Lesson 8

Development Tools

Introduction

Remix is an IDE for Solidity development.
It can run in the browser, or on the desktop.

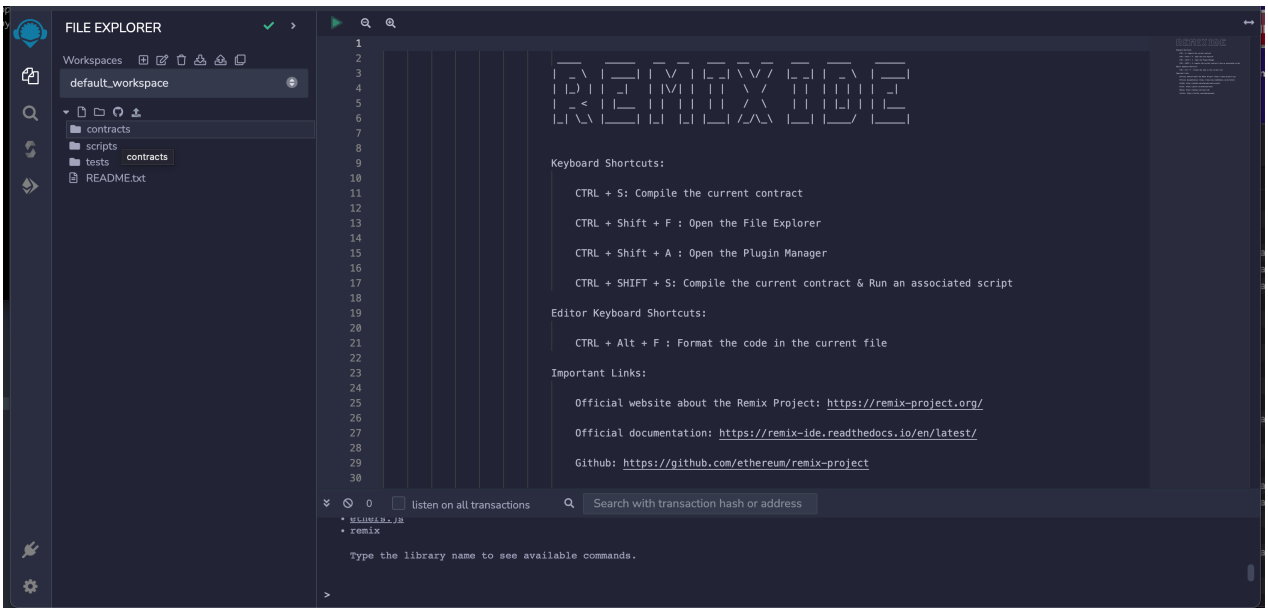
Browser : https://remix.ethereum.org/
Desktop : https://github.com/ethereum/remix-desktop/releases
or `npm install -g @remix-project/remixd`

Remix documentation!

