

Ethereum Smart contracts development

With Javascript (2022)



Andrea Lisi, andrealisi.12lj@gmail.com



Part 2

The Web3 library

An interface to interact with smart contracts





Web3

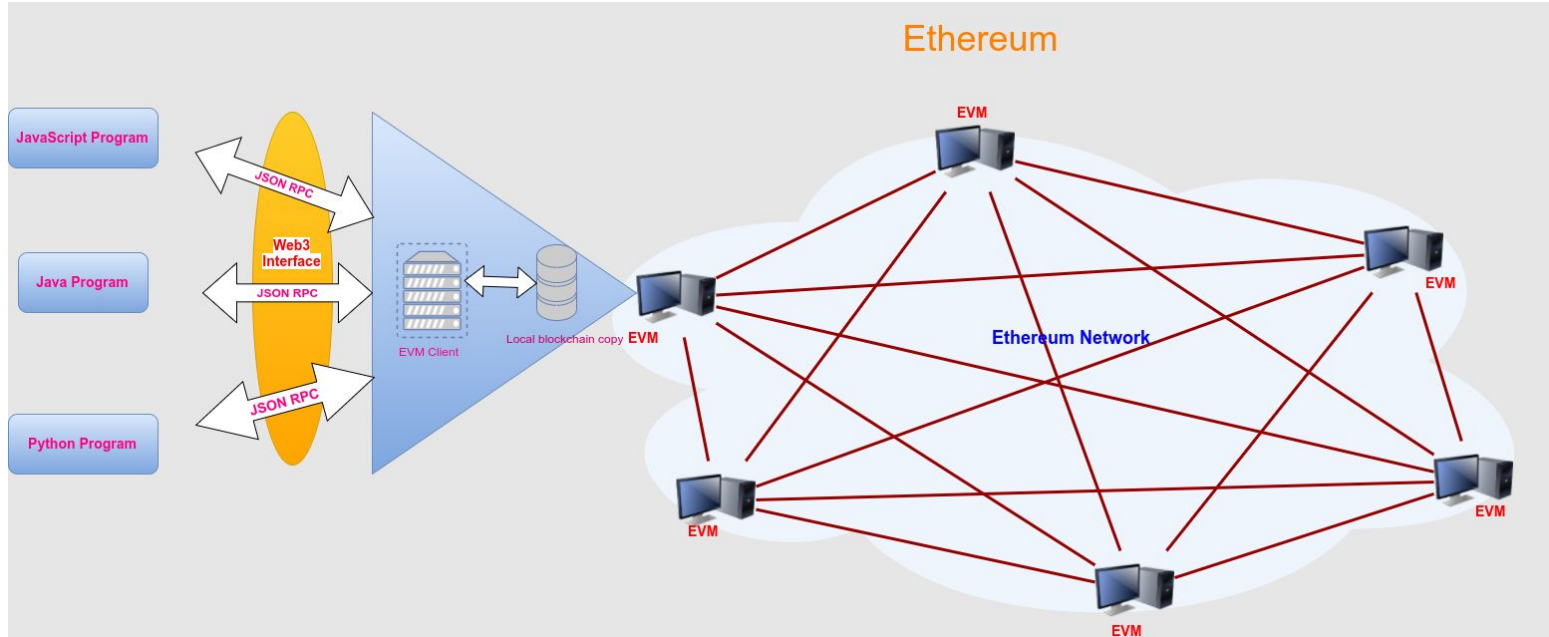
Web3 is a library to interact with the Ethereum network nodes with the RPC protocol, Remote Procedure Call

- Communications are asynchronous

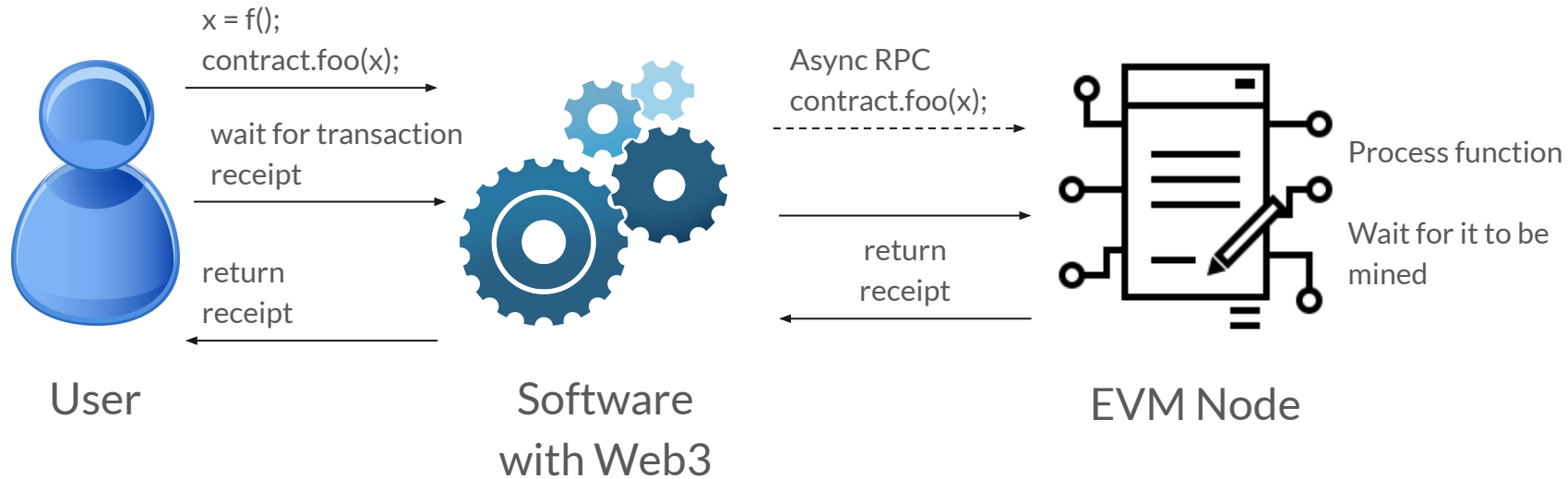
Software importing Web3 are able to communicate with smart contracts

