---

title: Testing simple smart contract with Waffle library description: Tutorial for beginners author: Ewa Kowalska tags: ["smart contracts", "solidity", "Waffle", "testing"] skill: beginner lang: en published: 2021-02-26

In this tutorial you'll learn how to {#in-this-tutorial-youll-learn-how-to}

- Test the changes of wallet balance
- Test emission of events with specified arguments
- Assert that a transaction was reverted

Assumptions {#assumptions}

- You can create a new JavaScript or TypeScript project
- You have some basic experience with tests in JavaScript
- You have used some package managers like yarn or npm
- You possess very basic knowledge of smart contracts and Solidity

Getting started {#getting-started}

The tutorial demonstrates test setup and run using yarn, but there is no problem if you prefer npm - I will provide proper references to the official Waffle documentation.

Install Dependencies {#install-dependencies}

Add ethereum-waffle and typescript dependencies to the dev dependencies of your project.

```
bash yarn add --dev ethereum-waffle ts-node typescript @types/jest
```

Example smart contract {#example-smart-contract}

During the tutorial we'll work on a simple smart contract example - EtherSplitter. It does not much apart from allowing anyone to send some wei and split it evenly between two predefined receivers. The split function requires the number of wei to be even, otherwise it will revert. For both receivers it performs a wei transfer followed by emission of the Transfer event.

Place the snippet of EtherSplitter code in `src/EtherSplitter.sol`.

```solidity pragma solidity ^0.6.0;

contract EtherSplitter { address payable receiver1; address payable receiver2;

```
event Transfer(address from, address to, uint256 amount);

constructor(address payable _address1, address payable _address2) public {
    receiver1 = _address1;
    receiver2 = _address2;
}

function split() public payable {
    require(msg.value % 2 == 0, 'Uneven wei amount not allowed');
    receiver1.transfer(msg.value / 2);
    emit Transfer(msg.sender, receiver1, msg.value / 2);
    receiver2.transfer(msg.value / 2);
    emit Transfer(msg.sender, receiver2, msg.value / 2);
}
```

} ```

## Compile the contract {#compile-the-contract}

To compile the contract add the following entry to the package.json file:
```

```
json "scripts": { "build": "waffle" }
```

Next, create the Waffle configuration file in the project root directory - `waffle.json` - and then paste the following configuration there:

```
json { "compilerType": "solcjs", "compilerVersion": "0.6.2", "sourceDirectory": "./src", "outputDirectory": "./build" }
```

Run `yarn build`. As the result, the `build` directory will appear with the EtherSplitter compiled contract in JSON format.

Test setup {#test-setup}

Testing with Waffle requires using Chai matchers and Mocha, so you need toadd them to your project. Update your package.json file and add the `test` entry in the scripts part:

```
json "scripts": { "build": "waffle", "test": "export NODE_ENV=test && mocha -r ts-node/register 'test/**/*.test.ts'" }
```

If you want to execute your tests, just run `yarn test` .

Testing {#testing}

Now create the `test` directory and create the new file `test\EtherSplitter.test.ts`. Copy the snippet below and paste it to our test file.

```ts import { expect, use } from "chai" import { Contract } from "ethers" import { deployContract, MockProvider, solidity } from "ethereum-waffle" import EtherSplitter from "../build/EtherSplitter.json"

use(solidity)

describe("Ether Splitter", () => { const [sender, receiver1, receiver2] = new MockProvider().getWallets() let splitter: Contract

beforeEach(async () => { splitter = await deployContract(sender, EtherSplitter, [ receiver1.address, receiver2.address, ]) })

// add the tests here }) ```
```

A few words before we start. The `MockProvider` comes up with a mock version of the blockchain. It also delivers mock wallets that will serve us for testing EtherSplitter contract. We can get up to ten wallets by calling `getWallets()` method on the provider. In the example, we get three wallets - for the sender and for two receivers.

Next, we declare a variable called 'splitter' - this is our mock EtherSplitter contract. It is created before each execution of a single test by the `deployContract` method. This method simulates deployment of a contract from the wallet passed as the first parameter (sender's wallet in our case). The second parameter is the ABI and the bytecode of the tested contract - we pass there the json file of the compiled EtherSplitter contract from the `build` directory. The third parameter is an array with the contract's constructor arguments, which in our case, are the two addresses of the receivers.

changeBalances {#changebalances}

First, we will check if the split method actually changes the balances of receivers' wallets. If we split 50 wei from senders account, we would expect the balances of both receivers to increase by 25 wei. We will use Waffle's `changeBalances` matcher:

```
ts it("Changes accounts balances", async () => { await expect(() => splitter.split({ value: 50 })).to.changeBalances( [receiver1, receiver2], [25, 25] ) })
```

As the first parameter of the matcher, we pass an array of receivers' wallets, and as the second - an array of expected increases on corresponding accounts. If we wanted to check the balance of one specific wallet, we could also use `changeBalance` matcher, which does not require passing arrays, as in the example below:

```
ts it("Changes account balance", async () => { await expect(() => splitter.split({ value: 50 })).to.changeBalance( receiver1, 25 ) })
```

Note that in both cases of `changeBalance` and `changeBalances` we pass the split function as a callback because the matcher needs to access the state of balances before and after the call.

Next, we'll test if the Transfer event was emitted after each transfer of wei. We'll turn to another matcher from Waffle:

Emit {#emit}

```ts it("Emits event on the transfer to the first receiver", async () => { await expect(splitter.split({ value: 50 })) .to.emit(splitter, "Transfer") .withArgs(sender.address, receiver1.address, 25) })

it("Emits event on the transfer to the second receiver", async () => { await expect(splitter.split({ value: 50 })) .to.emit(splitter, "Transfer") .withArgs(sender.address, receiver2.address, 25) }) ```

The `emit` matcher allows us to check if a contract emitted an event on calling a method. As the parameters to the `emit` matcher, we provide the mock contract that we predict to emit the event, along with the name of that event. In our case, the mock contract is `splitter` and the name of the event - `Transfer`. We can also verify the precise values of arguments that the event was emitted with - we pass as many arguments to `withArgs` matcher, as our event declaration expects. In case of EtherSplitter contract, we pass the addresses of the sender and the receiver along with the transferred wei amount.

## revertedWith {#revertedwith}

As the last example, we'll check if the transaction was reverted in case of uneven number of wei. We'll use `revertedWith` matcher:

```
ts it("Reverts when Vei amount uneven", async () => { await expect(splitter.split({ value: 51
})).to.be.revertedWith("Uneven wei amount not allowed") })
```

The test, if passed, will assure us that the transaction was reverted indeed. However, there must be also an exact match between the messages that we passed in `require` statement and the message we expect in `revertedWith`. If we go back to the code of EtherSplitter contract, in the `require` statement for the wei amount, we provide the message: 'Uneven wei amount not allowed'. This matches the message we expect in our test. If they were not equal, the test would fail.

# Congratulations! {#congratulations}

You've made your first big step towards testing smart contracts with Waffle! You might be interested in other Waffle tutorials:

- Testing ERC20 with Waffle
- Waffle: Dynamic mocking and testing contract calls
- Waffle say hello world tutorial with hardhat and ethers
```