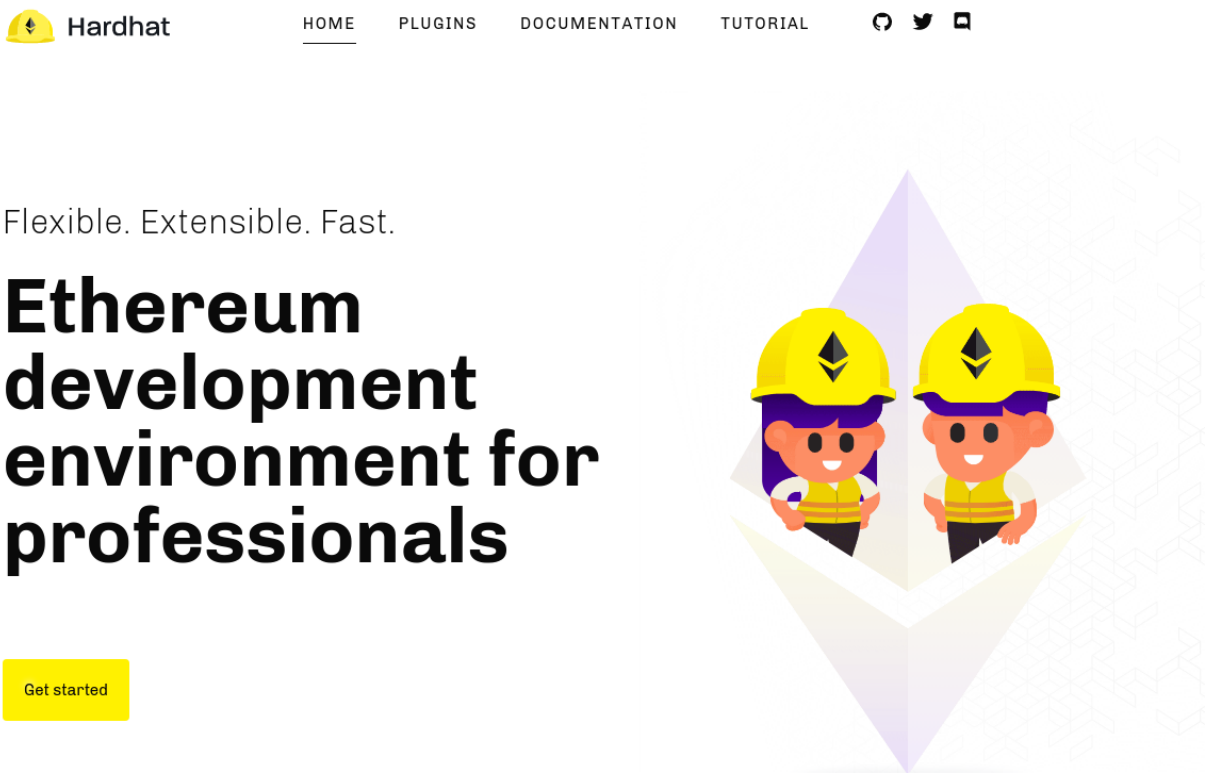
Remix has added more functionality recently, one useful feature is the ability to add log statements to your contract code.

Remix also contains some tutorials via the Learneth plugin.

Remix documentation



Introduction



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.
- Many useful plugins.
- Can be used alongside Foundry

Useful Plugins

Hardhat Toolbox

Includes

- ethers.js and the `hardhat-ethers` plugin.
- hardhat-waffle testing framework
- Mocha, Chai and Hardhat Chai Matchers plugin.
- Hardhat Network Helpers.
- hardhat-etherscan plugin.

- hardhat-gas-reporter plugin.
- solidity-coverage.
- Typechain.

Hardhat Foundry

See guide
Allows you to work with Foundry and Hardhat in the same project

Npm Packages

Open Zeppelin upgrade plugin

See details
For deploy and managing upgradable contracts

Tenderly Plugin

See details
Allows verification in Tenderly

Storage Layout

See Details

Gives a representation of storage

contract	state_variable	storage_slot	offset	type
ERC20	_balances	0	0	t_mapping(t_ac
ERC20	_allowances	1	0	t_mapping(t_ac
ERC20	_totalSupply	2	0	t_uint256
ERC20	_name	3	0	t_string_stora
ERC20	_symbol	4	0	t_string_stora
WatermelonToken	_balances	0	0	t_mapping(t_ac
WatermelonToken	_allowances	1	0	t_mapping(t_ac
WatermelonToken	_totalSupply	2	0	t_uint256
WatermelonToken	_name	3	0	t_string_stora
WatermelonToken	_symbol	4	0	t_string_stora

Hardhat Log Remover

See Details

Removes the console log statements from your contracts.

Installing Hardhat

Requirements

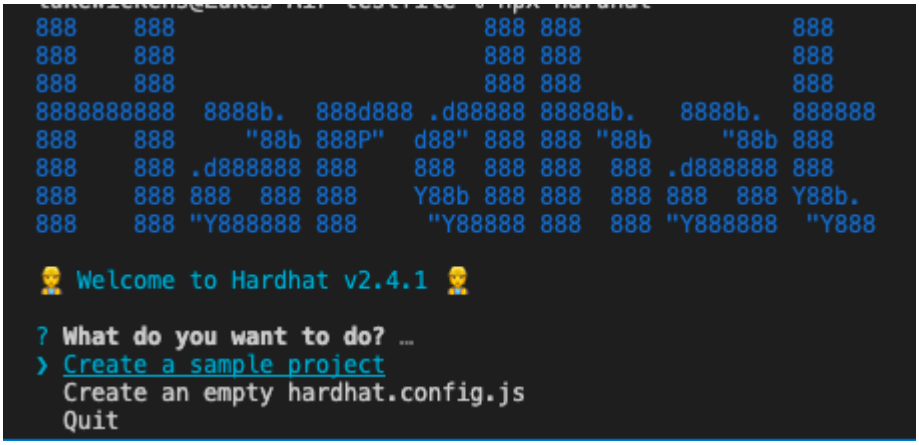
Node.js version 12 and above.

