waffle

Marek Kirejczyk

Jan 23, 2021

CONTENTS:

Waffle

Waffle is a library for writing and testing smart contracts.

Sweeter, simpler and faster than Truffle.

Works with ethers-js.

CHAPTER

ONE

PHILOSOPHY:

- **Simpler**: Minimalistic, few dependencies.
- **Sweeter**: Nice syntax, easy to extend.
- **Faster**: Focus on the speed of tests execution.

CHAPTER

TWO

FEATURES:

- Sweet set of chai matchers,
- Easy contract importing from npm modules,
- Fast compilation with native and dockerized solc,
- Typescript compatible,
- Fixtures that help write fast and maintainable test suites,
- Well documented.

CHAPTER

THREE

VERSIONS AND ETHERS COMPATIBILITY

- Use version `0.2.3` with ethers 3 and solidity 4,
- Use version `1.2.0` with ethers 4 and solidity 4,
- Use version `2.*.*` with ethers 4, solidity 4, 5 and ability to use native or dockerized solc.
- Use version `3.*.*` with ethers 5, solidity 4, 5, 6 and ability to use native, dockerized solc or dockerized vyper.

3.1 Getting Started

3.1.1 Installation

To get started, install `ethereum-waffle`:

Yarn

NPM

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

```
npm install --save-dev ethereum-waffle
```

3.1.2 Add external dependency

Add an external library by installing it with yarn or npm:

Yarn

NPM

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

```
npm install @openzeppelin/contracts -D
```

3.1.3 Writing a contract

Below is an example contract written in Solidity. Place it in `src/BasicToken.sol` file of your project:

```
pragma solidity ^0.6.0;

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

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

3.1.4 Compiling the contract

Add the following entry in the `package.json` of your project :

Note: Since Waffle 3.0.0 it recognises `waffle.json` as default configuration file. If your configuration file is called `waffle.json`, it's possible to use just `waffle` to build contracts.

Waffle 3.0.0

Waffle 2.5.0

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

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

In the `waffle.json` file of your project add the following entry:

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

Then run the following command:

Yarn

NPM

```
yarn build
```

```
npm run build
```

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

If you want to know more about how to configure Waffle, see *Configuration*.

3.1.5 Flattener

To flat your smart contracts run:

```
npx waffle flatten
```

In configuration file you can add optional field with path to flatten files:

```
{
  "flattenOutputDirectory": "./custom_flatten"
}
```

3.1.6 Writing tests

After you have successfully authored a Smart Contract you can now think about testing it. Fortunately for you, Waffle is packed with tools that help with that.

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

Run:

Yarn

NPM

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

```
npm install --save-dev mocha chai
```

Below is an example test file for the contract above written with Waffle. Place it under `test/BasicToken.test.ts` file in your project directory:

```
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]);
  });

  it('Assigns initial balance', async () => {
```

(continues on next page)

(continued from previous page)

```
    expect(await token.balanceOf(wallet.address)).to.equal(1000);
  });

  it('Transfer adds amount to destination account', async () => {
    await token.transfer(walletTo.address, 7);
    expect(await token.balanceOf(walletTo.address)).to.equal(7);
  });

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

  it('Can not transfer above the amount', async () => {
    await expect(token.transfer(walletTo.address, 1007)).to.be.reverted;
  });

  it('Can not transfer from empty account', async () => {
    const tokenFromOtherWallet = token.connect(walletTo);
    await expect(tokenFromOtherWallet.transfer(wallet.address, 1))
      .to.be.reverted;
  });

  it('Calls totalSupply on BasicToken contract', async () => {
    await token.totalSupply();
    expect('totalSupply').to.be.calledOnContract(token);
  });

  it('Calls balanceOf with sender address on BasicToken contract', async () => {
    await token.balanceOf(wallet.address);
    expect('balanceOf').to.be.calledOnContractWith(token, [wallet.address]);
  });
});
```

3.1.7 Running tests

Update your `package.json` file to include:

```
{
  "scripts": {
    "build": "waffle",
    "test": "export NODE_ENV=test && mocha",
  }
}
```

And run:

Yarn

NPM

