

Lesson 8 - Development Tools

Most popular tools

Remix

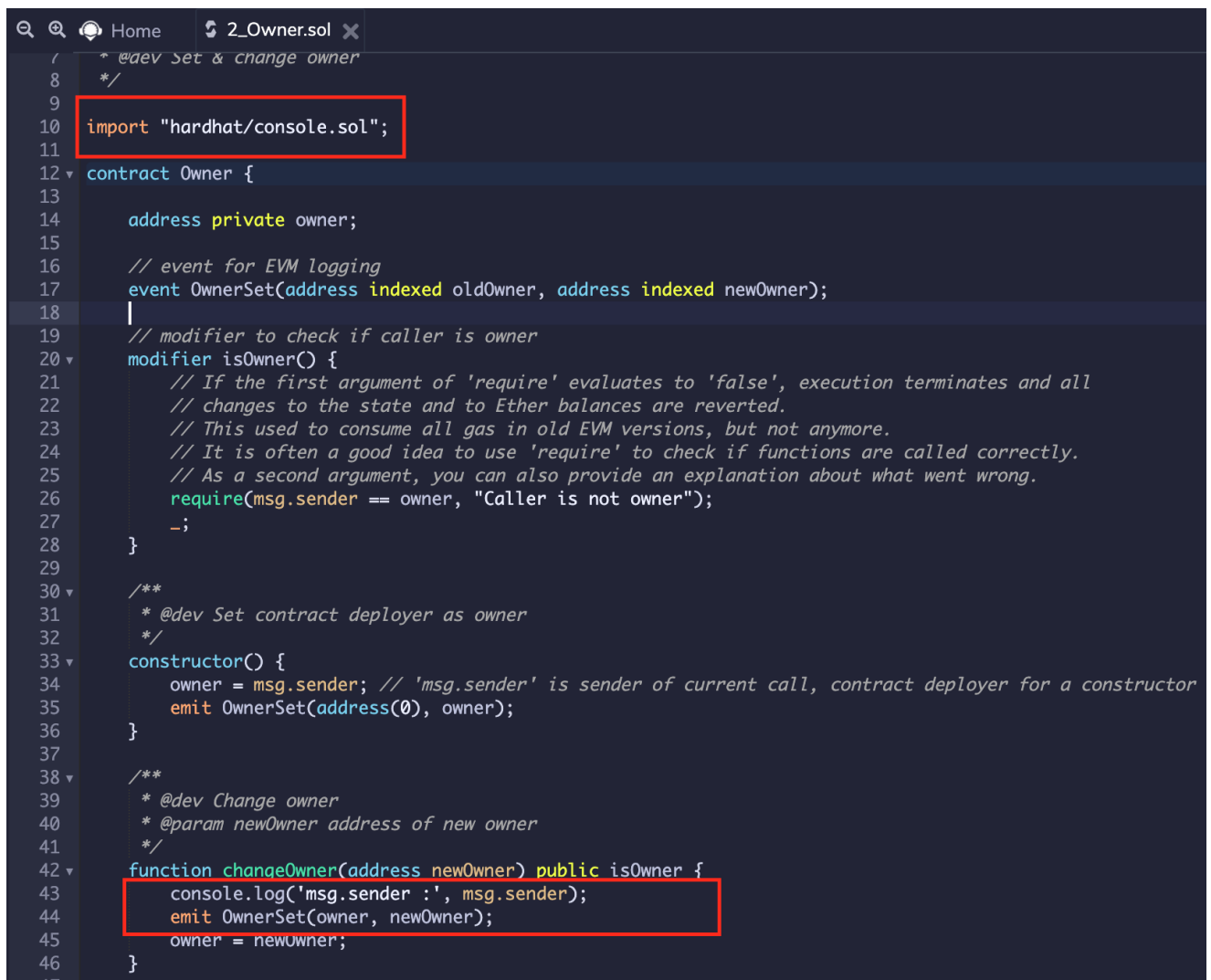
Browser : <https://remix.ethereum.org/>

Desktop : <https://github.com/ethereum/remix-desktop/releases>

or `npm install -g @remix-project/remixd`

Remix [documentation!](#)

Remix has added more functionality recently, one useful feature is the ability to add log statements to your contract code.



```
7  /* @dev Set & change owner */
8  */
9
10 import "hardhat/console.sol";
11
12 contract Owner {
13
14     address private owner;
15
16     // event for EVM logging
17     event OwnerSet(address indexed oldOwner, address indexed newOwner);
18
19     // modifier to check if caller is owner
20     modifier isOwner() {
21         // If the first argument of 'require' evaluates to 'false', execution terminates and all
22         // changes to the state and to Ether balances are reverted.
23         // This used to consume all gas in old EVM versions, but not anymore.
24         // It is often a good idea to use 'require' to check if functions are called correctly.
25         // As a second argument, you can also provide an explanation about what went wrong.
26         require(msg.sender == owner, "Caller is not owner");
27     }
28
29     /**
30     * @dev Set contract deployer as owner
31     */
32     constructor() {
33         owner = msg.sender; // 'msg.sender' is sender of current call, contract deployer for a constructor
34         emit OwnerSet(address(0), owner);
35     }
36
37     /**
38     * @dev Change owner
39     * @param newOwner address of new owner
40     */
41     function changeOwner(address newOwner) public isOwner {
42         console.log('msg.sender :', msg.sender);
43         emit OwnerSet(owner, newOwner);
44         owner = newOwner;
45     }
46 }
```

Remix also contains some tutorials via the [Learneth plugin](#).

See Lesson 2 notes for more details about Remix.



Flexible. Extensible. Fast.

Ethereum development environment for professionals

Get started



Hardhat Advantages

- Ability console.log inside Solidity file to help with debugging.
- Provides smart contract stack traces to aid debugging.
- You can choose to use Truffle/Web3 or Waffle/Ethers, making it very versatile.
- Lots of useful plugins.
- Becoming more popular - see npm package downloads chart below.

Requirements

Node.js version 12 and above.