Web3

Src: <http://www.dappuniversity.com/articles/web3-is-intro>



Web3





Web3 implementations

[W1] web3Js: JavaScript

[W2] web3J: Java

[W3] web3py: Python

[W4] web3.php: Php

[W5] hs-web3: Haskell





NodeJs and Npm

In this tutorial we are going to use an environment based on Javascript
We need **NodeJs** and **Npm** (Node Package Manager)





Requirements: NodeJs

NodeJs is an environment to execute Javascript code on your machine instead on the browser:

- Write server-side Javascript code
- Modern frameworks for web development (ReactJs, AngularJs etc...)
- And Javascript desktop applications (ElectronJs)
- Install NodeJs
 - <https://nodejs.org/en/docs/>



Requirements: Npm



Npm (Node Package Manager) is the tool to install NodeJs packages

- Local packages are installed in the `./node_modules/` directory
 - Libraries and utilities for a single project
- Global packages are all installed in a single folder in your system
 - CLI tools to be reused among many projects
- It is installed with NodeJs
 - <https://www.npmjs.com/get-npm>
 - <https://docs.npmjs.com/>



References, Web3

[W1] Web3Js: <https://github.com/ethereum/web3.js>

[W2] Web3J: <https://github.com/web3j/web3j>

[W3] Web3Py: <https://github.com/ethereum/web3.py>

[W4] Web3.php: <https://github.com/sc0Vu/web3.php>

[W5] hs-Web3: <https://github.com/airalab/hs-web3>



Part 3

Truffle framework

A framework for the development of smart contracts





Walkthrough

1. Introduce the Truffle framework
2. Init a new Truffle project
 - a. Project structure
 - b. Configuration file
3. Development
 - a. Coding
 - b. Compiling
 - c. Testing
 - d. Migrating



1. The Truffle Framework

Truffle is a CLI framework providing developers sweet tools for Ethereum smart contracts development. In particular:

- The Solidity **compiler**;
- A **migration** tool to deploy contracts to an Ethereum network;
- A **testing** environment;

- A NodeJs **console** to interact with the migrated contracts;
- An **execution** tool to automate commands inserted in the console



1. Installing Truffle

Truffle can be installed with npm

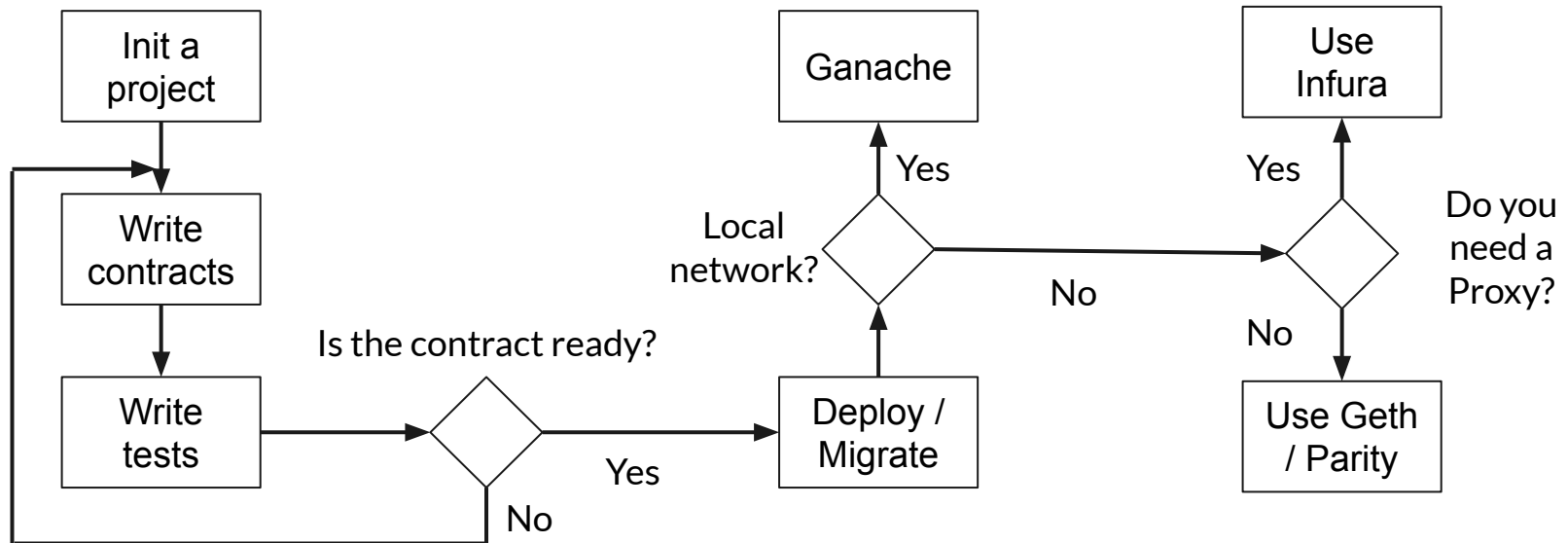
Requirements: **NodeJs** (v8.9.4 or later) and **npm**

