method file sync and the Tezos constant to call Taquito methods:

```

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

```

Replace the method `try`. After that, set up the initial storage state: the overall number of tokens and user balance:

```

try {
  const op = await tezos.contract.originate({
    // reading code from token.json
    code:

```



// setting the storage state in Michelson. Replace both addresses with your account address on the testnet, and numbers with the number of tokens you want to issue.

```
init:
  '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzrBdKafZ" (Pair { Elt
"tz1imn4fjJFwmNaiEWnAGdRrRHxzrBdKafZ" 1000 } 1000) } 1000)',
  })
```

The final code looks like this:

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

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

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

async function deploy() {
  const tezos = new TezosToolkit(provider);
  await importKey(
    tezos,
    "hoqfgsoy.qyisbhtk@tezos.example.org", // mail
    "ZnnZLS0v60", // password
    [
      "able", // passphrase
      "public",
      "usual",
      "hello",
      "october",
      "owner",
      "essence",
      "old",
      "author",
      "original",
      "various",
      "gossip",
      "core",
      "high",
      "hire"
    ]
  );
}
```




```

    ].join(' '),
    "2bed8dc244ee43a1e737096c4723263c269049d8" // private key
  );

  try {
    const op = await tezos.contract.originate({
      // reading code from token.json
      code: JSON.parse(fs.readFileSync("./token.json").toString()),
      // setting the storage state in Michelson. Replace both addresses with
      // your account address on the testnet, and numbers with the number of tokens you
      // want to issue.
      init:
        '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzrBdKafZ" (Pair { Elt
"tz1imn4fjJFwmNaiEWnAGdRrRHxzrBdKafZ" 1000 } 1000) } 1000)',
      })

    // deployment commences
    console.log('Awaiting confirmation...');
    const contract = await op.contract();
    // deployment report: amount of used gas, storage state
    console.log('Gas Used', op.consumedGas);
    console.log('Storage', await contract.storage());
    // operation hash one can use to find the contract in the blockchain explorer
    console.log('Operation hash:', op.hash);
  } catch (ex) {
    console.error(ex);
  }
}

deploy();

```

Open the terminal and execute `npx ts-node token-deploy.ts`. In a few minutes, Taquito will publish the token contract on the testnet and produce the operation's hash.

Find it in [florence.tzstats](https://florence.tzstats.com) and check the contract storage state. We have 1,000 tokens written therein, you will have the number you have specified.



Stats
Edonet2

236
Cycle
+5h 56min

Block
484,664

Activity

Bakers

Protocols

Election

Search by alias, hash or number

Tezos Contract ☆
KT1DUdLarkFG9tmA4uZCgbiHv5SJA9oUBw8G

General Contract QR code **Calls** Gas Used

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

Calls Entrypoints Contract Code **Storage** Bigmap #129,708

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

In the Bigmap field, find the records on token owner addresses. When publishing the contract, we sent all tokens to the test address.

Stats
Edonet2

236
Cycle
+5h 52min

Block
484,673

Activity

Bakers

Protocols

Election

Search by alias, hash or number

Tezos Contract ☆
KT1DUdLarkFG9tmA4uZCgbiHv5SJA9oUBw8G

General Contract QR code **Calls** Gas Used

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

Calls Entrypoints Contract Code **Storage** Bigmap #129,708

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



Sending tokens with Taquito

Remember how we added a number to the contract storage using Taquito? Now we do something more difficult: call the function transfer and send tokens to another user.

Install a Tezos wallet and create an account (in case you didn't do it last time). We recommend [Temple Wallet](#) as it supports Tezos testnets out of the box.

Open VS Code and create the file token-transfer.ts. Paste the code below therein:

```
// importing the methods of Taquito and the file with the test account data
acc.json

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

    // setting up the link to the testnet's public node
    private tezos: TezosToolkit;
    rpcUrl: string;

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

        // reading the mail, password, and the passphrase that can produce the private
        key
        this.tezos.setSignerProvider(InMemorySigner.fromFundraiser(acc.email,
        acc.password, acc.mnemonic.join(' ')))
    }

    // declaring the method transfer that has the following params:
    //
    // 1) contract: contract address;
    // 2) sender: sender address;
    // 3) receiver: recipient address;
    // 4) amount: amount of tokens to be sent.

    public transfer(contract: string, sender: string, receiver: string, amount:
    number) {
        this.tezos.contract
            .at(contract) // calling the contract at the address
    }
}
```



```

        .then((contract) => {
            console.log(`Sending ${amount} from ${sender} to ${receiver}...`);
            // calling the entry point transfer, send the reciever/sender
            addresses and the amount of tokens to be sent to it.
            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); // waiting for 1
            network confirmation
        })
        .then((hash) => console.log(`Hash: https://florence.tzstats.com/
        ${hash}`)) // getting the operation's hash
        .catch((error) => console.log(`Error: ${JSON.stringify(error, null,
        2)}`));

