How to develop DAPPs on Tezos

Pietro Abate - Nomadic Labs

12/02/2020

A DAPP, what is it ?

- A `Distributed Application`
- It's often a web application (my take)
- It's 80% presentation / 20% smart contract
- The smart contract should encode **only** the essential business logic needed to store information on the blochchain
- Everything else goes off-chain

Plan for today

- Intro about the Tezos blockchain in general
- What is Michelson and the Tezos smart contract platform
- Tooling : tezos client
- LIGO : Hight level smart contract language for Tezos
- Taquito : Glue to build Dapps on Tezos
- Small demo of a DAPP if we have time

During the presentation we are going to talk

- 80% about smart contracts !
- 20% about web development

The building Blocks(!) of a Blockchain

At an abstract level, we can see a blockchain as an immutable database that operates in a decentralized network. It is based on *simple* yet revolutionary ideas.

- Public Key Cryptography / Digital signature / Cryptographic Hash Functions
- A probabilistic solution to the byzantine generals problem for consensus among all nodes
- A p2p / gossip network for low level communication

Blockchains are often called `crypto-ledgers`, that is, an electronic book recording transactions where the identity of users and immutability of the book is cryptographically ensured.

Generic algorithm:

1. Send/receive/broadcast new transactions to all "participants" (**nodes**) on the network
2. Aggregate transaction into **blocks**
3. The next block is broadcasted by one (or several) nodes to the network
4. Nodes express their acceptance of a block by including its hash in the next block they create

Tezos Blockchain Framework

- Generic (other blockchains platforms can be implemented on our framework)
- Consensus protocol agnostic (possibly extracted code from specification ?)
- Strong emphasis on verified components
- Rigorous software engineering practices
- Written in OCaml
- Already used in production and continually evolving

Current Protocol (Babylon)

- Proposed in July together with Cryptium Labs
- Introduces Emmy+
- Reorganizes accounts (delegation from tz accounts, no more KT1 replaced by verified manager.tz)
- Improvements to smart contracts (entrypoints, multiple bigmaps, gas update)
- Small changes to voting procedure (proposal quorum, quorum caps)

Smart Contract Platform

- Popularized by Ethereum as general purpose programs
- The blockchain is the trusted intermediary (think escrow) that replaces many financial, notarial or insurance related functions
- In Tezos are simple programs to automate business logic
- With formal verification in mind

Smart contract in Tezos are

- A smart contract is a piece of code in the blockchain :
  - the code stored in a block by a user
  - other users can call this code
- A contract has state, can perform operations on the blockchain
  - Can passively interact with outside services (oracle)
  - Can call other smart contracts

Inter-contract interactions:

- contracts can emit operations (originations, transfer, delegations),
- emitted operations are run in a queue,
- either all operations succeed or no side-effect is performed, except taking the fees.

Why Michelson?

Michelson is Tezos smart contract language.

- Generic
- Safe
- Readable
- Easy gas accounting

Inherent tension:

- Generic and easy gas accounting suggest an assembly-like language
- Safe and readable suggest a high-level, functional, language

Michelson as a compromise

A low-level language with high-level primitives.

- Stack-based for easier gas calculation (no variables)
- Statically typed
- Functional
- Lispy

Michelson example

```
storage    (map string nat) ;
parameter  string ;
return     unit ;
code       { AMOUNT @sent; PUSH @required tez "5.00" ;
             COMPARE ; GT ; IF { FAIL } {} ;
             DUP ; DIP { CDR ; DUP } ;
             CAR ; DUP ;
             DIP {
                   GET ; IF_NONE { FAIL } {} ;
                   PUSH nat 1; ADD ; SOME
                 } ;
             UPDATE ; PUSH unit Unit; PAIR }
```

Michelson example (cont)

Initialization: sets the list of candidates

```
Map (Item "Tacos" 0)
    (Item "Baguette" 0)
    (Item "Kimchi" 0)
```

Tezos vs Ethereum vs Bitcoin

The main differences :

- Tezos has an on-chain governance system. Changes of Bitcoin and Ethereum are driven by the lead developers / foundation.
- Tezos is based on Delegated Proof-of-stake while Bitcoin and Ethereum on Proof-of-work (Ethereum has plans to move to a PoS consensus algorithm in the future)
- Tezos has Michelson while Ethereum has the EVM.

Solidity vs Michelson

- High Level (js-like) vs Low Level (with high-level primitives)
- Bytecode vs Readable code
- Virtual Machine (EVM) vs Interpreter

I don't compare Michelson with Bitcoin Scripts as they are two completely different tools.