Install Open Zeppelin Libraries

```
npm install @openzeppelin/contracts
```

Installation

Steps for installing and using Hardhat:

1. `$ npm install -D hardhat`
2. `$ npx hardhat`

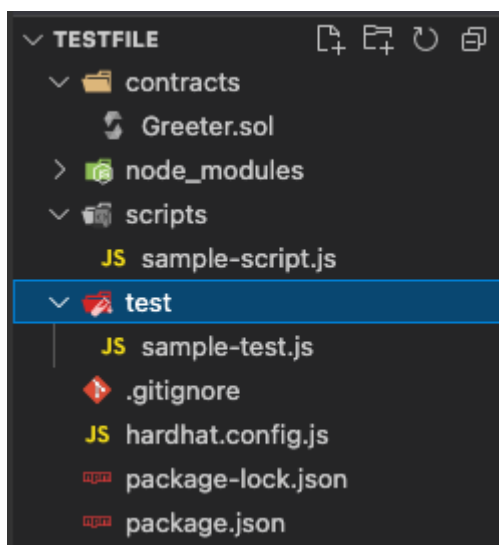
You should then see this in the terminal:

```
888 888 888 888 888
888 888 888 888 888
888 888 888 888 888
88888888888 8888b. 888d888 .d88888 88888b. 8888b. 888888
888 888 "88b 888P" d88" 888 888 "88b "88b 888
888 888 .d888888 888 888 888 888 .d888888 888
888 888 888 888 888 Y88b 888 888 888 888 Y88b.
888 888 "Y888888 888 "Y88888 888 888 "Y888888 "Y888

👤 Welcome to Hardhat v2.4.1 👤
? What do you want to do? ...
> Create a sample project
Create an empty hardhat.config.js
Quit
```

Choosing *Create a sample project* will:

- Create a contracts folder with a dummy contract 'Greeter.sol'
- Create a test folder with a sample test for the dummy contract.
- Create a scripts folder that contains the deployment script for the dummy contract.
- Installs node modules:
- @nomiclabs/hardhat-ethers
- @nomiclabs/hardhat-waffle
- And other necessary packages



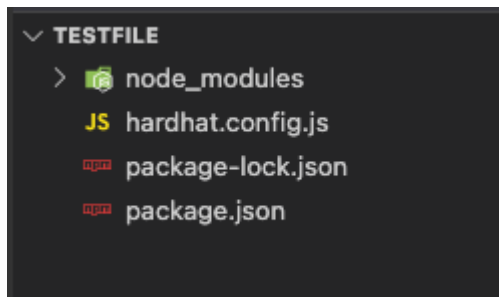
Choosing *Create and empty hardhat.config.js* will set up a new hardhat configuration file but without all of the dummy contracts, script files and tests. This method will not install any plugin (**ethers.js** / **Waffle**). If you want to use the **Web3.js** / **Truffle** plugin, this would be the options you would choose. If either of these sets of plugins are installed, you have to either put

```
require('@nomiclabs/hardhat-waffle')
```

or

```
require('@nomiclabs/hardhat-truffle5')
```

depending on the one that you are using.



After initializing hardhat , using `npx hardhat` again shows a list of commands:

```
Hardhat version 2.4.1
Usage: hardhat [GLOBAL OPTIONS] <TASK> [TASK OPTIONS]

GLOBAL OPTIONS:

  --config          A Hardhat config file.
  --emoji           Use emoji in messages.
  --help           Shows this message, or a task's help if its name is provided
  --max-memory      The maximum amount of memory that Hardhat can use.
  --network         The network to connect to.
  --show-stack-traces Show stack traces.
  --tsconfig        Reserved hardhat argument -- Has no effect.
  --verbose        Enables Hardhat verbose logging
  --version        Shows hardhat's version.

AVAILABLE TASKS:

  check          Check whatever you need
  clean          Clears the cache and deletes all artifacts
  compile        Compiles the entire project, building all artifacts
  console        Opens a hardhat console
  flatten        Flattens and prints contracts and their dependencies
  help           Prints this message
  node           Starts a JSON-RPC server on top of Hardhat Network
  run            Runs a user-defined script after compiling the project
  test           Runs mocha tests

To get help for a specific task run: npx hardhat help [task]
```

Example: Waffle & Ethers

dummyContract.sol

```

contracts > dummyContract.sol
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.0;
3
4  // imported ERC20 and Ownable contracts from the OpenZeppelin library.
5  import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
6  import "@openzeppelin/contracts/access/Ownable.sol";
7
8  /**
9   * @title A unique token that can be used in different context (eg: data or rental marketplace)
10  * @dev all the functions from the ERC20 tokens standard are available
11  * @author
12  */
13  contract DummyContract is ERC20("DummyToken", "DumTkn"), Ownable {
14      uint256 constant INITIAL_AMOUNT = 100;
15
16      constructor() {}
17
18      function setUp() external onlyOwner() {
19          _mint(msg.sender, INITIAL_AMOUNT);
20      }
21  }
22

```

We are going to use the same contract as used in the Truffle/Web3 example.

deploy_script.js

```

scripts > JS deploy_script.js > ...
1  const hre = require("hardhat");
2
3  async function main() {
4      // Hardhat always runs the compile task when running scripts with its command
5      // line interface.
6      //
7      // If this script is run directly using `node` you may want to call compile
8      // manually to make sure everything is compiled
9      // await hre.run('compile');
10
11     // We get the contract to deploy
12     const [deployer] = await ethers.getSigners();
13     console.log(`Deploying contracts with the account: ${deployer.address}`);
14
15     const balance = await deployer.getBalance();
16     console.log(`Account Balance: ${balance.toString()}`);
17
18     // We get the contract to deploy
19     const DummyContract = await hre.ethers.getContractFactory("DummyContract");
20     const dummyContract = await DummyContract.deploy();
21
22     console.log("Token deployed to:", dummyContract.address);
23 }
24

```

The file does the same thing as the *truffle_fixture.js* file in the Truffle/Web3 example. Essentially, it obtains the deployer information using **ethers.getSigners()** (which uses the first account in the list of generated accounts), then gets the balance of the deployer's

account and returns this information in the terminal. The contract is then deployed and the address is returned in the terminal.