```
yarn test
```

```
npm test
```

You should see the following output:

```
BasicToken
  ✓ Assigns initial balance (67ms)
  ✓ Transfer adds amount to destination account (524ms)
  ✓ Transfer emits event (309ms)
  ✓ Can not transfer above the amount (44ms)
  ✓ Can not transfer from empty account (78ms)
  ✓ Calls totalSupply on BasicToken contract (43ms)
  ✓ Calls balanceOf with sender address on BasicToken contract (45ms)


7 passing (5s)
```

If you want to know more about testing with Waffle, see *Basic testing*.

3.2 Compilation

3.2.1 Using third party libraries

One of the nice things about Waffle is that it enables you to import third party libraries when writing your smart contracts. All you need to do is install the library from `npm`.

For example you can install the popular `@openzeppelin/contracts` package:

Yarn

NPM

```
yarn add @openzeppelin/contracts
```

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

After installing a library, you can import it into your Solidity code:

```
pragma solidity ^0.6.0;

import "@openzeppelin/contracts/token/ERC721/ERC721Full.sol";
```

If you are using a custom `node_modules` location you can configure Waffle to recognize it. Change the `nodeModulesDirectory` setting in your `.waffle.json` file:

```
{
  "nodeModulesDirectory": "path/to/node_modules"
}
```

To read more about configuring Waffle, see *Configuration*.

3.2.2 Reducing compile times

By default, Waffle uses solc-js for compiling smart contracts. The package provides JavaScript bindings for the Solidity compiler. It is slow, but easy to use and install in the JS ecosystem.

Because we value speed and flexibility, we provide some alternatives that you can use with Waffle. There are two other options:

1. Installing solc directly on your computer, see *Using native solc*
2. Using solc installed in a docker container, see *Using dockerized solc*

3.2.3 Using native solc

This is the fastest option but comes with some downsides. The system wide installation means that you are stuck with a single Solidity version across all of your projects. Additionally it might be complicated to install the old versions of the compiler using this method.

We recommend this option if you only care about the latest solidity version.

You can find detailed installation instructions for native `solc` in the Solidity documentation.

Note: You need to install version compatible with your source files.

Change the `compilerType` setting in your `.waffle.json` file:

```
{
  "compilerType": "native"
}
```

To read more about configuring Waffle, see *Configuration*.

When compiling your smart contracts, Waffle will now use the native solc installation.

3.2.4 Using dockerized solc

This is the recommended option if you want flexibility when it comes to the compiler version. It is pretty easy to set up, especially if you have Docker installed.

If you don't have docker visit the Docker documentation to learn how to install it.

After you've installed docker you can install the Solidity compiler. Pull the docker container tagged with the version you are interested in, for example for version 0.4.24:

```
docker pull ethereum/solc:0.4.24
```

Then, change the `compilerType` setting in your `.waffle.json` file:

```
{
  "compilerType": "dockerized-solc",
  "compilerVersion": "0.4.24"
}
```

If no `compilerVersion` is specified the docker tag pulled defaults to `latest`. To read more about configuring Waffle, see *Configuration*.

When compiling your smart contracts, Waffle will now use the docker image you pulled.

3.2.5 Using dockerized vyper

This is the option if you have contracts in Vyper. You will need Docker installed.

To install docker visit the Docker documentation to learn how to do it.

To install dockerized Vyper pull the docker container tagged with the version you are interested in, for example for version 0.1.0:

```
docker pull vyperlang/vyper:0.1.0
```

Then, change the `compilerType` setting in your `.waffle.json` file:

```
{
  "compilerType": "dockerized-vyper",
  "compilerVersion": "0.1.0"
}
```

If no `compilerVersion` is specified the docker tag pulled defaults to `latest`. To read more about configuring Waffle, see *Configuration*.

When compiling your smart contracts, Waffle will now use the docker image you pulled.

3.3 Basic testing

3.3.1 Creating a provider

Creating a mock provider for your tests is super simple.