    }

}

```

Now create a file transfer.ts and paste the following code therein:

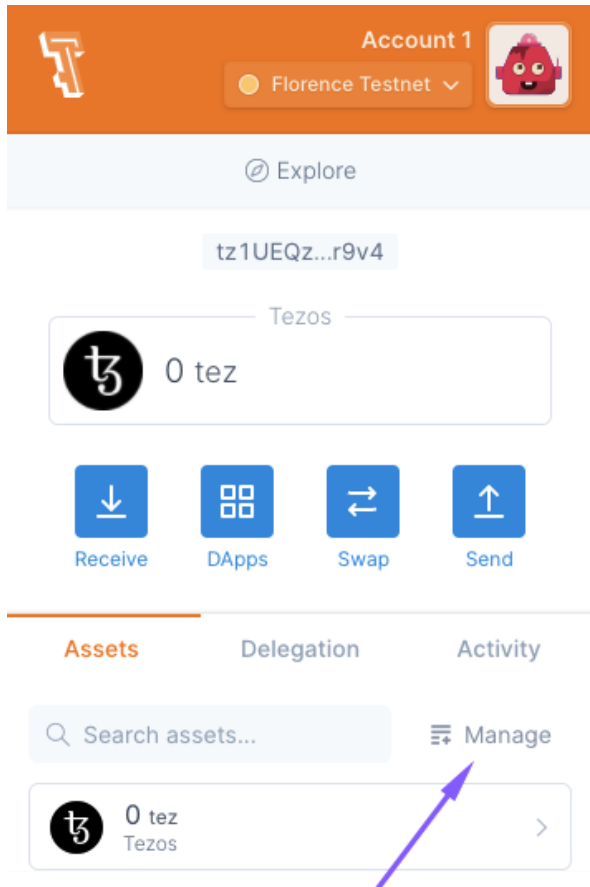
```

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

const RPC_URL = "https://florencenet.api.tez.ie"
const CONTRACT = "KT1DUdLarKFG9tmA4uZCgbiHv5SJA9oUBw8G" // address of the
published contract
const SENDER = "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" // public address of the
sender (find it in acc.json)
const RECEIVER = "tz1UEQzJbuaGJgwk6HwGwaKvjZ7rr9v4" // recipient's public
address (take it from the Tezos wallet you had created)
const AMOUNT = 3 // number of tokens to be sent, you can put another value here
new token_transfer(RPC_URL).transfer(CONTRACT, SENDER, RECEIVER, AMOUNT);

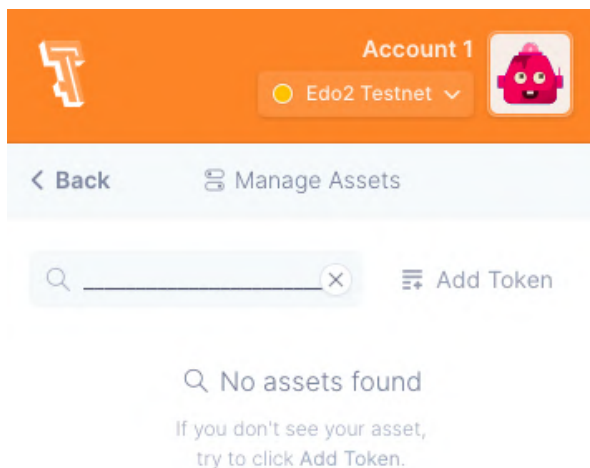
```

Open the console and execute `npx ts-node transfer.ts`. Wait for it to return the operation's hash. Then open Temple Wallet, press Tezos Mainnet on the upper side of the window and select Florence Testnet from the drop-down menu.



The wallet looks empty but the tokens are in fact there. Temple doesn't see them because the contract code has no metadata with its name and other params. We will cover metadata next time. Right now, however, we can add them manually and see tokens in the wallet.

Copy the token contract address from transfer.ts. Open Temple Wallet and press Manage on the right-hand side of Search Assets and then press Add Token.



Temple will open the browser with a tab where you can manually add the token. Choose the FA 1.2 standard in the drop-down menu Token type and then paste the smart contract address in the Address field.



Temple Account 1 Florence Testnet

< Back + Add Token

Token type

FA 1.2
FA 2

Address
Address of a deployed token contract.

KT10uvjm34tv34uncq09c23

Fill in the token data: its ticker, description, number of digits after the decimal point, and the link to the logo. Write whatever you wish, however, keep 0 (zero) in the Decimals field. Press Add Token to add the token with the preset parameters to the wallet.

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

Good job! Now Temple Wallet shows your token. It used the entry points of FA 1.2 so you can browse the balance and send tokens from the wallet's interface. That said, the recipient will still have to manually set up the metadata to see the transfer.

Assets Delegation Activity

Search assets... Manage

	0 tez Tezos	>
	69 TST Simple test token	>

Next time we'll deal with metadata and publish a full-fledged token that wallets will see straight away. And, what's more, we'll also issue an FA2 NFT.



Some conclusions

Tokens are records in the storage of smart contracts saying things like: “the address TZ1 owns 1,000 coins.” The tokens are transferred in the following manner:

1. User/app calls the smart contract.
2. The smart contract checks whether the user is able to send tokens.
3. The smart contract updates user balance records: reduces the sender’s balance by the amount of the transfer, and then adds it to the recipient’s balance.

Developers can write smart contracts from the scratch or use implementations, i.e. templates with basic functions. Advanced mechanisms like issuance or destruction of tokens have to be created on an individual basis.

Token Metadata and How to Issue NFT on Tezos

Last time, we talked about FA1.2 and published a token smart contract on a Tezos testnet. We, however, omitted metadata, so the token didn't show in the wallet automatically. This time, we'll be adding metadata to the smart contract so that wallets and blockchain explorers could see it. Then we'll study the NFT standard in Tezos and create a non-fungible token.



How apps see tokens

Tokens are not coins or separate files but records in the storage of smart contracts. If you have 100 kUSD in your wallet, it means that there is a record in the contract storage saying "Address TZ1 holds 100 tokens."

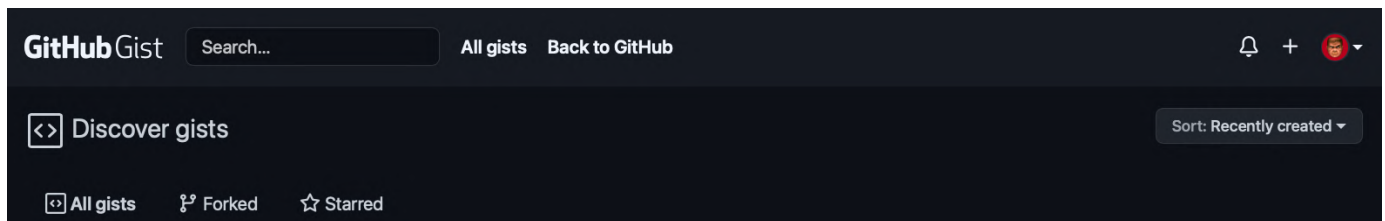
Aside from user balances, the storage also holds metadata, i.e. information about the smart contract itself: its name, brief description of the token, its ticker, number of digits after the decimal point, ID, a link to the logo, the owner's address, pseudo-entry points and other parameters.

Metadata is required for apps to see tokens and correctly display information on them. If there is no metadata, the app won't see the token, and if they are wrong, the display will also be faulty.

Create a JSON file with metadata

As of 2020, most Tezos developers use the standard TZIP-16 which dictates that metadata is stored in a JSON file. The developer has to upload the file on any public server, such as their own website, IPFS, or GitHub, and then put a link thereto in the smart contract's storage.

We'll be adding metadata to the token.ligo code we worked with last time. To do that, use Gist by Github: create a JSON file and get a uniform resource identifier (URI). Sign in or up on Github and go to Gist. Click + in the upper right corner of the window to create a new file.



Name the file fa12-metadata.json and paste the template in it:

```
{
  "symbol" : "Shorthand, ticker",
  "name" : "Full name",
  "decimals" : "0",
  "icon" : "a link to the picture with the extension of .jpg, .png or .svg",
  "description" : "Token description",
  "authors" : "author",
  "interfaces" : ["TZIP-007-2021-04-17", "TZIP-016-2021-04-17"]}

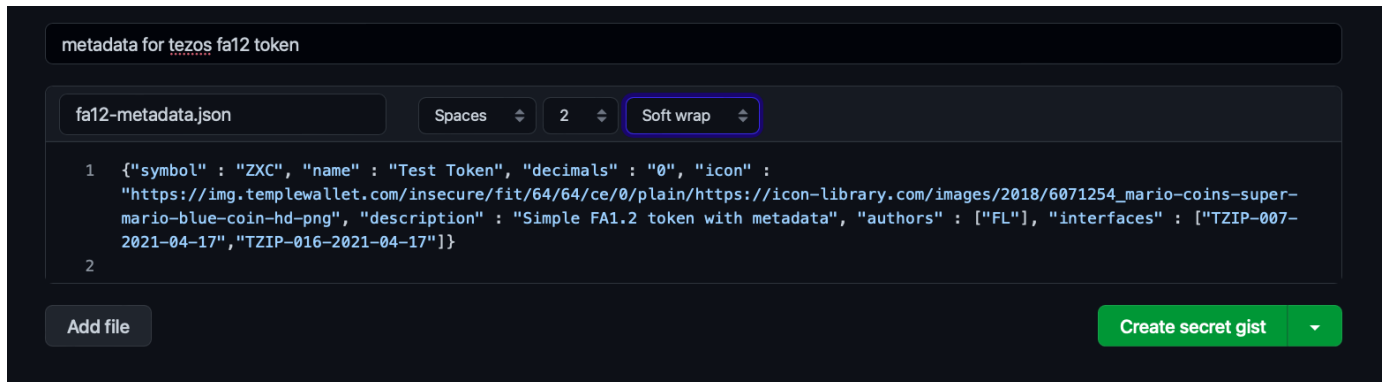
```

Fill in the fields by adding the ticker, the token's name, and add a link to the logo. Leave the fields decimals and interfaces untouched.

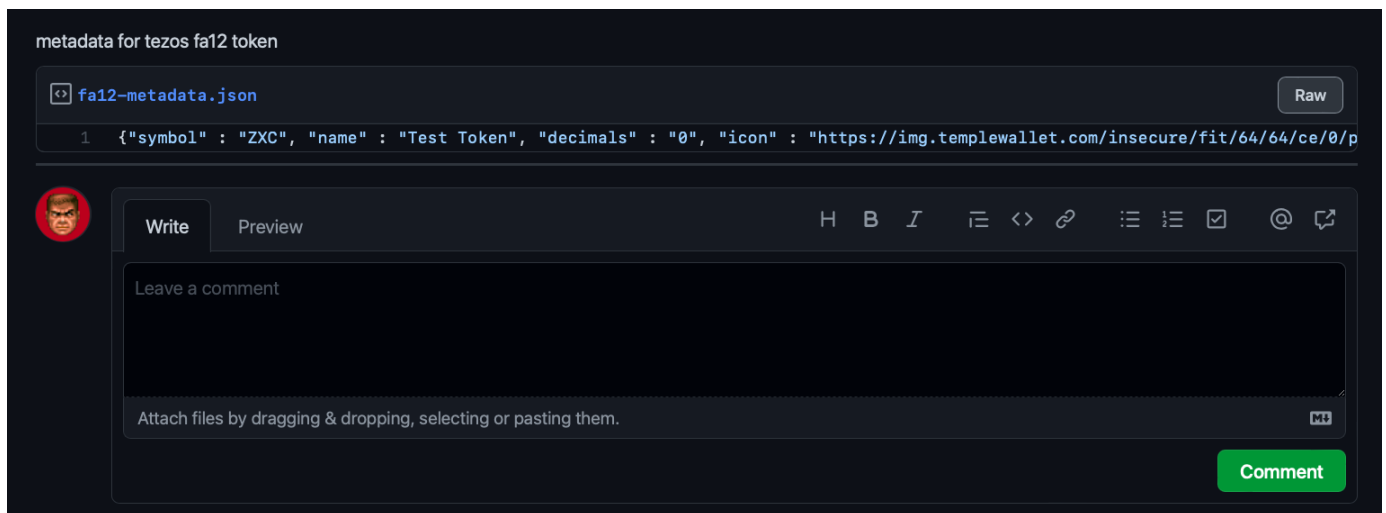
- **decimals** are the number of digits after the decimal point. If you set it over 0 (for instance, 3), wallets will show the token as 0.001. We keep the default value to avoid overcomplicating the example.
- **interfaces** are the rules of displaying metadata and the list of standard entry points the contract uses. Making alterations to this field may bring about problems with displaying the token in certain wallets.



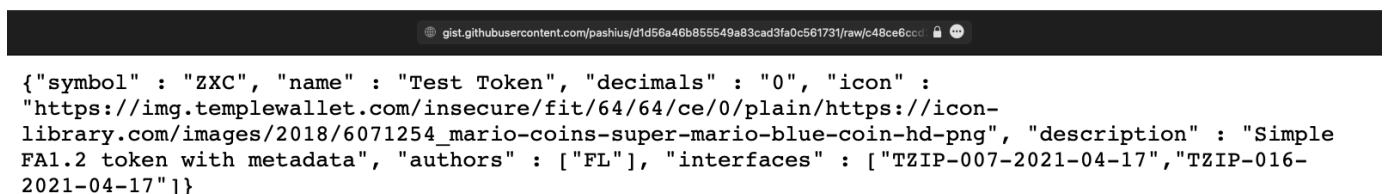
Open the drop-down menu from the green button and select Create public gist, then click the button. Thus you save the file and make it available to all users.



After refreshing the page, click Raw in the upper right corner of the window to open the file from a direct link.



Don't close the tab with the file, we'll need it in a few moments.





Adding metadata to the smart contract

A URI link to the metadata has to have a special format to be stored in the storage:

metadata : *big_map* (string, bytes). Let's add it to storage.

Start VS Code and open token.ligo. Find the declaration of storage type in the beginning and add the metadata format there.