The function `getContractFactory()`, which is called on ethers is an abstraction used to deploy new smart contracts. It is a special type of transaction called an initcode transaction. The contract bytecode is sent in the transaction, then it evaluates the code and allows you to create a new contract based upon that information. So in the example above, the `DummyContract` variable that the contract information is assigned to is sent through as the initcode. This then allows you to deploy an instance of that contract.

test.js

```
1  const { expect } = require("chai");
2  const { ethers } = require("hardhat");
3
4  describe("DummyContract", function () {
5    // Initialise variables
6    let DummyContract, dummyContract, owner, addr1, addr2;
7
8    beforeEach(async () => {
9      // Deploy a new instance of the contract
10     DummyContract = await ethers.getContractFactory("DummyContract");
11     dummyContract = await DummyContract.deploy();
12     // Get accounts and assign to pre-defined variables
13     [owner, addr1, addr2, _] = await ethers.getSigners();
14   });
15
16   describe("Deployment", () => {
17     it("Should be set with the Dummy Contract information", async () => {
18       // Failing test
19       expect(addr1.address).to.not.equal(await dummyContract.owner());
20       // Passing tests
21       expect(await dummyContract.owner()).to.equal(owner.address);
22       expect(await dummyContract.name()).to.equal("DummyToken");
23       expect(await dummyContract.symbol()).to.equal("DumTkn");
24     });
25   });
26 });
27
```

As you can see there is not much different between this test.js and the one used in the Truffle/Web3 project. The main difference comes in the `beforeEach` block. As explained in the `deploy-script.js` file, the `getContractFactory()` function allows the contract to be deployed, which is an extra step compared with Truffle and then the contract is deployed and assigned to the `dummyContract` variable. Then `getSigners()` is called which attributes account information (*public key, private key*) to each variable (*owner, addr1, addr2*). So the first address will be assigned to the owner, the second address to `addr1` and so on.

Specifying function caller and reverting

```

27 describe("setUp", () => {
28   it("Should not allow anyone but the owner to call", async () => {
29     await expect(() =>
30       dummyContract
31         .setUp({ from: addr1 })
32         .to.be.revertedWith("Ownable: caller is not the owner")
33     );
34   });
35
36   it("Should mint the initial amount to the contract owner", async () => {
37     const ownerBalanceBefore = await dummyContract.balanceOf(owner.address);
38     await dummyContract.setUp();
39     const ownerBalanceAfter = await dummyContract.balanceOf(owner.address);
40     expect(ownerBalanceAfter).to.equal(ownerBalanceBefore + 100);
41   });
42 });

```

When testing the `setUp()` function in the contract, we need to make sure that only the owner can call this. Therefore, we need to test what will happen when a different account tries to call it. We can do this in Truffle by specifying from inside of the function call (`{from: address}`). This tells the test to call the function from the address specified. We then check that the test is reverted and give the message that we expect to receive back. The method for doing the same thing using ethers.js is to use the connect method. The above example would now change to look like this.

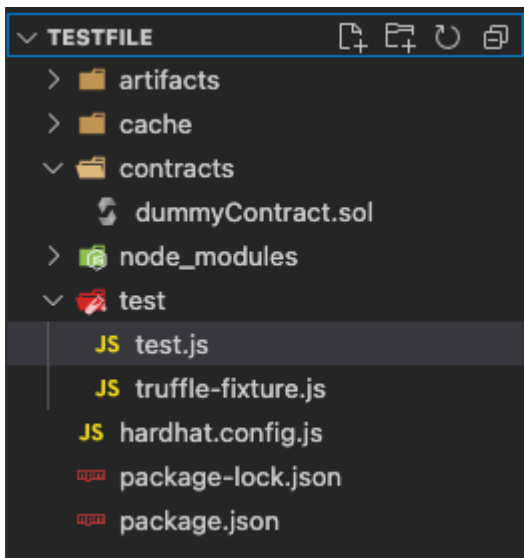
```

describe("setUp", () => {
  it("Should not allow anyone but the owner to call", async () => {
    await expect(() =>
      dummyContract
        .connect(addr1)
        .setUp()
        .to.be.revertedWith("Ownable: caller is not the owner")
    );
  });
});

```

Example: Truffle & Web3

Set up a new project in which we have a contract (*dummyContract.sol*), a test file (*test.js*) and a deployment script (*truffle-fixture.js*). The file structure can be seen below (artifacts and cache folders are created by Hardhat when you test the contract so these will not be there until you run `$ npm run hardhat test` for the first time).



dummyContract.sol

```
contracts > dummyContract.sol
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.0;
3
4  // imported ERC20 and Ownable contracts from the OpenZeppelin library.
5  import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
6  import "@openzeppelin/contracts/access/Ownable.sol";
7
8  /**
9   * @title A unique token that can be used in different context (eg: data or rental marketplace)
10  * @dev all the functions from the ERC20 tokens standard are available
11  * @author
12  */
13  contract DummyContract is ERC20("DummyToken", "DumTkn"), Ownable {
14      uint256 constant INITIAL_AMOUNT = 100;
15
16      constructor() {}
17
18      function setUp() external onlyOwner() {
19          _mint(msg.sender, INITIAL_AMOUNT);
20      }
21  }
22
```

Its a very basic contract that creates a new ERC20 coin (DummyToken). The only function it has is a setUp function that calls the mint function in the *ERC20 contract* that has been imported from the *OpenZeppelin library*, and mint the amount specified in the **INITIAL_AMOUNT** variable to the user that called the function. As we are utilising the *Ownable contract* for this function it will only allow the owner of the contract to call this function and therefore will only mint the initial amount to the owner's wallet address.

truffle-fixture.js


```

test > JS truffle-fixture.js > ...
1  //This file is used instead of a migrations file as the Truffle plugin does not fully support migrations yet.
2  //Instead, need to adapt the Migrations to become a hardhat-truffle fixture
3
4  const DummyContract = artifacts.require("DummyContract");
5
6  // Layout for hardhat deployment
7
8  module.exports = async () => {
9    const dummyContract = await DummyContract.new();
10   DummyContract.setAsDeployed(dummyContract);
11
12   console.log("Dummy Contract successfully deployed...");
13 };
14