Truffle, global installation:

```
$ (sudo) npm install -g truffle
```

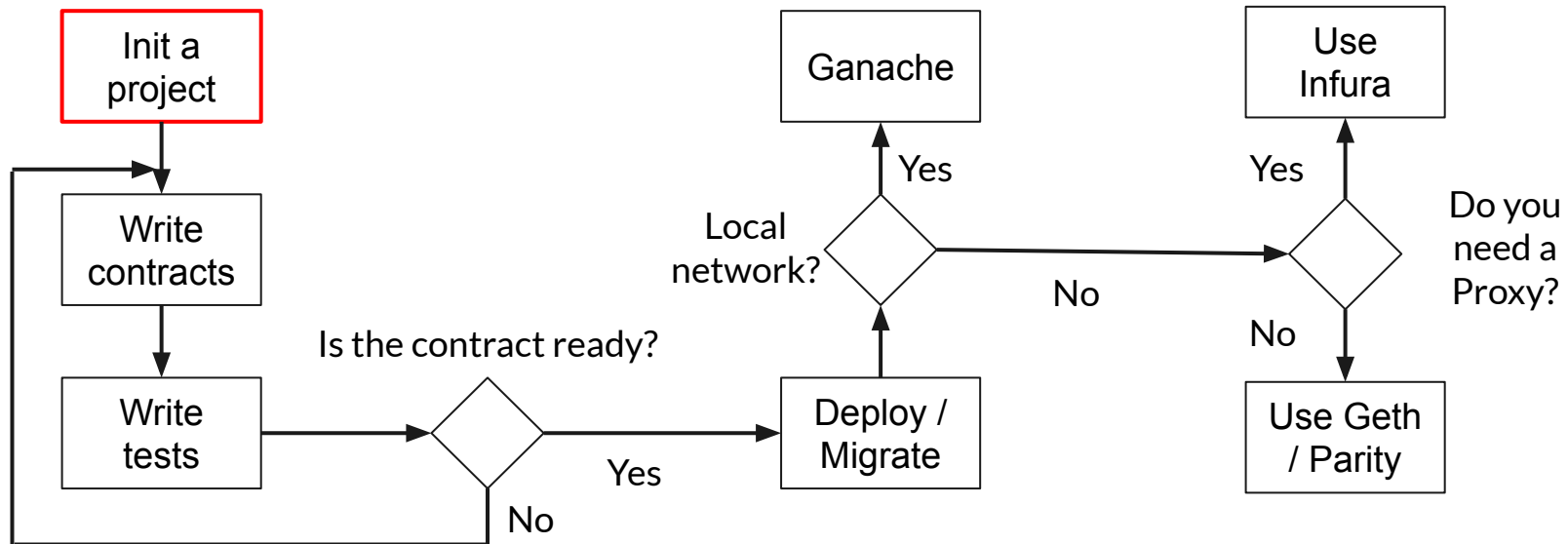


2. Truffle: workflow





2. Truffle: create a project





2. Truffle: create a project

```
$ truffle init
```

Initialize an empty Ethereum project in the current folder structured as:

- contracts/
- migrations/
- test/
- truffle-config.js



2. Truffle: create a project

```
$ truffle init
```

Initialize an empty Ethereum project in the current folder structured as:

- **contracts/** : contains the Solidity smart contracts
- migrations/
- test/
- truffle-config.js



2. Truffle: create a project

```
$ truffle init
```

Initialize an empty Ethereum project in the current folder structured as:

- contracts/
- **migrations/** : JS code that Truffle executes during the deployment
- test/
- truffle-config.js



2. Truffle: create a project

```
$ truffle init
```

Initialize an empty Ethereum project in the current folder structured as:

- contracts/
- migrations/
- **test/** : JS code that Truffle executes to test the contracts
- truffle-config.js