```
(* declaring the type of SC storage. It keeps the overall quantity of tokens and big_map
type storage is
record [
  metadata      : big_map (string, bytes)
  totalSupply   : amt;
  ledger        : big_map (address, account);
]
```

Contract Examples

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

```
Increment (PascalLIGO)
129 senderAccount.allowances[spender] := value;
130
131 (* updating the SC storage *)
132 s.ledger[Tezos.sender] := senderAccount;
133
134 } with (noOperations, s)
135
136 function getBalance (const owner : address; const contr : contract(amt); var s : storage) : return is
137 block {
138   //assigning account data to the constant ownerAccount
139   const ownerAccount : account = getAccount(owner, s);
140 }
141 //returning the account balance to the proxy contract
142 with (list [transaction(ownerAccount.balance, 0tz, contr)], s)
143
144 function getAllowance (const owner : address; const spender : address; const contr : contract(amt); var
145
Line 168, Column 1
```

[View in Try-Michelson IDE](#)

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

Configure

Compile Contract

Run

Access function

The function name from where your contract will start

main

☒ Output michelson in JSON format

Go to VS Code, open token.json and replace the old code with the new one, then save the file.

Now, storage has a metadata field: wallets and blockchain explorers will look for a link to a JSON file with metadata there. Right now, the field is empty. We'll specify the file's URI in the contract's storage prior to deploying it. For the Michelson VM to read the link, it has to be translated into bytes. Go back to the URI tab at meta.json and copy the address.

gist.github.com/pashius/d1d56a46b855549a83cad3fa0c561731/raw/c48ce6cc0

```
{ "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"] }
```



Open an online converter like [Onlinestringtools](https://onlinestringtools.com/). Put the link's text in String and unflag Add Whitespaces to remove empty spaces between bytes. The converter will produce a long number in the field Bytes. It is the link in byte format. We'll use it later, so don't close the tab.

string

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

Import from file Save as... Copy to clipboard

bytes

`68747470733a2f2f7261772e67697468756275736572636f6e746556e742e636f6d2f706173686975732f746f6b6556e2d6d657461646174612f6d61696e2f6d6574612e6a736f6e`

Chain with... Save as... Copy to clipboard

string to bytes converter options

Full Bytes **Delimiter**

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

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

Go to VS Code and create a file called `storage.tz`. Paste the storage template there:

```
'(Pair (Pair { Elt "public address from acc.json" (Pair { Elt "public address from acc.json" number of tokens } number of tokens) } { Elt "" 0xlink to meta.json in byte format }) number of tokens)'
```

Fill in the template's fields. make sure you leave an empty string "" under the link to meta.json. It is the key that helps the VM understand that what follows is a link to metadata.

The public address must be in quotation marks, while the number of tokens and the link must not. Make sure to write 0x before the byte-format link, otherwise, the compiler won't be able to read it.