```

As can be seen from the comments at the top of this file, using Hardhat is slightly different than Truffle when it comes to deployment. Normally there is a migrations file that would contain the deployment script for the contracts. However, **this feature has not yet been implemented in Hardhat**, so a truffle-fixture file has to be used instead. This requires in the contract(s) that you need to deploy, then creates a new version of that contract(s) and then sets it as deployed.

hardhat-config.js

```

JS hardhat.config.js > ...
1  /**
2   * @type import('hardhat/config').HardhatUserConfig
3   */
4  require("@nomiclabs/hardhat-truffle5"); // Truffle plugin
5  require("@nomiclabs/hardhat-ganache"); // Ganache plugin
6
7  // To test: npx hardhat test
8
9  module.exports = {
10   defaultNetwork: "ganache",
11   networks: {
12     ganache: {
13       url: "http://127.0.0.1:7545",
14       gasLimit: 6000000000,
15       defaultBalanceEther: 1000,
16     },
17   },
18   solidity: "0.8.0",
19 };
20

```

The hardhat-config.js file is where you require the **Truffle/Waffle** plugins. **There is no need to require Web3/Ethers** as Truffle/Waffle already do this .

In the example above, the Ganache plugin has also been required as this will spin up a local blockchain on the specified URL. You can also specify other parameters such as the default ETH balance of each user. This plugin automatically starts Ganache before running your tests and stops it after.

You can also achieve the same thing by starting Ganache manually and running the command `$ npx hardhat --network localhost test`

test.js

```
test > JS test.js > ...
1  const { assert, expect } = require("chai");
2  const DummyContract = artifacts.require("DummyContract");
3
4  contract("Dummy Contract", ([owner, user1]) => {
5      let dummyContract;
6
7      // Before any tests are run, deploy the contract to be tested.
8      before(async () => {
9          dummyContract = await DummyContract.deployed();
10     });
11
12     // Before each describe block is run.
13     beforeEach(async () => {
14         // This deploys a new version on the DummyContract so that everything is reset.
15         dummyContract = await DummyContract.new();
16     });
17
18     // Using Chai assert statements.
19     describe("Contract deployment", async () => {
20         it("Should know information about the contract", async () => {
21             // Failing test.
22             assert.notEqual(await dummyContract.owner(), user1);
23             // Passing test.
24             assert.equal(await dummyContract.owner(), owner);
25             assert.equal(await dummyContract.symbol(), "DumTkn");
26             assert.equal(await dummyContract.name(), "DummyToken");
27         });
28     });
29 });
30
```

This version of the tests are using Chai assert. The way you write tests in Hardhat is almost identical to Truffle.

To run the tests use command `$ npx hardhat test`

If you want to run a specific test file, specify the name after the above command (including the path) `$ npx hardhat test test/test.js`.

```
Contract: Dummy Contract
Dummy Contract successfully deployed...
Contract deployment
  ✓ Should know information about the contract (159ms)

1 passing (741ms)
```

Using Chai expect the tests would be written like this:

```
// Using Chai expect statements.
describe("Contract deployment", async () => {
  it("Should know information about the contract", async () => {
    // Failing test.
    expect(user1).to.not.equal(await dummyContract.owner());
    // Passing test.
    expect(await dummyContract.owner()).to.equal(owner);
    expect(await dummyContract.symbol()).to.equal("DumTkn");
    expect(await dummyContract.name()).to.equal("DummyToken");
  });
});
```

Web3.js

Web3 is automatically used when the Truffle plugin has been required in the hardhat-config.js file. Web3.js is available in the global scope. In the below example, I have added extra code into the before block that gets a list of accounts from web3 and then displays them in the console. These accounts will be different every time as a new blockchain instance is used on each test run.

```
7 // Before any tests are run, deploy the contract to be tested.
8 before(async () => {
9   dummyContract = await DummyContract.deployed();
10
11   // Gets the list of accounts from Web3
12   let accounts = await web3.eth.getAccounts();
13   // Display accounts
14   var count = 0;
15   for (var account in accounts) {
16     console.log(`Account${count}: ${accounts[account]} \n`);
17     count++;
18   }
19 });
20
```

```

Contract: Dummy Contract
Dummy Contract successfully deployed...
Account0: 0x73A351302a5eb8Dd9a7214Ab648E66a3CE57318b
Account1: 0x35DcA3c49dA2bBBE585c07423853d4cE3022E5a3
Account2: 0x7B4C13C03CDD1B54957764B49bAC9E956F6BF2B9
Account3: 0x800CB77eE90dD75eC91e21c1F125Efa2c3625850
Account4: 0x9B5C44406F68e294942e73E7Ce84D32A41F86c03
Account5: 0xB29C89254E28754DC54dDB851f163Fb65Fa05797
Account6: 0xD0719f6311c498346339F6326871e53AB13838C8
Account7: 0x77FB58fF96649241d44B0012723E5a7bdb5Cb73f
Account8: 0x954C6F340FFb6fbe7CDf1172f052484F12df2D56
Account9: 0x6BA65e08AC2883a9c019bF61f69028A4E1411f11

Contract deployment
  ✓ Should know information about the contract (163ms)

Contract deployment
  ✓ Should know information about the contract (169ms)

2 passing (1s)

```

Console log from within contracts

See [Docs](#)

- You can use it in calls and transactions. It works with `view` functions, but not in `pure` ones.
- It always works, regardless of the call or transaction failing or being successful.
- To use it you need to import `hardhat/console.sol`.
- You can call `console.log` with up to 4 parameters in any order of following types:
 - `uint`
 - `string`
 - `bool`
 - `address`
- There's also the single parameter API for the types above, and additionally `bytes`, `bytes1` ... up to `bytes32`:
 - `console.logInt(int i)`
 - `console.logUint(uint i)`
 - `console.logString(string memory s)`
 - `console.logBool(bool b)`
 - `console.logAddress(address a)`
 - `console.logBytes(bytes memory b)`
 - `console.logBytes1(bytes1 b)`

- `console.logBytes2(bytes2 b)`
- ...
- `console.logBytes32(bytes32 b)`
- `console.log` implements the same formatting options that can be found in Node.js' `console.log` ([opens new window](#)), which in turn uses `util.format` ([opens new window](#)).
 - Example: `console.log("Changing owner from %s to %s", currentOwner, newOwner)`
- `console.log` is implemented in standard Solidity and then detected in Hardhat Network. This makes its compilation work with any other tools (like Remix, Waffle or Truffle).
- `console.log` calls can run in other networks, like mainnet, kovan, ropsten, etc. They do nothing in those networks, but do spend a minimal amount of gas.
- `console.log` output can also be viewed for testnets and mainnet via [Tenderly](#) ([opens new window](#)).
- `console.log` works by sending static calls to a well-known contract address. At runtime, Hardhat Network detects calls to that address, decodes the input data to the calls, and writes it to the console.