High level languages

- Michelson is a low level programming language
- It is efficient, and easily verifiable, but can be difficult for a programmer
- There are many ongoing efforts to create high-level languages that compile down to Michelson
- Examples of such languages are PascalLIGO, CamlLIGO, ReasonLIGO or SmartPy

A few links

LIGO	`https://www.ligolang.org`
	Pascal-like syntax (also OCaml and Reason), strongly typed
SmartPy	`https://smartpy.io`
	Python as a metalanguage, type inference, tests in Python
Morley	`http://hackage.haskell.org/package/morley`
	Haskell library
Albert	`https://albert-lang.io`
	formally-verified, intermediate language
SCaml	`https://gitlab.com/dailambda/scaml`
	Subset of OCaml, benefits from its ecosystem
Archetype	`http://archetype-lang.org`
	state machine, assets, formally verifiable
Juvix	`https://juvix.org`
	dependent-linearly-typed, formally verifiable

Tools and Requirements

- Debian based machine or VM
- `docker`
- `python3`
- `snapd`

First we need a node

To run one node on `localhost` we issue the following command :

```
$ alias teztool='docker run -it -v $PWD:/mnt/pwd \
  -e MODE=dind -e DIND_PWD=$PWD \
  -v /var/run/docker.sock:/var/run/docker.sock \
  registry.gitlab.com/nomadic-labs/teztool:latest'

$ teztool babylonnet sandbox --time-between-blocks 10 \
    start 18732
```

This will initialize a node

- listening for RPC on port 18732 (rpc port)
- The node will initialize and run the *babylon* protocol
- Will start a baker and create a block every 10 seconds

Tezos Client

- We install a snap binary
- snap is a container based sw distribution platform

```
$ wget https://gitlab.com/abate/tezos-snapcraft/-/raw\
  /master/snaps/tezos_5.1.0_multi.snap?inline=false

$ sudo snap install tezos_5.1.0_multi.snap --dangerous
```

And now finally we can talk with our node or with any Tezos node out there

```
$ export PATH=/snap/bin/:$PATH
$ tezos.client man
```

Tezos Client (cont)

We can keep track of the level that our local node has reached :

```
$ tezos.client -A localhost -P 18732 bootstrapped
```

Notice we connect to the node on `localhost:18732`

Or check the balance :

```
$ tezos.client get balance for bob
```

(We'll play with this in a moment)

Tezos Client : Global options

We have already seen a few options used to wrap the tezos client and work in our sandbox.

To create a new configuration file based on the current options we can use the command `tezos.client config init`.

```
$ tezos.client -A localhost -P 18732 config init
```

- `-A` : the ip address of the node host (accepts ipv4 and ipv6 addresses)
- `-P` : the RPC port of the node

Sandbox Test accounts (cont)

The client on our machine does not know these accounts, so we need to add them.

```
$ tezos.client list known addresses

tezos.client import secret key bootstrap1 \
  unencrypted:edsk3gUfUPyBSfrS9CCgmCiQsTC...
tezos.client import secret key bootstrap2 \
  unencrypted:edsk39qAm1fiMjgmPkw1EgQYkMz...
tezos.client import secret key bootstrap3 \
  unencrypted:edsk4ArLQgBTLWG5FJmnGnT689V...
tezos.client import secret key bootstrap4 \
  unencrypted:edsk2uqQB9AY4FvioK2YMdfmyMr...
tezos.client import secret key bootstrap5 \
  unencrypted:edsk4QLrcijEffxV31gGdN2HU7U...
```

Sandbox Test accounts (cont)

```
$ tezos.client list known addresses

bob: tz1M4zWSnYfsVyTLqL3hsHifuwwLWo2J196z (encrypted sk known)
bootstrap5: tz1ddb9NMYHZi5... (unencrypted sk known)
bootstrap4: tz1b7tUupMgCNw... (unencrypted sk known)
bootstrap3: tz1faswCTDciRz... (unencrypted sk known)
bootstrap2: tz1gjaF81ZRRvd... (unencrypted sk known)
bootstrap1: tz1KqTpEZ7Yob7... (unencrypted sk known)
```

Transactions

One of the basic uses of the `tezos.client` is to add transactions to the blochchain and to check account balances.

First let's check how many tokens are associated to the account `bootstrap1`

```
 tezos.client get balance for bootstrap1
4000000 tz
```

LIGO

- Is a programming language that compiles to Michelson
- Features variables, expressions, function calls, data types...
- Is available in different flavors:
  - PascaILIGO, an imperative Pascal-like language
  - CameLIGO, inspired by the pure subset of OCaml
  - ReasonLIGO, inspired by ReasonML, which features a JS-like syntax

Install LIGO

LIGO is distributed in a number of way. My personal way of using ligo is using their docker image.

