Hardhat Quick Guide

Contents

Overview 2
Creating a Hardhat project: 2
Configuring Hardhat: 3
Compiling a Hardhat project 4
Deploying smart contracts with Hardhat 4
The Hardhat Network: 5
Deploying a smart contract to a local Hardhat Network 5
Debugging with Hardhat 5
Testing with Hardhat 6
Creating Tasks with Hardhat 9
Subtasks 9
The hardhat-ethers plugin 10
Using Hardhat with Ganache 10

Overview

Hardhat documentation: https://hardhat.org/getting-started

Hardhat is a development environment to compile, deploy, test, and debug Ethereum smart contracts. Hardhat comes built-in with the Hardhat Network (a local Ethereum network for development) and the Hardhat Runner - the CLI to interact with Hardhat that acts as an extensible task runner. To see all available tasks, run: npx hardhat

For example, running the compile task: npx hardhat compile

Requirements to run Hardhat:

Install the latest LTS version of Node.js: https://nodejs.org/en/download

Creating a Hardhat project:

To setup a basic Hardhat project, execute the following commands:

- In your project folder, create a package.json file: ***npm init***

- Install Hardhat locally: ***npm i hardhat -D*** => this installs Hardhat as development dependency

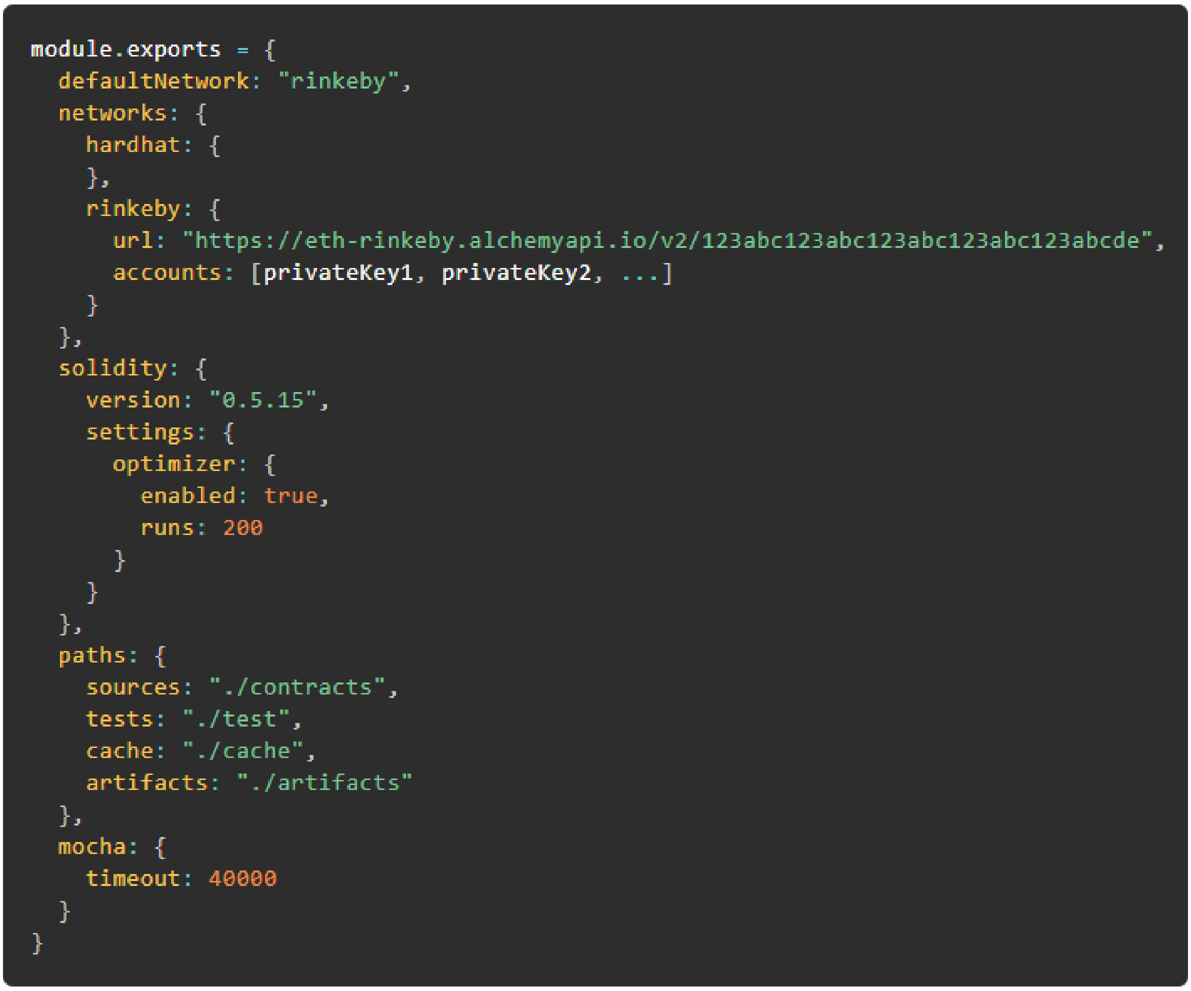
- Create a hardhat project: ***npx hardhat*** => select: Create a basic sample project

- Install the following plugins: ***npm i -D @nomiclabs/hardhat-ethers ethers @nomiclabs/hardhat-waffle ethereum-waffle chai***

Configuring Hardhat:

The configuration file (hardhat.config.js) in the root folder of your project is always executed on startup before anything else happens - for example when we run a task.