```
taq-test > stuff > storage.tz
1   '(Pair (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
2   |         |         | (Pair { Elt "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" 1000 } 1000) }
3   |         |         | { Elt ""
4   |         |         | 0x68747470733a2f2f7261772e67697468756275736572636f6e746556e742e636f6d2f706173686975732f746f6b6556e2d6d657461646174612f6d61696e2f6d6574612e6a736f6e
5   |         |         | 1000) '
6
```

Copy the code from `storage.js` and switch to `token-deploy.js`. Replace the old storage state record with the new one.



```
try {
  const op = await tezoz.contract.originate({
    // reading code from token.json
    code: JSON.parse(fs.readFileSync("./token.json").toString()),
    // setting the storage state in Michelson. Replace both addresses with your account
    init:
      '(Pair { Elt "tz1mn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" (Pair { Elt "tz1mn4fjJFv'
  })
}
```

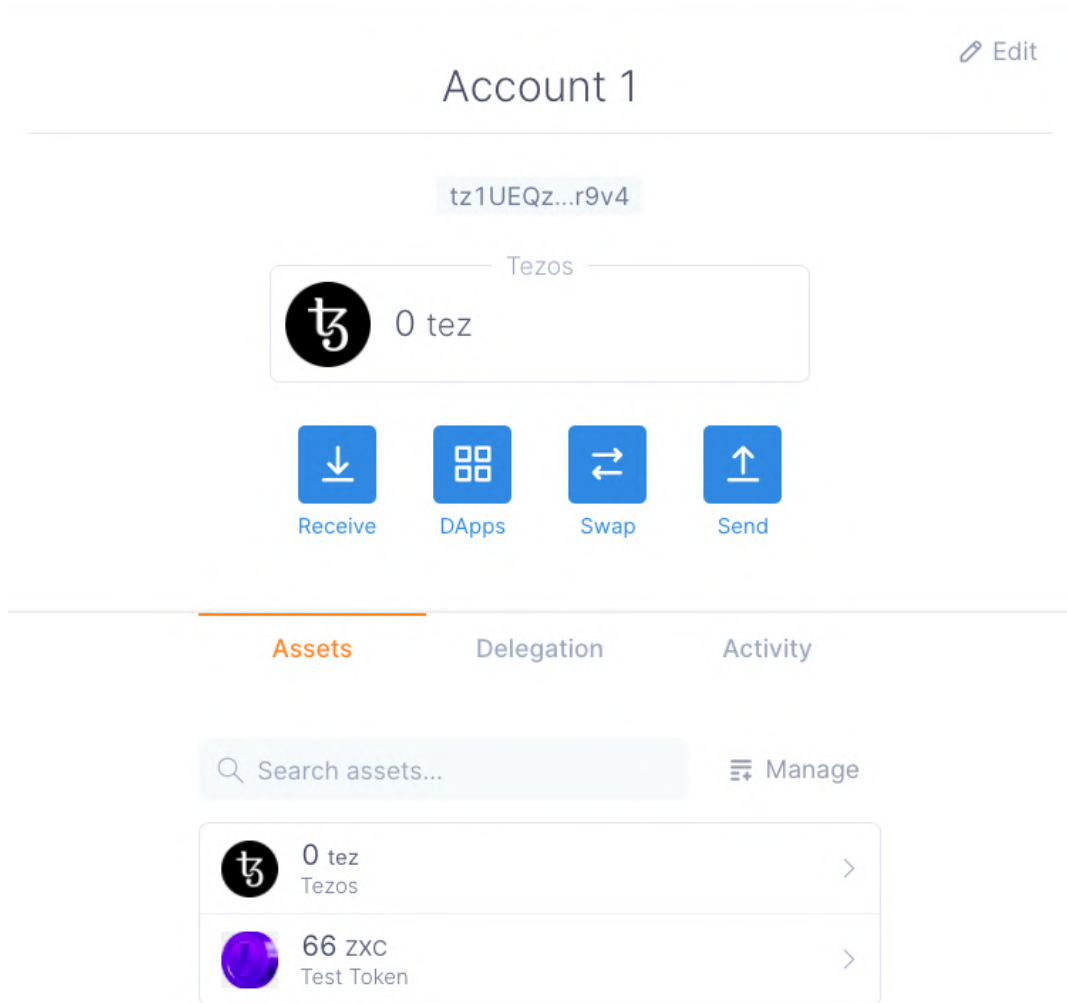
Open the terminal in VS Code. Make sure you are in `taq-test` and execute the following command:
`npx ts-node token-deploy-ts`

Taquito will produce a link to the operation's hash. Go to tzstats.io, find the contract address from the hash, and copy it. Check if the wallets see metadata: prepare and execute a transaction. Open `transfer.ts` and replace the old token address with the newly published one.

