Write and deploy my first dApp part.2

Prerequisites

Node.js: javascript-like language that runs on the Chrome Javascript engine
—> fast
—> interpreted language
https://nodejs.org/en/

Ganache-cli (new testrpc) simulates a full node behaviour.

```
npm install -g ganache-cli
```

See https://docs.nethereum.com/en/latest/ethereum-and-clients/ganache-cli/ for the different options supported by ganache-cli.

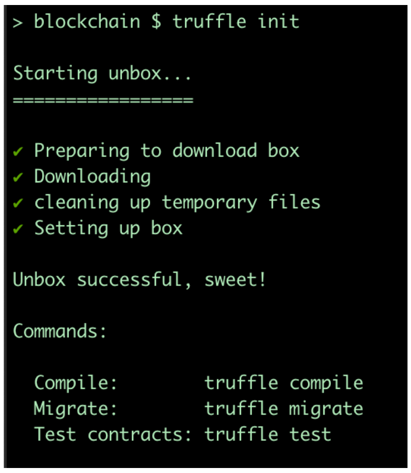
Download the truffle environment to deploy smart-contracts written in Solidity. Ganache-cli must be running besides.

```
npm install -g truffle
```

Create an empty directory (here, blockchain) and run

```
truffle init
```

Truffle creates a project, with configuration files and folders for the different contracts written in Solidity.



```
> blockchain $ truffle init

Starting unbox...
=================

✓ Preparing to download box
✓ Downloading
✓ cleaning up temporary files
✓ Setting up box

Unbox successful, sweet!

Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
```

Initialise Truffle

Open truffle-config.js and replace the content with

```
module.exports = {
     networks: {
          development: {
               host: "127.0.0.1",
               port: 8545,
               network_id: "*",
          }
     }
}
```

`network_id` matches the network created by ganache-cli; if you know the number (e.g. 3454, see option --`networkId` of ganache-cli) you can fill the field. Otherwise `"*"` matches any network.

Write a smart contract in Solidity

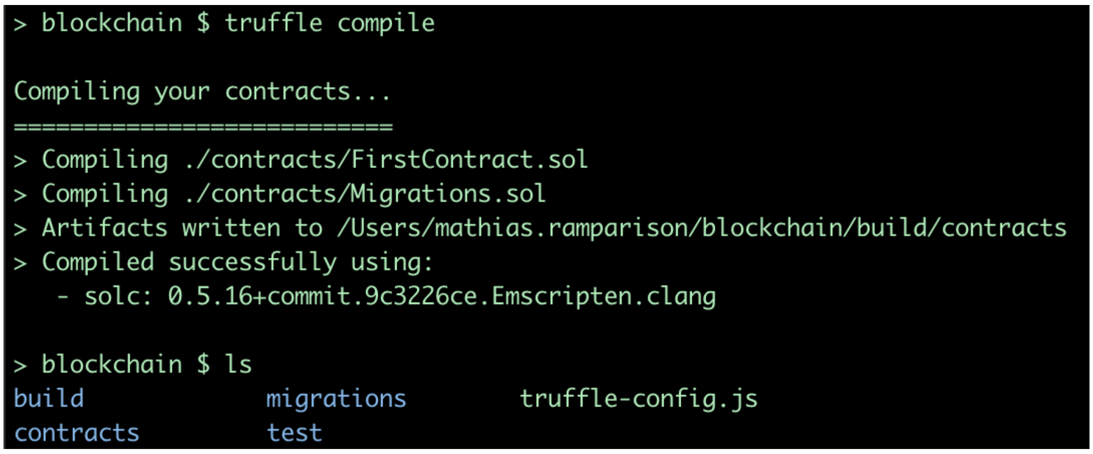
Create a new file FirstContract.sol in the `contracts` folder. By default, if no account is specified, truffle uses the first account created by ganache-cli to pay the transactions.