Sample configuration file: https://hardhat.org/config



```
module.exports = {
  defaultNetwork: "rinkeby",
  networks: {
    hardhat: {
    },
    rinkeby: {
      url: "https://eth-rinkeby.alchemyapi.io/v2/123abc123abc123abc123abc123abcde",
      accounts: [privateKey1, privateKey2, ...]
    }
  },
  solidity: {
    version: "0.5.15",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
  paths: {
    sources: "./contracts",
    tests: "./test",
    cache: "./cache",
    artifacts: "./artifacts"
  },
  mocha: {
    timeout: 40000
  }
}
```

It is not required to add hardhat or localhost (http://127.0.0.1:8545) to the list of networks - they are always available.

Compiling a Hardhat project

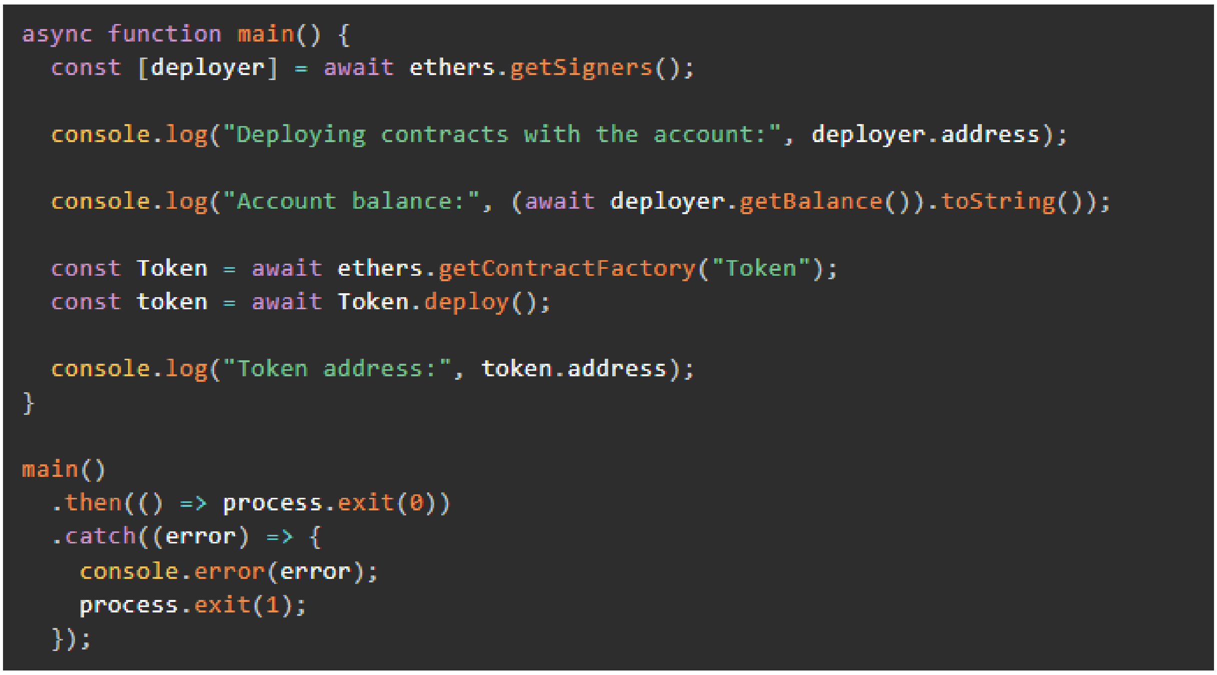
To compile all the smart contracts in your project, execute: npx hardhat compile

Compiling smart contracts with Hardhat generates two files per compiled contract: an artifact and a debug file. The artifact has all the information that is required to deploy and interact with the contract: contractName, abi, bytecode...

The HRE has an artifacts object with helper methods. For example, you can get a list with the paths to all artifacts by calling hre.artifacts.getArtifactPaths().You can also read an artifact using the name of the contract by calling hre.artifacts.readArtifact("Bar"), which will return the content of the artifact for the Bar contract.