```
TS transfer.ts > [?] CONTRACT
1 import { token_transfer } from './token-transfer';
2
3 const RPC_URL = "https://florencenet.api.tez.ie"
4 const CONTRACT = "KT1DUdLarKFG9tmA4uZCgbiHv5SJA9oUBw8G" //address of the published contract
5 const SENDER = "tz1mn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" //public address of the sender (f
6 const RECEIVER = "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" // recipient's public address (f
7 const AMOUNT = 3 //number of tokens to be sent, you can put another value here
8 new token_transfer(RPC_URL).transfer(CONTRACT, SENDER, RECEIVER, AMOUNT);
9
```

Open the terminal and execute:
`npx ts-node transfer.ts`

Wait for the transaction to be confirmed and check the wallet. If you've done everything right, it will display your tokens.



Here comes the FA 2

The uniqueness of NFTs (non-fungible tokens) rests on two factors: the issuer contract address and ID. Fungible tokens have no IDs.

NFTs can be issued through a simple contract with two functions: transfer to transfer the tokens, and update to update user balances. Technically, such a token will be unique. Still, if you don't stick to FA1.2 and FA2 standards, wallets will never see this NFT, and you will have to use the Tezos client and the terminal to send the token.

To make working with NFTs simpler, Tezos devs created the standard FA2. It describes the interface for working with fungible, non-fungible, non-transferable, and other sorts of tokens. FA2 is way more complicated than FA1.2 but it gives more leeway to developers. Thus, one can issue several tokens within the same contract or put transactions in batches to save gas.

The code of an FA2 token consists of four parts:

- messages about basic errors;
- interfaces: declaring all custom data types;
- functions for work with operators, i.e. users that can transfer tokens;
- core: metadata, storage, token transfer and balance check functions.



Implementing the NFT standard FA2 in LIGO looks like this:

```
// 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

// declaring the token id type, a natural number
type token_id is nat

// declaring the types of input parameters the token transfer function assumes:
// recipient address, id, and the number of tokens. Adding the sender's address to
// the type transfer
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);
]

// declaring types to read the balance: the owner's address, token id
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));
]

// declaring the operator type (the address that can send tokens)
type operator_param is
[@layout:comb]
record [
  owner : address;
  operator : address;
  token_id: token_id;
]

// declaring the type of params required to update the list of operators
type update_operator is
[@layout:comb]
| Add_operator of operator_param
| Remove_operator of operator_param

// declaring the type that contains NFT metadata: token ID and link to the JSON
file
type token_info is (token_id * map(string, bytes))

type token_metadata is
big_map (token_id, token_info)

```



```
// declaring the type with the link to the smart contract's metadata. The data
will be shown in the wallet
type metadata is
  big_map(string, bytes)

// declaring the type that can store records on several tokens and their metadata
in the same contract
type token_metadata_param is
  [@layout:comb]
  record [
    token_ids : list(token_id);
    handler : (list(token_metadata)) -> unit;
  ]

// declaring the pseudo-entry points: token transfer, balance check, operator
update, and metadata check
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

// declaring the data types to change permissions to transfer tokens. E.g., they
can create a token that can't be sent elsewhere
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  
  
// declaring the type that stores records on operators in the same big_map  
type operator_storage is big_map ((address * (address * token_id)), unit)
```



```
// declaring the function for updating the list of operators
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

// declaring the function that checks if the user can update the list of
operators
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

// declaring the function that checks if the user can update the list of owner
addresses, and only in that case calls the update function
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

// declaring the function that checks the permissions to transfer tokens. If the
user can't send a token, the function terminates the contract's execution
```



```

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

// declaring the function for the owner to transfer the token
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)

// declaring the function that collects all transactions of the same token in one
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

// declaring the data type to store records on which address keeps the token with
the given id
type ledger is big_map (token_id, address)

// declaring the contract storage: TZIP-16 metadata, ledger of addresses and
tokens, list of operators, and on-chain metadata
type collection_storage is record [
  metadata : big_map (string, bytes);
  ledger : ledger;
  operators : operator_storage;
  token_metadata : token_metadata;
]

// declaring the token transfer function. It will get the token's id, the
addresses of the sender and the recipient, and checks whether the sender has the
right to transfer the token
function transfer (
  const txs : list(transfer);
  const validate : operator_validator;
  const ops_storage : operator_storage;
  const ledger : ledger) : ledger is block {
  // checking the sender's right to transfer the token
  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
      // checking the number of transferred NFT. We imply that the contract
has issued just 1 token with this ID
      //if the user wants to send 0, 0.5, 2 or any other number of tokens, the
function will terminate the contract's execution
      if (dst.amount = 0n) then
        ll
      else if (dst.amount /= 1n)
        then (failwith(fa2_insufficient_balance): ledger)
      else block {
        const owner = Big_map.find_opt(dst.token_id, ll);
```




```

    } with
      case owner of
        Some (o) ->
          // checking whether the sender has the token
          if (o /= tx.from_)
            then (failwith(fa2_insufficient_balance) : ledger)
            else Big_map.update(dst.token_id, Some(dst.to_), ll)
          | None -> (failwith(fa2_token_undefined) : ledger)
        end
      ,
      tx.txs,
      1
    )
  } with List.fold(make_transfer, txs, ledger)

// declaring the function that returns the sender's balance
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)

// declaring the function with pseudo-entry points which underpin the very FA2
standard
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);

```