Copy/Paste the following code in the file.

```
// version of solidity to use for the compilation, above 0.5.0
pragma solidity >=0.5.0;

// the contract
contract FirstContract {
    // variables used in the contract
    string private myText;

    // constructor of the contract
    constructor() public {
     myText = "Hello World";
    }

    //function that returns the content of myText
     function getText() public view returns(string memory){
     return myText;
     }

    //function to assign a new content to myText
    function setText(string memory _myText) public {
     myText = _myText;
     }

}
```

COMPILE THE CODE

Now, we can compile the code with `truffle compile`



```
> blockchain $ truffle compile

Compiling your contracts...
===========================
> Compiling ./contracts/FirstContract.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/mathias.ramparison/blockchain/build/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

> blockchain $ ls
build               migrations          truffle-config.js
contracts           test
```

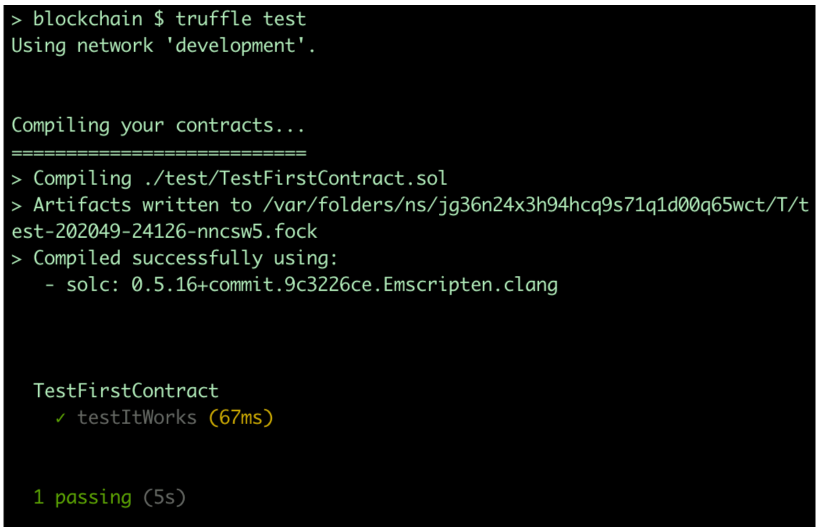
A new folder `build` has been created, with a subfolder `contracts`. In this new folder, there is the compiled code (along with other informations) of FirstContract, in the JSON format. It is called artifact.
In the terminal where ganache-cli is running, a new transaction has been created. Compiling code on the network (here, the local one supported by ganache-cli) costs “gas” (ETH) so the first account must see its balance decrease.

TEST THE SMART CONTRACT

We can first test the smart contract by running a test contract on the local blockchain.
Create a file TestFirstContract.sol in the folder `test`
Copy/Paste the following code

```
pragma solidity >=0.5.0;
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/FirstContract.sol";

contract TestFirstContract {

    function testItWorks() public {

        // Get the deployed contract
          FirstContract firstcontract =
FirstContract(DeployedAddresses.FirstContract());

        // Call getText function in deployed contract
          string memory text = firstcontract.getText();

        // Assert the function returns the correct initial text
          Assert.equal(text, "Hello World", "it works");
     }
}
```

Run truffle test. It may compile the code again, and will return a positive result.



```
> blockchain $ truffle test
Using network 'development'.


Compiling your contracts...
===========================
> Compiling ./test/TestFirstContract.sol
> Artifacts written to /var/folders/ns/jg36n24x3h94hcq9s71q1d00q65wct/T/t
est-202049-24126-nncsw5.fock
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang



  TestFirstContract
    ✓ testItWorks (67ms)


  1 passing (5s)
```

Running the test costs ETH to the first account created by ganache-cli.

DEPLOY THE SMART CONTRACT ON THE BLOCKCHAIN

To be able to interact with the smart contract on the blockchain, we must “migrate” the compiled code.To perform the migration, truffle needs instructions.

Create a new file 2_deploy_contract.js in the `migration` folder and copy/paste the following code.

```
// requires the compiled code and other informations
const FirstContract = artifacts.require("FirstContract");
// deploys the contract named FirstContract from the JSON artifact
module.exports = function(deployer) {
     deployer.deploy(FirstContract);
};
```

Deploy the compiled code with `truffle migrate`
The final cost of the deployment is displayed.

Interact with the smart contract

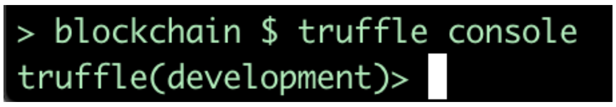
At a low level, we can interact with Ethereum nodes with a remote procedure call (RPC) protocol encoded in JSON (see https://github.com/ethereum/wiki/wiki/JSON-RPC).

The web3 library, installed in the first part of this practical session offers a high-level interface for JSON-RPCalls.

Truffle offers an abstraction of the web3 interface for the interactions with Ethereum nodes and especially smart contracts. Truffle is commonly used among smart contracts developers.