2. Truffle: create a project

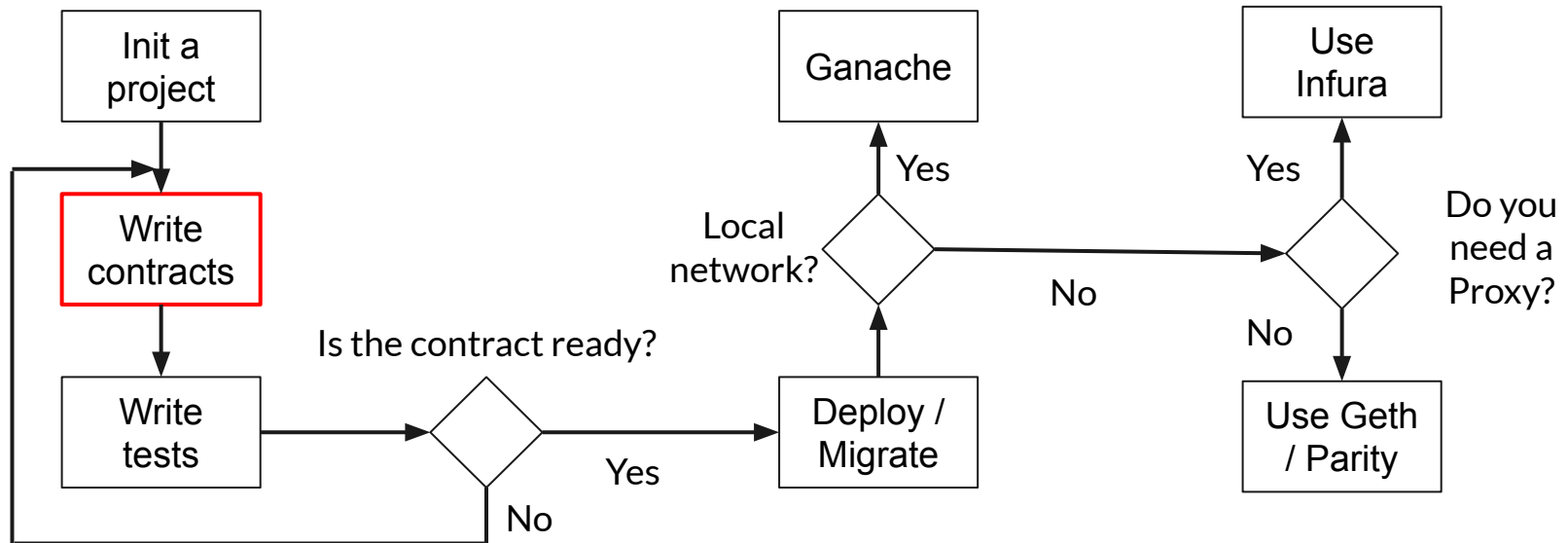
```
$ truffle init
```

Initialize an empty Ethereum project in the current folder structured as:

- contracts/
- migrations/
- test/
- **truffle-config.js** : JS Truffle configuration file



3.a. Truffle: write contracts





3.a. Example contract



```
// ./contracts/MyContract.sol
contract MyContract {
    uint public value;

    constructor() public {
        value = 1;
    }
    function increase(uint _v) public {
        value = value + _v;
    }
    function get_square() public view returns (uint) {
        return value * value;
    }
}
```



3.b. Truffle: compilation

We save our contract in the *contracts/* folder

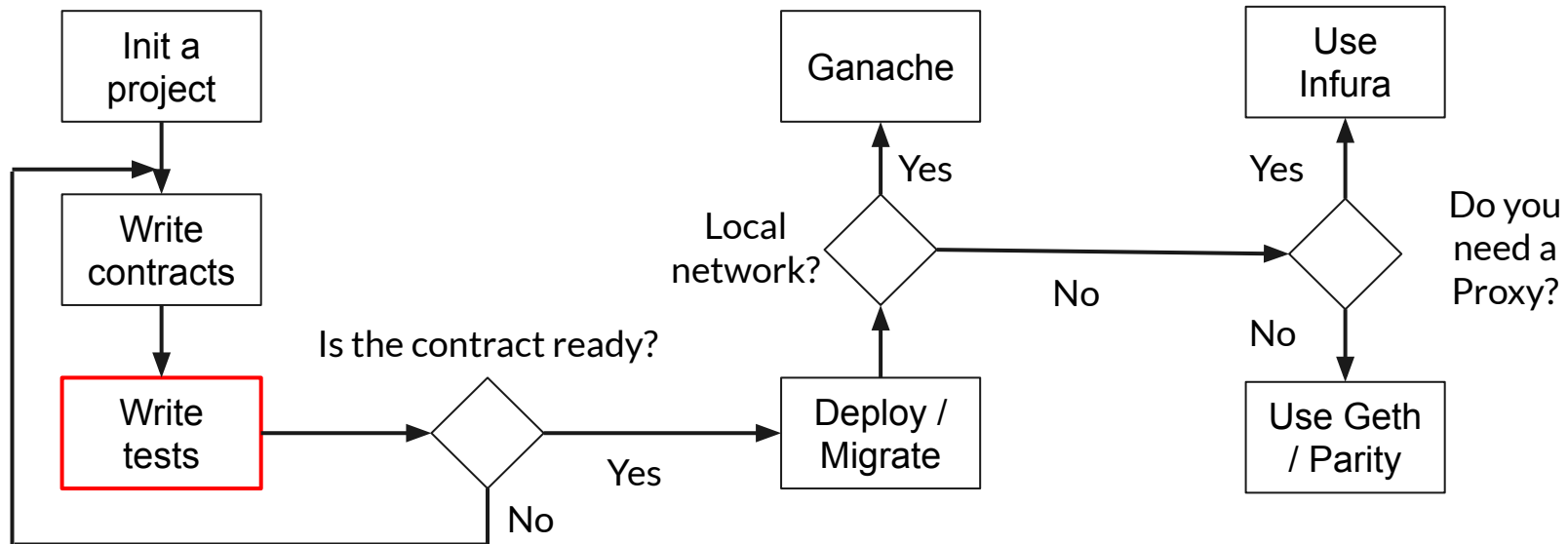
We can **compile** the smart contract(s) in the *contracts/* folder with the following command:

```
$ truffle compile
```

It creates the *build/* folder with the results of the compilation in .json format, including the ABI (Abstract Binary Interface) and the bytecode



3.c. Truffle: testing





3.c. Truffle: testing

Truffle uses the [Mocha](#) testing framework, and the [Chai](#) assertion library

Extra assertion commands are provided by **truffle-assertions** npm package

- Test events parameters and reverted executions
- <https://www.npmjs.com/package/truffle-assertions>



3.c. Truffle: testing

How are tests executed?