Copy this one-liner in `~/.local/bin`

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

and maybe

```
export PATH=~/.local/bin:$PATH
```

Et Voila

```
$ ligo
Unable to find image 'nomadiclabs/ligo:latest' locally
latest: Pulling from nomadiclabs/ligo
e7c96db7181b: Pull complete
85d2217d151f: Pull complete
9767b756e420: Pull complete
Digest: sha256:a1228ded1e4d25784d3bd50e06e3d7ee273ba9eff03c30e
Status: Downloaded newer image for nomadiclabs/ligo:latest

LIGO needs a command. Do ligo --help
```

Built-in types `Nat`, `Int`, `Tez`

- Ex. 1 : int , 1n : nat, 5mutez
- `nat + int` produces `int`
- `tez + tez` produces `tez`
- you can't add `tez + int` or `tez + nat`
- you can't add `nat + int`
- subtraction of two nats, yields an int
- you can't subtract two nats
- you can multiply `nat` and `tez`
- division of two `tez` values results into a `nat`

Built-in types Strings

```
const name: string = "Alice";
const greeting: string = "Hello";
// Hello Alice
const full_greeting: string = greeting ^ " " ^ name;
// length = 5
const length: nat = size(name);
```

Notice that in `Pascal ligo` we have type annotations.

Built-in types `Bool`

```
const a: bool = True;
const b: bool = False;
```

Nothing much to say here ...

Built-in types Sum Types

```
type action is
  | Increment of int
  | Decrement of int
  | Reset of unit
```

Sum types are particularly useful to declare actions/entrypoints for our contract.

Functions

```
function add(const a: int; const b: int): int is begin
  const result: int = a + b;
  end with result;
```

The function body consists of two parts:

- block {} - logic of the function
- with - the return value of the function

A function can also be *block-less*

Variables and Const

No more **push/pop** from the stack as in Michelson.

```
const c : nat = 10n;

var start: timestamp := \"1970-01-00T00:00:01Z\";
```

- Constants are immutable by design, which means their values cannot be reassigned
- Variables, unlike constants, are mutable. They cannot be declared in a global scope, but they can be declared and used within functions, or as function parameters.

For Loops

```
function for_sum (var n : nat):
  int is block {
    var acc : int := 0 ;
    for i := 1 to int(n)
      begin
        acc := acc + i;
      end
} with acc
```

We also have `while` loops in the language

Conditionals

```
type magnitude is Small | Large // See variant types.

function compare (const n : nat) : magnitude is
  if n < 10n then Small (Unit) else Large (Unit)
  // Unit is needed for now.


if x < y then
  block {
    const z : nat = x;
    x := y; y := z
  }
else skip;
```

Option Values

```
type dinner is option(string);

// stay hungry
const p1: dinner = None;
// have some hamburgers
const p2: dinner = Some("Hamburgers")
```

Pattern Matching

```
type dinner is option(string);
function is_hungry(const dinner: dinner):
  bool is // block-less function
    with (
        case dinner of
            | None -> True
            | Some(d) -> False
        end
    )
```

Record Types

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

Create a record :

```
const alice : user =
  record [ id = 1n; is_admin = True; name = "Alice" ]
```

Access a record as

```
const alice_admin : bool = alice.is_admin
```

Maps

```
type move is int * int

type register is map (address, move)
```

Initialize the map :

```
const moves : register =
  map [
    ("tz1KqTpEZ...." : address) -> (1,2);
    ("tz1gjaF81...." : address) -> (0,3)]
```

Big_maps are not different

```
type move is int * int

type register is big_map (address, move)


const moves : register =
  big_map [
    ("tz1KqTpEZ...." : address) -> (1,2);
    ("tz1gjaF81...." : address) -> (0,3)]
```

All together : Our First Contract

```
type action is
| Increment of int
| Decrement of int
| Reset of unit

function main (const p : action ; const s : int) :
  (list(operation) * int) is
 block { skip } with ((nil : list(operation)),
  case p of
  | Increment(n) -> s + n
  | Decrement(n) -> s - n
  | Reset(n) -> 0
 end)
```

Tezos Specific functions : PACK/UNPACK

- Michelson provides the PACK and UNPACK instructions for data serialization.
- `PACK` converts Michelson data structures into a binary format
- `UNPACK` reverses that transformation.