Deploying smart contracts with Hardhat

Sample deployment script in the scripts folder: deploy.js



```
async function main() {
  const [deployer] = await ethers.getSigners();

  console.log("Deploying contracts with the account:", deployer.address);

  console.log("Account balance:", (await deployer.getBalance()).toString());

  const Token = await ethers.getContractFactory("Token");
  const token = await Token.deploy();

  console.log("Token address:", token.address);
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

To run the script: ***npx hardhat run scripts/deploy.js --network <network-name>***

The Hardhat Network:

By default, Hardhat will always create an in-memory instance of the Hardhat Network whenever a task (this could also be a script or tests) is executed and all of Hardhat's plugins (ethers.js, Waffle...) will connect directly to this network's provider.

It's also possible to run the Hardhat Network in a standalone fashion so that external clients (like MetaMask or your DAPP) can connect to it.

Deploying a smart contract to a local Hardhat Network

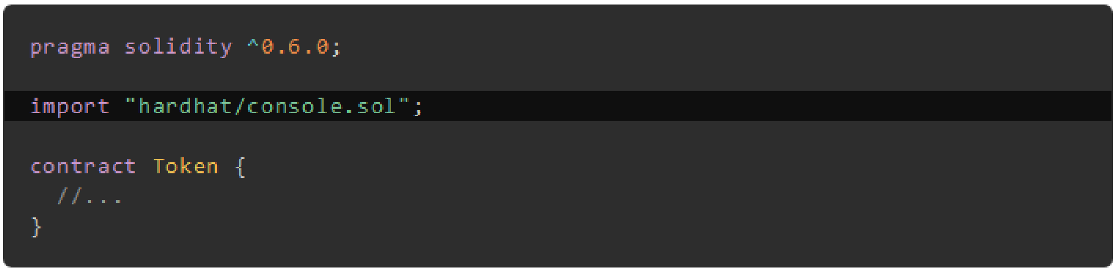
- Open a command window and run: ***npx hardhat node*** => this starts a local Hardhat Network, that exposes a JSON-RPC interface to the Network through: http://127.0.0.1:8545

- In a second command window, run: ***npx hardhat run scripts/deploy.js --network localhost*** => localhost needs to be specified explicitly, deploy.js is your deployment script

- If you need MetaMask, select the "localhost" network in Metamask and make sure, the RPC URL is: http://127.0.0.1:8545, you may also need to set the Chain ID to: 31337

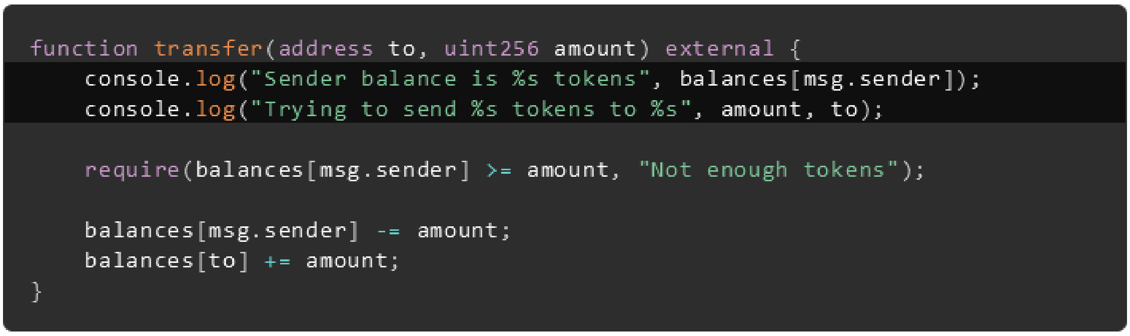
Debugging with Hardhat

When running your contracts and tests on the Hardhat Network you can print logging messages and contract variables using ***console.log(...)*** in your Solidity code. To use the logging feature, you have to import hardhat/console.sol in your contract code:



```
pragma solidity ^0.6.0;

import "hardhat/console.sol";