We can use a local network (blockchain) for testing

- Truffle can create one on the fly while running the tests
 - Solution adopted now (no configs are needed)
- Truffle can connect to an instance running in the background
 - More on that on the “Migration” slides



3.c. Truffle: testing

A testing file requires to:

- Create a file *test_mycontract.js* inside the *test/* folder
- Import the compiled smart contract from the *build/* folder
- Code testing scripts
 - Web3 calls to smart contract functions are **asynchronous**, which are implemented by **Promise** objects in Javascript



3.c. Truffle: testing

```
// ./test/test_mycontract.js
// Import the contracts to test in Truffle from build/
  // MyContract is a template, not an instance (e.g. the Class, not the object)
const MyContract = artifacts.require("MyContract"); // ./build/MyContract.json
// Create a testing environment
// accounts are Ethereum accounts, injected by Truffle. More on that later
contract("Testing MyContract", accounts => {
  // Create a test
  it("Should test the constructor", function() {
    // 1 Create a known state
    // 2 Execute the operation to test
    // 3 Test the conditions
  }); });
```



3.c. Javascript promises

When you call a smart contract function with Web3 you get as result a Javascript Promise object:

- `const promise = contract.function_name(params);`

When a Promise is completed, we can use its result within the *then* statement:

- `promise.then((result) => { // use your result here })`

More on [Javascript Promises](#)



3.c. Truffle: testing

```
// ./test/test_mycontract.js
it("Should test the constructor", function() {
  // 1 Create a known state: not here, we test the constructor
  // 2 Execute the operation to test, i.e. the constructor
  return MyContract.new() // Create a new contract
  .then(instance => {
    // "instance" is a constructed instance of MyContract (i.e. the object)
    // 3 Test the condition: check if "value" is 1
    return instance.value().then(v => {
      // "v" is the result of value(): solidity uint256 are BigNumber objects
      assert.equal(v.toNumber(), 1, "Value should be initialized at 1");
    });
  });
}); // end return MyContract.new }); // end it()
```



3.c. Truffle: testing with async/await

To lighten the code it is possible to use the `async / await` syntax. This results to a complete synchronous code, but it is enough for our goals that do not require heavy asynchronous programming

```
it("Should test the constructor with async/await", async function() {  
    // 1 Create a known state: not here, we test the constructor  
    // 2 Execute the operation to test, i.e. the constructor  
    const instance = await MyContract.new();  
    // 3 Test the condition: i.e. if value is 1  
    const v = await instance.value();  
    assert.equal(v.toNumber(), 1, "Value should be initialized at 1");  
});
```




3.c. Truffle: testing

After a test has been written:

- Execute ***all*** the test files inside test/ folder with:

```
$ truffle test
```
- Execute ***a single*** test file inside test/

```
$ truffle test test/test_mycontract.js
```



3.c. Smart contract return values



When executed with Web3, functions which are labeled as **view / pure** return the value of the Solidity return statement. Such functions won't be put in a block (mined) and they do not cost a fee to the caller

On the other hand, non-view/pure functions are so called within **transactions**, meaning that they will be mined, put in a block and they do cost a fee. The return value IS NOT the value expressed by the Solidity return statement, but a **transaction**



3.c. Our contract (modified)



```
contract MyContract {  
  
    uint public value;  
  
    constructor() public {  
        value = 1;  
    }  
    function increase(uint _v) public returns (uint) {  
        value = value + _v;  
        return value; // new  
    }  
    function get_square() public view returns (uint) {  
        return value * value;  
    }  
}
```



3.c. Truffle: testing transactions

The `assert.equal()` condition should fail because `result` is not the integer as expected by the Solidity code, but it is an object, i.e. the transaction with information like block number, gas used etc...

```
it("Should increase the value by 41", async function() {
  // 1 Create a known state
  const instance = await MyContract.new();
  // 2 Execute the operation to test
  const result = await instance.increase(41);
  // 3 Test the condition
  assert.equal(result.toNumber(), 42, "The result should be 42");
});
```




3.c. Smart contract return values

How can I get the return value of a transaction?

Ideas:

- Emit an event, with the result the argument of the event
 - Events and their data are contained in the **logs** field of the transaction object
- Call a view function right after the transaction (free of gas)

More information on

<https://truffleframework.com/docs/truffle/getting-started/interacting-with-your-contracts>



3.c. Smart contract return values

Note

If another smart contract calls the *increase()* function gets the return value as expected

```
contract Incrementer {
    uint public number;

    function increase() public {
        number = number + 1;
        return number;
    }
}
```

```
contract MyContract {
    function callContract(Incrementer
        _contract) public {
        value = value + _contract.increase();
    }
}
```