In LIGO this can be accessed with the function `bytes_pack` and `bytes_unpack`

```
function id_string (const p : string) : option (string)
is block {
  const packed : bytes = bytes_pack (p)
} with (bytes_unpack (packed) : option (string))
```

Tezos Specific functions : Hash

We can hash keys of value in LIGO using the function `crypto_hash_key`

```
function check_hash_key (const kh1 : key_hash;
                         const k2 : key) :
                         bool * key_hash is
  block {
    var ret : bool := False;
    var kh2 : key_hash := crypto_hash_key (k2);
    if kh1 = kh2 then ret := True else skip
  } with (ret, kh2)
```

This function gets a hashed key and a not hashed key and verify if the are the same.

Tezos Specific functions : Checking Signatures

```
function check_signature
    (const pk      : key;
     const signed  : signature;
     const msg     : bytes) : bool
  is crypto_check (pk, signed, msg)
```

This function check if the message was signed by the owner of `pk`

Tezos Specific functions : Self Address

- Often you want to get the address of the contract being executed.
- You can do it with `self_address`.

```
const current_addr : address = self_address
```

This works at top level, but not in functions.

Lets' compile it

```
scripts/ligo compile-contract increment.ligo main
```

```
{ parameter (or (or (int %decrement) (int %increment))
(unit %reset)) ;
  storage int ;
  code { DUP ; CDR ; DIP { DUP } ; SWAP ; CAR ;
         IF_LEFT
           { DUP ;
             IF_LEFT
               { DIP 2 { DUP } ; DIG 2 ;
                 DIP { DUP } ; SUB ; DIP { DROP } }
               { DIP 2 { DUP } ; DIG 2 ;
                 DIP { DUP } ; ADD ; DIP { DROP } } ;
             DIP { DROP } }
           { DROP ; PUSH int 0 } ;
         NIL operation ; PAIR ; DIP { DROP 2 } } }
```

Originate

```
tezos.client originate contract increment
  transferring 0 from bootstrap1
  running michelson/increment.ligo.tz
  --init "1" --burn-cap 0.48
```

- we originate a new contract and we call it `increment`
- we can decide to transfer token to the contract
- these token come from an identity (`bootstrap1`)
- `--init "1"` is the initial value for the storage
- `--burn-cap 0.48` is the fee we pay to the chain to originate the contract

Check the storage of the contract

We can use the `tezos.client` to have a look at the storage of the contract

```
$ tezos.client get contract storage for increment
3

$ tezos.client get contract storage for first
Pair (Pair "great feature" {})
  { Elt 1 10 ; Elt 2 0 ; Elt 3 0 }
```

Using the RPC interface

Using `tezos.client -l`

```
>>>>2: http://localhost:18734/chains/main/blocks\
  /head/context/contracts\
  /KT1W6VyBDfxeFMNMJhRzoNbS12freH9KiYCu/storage
<<<<2: 200 OK
  { "prim": "Pair",
    "args":
      [ { "prim": "Pair", "args": [
      { "string": "great feature" }, [] ] },
        [ { "prim": "Elt", "args": [
        { "int": "1" }, { "int": "10" } ] },
          { "prim": "Elt", "args": [
          { "int": "2" }, { "int": "0" } ] },
          { "prim": "Elt", "args": [
          { "int": "3" }, { "int": "0" } ] } ] ] }
```

Demo

A Few more LIGO Snippets.

This is the part that never works.

Taquito

Is a typescript library to interact with a Tezos node

- It's Easy to Use : versioned, releases, published to npmjs.org
- Includes a set of ready-made React components
- Taquito has no reliance on any stack by default
- Taquito comes complete with a well-documented API
- Open source and friendly developers
- https://tezostaquito.io

Installing the library

```
yarn install @taquito/taquito
```

Opinion : `yarn` bettern than `npm`

And instantiate an instance of the library

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

const tezos = new TezosToolkit();
```

Configure the node

```
import { Tezos } from '@taquito/taquito';

Tezos.setProvider({ rpc: 'your_rpc' });
```

If you are developing this is usually configure to talk to a sandbox instance

```
import { Tezos } from '@taquito/taquito';

Tezos.setProvider({ rpc: 'localhost:18732' });
```

You might have problem configuring the `cors` policy.

- The sandbox uses very open cors policy.
- In production, you should configure a node with strict policies.

TezBridge (detour)

Why we need another component ?

- TezBridge is a connector between Tezos and DApps.
- https://docs.tezbridge.com/
- TezBridge is a pure web application
- A modern web browser is the only software required.

What it can do for us :

- It can do Key generation for you (nice for testing)
- Key import : People can import all kinds of keys into the TezBridge(ed25519/secp256k1/p256/mnemonic/faucet)
- Local signer : act a bit like metamask
- Hardware signer : TezBridge currently supports Ledger with USB port

We need to configure a singer now

We use TezBridge in this example :