Useful plugins

1. Solidity-coverage (<https://hardhat.org/plugins/solidity-coverage.html>): Test coverage for Solidity contracts.
2. Hardhat-etherscan (<https://hardhat.org/plugins/nomiclabs-hardhat-etherscan.html>): Verifies contract deployment on etherscan.
3. Hardhat-gas-reported (<https://hardhat.org/plugins/hardhat-gas-reporter.html>): Shows gas usage per unit test and gives monetary cost (in specified currency).
4. Hardhat-tracer (<https://hardhat.org/plugins/hardhat-tracer.html>): Shows emitted events when running tests.
5. Hardhat Log Remover (<https://www.npmjs.com/package/hardhat-log-remover>)
Removes console log and related import statements

Quick Guide

Initialising a project from scratch:

- In the terminal `$ npm install --save-dev hardhat`
- Then run `$ npx hardhat`
- Then select `create an empty hardhat-config.js`
- If using Waffle/Ethers, select `y` when prompted to install those packages
- If using Truffle/Web3, select `n`
- Once the project has been set up, if using Truffle/Web3, open `hardhat-config.js` and require the `truffle-5` plugin.

- If you are planning on using **Ganache**, also require the ganache plugin.
 - You have to specify the default network to be **Ganache**.
 - In the project main directory, create a test folder, scripts folder and a contracts folder.
-

Foundry

Foundry is a smart contract development toolchain written in Solidity.

Forge is the CLI tool to initialise, build and test Foundry projects.

Foundry allows fast (milliseconds) and sophisticated Solidity tests, e.g. reading and writing directly to storage slots.

Getting Started

There is an example project in our [repo](#)

1. Install Foundry
 - [Steps for various operation systems.](#)
2. Initialise a project called **lets_forge**
 - `forge init lets_forge`
3. Build the project
 - `forge build`
4. Verify install has worked by running the tests
 - `forge test`

Folders `out` and `cache` have been generated to store the contract artifact (ABI) and cached data, respectively.

Selecting solc version used by system: <https://github.com/crytic/solc-select>.

Testing: Basics

Smart contracts are tested using smart contracts, which is the secret to Foundry's speed since there is no additional compilation being carried out.

A smart contract e.g. `MyContract.sol` is tested using a file named `MyContract.t.sol`:

```
|—— src
|  |—— MyContract.sol
|  |—— test
|  |—— MyContract.t.sol
```

`MyContract.t.sol` will import the contract under test in order to access its functions.

First Contract

Create a contract called `A.sol` and save it in `src` with the following contents:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

/// @title Encode smart contract A
/// @author Extropy.io
contract A {
    uint256 number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256) {
        return number;
    }
}
```


Then create a file to test the smart contract, for example `A.t.sol` in `test` with the following contents:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

// Standard test libs
import "forge-std/Test.sol";
import "forge-std/Vm.sol";

// Contract under test
import {A} from "../src/A.sol";

contract ATest is Test {
    // Variable for contract instance
    A private a;

    function setUp() public {
        // Instantiate new contract instance
        a = new A();
    }

    function test_Log() public {
        // Various log examples
        emit log("here");
        emit log_address(address(this));
        // HEVM_ADDRESS is a special reserved address for the VM
        emit log_address(HEVM_ADDRESS);
    }

    function test_GetValue() public {
        assertTrue(a.retrieve() == 0);
    }

    function test_SetValue() public {
        uint256 x = 123;
        a.store(x);
        assertTrue(a.retrieve() == 123);
    }

    // Define the value(s) being fuzzed as an input argument
    function test_FuzzValue(uint256 _value) public {
        // Define the boundaries for the fuzzing, in this case 0
    }
}
```

```

and 99999
    _value = bound(_value, 0, 99999);
    // Call contract function with value
    a.store(_value);
    // Perform validation
    assertTrue(a.retrieve() == _value);
}
}

```

Finally run the test with `forge test -vv` to see the results and all logs.

Run tests

```
forge test
```

To print event logs:

```
forge test -vv
```

To print trace of any failed tests:

```
forge test -vvv
```

To print trace of all tests:

```
forge test -vvvv
```

To run a specific test:

```
forge test --match-test test_myTest
```

Dependencies

Forge uses [git submodules](#) so it works with any GitHub repo that contains smart contracts.

Adding Dependencies

To install e.g. OpenZeppelin contracts from the [repo](#) one would run

```
forge install OpenZeppelin/openzeppelin-contracts.
```

The repo will have been cloned into the `lib` folder.

Dependencies can be updated by running `forge update`.

Removing Dependencies

Dependencies can be removed by running

```
forge remove openzeppelin-contracts,
```

which is equivalent to

```
forge remove OpenZeppelin/openzeppelin-contracts.
```

Integrating with Existing Hardhat project

Foundry can work with Hardhat-style projects where dependencies are npm packages (stored in `node_modules`) and contracts are stored in `contracts` as opposed to source.

1. Copy lib/forge-std from a newly-created empty Foundry project to this Hardhat project directory.
2. Copy foundry.toml configuration to this Hardhat project directory and change src, out, test, cache_path in it:

```
[default]
src = 'contracts'
out = 'out'
libs = ['node_modules', 'lib']
test = 'test/foundry'
cache_path = 'forge-cache'
```

3. Create a remappings.txt to make Foundry project work well with VS Code Solidity extension:

```
ds-test/=lib/forge-std/lib/ds-test/src/
forge-std/=lib/forge-std/src/
```

4. Make a sub-directory test/foundry and write Foundry tests in it.

Foundry test works in this existing Hardhat project. As the Hardhat project is not touched and it can work as before.