```

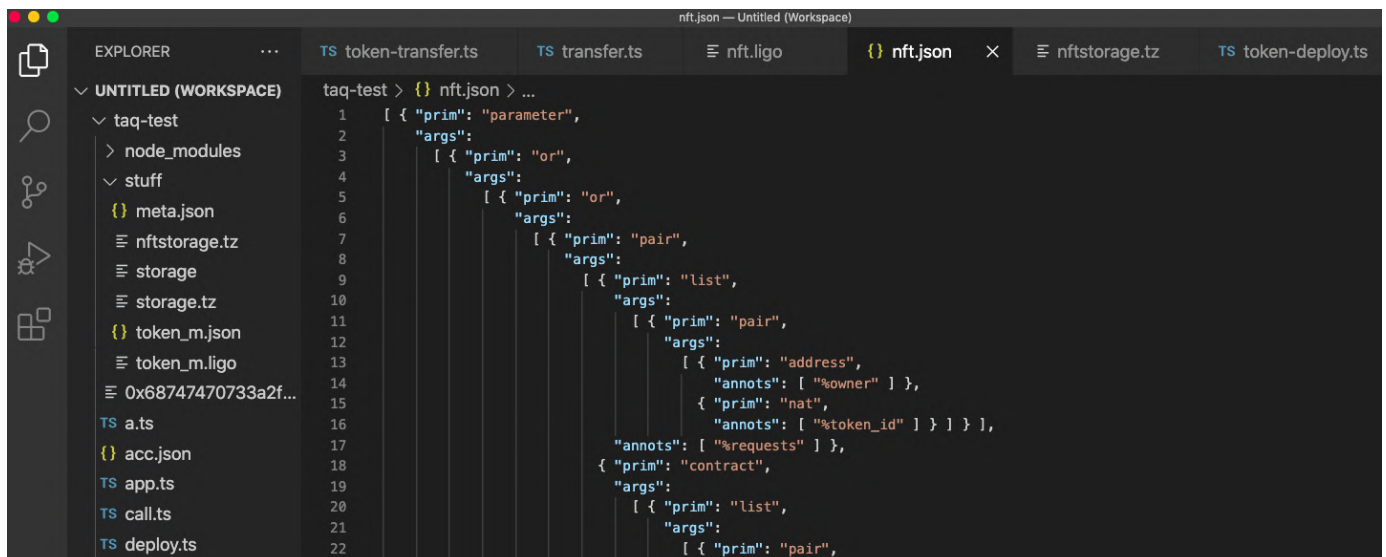
    const new_storage = storage with record [ ledger = new_ledger ]
  } with ((list [] : list(operation)), new_storage)
| Balance_of (p) -> block {
    const op = get_balance (p, storage.ledger);
  } with (list [op], storage)
| Update_operators (updates) -> block {
    const new_operators = fa2_update_operators(updates, 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

```

Issuing NFT on FA 2

Open VS Code and create `nft.ligo` in `taq-test`. Paste therein the contents of the window above or copy the code from [Gist](#).

Compile the code in LIGO into JSON. Copy the contents of the window file and go to ide.ligolang.org. There, select Compile Contract in the drop-down list and flag Output Michelson in JSON Format. Press Run. The compiler will produce the final code under the editor window. Create `nft.json` in VS Code and paste the code there.



Prepare two JSON files with metadata: for NFT and the smart contract. Go to [Gist](#) and click + in the upper right corner to create a new file. Name the file `nft_meta.json` and paste the following there:



```
{
  "symbol" : "ticker",
  "name" : "name of the NFT",
  "description" : "Description",
  "decimals" : "0",
  "isBooleanAmount" : true,
  "artifactUri" : "link to the tokenised object",
  "thumbnailUri" : "logo of theNFT",

  "minter" : "token issuer name",
  "interfaces" : ["TZIP-007-2021-04-17", "TZIP-016-2021-04-17", "TZIP-21"]}
```

Fill in all the fields except interfaces. Specify the links to the tokenised object and the token's logo as <https://...> rather than in byte format. Open the drop-down menu on the green button and select Create public gist and click the button. Thus you save the file and make it available to any user. After refreshing the page, click Raw to open the file via a direct link. Don't close the tab. In Gist, create `contract_meta.json` and paste therein the following template:

```
{
  "name" : "contract name",
  "description" : "contract description",

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

Fill in the contract's metadata, click on Create Public Gist, and then press Raw. Don't close the tab, we'll need it in a few moments. Meanwhile, prepare the script for publishing NFTs. Create `nft-deploy.ts` and paste therein the code from `token-deploy.js`. Find the method code: `JSON.parse()` and replace the readable file's name:

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

```
try {
  const op = await tezos.contract.originate({
    // reading code from token.json
    code: JSON.parse(fs.readFileSync("./nft.json").toString()),
    // setting the storage state in Michelson. Replace both addresses with your ac
    init:
      '(Pair { Elt "tz1imn4fjJFwmNaiEWnAGdRrRHxzxrBdKafZ" (Pair { Elt "tz1imn4fj
```



To publish the smart contract, we have to specify the state of the storage. To do that, create the file `nft-storage` with no extension and paste the following therein:

```
'(Pair (Pair { Elt id "your wallet address" } { Elt "" 0x link to contract
metadata in byte format}) { Elt (Pair "your wallet address" "your wallet address"
id) Unit } { Elt 0 (Pair id { Elt "" 0x link to contract metadata in byte format
}) })')
```