Installation

Steps for installing and using Hardhat:

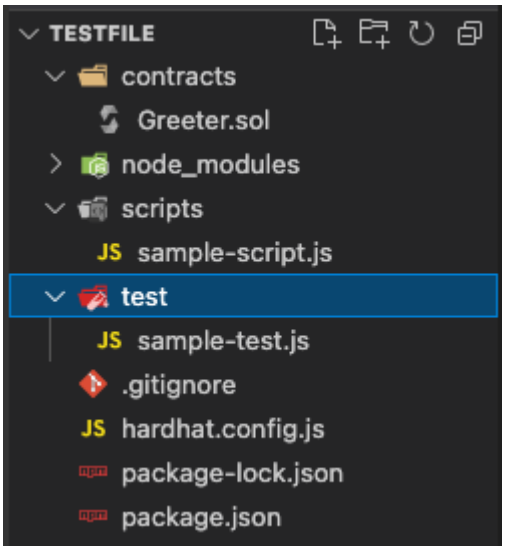
1. `npm install –D hardhat`
2. `npx hardhat`

You should then see this in the terminal:



Choosing *Create a sample project* will:

- Create a contracts folder with a dummy contract *'Lock.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



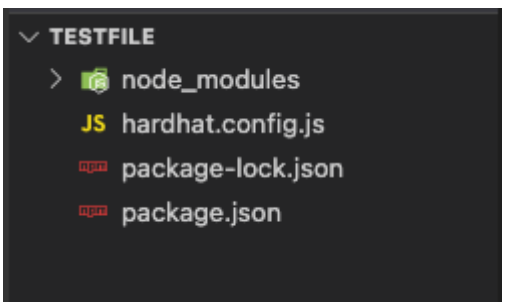
Choosing *Create and empty hardhat.config.js* will set up a new hardhat configuration file but without all of the dummy contracts, script files and tests. This method will not install any plugin (**ethers.js / Waffle**). If you want to use the **Web3.js / Truffle** plugin, this would be the options you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