See folder `hardhat-foundry`. Both test suites function:

```
npx hardhat test
```

```
forge test -vvv
```

Deploying Contracts

`forge create` allows you to deploy contracts to a blockchain.

To deploy the previous contract to e.g. Ganache using an account with private key `4e0...9b` you would type:

```
forge create A --legacy --contracts src/A.sol --private-key  
4e0...9b --rpc-url http://127.0.0.1:8545
```

Result:

```
Deployer: 0x536f8222c676b6566ff34e539022de6c1c22cc06  
Deployed to: 0x79bb7a73d02b6d7e2e98848d26ad911720421df0  
Transaction hash:  
0xc5bb34ee82dc2f57bd7f7862eec440576a7cc7cfe4533392192704fd44653b  
68
```

The `--legacy` flag is used since Ganache doesn't support EIP-1559 transactions. Hardhat (`npx hardhat node`) circumvents this issue.

Interacting with Contracts

```
cast call 0x79bb7a73d02b6d7e2e98848d26ad911720421df0 "retrieve()" --  
-rpc-url http://127.0.0.1:8545
```

Other Tools

Truffle

Truffle is an Ethereum development environment which has built-in compiling and migration, and testing frameworks.

Truffle is part of the Truffle Suite. The Suite contains a range of Ethereum development tools.

Truffle : <https://trufflesuite.com/>

The screenshot displays the Truffle Suite website with a dark background. It features three tool cards: Truffle (11,558 stars), Ganache (3,601 stars), and Drizzle (616 stars). Each card includes a title, a brief description, and buttons for 'LEARN MORE', 'GITHUB REPO', and 'DOCS'.

TRUFFLE Star 11,558

SMART CONTRACTS MADE SWEETER

A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier.

[LEARN MORE](#) [GITHUB REPO](#) [DOCS](#)

Ganache Star 3,601

ONE CLICK BLOCKCHAIN

A personal blockchain for Ethereum development you can use to deploy contracts, develop your applications, and run tests. It is available as both a desktop application as well as a command-line tool (formerly known as the TestRPC). Ganache is available for Windows, Mac, and Linux.

[LEARN MORE](#) [GITHUB REPO](#) [DOCS](#)

drizzle Star 616

FRESH CHAIN-DATA FOR FRONT-ENDS

A collection of front-end libraries that make writing dapp front-ends easier and more predictable. The core of Drizzle is based on a Redux store, so you have access to the spectacular development tools around Redux. We take care of synchronizing your contract data, transaction data and more.

[LEARN MORE](#) [GITHUB REPO](#) [DOCS](#)

Guide to Truffle

1. The folder structure

```
contracts
| Migrations.sol
| VolcanoCoin.sol
|
migrations
| 1_initial_migration.js
|
node_modules
| @openzeppelin/contracts
|
test
|
package-lock.json
|
```

| package.json

|

| truffle-config.json

Understanding the folder structure

contracts

The location for all your contracts.

```
``Migrations.sol``
```

The contracts directory is set up with Migrations.sol. Migrations.sol keeps a track of the migrations to the network.

migrations

The location for all your deployment scripts.

1_initial_migration.js

The migrations directory is set up by default with 1_initial_migration.js. These deployment scripts are run when migrating the contracts to the network.

The file's number acts as a reference, that is then recorded in Migrations.sol on deployment.

For example, to create another migration, the file name would be 2_second_migration.js and this would be referenced as the 2nd migration inside the Migrations contract.

node_modules

Location for npm packages.

test

The location for test files. It's best practice to name your test files as myContractsName.test.js

package.json

A JSON that holds metadata for your project.

truffle-config.js

The network configuration file (see below).

2. Setting up the network

truffle-config.js is your configuration file. It has been set up to enable configuration of different network settings.

In our file, we have set up a development network at local host port 8545. 8545 is the standard port for Ethereum.

The network settings must always be exported in an object.

3. Ganache CLI

Ganache CLI is also part of the Truffle suite. It simulates your own personal blockchain for Ethereum development.

Open the terminal and `cd` to `TruffleIntroduction` if you are not already in that directory

In a terminal, run the command `npm run ganache-cli`. You should see a list of 10 test accounts with their private keys, with 100 test Ether to play around with,

Ganache provides a mnemonic key if you want to import the wallet.

It is listening on 127.0.0.1:8545, which is standard for a local Ethereum blockchain network.

Available Accounts

```
=====
(0) 0xeE093F79F43800A94A8732c9F81B3ae3C93eAEd3 (100 ETH)
(1) 0xB1501A9b99F7d99267741bE5975f0eff143163FA (100 ETH)
(2) 0x1974A9b5c35B031405e8275D38f101e7339903cE (100 ETH)
(3) 0xD594155F44b07fdeBdCa2e9DdD00c1e15c879105 (100 ETH)
(4) 0xBdefDCfADaeFED3A71A246748373299e59D4a3C3 (100 ETH)
(5) 0x4eDA1d5baB7DaA46F184C4E2320A1b6a79a4DE48 (100 ETH)
(6) 0x8fE2aDD4B7EBd7EaAD15A7Dcef72528E8c7CF231 (100 ETH)
(7) 0x7399Ad93428113bf1bD826E6666A9395e7dE0086 (100 ETH)
(8) 0x1Bc7b99Ec844eB9F3BA4b52fB85A116AF4553e46 (100 ETH)
(9) 0x20768DD1561A00dc9cF9d3D3E64e51dEC93e8bB8 (100 ETH)
```