contract Token {
  //...
}
```

Now you can use logging in your contract functions:

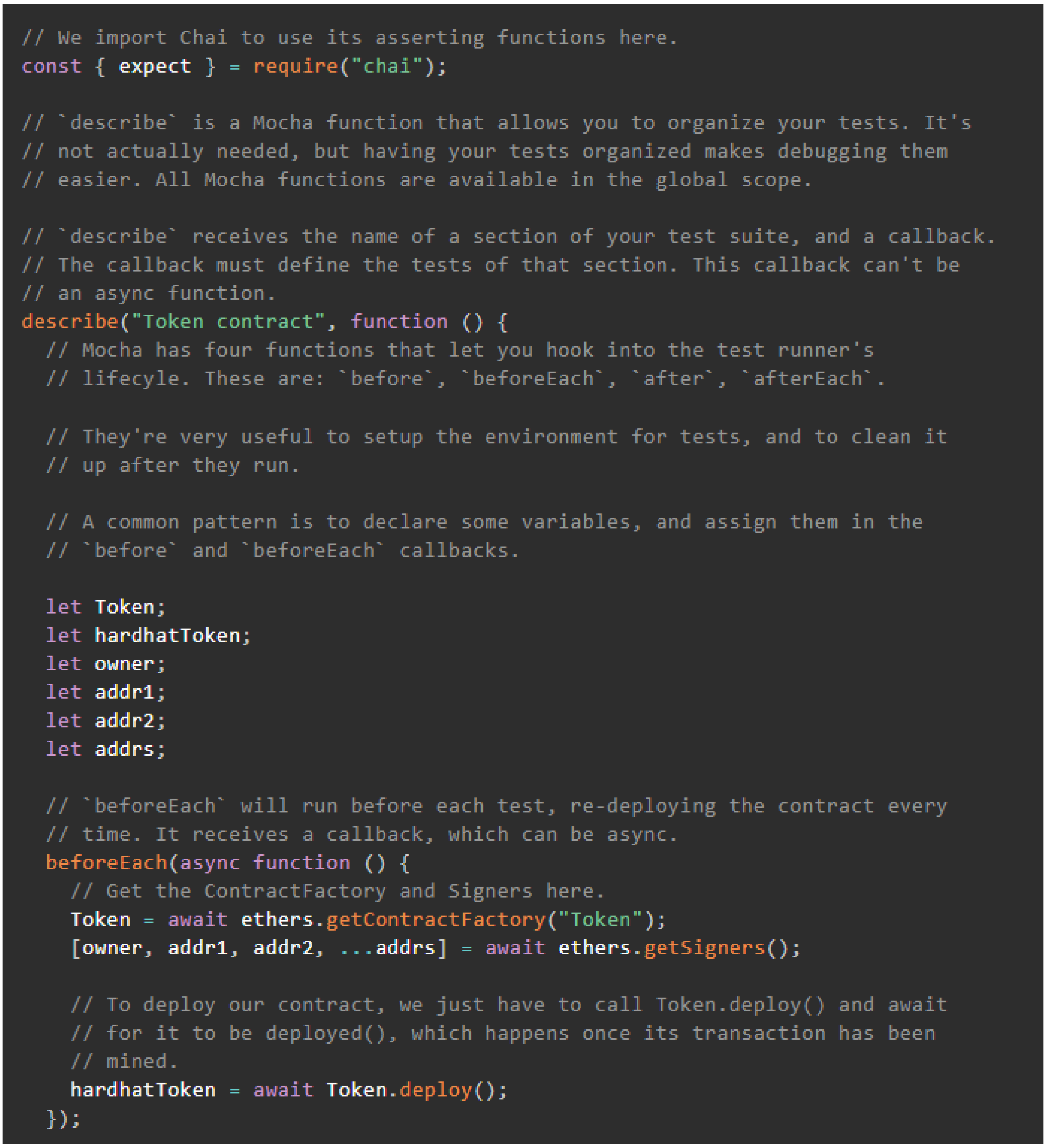


```
function transfer(address to, uint256 amount) external {
    console.log("Sender balance is %s tokens", balances[msg.sender]);
    console.log("Trying to send %s tokens to %s", amount, to);

    require(balances[msg.sender] >= amount, "Not enough tokens");