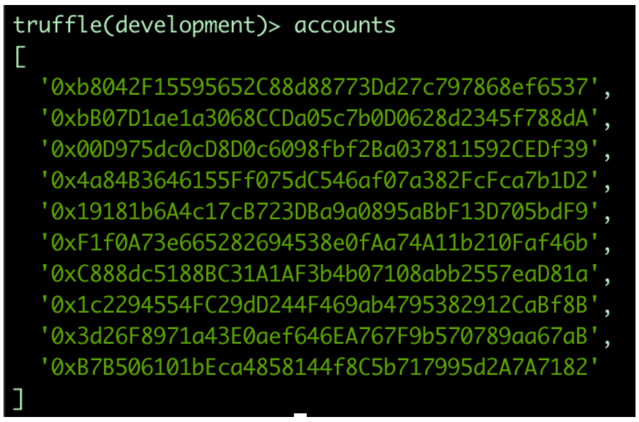
(high-level) Truffle >> web3 >> JSON-RPC (low level)

Truffle encompasses its own command line interface (CLI).
Start the truffle cli with `truffle console`



```
> blockchain $ truffle console
truffle(development)>
```

Similarly to web3, we can display the accounts

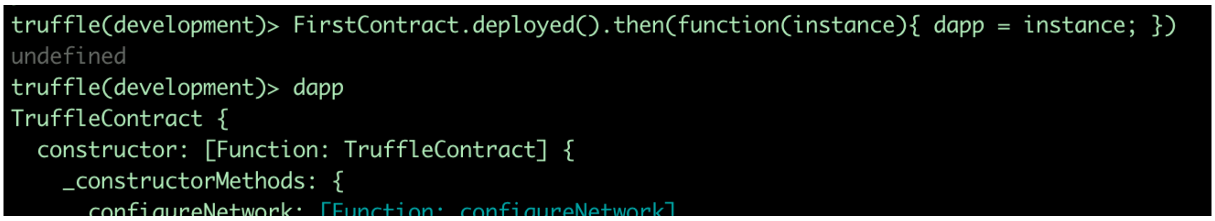


```
truffle(development)> accounts
[
  '0xb8042F15595652C88d88773Dd27c797868ef6537',
  '0xbB07D1ae1a3068CCDa05c7b0D0628d2345f788dA',
  '0x00D975dc0cD8D0c6098fbf2Ba037811592CEDf39',
  '0x4a84B3646155Ff075dC546af07a382FcFca7b1D2',
  '0x19181b6A4c17cB723DBa9a0895aBbF13D705bdF9',
  '0xF1f0A73e665282694538e0fAa74A11b210Faf46b',
  '0xC888dc5188BC31A1AF3b4b07108abb2557eaD81a',
  '0x1c2294554FC29dD244F469ab4795382912CaBf8B',
  '0x3d26F8971a43E0aef646EA767F9b570789aa67aB',
  '0xB7B506101bEca4858144f8C5b717995d2A7A7182'
]
```

In the truffle cli, instantiate the contract previously deployed with

```
FirstContract.deployed().then(function(instance){ dapp = instance; })
```

FirstContract is instantiated in the variable `dapp`
You can display the details of your contract (i.e., content of FirstContract.json)…

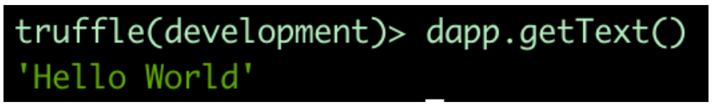


```
truffle(development)> FirstContract.deployed().then(function(instance){ dapp = instance; })
undefined
truffle(development)> dapp
TruffleContract {
  constructor: [Function: TruffleContract] {
    _constructorMethods: {
      configureNetwork: [Function: configureNetwork]
```

… and the file is quite huge.

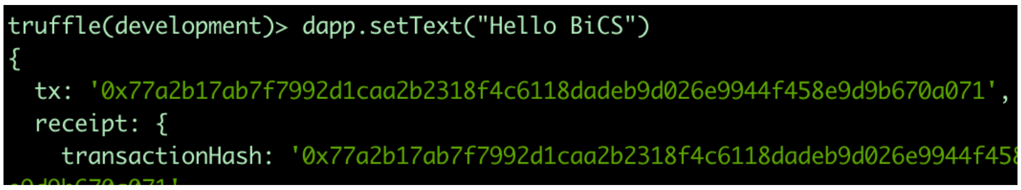
CALL THE FUNCTIONS OF YOUR SMART CONTRACT [1]

The getText() function is just a call (`eth_call` in ganache-cli) to get the content of a variable in the smart contract. Therefore it is free of charges (no fee).



```
truffle(development)> dapp.getText()
'Hello World'
```

On the contrary, the setText() function is an actual transaction that requires a computation from the nodes of the network (`eth_sendTransaction` in ganache-cli).