Private Keys

```
=====
(0) 0xd294bc56e6faa1ccf0ec9df385d4dce56b2d2528c3a743deeded7bff9d2a4c81
(1) 0x17351595dde947b997d49ba0b1827d91c6711e9eed3053ff3a6b10a3c8f77b99
(2) 0x57efafa3b0c26cc58eaa1c193184bc6ac4402b972a198f9f8fec653f9e504c7a
(3) 0x25efa9b37898ffd5b6f6dae6590b5daf40912fe80299ab69e3b1f3f33ed87d00
(4) 0xdcd76e65213b7802ac74e88526ba184e4664268a3924df087cdb14ba8673a84
(5) 0xc76a8341609a12a0662b137c98f786d305c31fa5b440a0f98e916728fea4f594
(6) 0x847df3315969cfb7df11c9793812881c3d3ca8901ebae06023c36e6e577b1e08
(7) 0xbbfa57076cdbad7cb87d19c7819fbd4be1adcdc0e454e307a414fc18048a6c79
(8) 0xf75a11d0b597d4df35fc04f7e4e60f881f32203b72e777b88d3e986cc5fd9687
(9) 0xdfd81cda96335756e2e071c2cfe5f9e7200f291a02acf8272cd53ac7c4f2de82
```

HD Wallet

```
=====
Mnemonic:      since fever token stand there end forest chicken install embody segment fire
Base HD Path:  m/44'/60'/0'/0/{account_index}
```

Gas Price

```
=====
20000000000
```

Gas Limit

```
=====
6721975
```

Call Gas Limit

```
=====
9007199254740991
```

```
Listening on 127.0.0.1:8545
eth_blockNumber
```

4. Truffle console

Truffle console is an interactive console that connects you to any Ethereum client.

In another terminal, run the command `npm run truffle console` which opens up a development console.

Some useful commands

`accounts` - return a list of available accounts.

`compile` - compiles the contract and generates the ABI in JSON format.

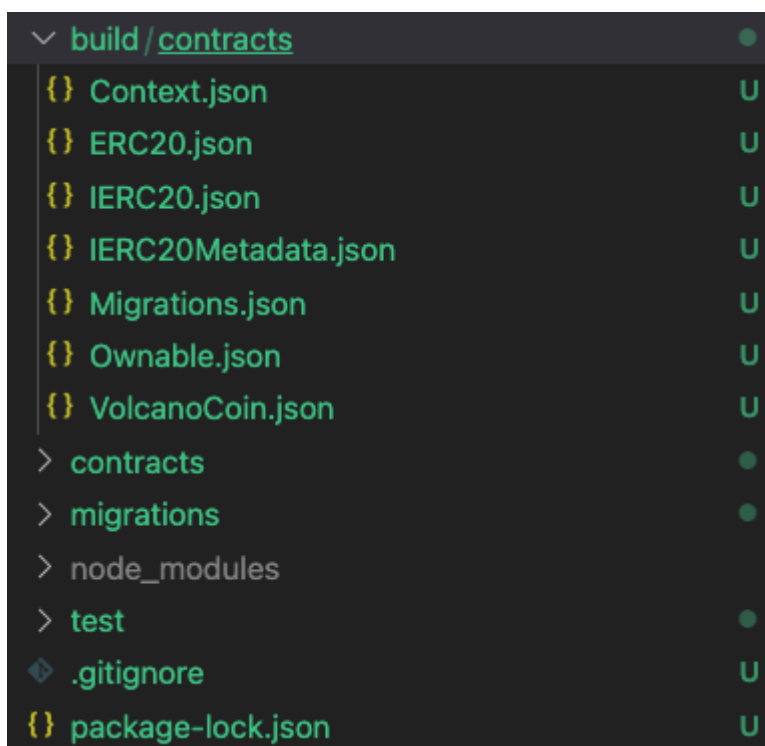
`migrate` - migrates the contract to the blockchain.

`migrate --reset` - Resets and runs all your migrations from the start.

5. Compile

Run `compile` in the truffle console. Every contract and inherited contract is compiled and the ABI (Application Binary Interface) is generated. You will see a new `build/contracts` folder has been generated, and each compiled contract has `json` file containing the formatted ABI.

```
Compiling your contracts...
=====
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/VolcanoCoin.sol
> Compiling ./node_modules/@openzeppelin/contracts/access/Ownable.sol
> Compiling ./node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol
> Compiling ./node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol
> Compiling ./node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol
> Compiling ./node_modules/@openzeppelin/contracts/utils/Context.sol
> Artifacts written to /Users/Kirsty/extropy/codeacademy/materials/Week 2 - introduction to Truffle/build/contracts
> Compiled successfully using:
   - solc: 0.8.4+commit.c7e474f2.Emscripten.clang
```



Recap: ABI (Application Binary Interface)

The ABI is the standard way to interact with smart contracts on the Ethereum network. It is a JSON which contains the names, inputs and outputs for the smart contract functions.

When a contract's function is called, the RPC node encodes the JSON data into Ethereum bytecode and broadcasts it to the network.

6. Migrations

Go to `1_initial_migration.js` in the migrations folder. You would normally see this by default:

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

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

We have imported the VolcanoCoin contract and staged the contract for deployment for you. You should see that the file has been changed to this:

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

module.exports = async function (deployer) {
  await deployer.deploy(Migrations);
  await deployer.deploy(VolcanoCoin);
};
```

artifacts

This is a JSON file that contains information regarding the contract such as the ABI, compiler version, contract bytecode.

Example:

```
{
  "contractName": "Migrations",
  "abi": [],
  "bytecode": "0x...",
  "sourceMap": "",
  "deployedSourceMap": "",
  "sourcePath": "/project/contracts/Migrations.sol",
```

```

"ast": {},
"legacyAST": {},
"compiler": {
  "name": "solc",
  "version": "0.8.0+commit.1d4f565a.Emscripten.clang"
},
"networks": {
  "17": {
    "events": {},
    "links": {},
    "address": "0x42a8d8ba55faAA734Ee07eD3179047169be5419e",
    "transactionHash":
"0x4e5fc578b9ea44401047fc010dd1ee20cd899b8091fe4304ce0f955dc1d4d
b5b"
  }
},
"schemaVersion": "3.0.0",
"updatedAt": "2021-07-11T17:59:54.615Z",
"devdoc": {},
"userdoc": {}
}

```

7. Deploy

In the truffle development console, run `migrate`.