    balances[msg.sender] -= amount;
    balances[to] += amount;
}
```

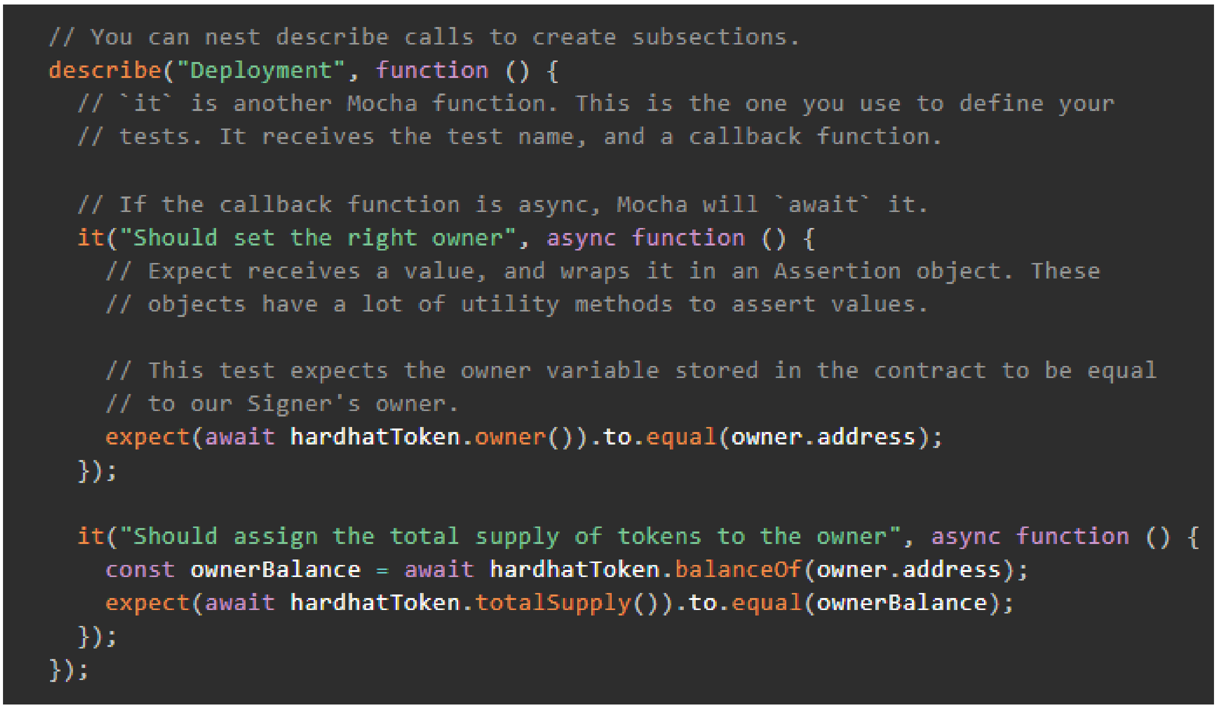
Testing with Hardhat

Hardhat uses Mocha (https://mochajs.org/) as test runner. We are also using Chai (https://www.chaijs.com/) which is an assertions library. These asserting functions are called "matchers", and the ones we're using here actually come from Waffle (library for smart contract testing: https://ethereum-waffle.readthedocs.io/en/latest/). This is why we're using the @nomiclabs/hardhat-waffle plugin, which makes it easier to assert values from Ethereum

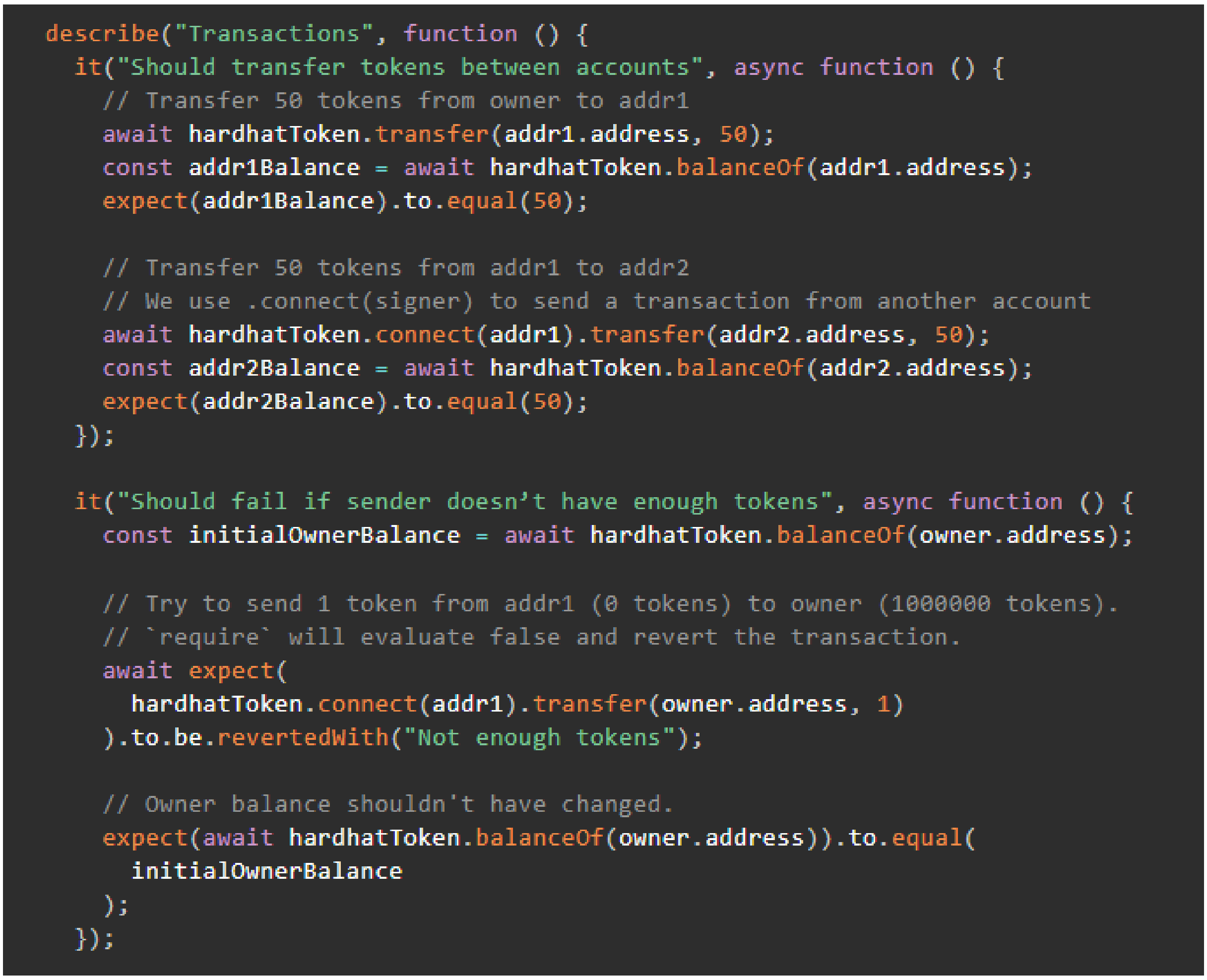
Test example: https://hardhat.org/tutorial/testing-contracts



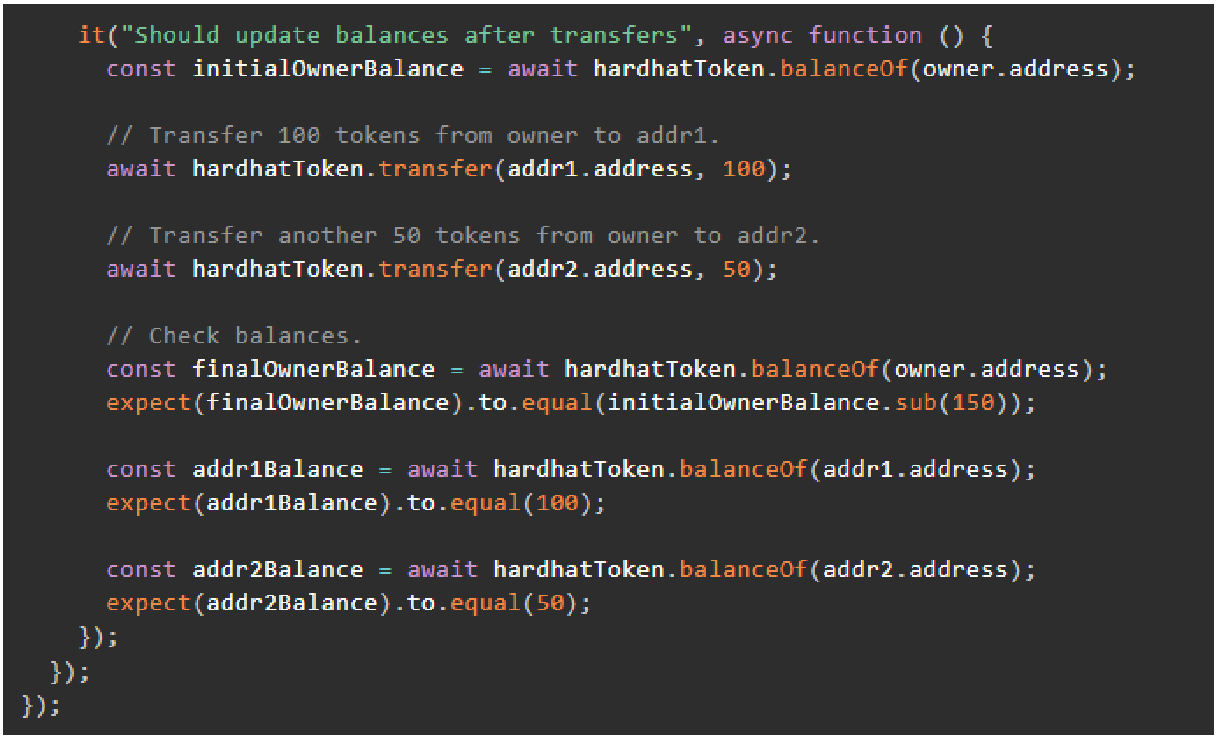
```
// We import Chai to use its asserting functions here.
const { expect } = require("chai");