```
import { Tezos } from '@taquito/taquito';

import { TezBridgeSigner } from '@taquito/tezbridge-signer';

Tezos.setProvider({ signer: new TezBridgeSigner() });
```

Get the Balance

```
Tezos.setProvider({ rpc: 'https://api.tez.ie/rpc/mainnet' });

Tezos.tz.getBalance('tz1NAozDvi5e7frVq9cUaC3uXQQannemB8Jw')
  .then(balance =>
    render(`${balance.toNumber() / 1000000} tz`))
  .catch(error => render(JSON.stringify(error)));
```

While developing it's handy to import private keys in taquito

```
Tezos.importKey(
  'p2sk2obfVMEuPUnadAConLWk7Tf4Dt3n4svSgJwrgpamRqJXvaYcg1');
// note this is a `p2sk` : private key
// only good for testing
```

Or import a faucet account

```
Tezos.importKey(
  FAUCET_KEY.email,
  FAUCET_KEY.password,
  FAUCET_KEY.mnemonic.join(' '),
  FAUCET_KEY.secret
);
```

Transfer

```
Tezos.setProvider({ rpc: 'https://api.tez.ie/rpc/babylonnet' })

render(`Fetching a private key...`);
fetch('https://api.tez.ie/keys/babylonnet/', {
  [...] // complete snippet on the taquito website
  .then(() => {
    const amount = 2;
    const address = 'tz1h3rQ8wBxFd8L9B3d7Jhaawu6Z568XU3xY';
    render(`Transfering ${amount} tz to ${address}...`);
    return Tezos.contract.transfer({
      to: address, amount: amount });
  })
  .then(op => {
    return op.confirmation(); })
  .then(block => render(`Block height: ${block}`))
  .catch(error => render(
    `Error: ${error} ${JSON.stringify(error, null, 2)}`));
```

Interacting with a smart contract

```
Tezos.setProvider({ rpc: 'https://api.tez.ie/rpc/babylonnet' })

render(`Fetching a private key...`);
fetch('https://api.tez.ie/keys/babylonnet/', {
  [...] // complete snippet on the taquito website
  .then(() => Tezos.contract.at(
  'KT1LjpCPTqGajeaXfLM3WV7csatSgyZcTDQ8'))
  .then(contract => {
    const i = 7;
    render(`Incrementing storage value by ${i}...`);
    return contract.methods.increment(i).send();
  })
  .then(op => op.confirmation())
  .then(block => render(`Block height: ${block}`))
  .catch(error => render(
    `Error: ${JSON.stringify(error, null, 2)}`));
```

ReasonML

- ReasonML let you write simple, fast and quality type safe code
- It has a functional flavour, and it is based on the OCaml compiler
- It provides types without hassle : Powerful, safe type inference mean no type annotations
- Easy JavaScript interop : Use packages from NPM/Yarn with minimum hassle
- Flexible & Fun : can be used to make websites, animations, games, servers, cli tools, and more!
- This is how we do roll, but all the tools I presented today are not linked to `reasonml`
- https://reasonml.github.io/

ReasonML binding

- It's a pull request on the taquito gitlab : https://github.com/ecadlabs/taquito/pull/234
- Provides a minimal type-safe binding on top of the taquito library
- To use it `yarn add https://gitlab.com/abate/taquito-bs`

Not that different to JS

```
let bootstrap1 = "tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx";
let bootstrap1_sk = "edsk3gUfUPyBSfrS9CCgmCiQsTCHGkviBDusMxDJs
let inMemorySigner = new_inMemorySigner(bootstrap1_sk);

Taquito.tezos -> setProvider({
  rpc : "http://localhost:18732", signer: inMemorySigner});

Js.log("retrieve balance of " ++ bootstrap1);
tezos -> Basic.get_balance(bootstrap1) -> Js.log;
```

Re-Demo

A Complete ReasonReact Application.

This is the part that sometimes works.

Conclusions and Take-aways

- We provide the necessary tools to develop DAPPs on Tezos
- LIGO is a nice language to write smart contract (but there are many others out there)
- The smart contract should encode **only** the essential business logic needed to store information on the blochchain
- Everything else goes off-chain
- Taquito is a rich library to write DAPPs
- ReasonReact is a nice language to do web programming.

Questions?

Contact Us

Pietro Abate pietro.abate@nomadic-labs.com