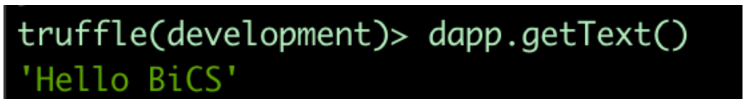
```
truffle(development)> dapp.setText("Hello BiCS")
{
  tx: '0x77a2b17ab7f7992d1caa2b2318f4c6118dadeb9d026e9944f458e9d9b670a071',
  receipt: {
    transactionHash: '0x77a2b17ab7f7992d1caa2b2318f4c6118dadeb9d026e9944f45
```

By default, the author of the transaction is the first account (account[0]) created by ganache-cli. To specify the author of the transaction, use for example
`dapp.setText("Hello BiCS", { from: accounts[1] })`

You should see that the ETH balance has dropped slightly.
In another terminal where web3 is initialised, with node.js started:

```
web3.eth.getBalance("0xb8042F15595652C88d88773Dd27c797868ef6537").then(console.log);
```

Now we try again the getText() function:



```
truffle(development)> dapp.getText()
'Hello BiCS'
```

[1]To interact with your smart contract functions directly with node.js and web3, see appendix.

Project

Build a smart contract that allows the different accounts to vote.

It is possible based on the simple interaction functions presented here and with private and public variables in the smart contract.

A simple array works

```
[name1, #votes1 ]
[name2, #votes2 ]
[name2, #votes2 ]
…
```

but your are free to choose your format and what are the votes about, i.e. the possible options.

A user must be able to vote for an option by authoring a transaction to the contract, with the appropriate function. Voting, as it modifies the variables in the smart contract, should cost ETH. When the user votes, the vote function should return and print a log in the terminal of the user such as `address XXX voted for option XXX`.

A user must be able to display the different vote options with a call function, which is free.

A user must be able to display the results of the vote. However results must be encrypted on the blockchain (i.e., `#votes1` is not an integer). Only one user must be able to decrypt the results with a passphrase for example using an additional function of the smart contract `getResults(passphrase)`.

A user is allowed to vote only once.

For your tests, you can put your code in a file.js and run it with `truffle exec file.js`

Question: Is the voting procedure transparent (i.e., anyone can see what an account is voting for)?

To go further: the vote is open for a limited amount of time, e.g. for 20 minutes after the smart contract is deployed.

Appendix

We can directly interact with the deployed smart contract from a terminal with node.js started.
In the FirstContract.json file with the code compiled, there is the application binary interface (ABI) of the smart contract.
Retrieve the ABI of your smart contract FirstContract with

```
const fs = require('fs');
const contract = JSON.parse(fs.readFileSync('./build/contracts/
FirstContract.json', ‘utf8'));
```

Create the smart contract object with

```
var FirstContract = new web3.eth.Contract(contract.abi,
'0x321BC6AA86Fe403C6c6b311074Be7340F9762530');
```

Where the second argument is the address of your smart contract, that you can find also in the FirstContract.json file.

Now, you can call the getText() function with

```
FirstContract.methods.getText()
     .call(function(error, result){
          console.error(error); // optional
          console.log(result);
     });
```

You can create a transaction from an account to call the setText() function with

```
FirstContract.methods.setText(“my other text")
  .send({ from: '0xb8042F15595652C88d88773Dd27c797868ef6537' })
  .then(receipt => { console.log(“ok")
 });
```

Where the address is the author of the transaction.

NOTE
The above code works with Web3 1.0.x beta, which you normally should have.
If you have Web3 0.x.x released, try the following code after parsing the JSON file

```
var FirstContract = new web3.eth.Contract(contract.abi);
var contractInstance = FirstContract.at(‘___the address of the contract__’);
contractInstance.getGreeting(
  { from: ___the address of the author___, gas: 100000000000000 },
  (err, res) => { console.log("ok") }
);
```