// `describe` is a Mocha function that allows you to organize your tests. It's
// not actually needed, but having your tests organized makes debugging them
// easier. All Mocha functions are available in the global scope.

// `describe` receives the name of a section of your test suite, and a callback.
// The callback must define the tests of that section. This callback can't be
// an async function.
describe("Token contract", function () {
  // Mocha has four functions that let you hook into the test runner's
  // lifecyle. These are: `before`, `beforeEach`, `after`, `afterEach`.

  // They're very useful to setup the environment for tests, and to clean it
  // up after they run.

  // A common pattern is to declare some variables, and assign them in the
  // `before` and `beforeEach` callbacks.

  let Token;
  let hardhatToken;
  let owner;
  let addr1;
  let addr2;
  let addrs;

  // `beforeEach` will run before each test, re-deploying the contract every
  // time. It receives a callback, which can be async.
  beforeEach(async function () {
    // Get the ContractFactory and Signers here.
    Token = await ethers.getContractFactory("Token");
    [owner, addr1, addr2, ...addrs] = await ethers.getSigners();

    // To deploy our contract, we just have to call Token.deploy() and await
    // for it to be deployed(), which happens once its transaction has been
    // mined.
    hardhatToken = await Token.deploy();
  });
```



```
// You can nest describe calls to create subsections.
describe("Deployment", function () {
  // `it` is another Mocha function. This is the one you use to define your
  // tests. It receives the test name, and a callback function.

  // If the callback function is async, Mocha will `await` it.
  it("Should set the right owner", async function () {
    // Expect receives a value, and wraps it in an Assertion object. These
    // objects have a lot of utility methods to assert values.

    // This test expects the owner variable stored in the contract to be equal
    // to our Signer's owner.
    expect(await hardhatToken.owner()).to.equal(owner.address);
  });

  it("Should assign the total supply of tokens to the owner", async function () {
    const ownerBalance = await hardhatToken.balanceOf(owner.address);
    expect(await hardhatToken.totalSupply()).to.equal(ownerBalance);
  });
});
```



```
describe("Transactions", function () {
  it("Should transfer tokens between accounts", async function () {
    // Transfer 50 tokens from owner to addr1
    await hardhatToken.transfer(addr1.address, 50);
    const addr1Balance = await hardhatToken.balanceOf(addr1.address);
    expect(addr1Balance).to.equal(50);

    // Transfer 50 tokens from addr1 to addr2
    // We use .connect(signer) to send a transaction from another account
    await hardhatToken.connect(addr1).transfer(addr2.address, 50);
    const addr2Balance = await hardhatToken.balanceOf(addr2.address);
    expect(addr2Balance).to.equal(50);
  });

  it("Should fail if sender doesn’t have enough tokens", async function () {
    const initialOwnerBalance = await hardhatToken.balanceOf(owner.address);

    // Try to send 1 token from addr1 (0 tokens) to owner (1000000 tokens).
    // `require` will evaluate false and revert the transaction.
    await expect(
      hardhatToken.connect(addr1).transfer(owner.address, 1)
    ).to.be.revertedWith("Not enough tokens");

    // Owner balance shouldn't have changed.
    expect(await hardhatToken.balanceOf(owner.address)).to.equal(
      initialOwnerBalance
    );
  });
```



```
    it("Should update balances after transfers", async function () {
      const initialOwnerBalance = await hardhatToken.balanceOf(owner.address);

      // Transfer 100 tokens from owner to addr1.
      await hardhatToken.transfer(addr1.address, 100);

      // Transfer another 50 tokens from owner to addr2.
      await hardhatToken.transfer(addr2.address, 50);

      // Check balances.
      const finalOwnerBalance = await hardhatToken.balanceOf(owner.address);
      expect(finalOwnerBalance).to.equal(initialOwnerBalance.sub(150));

      const addr1Balance = await hardhatToken.balanceOf(addr1.address);
      expect(addr1Balance).to.equal(100);

      const addr2Balance = await hardhatToken.balanceOf(addr2.address);
      expect(addr2Balance).to.equal(50);
    });
  });
});
```

To execute your test, run the following command: npx hardhat test

Key points of the provided Test file:

- **Chai**: provides asserting functions for our tests

- **describe**: allows us to organize our tests into groups. We can also nest describe calls to create subsections.

- **beforeEach**: this is a special Mocha function that is run before each test - in our example, we are re-deploying our contract before each test is run. Additional hooks provided by Mocha are: before, after and afterEach.

- **it**: another Mocha function that allows us to define individual tests

- **expect**: assertion function that receives a value, and wraps it in an Assertion object. These objects have a lot of utility methods to assert values - like: to, be, equal, above, gt, lt, within, revertedWith, emit... => https://ethereum-waffle.readthedocs.io/en/latest/matchers.html

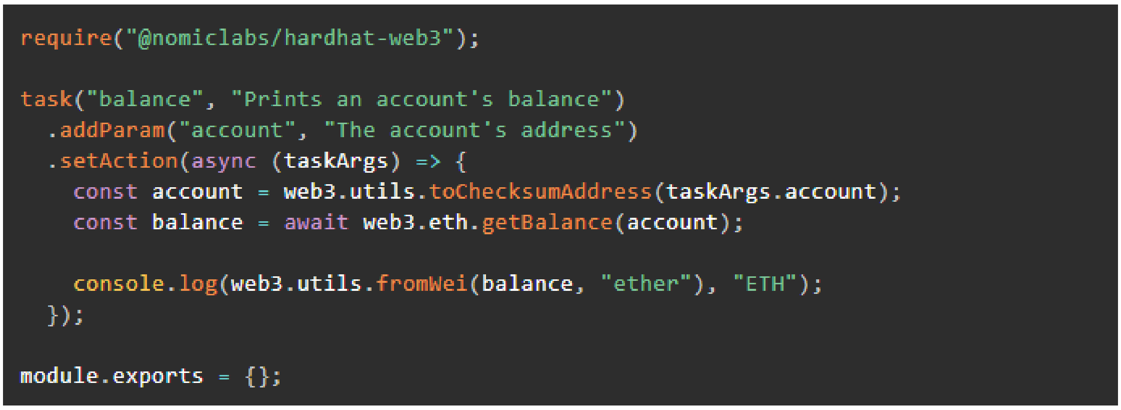
Creating Tasks with Hardhat

Tasks are the core component used for automation. To see the currently available tasks in your project, run npx hardhat. The most important tasks: compile, test, run, node, help

New Tasks can be added to the hardhat.config.js file. For more complex tasks, it may be a good idea to split the code into several files and require them from the configuration file

Creating a task is done by calling the task function. When you add a parameter to a task, Hardhat will handle its help messages for you:

Sample Task: https://hardhat.org/guides/create-task