Fill in the fields in the template. The code shall be one string without interruptions. The token's id has to be written without quotation marks. Don't use any script other than Latin in the token's name as Michelson works exclusively with Unicode. We issue the token with `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
```

Put the storage state code in `nft-deploy.ts` in the field `init`. Don't forget to separate it with apostrophes and put a comma. Our script looked like that:

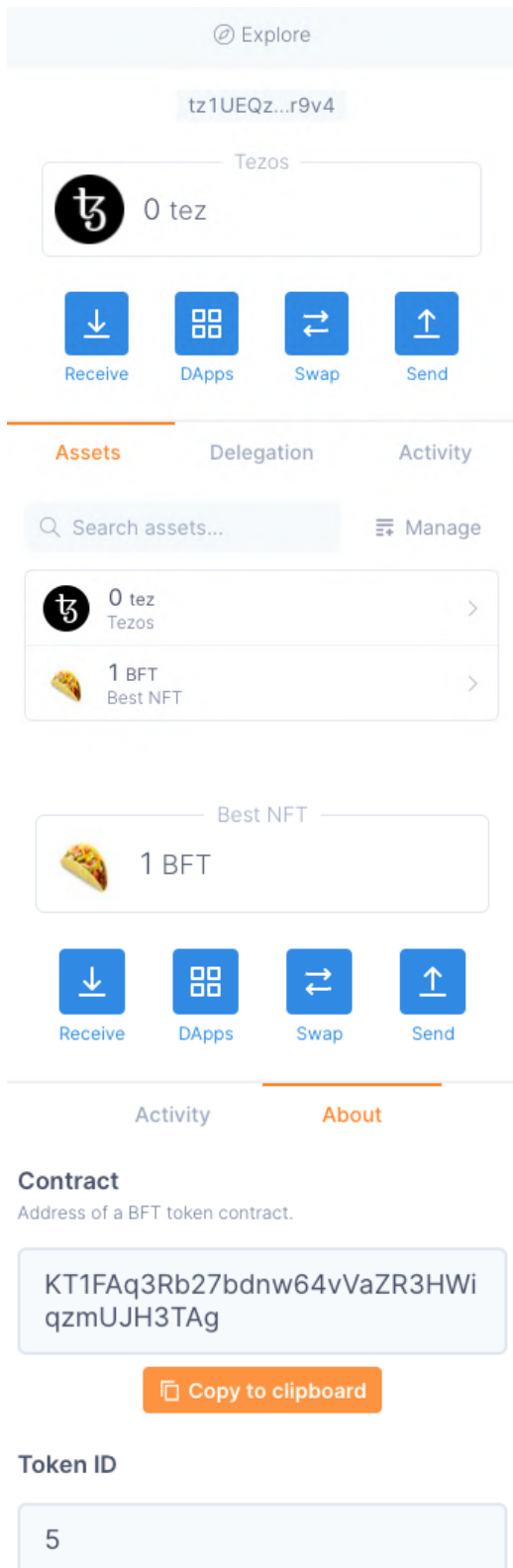
```
try {
  const op = await tezos.contract.originate({
    // reading code from token.json
    code: JSON.parse(fs.readFileSync("./nft.json").toString()),
    // setting the storage state in Michelson. Replace both addresses with your
    account address on the testnet, and numbers with the number of tokens you want
    to issue.
    init:
      '(Pair (Pair { Elt 5 "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" } { Elt ""
      0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f70617368
      6975732f62393134313431643663343739656531303034326532376265613831346265342f726177
      2f336461356234376461623031623863303935346338373539306636366564323230323235636438
      322f636f6e74726163745f6d6574612e6a736f6e }) { Elt (Pair
      "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4" "tz1UEQzJbuaGJgwvkekk6HwGwaKvjZ7rr9v4"
      5) Unit } { Elt 0 (Pair 5 { Elt ""
      0x68747470733a2f2f676973742e67697468756275736572636f6e74656e742e636f6d2f70617368
      6975732f62663063653337333738313431636230393033396564323461363134316166372f726177
      2f656636373766643462623639656436316439626438666361616234366263663830306263373664
      622f6e66746d6574322e6a736f6e }) })',
  })
```



Open the terminal and execute:

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

Wait for the deployment to end and check your wallet. The NFT will arrive in about a minute.



If the deployment was fault-free yet the wallet fails to show the token, check the network. The wallet may be connected to the mainnet rather than the testnet. As a last resort, add the token manually by pressing Manage. Sometimes, when new tokens are issued directly into the user's address, the wallet may not display them until the first transaction. You will see the token's id and the availability of the NFT on your balance but not the tokenised content. Wallets are unable to show it as they lack the interface to read the field artifactUri, unlike NFT marketplaces.

Go to the blockchain explorer [BetterCallDev](https://bettercalldev.com/) and find your contract from the operation hash or the address. Then go to the tab Tokens. The link to the tokenised content is in the field artifact_uri.



The screenshot shows the FA2 interface with the 'TOKENS (1)' tab selected. It displays the following information:

TOKEN ID	METADATA URI	
5	https://gist.githubusercontent.co...	{ } RAW JSON

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

In order to issue or burn an NFT, you have to add respective functions to the contract code. We omitted that to avoid overcomplicating the first acquaintance with FA2.

Final conclusions

Metadata is a standardised description of a smart contract/token that includes the name, ticker, description, decimals, logo link, supported interfaces, and other data. The simplest way to add metadata is to use the standard TZIP-16. According to it, info on the token and the smart contract is stored in JSON files that have to be stored on a public server like GitHub. After that, add the links to the storage. This approach's main advantage is that you can revise files in case there has been a mistake.

NFT exist thanks to metadata. The developer records the tokens identifier and the link to the tokenised content therein. The connection between the ID and the contract address is unique, so NFT with similar content and metadata will still be different from each other.