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

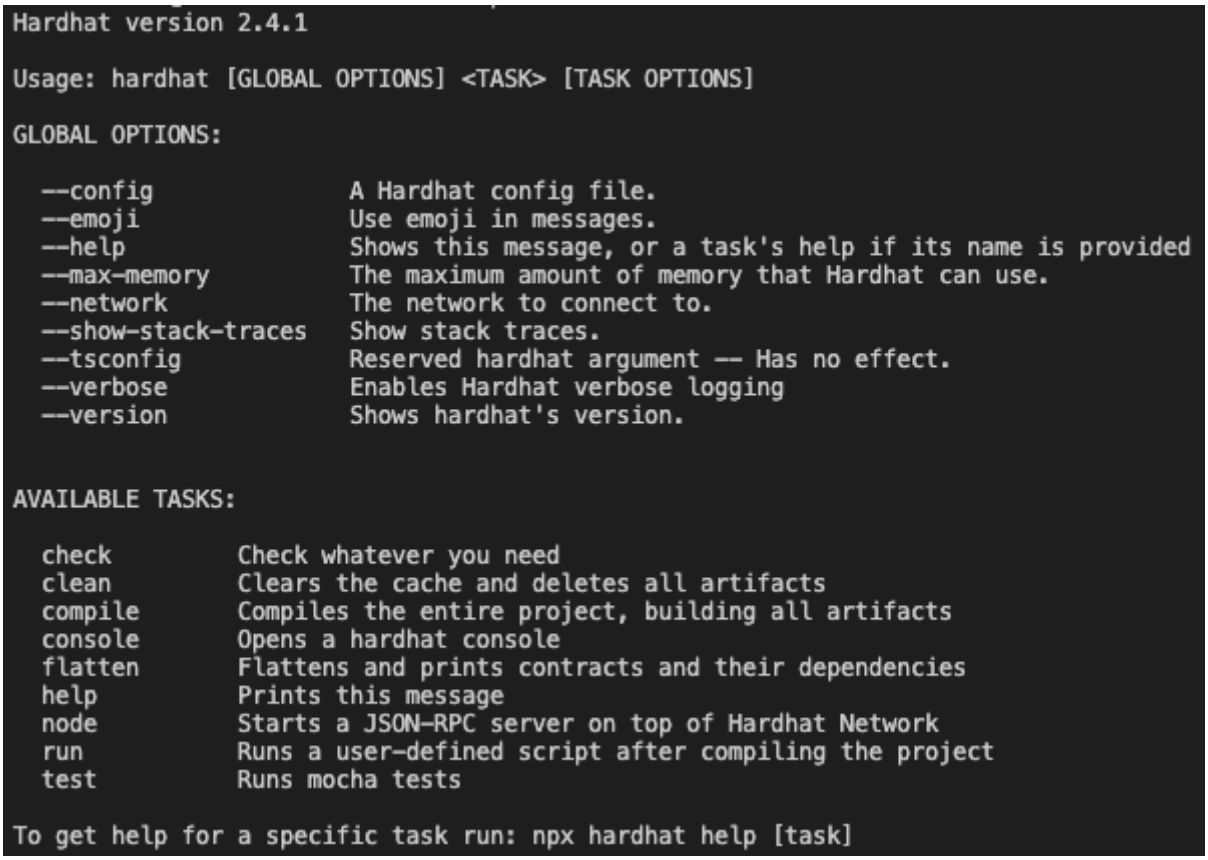
or

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

depending on the one that you are using.



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



Install Open Zeppelin Libraries

`npm install @openzeppelin/contracts`

Using Waffle & Ethers

dummyContract.sol

contracts > dummyContract.sol

```sol
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

// imported ERC20 and Ownable contracts from the OpenZeppelin library.
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

/**
 * @title A unique token that can be used in different context (eg: data or rental marketplace)
 * @dev all the functions from the ERC20 tokens standard are available
 * @author
 */
contract DummyContract is ERC20("DummyToken", "DumTkn"), Ownable {
    uint256 constant INITIAL_AMOUNT = 100;

    constructor() {}

    function setUp() external onlyOwner() {
        _mint(msg.sender, INITIAL_AMOUNT);
    }
}

```

deploy_script.js

scripts > JS deploy_script.js > ...

```js
const hre = require("hardhat");

async function main() {
  // Hardhat always runs the compile task when running scripts with its command
  // line interface.
  //
  // If this script is run directly using `node` you may want to call compile
  // manually to make sure everything is compiled
  // await hre.run('compile');

  // We get the contract to deploy
  const [deployer] = await ethers.getSigners();
  console.log(`Deploying contracts with the account: ${deployer.address}`);

  const balance = await deployer.getBalance();
  console.log(`Account Balance: ${balance.toString()}`);

  // We get the contract to deploy
  const DummyContract = await hre.ethers.getContractFactory("DummyContract");
  const dummyContract = await DummyContract.deploy();

  console.log("Token deployed to:", dummyContract.address);
}

```

This file 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
```

The **getContractFactory()** function allows the contract to be deployed, 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.
The method for doing this 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")
    );
  });
```

Console log from within contracts

See Docs
To enable it within your contract include the import :

```
import "hardhat/console.sol";
```

- 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.

Hardhat Network

See Documentation

Hardhat network is a local node that is useful for testing.
It is started automatically when you run tests in Hardhat.

Running in standalone mode

Start with
`npx hardhat node`
This exposes the usual RPC endpoints, you can then connect to this node from example a wallet.
by default this is available at
`http://127.0.0.1:8545`

If you want to connect Hardhat to this node, you can specify
`--network localhost`

By default blocks will be mined as they are needed, but a number of modes are available, see Mining modes

Forking Networks

See Guide
You will need connection to an archive node, via providers such as Alchemy or Infura.

If `<key>` is your Alchemy key
then from the command line run :
`npx hardhat node --fork https://eth-mainnet.alchemyapi.io/v2/<key>`

You can also configure this in the project config file with

```
networks: {
  hardhat: {
```

```
    forking: {
      url: "https://eth-mainnet.alchemyapi.io/v2/<key>",
    }
  }
}
```

When forking the mainnet it is useful to pin to a specific block, this allows repeatable tests, and allows hardhat to cache data when restarting the network.