3.c. Truffle: testing, conclusions

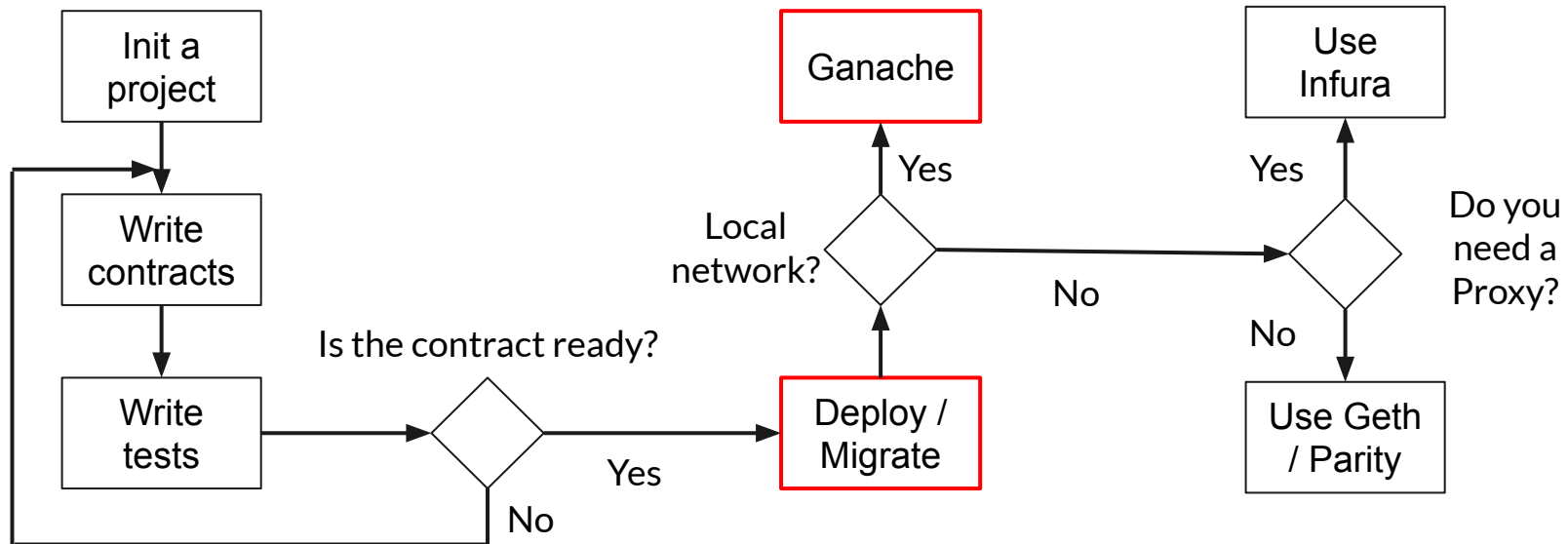
Testing smart contracts with Truffle can be tricky at the beginning , but it is very straightforward once understood how it works

Visit the Mocha and Chai pages for more documentation

So far we have seen how to test smart contracts without providing any network (blockchain)



3.d. Truffle: migrating





3.d. Truffle: migrating

As soon we feel confident of our contracts we can migrate (deploy) them to a target network

To migrate our contracts we need to:

- Write in our *truffle_config.js* file the target network settings
- Write our migration script
 - In this script we decide which contracts we migrate
- Execute the migration script with the command provided by Truffle

3.d. Truffle: migrating



We can start with a local (simulated) network for the development

The Truffle suite provides Ganache

- The CLI version: <https://github.com/trufflesuite/ganache>
 - Note: before 2022 the CLI version used to be called ganache-cli
- The GUI version: <https://www.trufflesuite.com/ganache>

Ganache CLI, how it looks like

```
File Edit View Search Terminal Help
allc@allc:~/Documents/PHD/SupportP2P2020/lezioni/example$ ganache-cli
Ganache CLI v6.5.0 (ganache-core: 2.6.0)

Available Accounts
=====
(0) 0x015cc24d96037a9130acdfcd113f3dad398273ca (~100 ETH)
(1) 0x183834fcc483997b53cf5fd644c96fc6110d7f0c (~100 ETH)
(2) 0xd1d9e7382ba1a0a10fc77499715937d234a57550 (~100 ETH)
(3) 0x3a0b1a43dae7c66a59b29427bc4fbc7b106f4b6e (~100 ETH)
(4) 0x0eb8274f5a09c933a5fa8b793c9feac19b1d6e2c (~100 ETH)
(5) 0x27a19ff14eb9a29ec800254b42849fa4135b50a3 (~100 ETH)
(6) 0x2886f438aa9d2594843618b721b6c1741871e91b (~100 ETH)
(7) 0x08bf94b966897fb2a3d0212104b2eba30fb470f6 (~100 ETH)
(8) 0xb9f97e6baa0a22cb7387adbe1f330a5aec6b95aa (~100 ETH)
(9) 0x4b4db6b8101ca0a8352664f72d7e8ede941b6dc1 (~100 ETH)

Private Keys
=====
(0) 0x59ec464379dfc2c0b67da8d55657ca5be653b16ac98ffe07094631322c84b489
(1) 0x32b07708eee00e9086e0718d05b92d55e749c081b88f3c01b1143c812701eeeb
(2) 0x69684baefb82b558a959752c9f405cb2d018fb15e3910333cf3e21475fbb8ff
(3) 0x7b6294a1aa37ecd8c996ff4224646a9455d22b71f2c2165e0578756cde2f307c
(4) 0x0af8fa44d16c95fd83c01f1b0d7173d3d7740c69e02d804495c5340fa859d67e
(5) 0x21f3a73c9369fdd4939ea22e03a49d6c227328308bbf6cca1580df5b1a327c78
(6) 0xb3f26374bf86a81ab5efa6a9a483b34ae84497db17f1a979af05f8ca0783abad
(7) 0xf94e23db254fe7bebf08dceba7839a6279da04dc73116864ce4bf120414b244
(8) 0xbe52cffd30b13f0c35d3512f26f763ec3f651f027bb34278c1eab5ba38e5a33c
(9) 0x09469843de48e4e3fa7d30c148e63d71f203928b167f018b5d3397320fde190

HD Wallet
=====
Mnemonic:      uncover absent point off reduce scorpion world small table reveal wide siren
Base HD Path:  m/44'/60'/0'/0/{account_index}

Gas Price
=====
20000000000

Gas Limit
=====
6721975

Listening on 127.0.0.1:8545
```



Ganache GUI, how it looks like



Ganache

ACCOUNTS BLOCKS TRANSACTIONS LOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK 0	GAS PRICE 20000000000	GAS LIMIT 6712390	NETWORK ID 5777	RPC SERVER HTTP://127.0.0.1:7545	MINING STATUS AUTOMINING
--------------------	--------------------------	----------------------	--------------------	-------------------------------------	-----------------------------

