---

title: Testing ERC-20 tokens with Waffle description: Learn how test Solidity smart contracts and use smart contract matchers with Waffle. author: Vladislav Starostenko tags: ["waffle", "smart contracts", "solidity", "testing", "erc-20"] skill: intermediate lang: en published: 2020-10-16

In this tutorial you will learn how to:

- Write tests for smart contracts with Waffle
- Use some popular matchers to test smart contracts with Waffle

Assumptions:

- you can get around in a terminal,
- you can create a new `JavaScript` project,
- you've written a few lines of `Solidity` code,
- you've written a few tests in `JavaScript`,
- you've used `yarn` or `npm`, JavaScripts's package installer.

Again, if any of these are untrue, or you don't plan to reproduce the code in this article, you can likely still follow along just fine.

A few words about Waffle {#a-few-words-about-waffle}

Waffle is the most advanced library for writing and testing smart contracts.

Works with the JavaScript API ethers-js.

You can read more details in the Waffle documentation !

The quick tutorial {#the-quick-tutorial}

First things first, create new `JavaScript` or `TypeScript` project (I'll use `TS`, but if you use `JS` it's not a problem) :

Somewhat like this :

▶ package.json
▶ tsconfig.json
▶ .gitignore
▶ .eslintrc.js

Step #1: Install waffle in your project Link to doc {#step-1-install-waffle-in-your-project}

To get started, install `ethereum-waffle`. In this tutorial, I'll use `yarn`, so to install `ethereum-waffle` run:

```
bash yarn add --dev ethereum-waffle
```

Step #2: Write a smart contract Link to doc {#step-2-write-a-smart-contract}

In this tutorial, I'll use ERC20 token from OpenZeppelin.

So, add `OpenZeppelin` by installing it with `yarn`:

```
bash yarn add @openzeppelin/contracts -D
```

Then create `BasicToken.sol` contract in `src` directory:

```solidity pragma solidity ^0.6.0;
```

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// Example class - a mock class derived from ERC20 contract BasicToken is ERC20 { constructor(uint256 initialBalance) ERC20("Basic", "BSC") public { _mint(msg.sender, initialBalance); } }

```

## Step #3: Compile your smart contract Link to doc {#step-3-compile-your-smart-contract}

To compile your smart contract add the following entry in the `package.json` of your project :

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

Also, add `waffle.json` file in the main directory of your project.

An example of `waffle.json` configuration:

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

You can read more about the Waffle configuration here.

Then just run `yarn build` to compile your smart contract.

You should see that Waffle compiled your contract and placed the resulting JSON output inside the `build` directory.

▶ BasicToken.json

Step #4: Test your smart contract Link to doc {#step-4-test-your-smart-contract}

Step #4.1 Install necessary dependencies Link to doc {#step-4-1}

After we have successfully authored a Smart Contract we can test it. We will use `Waffle` to do it.

Tests in `Waffle` are written using `Mocha` alongside with `Chai`. We can use a different test environment, but `Waffle` matchers only work with `Chai`.

So, we need to add `Chai` to our dependencies :

```
bash yarn add --dev mocha chai
```

Step #4.2 Create test file Link to doc {#step-4-2}

To write our test we need to create `BasicToken.test.ts` file in our test directory.

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

use(solidity)

describe("BasicToken", () => { const [wallet, walletTo] = new MockProvider().getWallets() let token: Contract

beforeEach(async () => { token = await deployContract(wallet, BasicToken, [1000]) }) }) ```

So, we use `deployContract` method from `Waffle` to deploy our token. As arguments, we should pass `wallet`, the compiled json file of our contract and default balance.

`Waffle` also allows us to create a `wallet`, which makes it very easy to deploy a contract.

You can read more about `wallet` here and you can read more about the deploying functionhere.

Let's write a simple test to check balance of our wallet. Since we submitted the value 1000 during the deployment our contract, the balance of our wallet must be 1000 tokens, which we can check in the first test.

```
ts it("Assigns initial balance", async () => { expect(await token.balanceOf(wallet.address)).to.equal(1000) })
```

To run the test use `yarn test`

Step #4.3 Emitting events Link to doc {#step-4-3}

In this tutorial, I want to show you the most useful matchers of`Waffle`, so let's start with the first one.

`Waffle` allows us to test what events where emitted.

In this tutorial, I'll test the `transfer` method of our contract.

In this test, I'll make a transfer from one wallet to another and check whether the`Transfer` event was called.

```
ts it("Transfer emits event", async () => { await expect(token.transfer(walletTo.address, 7)) .to.emit(token, "Transfer") .withArgs(wallet.address, walletTo.address, 7) })
```

Also, a big advantage of this matcher is that we can check which arguments this event was called with by adding`withArgs` to our test.

This will allow us to be sure that our function is being called correctly!

Step #4.4 Revert with message Link to doc {#step-4-4}

`Waffle` allows us to test what message it was reverted with.

We will use `revertedWith` matcher in our test to check it.

We can write a test in which we will perform a transfer for an amount greater than we have on our wallet. And then we'll check if the transaction reverted with the exact message!

```
ts it("Can not transfer above the amount", async () => { await expect(token.transfer(walletTo.address, 1007)).to.be.revertedWith( "VM Exception while processing transaction: revert ERC20: transfer amount exceeds balance" ) })
```

Step #4.5 Change-token-balance Link to doc {#step-4-5}

`Waffle` allows us to check for changes in the balances of the wallets!

We can use the `changeTokenBalance` matcher to check the balance change or the`changeTokenBalances` for a multiple account.

The matcher can accept `numbers`, `strings` and `BigNumbers` as a balance change, while the address should be specified as a wallet or a contract.

Let's write the next test:

```
ts it("Send transaction changes receiver balance", async () => { await expect(() => wallet.sendTransaction({ to: walletTo.address, gasPrice: 0, value: 200 }) ).to.changeBalance(walletTo, 200) })
```

The above is a test for a single wallet.

And the next one for multiple wallets:

```
ts it("Send transaction changes sender and receiver balances", async () => { await expect(() => wallet.sendTransaction({ to: walletTo.address, gasPrice: 0, value: 200 }) ).to.changeBalances([wallet, walletTo], [-200, 200]) })
```

The transaction is expected to be passed as a callback (we need to check the balance before the call) or as a transaction response.

Congratulations {#congratulations}

Congratulations! You've made it through my tutorial. You've taken your first big step towards testing smart contracts with Waffle.

Code from this tutorial you can be find[here](#).

More documentation about `Waffle` available [here](#).