```
require("@nomiclabs/hardhat-web3");

task("balance", "Prints an account's balance")
  .addParam("account", "The account's address")
  .setAction(async (taskArgs) => {
    const account = web3.utils.toChecksumAddress(taskArgs.account);
    const balance = await web3.eth.getBalance(account);

    console.log(web3.utils.fromWei(balance, "ether"), "ETH");
  });

module.exports = {};
```

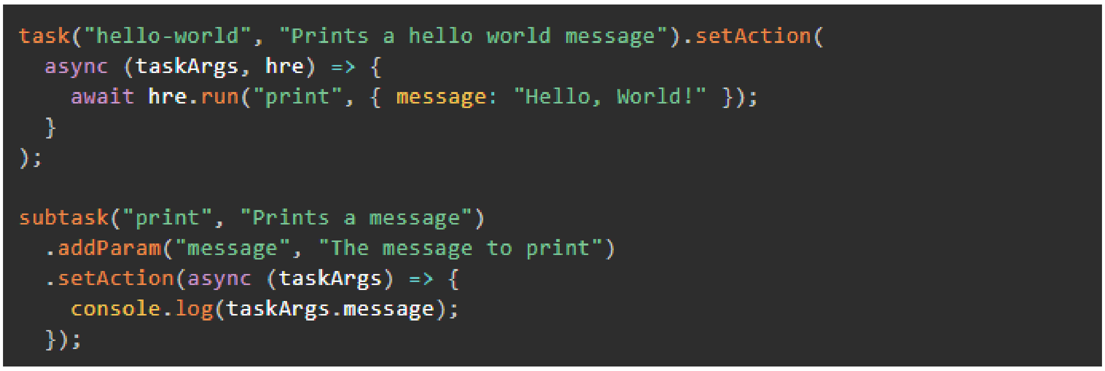
To run this task, we execute: ***npx hardhat balance --account 0xabc123...***

Adding an optional parameter can look like this:

task("balance", "Prints an account's balance")
 .addOptionalParam("account", "The account's address")

Subtasks

Creating tasks with lots of logic makes it hard to extend or customize them. Making multiple small and focused tasks that call each other is a better way to allow for extension. To run a subtask, or any task whatsoever, you can use the run function. It takes two arguments: the name of the task to be run, and an object with its arguments.



```
task("hello-world", "Prints a hello world message").setAction(
  async (taskArgs, hre) => {
    await hre.run("print", { message: "Hello, World!" });
  }
);

subtask("print", "Prints a message")
  .addParam("message", "The message to print")
  .setAction(async (taskArgs) => {
    console.log(taskArgs.message);
  });
```

The hardhat-ethers plugin

This plugin adds the ethers.js library to Hardhat, which allows you to interact with the Ethereum blockchain in a simple way. The plugins adds an ethers object to the Hardhat Runtime Environment. This object has the same API as ethers.js, with some extra Hardhat-specific functionality.

Installation: ***npm i --D @nomiclabs/hardhat-ethers ethers***

Add the following statement to your hardhat.config.js:

require("@nomiclabs/hardhat-ethers");

Hardhat-ethers adds a provider object to ethers (ethers.provider), which is automatically connected to the selected network.

Additional functions that are added to the ethers object:

getContractFactory(name: string, signer?: ethers.Signer): Promise<ethers.ContractFactory>;

getContractAt(name: string, address: string, signer?: ethers.Signer): Promise<ethers.Contract>;

getSigners() => Promise<ethers.Signer[]>;

getContractFactoryFromArtifact(artifact: Artifact, signer?: ethers.Signer): Promise<ethers.ContractFactory>;

Using Hardhat with Ganache

Just start Ganache and then run Hardhat with (for example to deploy a contract):

npx hardhat --network localhost run scripts/deploy.js