MNEMONIC
candy maple cake sugar pudding cream honey rich smooth crumble sweet treat

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX	
0x627306090abaB3A6e1400e9345bC60c78a8BEf57	100.00 ETH	0	0	
0xf17f52151EbEF6C7334FAD080c5704D77216b732	100.00 ETH	0	1	
0xC5fdf4076b8F3A5357c5E395ab970B5B54098Fef	100.00 ETH	0	2	
0x821aEa9a577a9b44299B9c15c88cf3087F3b5544	100.00 ETH	0	3	
0x0111111111111111111111111111111111111111	100.00 ETH	0	4	

45

3.d. Truffle: migrating, config



We need to tell Truffle to target our Ganache network

In our configuration file *truffle_config.js*:

- We create an identifier for our **network** called “development”
- Ganache **host** is local host, i.e. “127.0.0.1”
- Ganache **port** can be set, assuming “8545”
- In this case **network_id** can be anything, represented with “*”

During initialization Truffle creates a pre-compiled config file, all commented. Therefore we need only to un-comment the interested portions

3.d. Truffle: migrating, config



development is a special name for a network. When targeting a network with a Truffle command “development” can be omitted

The fields in *truffle_config.js* are shown below:

```
networks: {
  development: {           // Network name
    host: "127.0.0.1",     // Localhost (default: none)
    port: 8545,            // Ethereum port (default: none)
    network_id: "*",      // Any network for local (default: none)
  },
}
```

3.d. Truffle: migrating, scripts



The migration scripts are placed in the *migrations/* folder

Truffle executes the scripts in that folder using a lexicographic order. Typically these scripts are called `1_***.js`, `2_+++js`

This folder contains by default the `1_initial_migration.js` script, which migrates the `Migration.sol` contracts, useful for Truffle

3.d. Truffle: migrating, scripts



The default `1_initial_migrations.js` file:

```
// Import the contracts to migrate from build/  
const Migrations = artifacts.require("Migrations"); // ./build/Migrations.json  
  
// The function to execute during the migration  
module.exports = function(deployer) {  
  // Deploy the Migrations contract, i.e. an instance of Migration on the target network  
  // This command executes the constructor. If Migration would have had parameters  
  // in its constructor, they should have been as following arguments of deploy()  
  deployer.deploy(Migrations);  
};
```

3.d. Truffle: migrating



After being sure that Ganache is running, we can execute our scripts in *migrations/* with:

```
$ truffle migrate --reset --network development
```

- `--network "net"` specifies the target network named *net*
 - *development* is the default one, therefore the "`--network development`" option can be omitted
- Truffle does not re-migrate up-to-date contracts
 - `--reset` forces Truffle to migrate all the contracts
- You should see the balance of the first account decreased by a little

3.d. Truffle: migrating, scripts



We can modify `1_initial_migrations.js` to migrate also our contract:

```
const Migrations = artifacts.require("Migrations");
const MyContract = artifacts.require("MyContract");

module.exports = function(deployer) {
  deployer.deploy(Migrations);
  deployer.deploy(MyContract);
};
```

3.d. Truffle: migrating, scripts



We can get the network name, in case we have many, to filter execution:

```
const Migrations = artifacts.require("Migrations");
const MyContract = artifacts.require("MyContract");

// These inputs are injected by Truffle
module.exports = function(deployer, network) {
  deployer.deploy(Migrations);
  if(network == "development") {
    deployer.deploy(MyContract);
  }
};
```

3.d. Truffle: migrating, scripts



We can get the accounts of our target network:

```
const Migrations = artifacts.require("Migrations");
const MyContract = artifacts.require("MyContract");

// These inputs are injected by Truffle
module.exports = function(deployer, network, accounts) {
  deployer.deploy(Migrations);
  if(network == "development") {
    deployer.deploy(MyContract, {from: accounts[1]}); // Use your second account to deploy
  }
};
```

3.d. Truffle: migrating, scripts



Do not forget that these functions return Promises:

```
const Migrations = artifacts.require("Migrations");
const MyContract = artifacts.require("MyContract");

// These inputs are injected by Truffle
module.exports = async (deployer, network, accounts) => {
  await deployer.deploy(Migrations);
  if(network == "development") {
    const instance = await deployer.deploy(Migrations, {from: accounts[2]});
    // Do stuff with instance...
  }
};
```



3.d. Truffle: migrating, conclusions

Migrating means deploying a contract to a target network. This network is specified in the *truffle_config.js* file

An example of local network is Ganache



3.d. Truffle: migrating, conclusions

Warning

When exposing the “development” network the “on the fly” blockchain as explained in 3.c. *Truffle: testing* does not work anymore, and if the target network is not running Truffle will complain

This because omitting the `--network` flag Truffle uses “development” by default, **only if exposed**



Truffle: conclusions

Truffle eases the workflow to develop, test and migrate smart contracts

The suite provides other tools we didn't see, like its console and a way to execute scripts within the Truffle environment

- Type `$ truffle help` to see the list of available cmds

If the smart contracts are small, and few checks are required, then Remix is enough



Truffle: conclusions

Installing Truffle you install also the Solidity compiler and Web3Js

During the execution of Truffle cmds Web3Js injected by Truffle, and so there is no need to import the library

But many examples with Truffle use the Web3 wrapper **truffle-contract**

- It makes calling smart contract functions more intuitive
- All the examples in the Truffle page use this wrapper
- <https://github.com/trufflesuite/truffle/tree/master/packages/contract>



Truffle: docs

Config file: <https://truffleframework.com/docs/truffle/reference/configuration>

Compilation: <https://truffleframework.com/docs/truffle/getting-started/compiling-contracts>

Testing with Js: <https://truffleframework.com/docs/truffle/testing/writing-tests-in-javascript>

Migration: <https://truffleframework.com/docs/truffle/getting-started/running-migrations>

And online tutorials...