```

Starting migrations...
=====
> Network name:      'development'
> Network id:        1625841450743
> Block gas limit: 6721975 (0x6691b7)

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash:
0x29145a69717660320f9d242a0a6c73e68ef2b94e24cf4611e3e83971b1c732
54
> Blocks: 0           Seconds: 0
> contract address:
0x1E4DC90c851ee64edd5B2b0Cc00f94806d620304

```

```
> block number:      3
> block timestamp:    1626089841
> account:
0xeE093F79F43800A94A8732c9F81B3ae3C93eAEd3
> balance:            99.98927406
> gas used:            246892 (0x3c46c)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.00493784 ETH
```

Deploying 'VolcanoCoin'

```
-----
> transaction hash:
0xca291d3f8ac775cdd4df728b11045fe2dfd6d3ec317d2b8a38bc58841d663e
87
> Blocks: 0           Seconds: 0
> contract address:
0x8B1357c98f1316C4ECF965E160F240EbcB1d7503
> block number:       4
> block timestamp:    1626089842
> account:
0xeE093F79F43800A94A8732c9F81B3ae3C93eAEd3
> balance:            99.95221212
> gas used:            1853097 (0x1c46a9)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.03706194 ETH
```

> Saving migration to chain.

> Saving artifacts

```
-----
> Total cost:          0.04199978 ETH
```

Summary

```
=====
> Total deployments:   2
> Final cost:          0.04199978 ETH
```

Common errors

- When the Solidity code is altered, the contract must be re-compiled to reflect the latest changes in the contract. If you need to re-compile your contract, you will also need to re-deploy it.
- Check that the URL in the configuration file is correct. For local host it is standard to be 127.0.0.1:8545 for the Ethereum network.

If you have copied a contract from Remix to your Truffle project, ensure that you have the exact copy of the contract and the correct ABI. Even the smallest, insignificant difference (i.e additional line space, comments) can cause errors!

- In your migrations file, the name of the contract must be the exact same.
- Check the migrations contract - does it include the pragma version you're using?
- Are you having migration issues? Try resetting the migrations by running `migrate --reset` in the Truffle console, or `truffle migrate --reset` if you're outside of the console.

Setting up locally

Before setting up, ensure that you have node and npm installed. Here's a useful guide - <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>

```
$ mkdir myProject
```

```
$ cd myProject
```

`$ npm init -y` to initialise a new npm project. If this does not work, try `npm init` and manually input your project information when prompted.

```
$ npm install -g --save truffle
```

In `truffle-config.js`, uncomment your development network settings.

In `Migrations.sol`, change the pragma version to include the version you're using.

Testing with Truffle

Test files written in javascript or solidity can be added to the test directory and will run as a result of the **test** command from the console. The overall results of the test suite are listed in the console.

```

truffle(development)> test
Using network 'development'.

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Contract: Volcano
  1) should mint 10000 VolcanoCoin
     > No events were emitted

0 passing (330ms)
1 failing

1) Contract: Volcano
   should mint 10000 VolcanoCoin:

     Minting Failed
     + expected - actual

     -10000
     +9999

     at Context.<anonymous> (test/volcano.test.js:7:12)
     at processTicksAndRejections (internal/process/task_queues.js:95:5)

```

the test starts with *it* , you can exclude tests by changing this to *xit* or set only one test to run with *it.only*

You will need to consider the atomicity of the tests, and decide what setup should be done.

Before a contract section is run, truffle re deploys your contracts so that you have a clean starting point.

Within the contract section you can use `before` and `beforeEach`

Testing for Transaction Errors and Events

A useful package 'truffle-assertions' allow us to test for transaction failure and events.

See : <https://www.npmjs.com/package/truffle-assertions>

Testing a revert (or require) statement

It is useful to be able to test that the `require` and `revert` statements in our functions are working correctly.

For example if we have a function **registerDetails** which uses the `onlyOwner` modifier, we would expect it to revert if it is called from a different address.

We can test this with the following test

(You need to specify the exact error message that is returned from the `require` statement.)

```
it('Checks only owner can add an address', async () => {
  const hacker = '0xAE0F0e3bc47Aa699De97373822D825E4A3665698'
  await truffleAssert.reverts(
    RegistryContract.registerDetails('John Smith', { from:
hacker }),
    'Ownable: caller is not the owner'
  )
})
```

Similarly we can test that events have been emitted with

```
truffleAssert.eventEmitted(result, 'TestEvent', { param1: 10,
param2: 20 });
```

Here param1 and param2 are the values passed in the event.

There are a number of different options available, please see the documentation for those.

There are further libraries to help you test, for example the passage of time.

<https://docs.openzeppelin.com/test-helpers/0.5/>

The use of .call in Truffle

We interact with write and view functions differently.

To indicate the difference we typically use .call to show we are interacting with a view function,

for example in a unit test we can use

```
const totalSupply = await instance.totalSupply.call();
```

However Truffle has the ability to know which of our contract functions are view functions (from the abi), and which are write functions,

so using the .call syntax in Truffle is optional

If we write

```
const totalSupply = await instance.totalSupply();
```

Truffle realises that this function is a view function and will create the correct type of RPC.

Workflow

A workflow that has worked well for us

- Explore / experiment in Remix - good for trying out syntax and quick checks on functionality
- Move to Truffle, write unit tests for TDD, and / or further develop the contracts

- Having a suite of tests gives confidence for refactoring
- Write the UI using mocks objects is necessary
- Use traditional techniques such as CI/CD.

Useful Resources

<https://www.trufflesuite.com/docs/truffle/overview>

<https://github.com/trufflesuite/ganache-cli/blob/master/README.md>

<https://www.trufflesuite.com/docs/truffle/testing/writing-tests-in-javascript>

<https://docs.openzeppelin.com/test-helpers/0.5/>

<https://github.com/MolochVentures/moloch/tree/4e786db8a4aa3158287e0935dcbc7b1e43416e38/test#moloch-testing-guide>

Brownie

A python based environment similar to truffle

Uses web3.py for interaction with contracts

See [documentation](#)

Rare skills competition

[Details](#)