Extra

Other ways to interact with the smart contracts





Retrieve contract instances

How to get an instance of a contract:

- `.new()`, `.deployed()` and `.at()`

```
it("Should retrieve the instance of a contract", async function() {
  const address = "0x001d3...f1f086ba0f9"; // A contract address
  const _new = await MyContract.new(); // Create a new contract, return the instance
  const last = await MyContract.deployed(); // Get the *last* deployed instance of
  MyContract (typically used in combination con truffle migrate)
  const that = await MyContract.at(address); // Get the *deployed* instance of
  MyContract with address "address"
});
```



Transaction parameters

In Solidity we get the special constructs `msg.value`, `msg.sender`, etc

- `msg.sender` is the account invoking the function
- When omitted the default account is `[0]`

```
it("Should send Ether to a payable function", async function() {  
  
    const instance = await MyContract.new();  
    const tx = await instance.foo(41); // default account is accounts[0]  
  
});
```

<https://www.trufflesuite.com/docs/truffle/getting-started/interacting-with-your-contracts#making-a-transaction>



Transaction parameters

In Solidity we get the special constructs `msg.value`, `msg.sender`, etc

- `msg.sender` is the account invoking the function
- Otherwise it can be specified with a special last-param object

```
it("Should send Ether to a payable function", async function() {
  const alice = accounts[3];
  const instance = await MyContract.new();
  const tx = await instance.foo(41, {from: alice});
});
```

<https://www.trufflesuite.com/docs/truffle/getting-started/interacting-with-your-contracts#making-a-transaction>



Transaction parameters

In Solidity we get the special constructs `msg.value`, `msg.sender`, etc

- `msg.value` must be provided, if required
- It is a field of the special object

```
it("Should send Ether to a payable function", async function() {
  const alice = accounts[3];
  const instance = await MyContract.new();
  const tx = await instance.foo(41, {from: alice, value: 10000000}); // wei
});
```

<https://www.trufflesuite.com/docs/truffle/getting-started/interacting-with-your-contracts#making-a-transaction>



Manual interaction

Truffle provides two Javascript consoles to manually interact with your contracts:

- **Console:** Connects to a network like Ganache (or also the Ethereum main and test networks) and you interact with it
- **Develop:** Similar, but it creates an network on-the-fly
- <https://www.trufflesuite.com/docs/truffle/getting-started/using-truffle-develop-and-the-console>



Automatic interaction

Otherwise is possible to execute automatically the operations we write in our console:

- Write a script file *script.js*
- Execute it inside the Truffle environment (and therefore with access to Web3 etc)
 - `truffle exec script.js`
- <https://www.trufflesuite.com/docs/truffle/getting-started/writing-external-scripts>



Troubleshooting

Typically errors have not a not clear code or message. Here a few hints:

- Be consistent with the compiler version both in the smart contracts (pragma) and in *truffle_config.js*. Type `truffle compile --list` to see the list of available compilers. Truffle automatically fetches the version you provide if not installed



Web3 in Python

For those who prefer Python to interact and test smart contracts web3py is easy to use

- You need to create your virtualenv and install web3
- You need to have a working network (Ganache is fine)
- You need to compile your contracts and get the ABI and Bytecode (either with Truffle or Remix)
- Write your Python code
- https://www.youtube.com/watch?v=SAi5rYFh7yw&list=PLS5SEs8ZftgVn38FOhXvLc0PoX_0hnJO9



Web3 in Python

```
import json
from web3 import Web3
# Init Web3
ganache_url = 'HTTP://127.0.0.1:8545'
web3 = Web3(Web3.HTTPProvider(ganache_url))
print("Is web3 connected: ", web3.isConnected())

# Get the data needed to create a contract
bytecode = "60806040..."
with open('Greeting.json') as json_abi:
    abi = json.load(json_abi)

# Now with the ABI and bytecode we can instantiate the Greetings contract
```



Web3 in Python

```
# Create a Greeting contract template. Greeter is "the Java class"
Greeter = web3.eth.contract(abi=abi, bytecode=bytecode)
# Call the constructor (async), you get only the transaction hash so far
# A non-view function needs to be invoked with .transact()
tx_hash = Greeter.constructor().transact()
print(web3.toHex(tx_hash))
# Wait for the transaction to complete, get the result receipt
tx_receipt = web3.eth.waitForTransactionReceipt(tx_hash)
print(tx_receipt) # receipt

# Get the contract instance reference (i.e. the "Java object")
contract = web3.eth.contract(abi=abi, address=tx_receipt.contractAddress)
# new_contract is the contract instance and you can finally call its functions
```



Web3 in Python

```
# Call functions
# greet() is a view function, it can be invoked with .call()
print("Contract greet: ", contract.functions.greet().call())

# setGreeting(string) is a transaction, it can be invoked with .transact() as the
# constructor
tx_hash = contract.functions.setGreeting("Hola").transact()
web3.eth.waitForTransactionReceipt(tx_hash)

# calling again greet() should return a different result
print("Contract greet: ", contract.functions.greet().call())
```