```
import { MockProvider } from 'ethereum-waffle';
const provider = new MockProvider();
```

This class takes an optional `MockProviderOptions` parameter in the constructor. Then the `ganacheOptions` from `MockProviderOptions` are passed to the underlying ganache-core implementation. You can read more about the options here.

3.3.2 Getting wallets

To obtain wallets that have been prefunded with eth use the provider

```
import { MockProvider } from 'ethereum-waffle';

const provider = new MockProvider();
const [wallet, otherWallet] = provider.getWallets();

// or use a shorthand

const [wallet, otherWallet] = new MockProvider().getWallets();
```

By default this method returns 10 wallets. You can modify the returned wallets, by changing MockProvider configuration.

Waffle 3.0.0

Waffle 2.5.0

```
import { MockProvider } from 'ethereum-waffle';
const provider = new MockProvider({
    ganacheOptions: {
      accounts: [{balance: 'BALANCE IN WEI', secretKey: 'PRIVATE KEY'}]
    }
});
const wallets = provider.getWallets();
```

```
import { MockProvider } from 'ethereum-waffle';
const provider = new MockProvider({
  accounts: [{balance: 'BALANCE IN WEI', secretKey: 'PRIVATE KEY'}]
});
const wallets = provider.getWallets();
```

You can also get an empty random wallet by calling:

```
import { MockProvider } from 'ethereum-waffle';
const provider = new MockProvider();
const wallet = provider.createEmptyWallet();
```

3.3.3 Deploying contracts

Once you compile your contracts using waffle, you can deploy them in your javascript code. It accepts three arguments:

- wallet to send the deploy transaction
- contract information (abi and bytecode)
- contract constructor arguments

Deploy a contract:

```
import BasicTokenMock from "build/BasicTokenMock.json";

token = await deployContract(wallet, BasicTokenMock, [wallet.address, 1000]);
```

The contract information can be one of the following formats:

```
interface StandardContractJSON {
  abi: any;
  evm: {bytecode: {object: any}};
}

interface SimpleContractJSON {
  abi: any[];
  bytecode: string;
}
```

3.3.4 Linking

Link a library:

```
myLibrary = await deployContract(wallet, MyLibrary, []);
link(LibraryConsumer, 'contracts/MyLibrary.sol:MyLibrary', myLibrary.address);
libraryConsumer = await deployContract(wallet, LibraryConsumer, []);
```

Note: You need to use a fully qualified name as the second parameter of the link function (path to the file relative to the root of the project, followed by a colon and the contract name).

3.4 Chai matchers

A set of sweet chai matchers, makes your test easy to write and read. Before you can start using the matchers, you have to tell chai to use the solidity plugin:

```
import chai from "chai";
import { solidity } from "ethereum-waffle";

chai.use(solidity);
```

Below is the list of available matchers:

3.4.1 Bignumbers

Testing equality of big numbers:

```
expect(await token.balanceOf(wallet.address)).to.equal(993);
```

Available matchers for BigNumbers are: *equal*, *eq*, *above*, *gt*, *gte*, *below*, *lt*, *lte*, *least*, *most*.

3.4.2 Emitting events

Testing what events were emitted with what arguments:

```
await expect(token.transfer(walletTo.address, 7))
  .to.emit(token, 'Transfer')
  .withArgs(wallet.address, walletTo.address, 7);
```

Note: The matcher will match `indexed` event parameters of type `string` or `bytes` even if the expected argument is not hashed using `keccack256` first.

Testing with indexed bytes or string parameters. These two examples are equivalent

```
await expect(contract.addAddress("street", "city"))
  .to.emit(contract, 'AddAddress')
  .withArgs("street", "city");
```

(continues on next page)

(continued from previous page)

```
const hashedStreet = ethers.utils.keccak256(ethers.utils.toUtf8Bytes("street"));
const hashedCity = ethers.utils.keccak256(ethers.utils.toUtf8Bytes("city"));
await expect(contract.addAddress(street, city))
  .to.emit(contract, 'AddAddress')
  .withArgs(hashedStreet, hashedCity);
```

3.4.3 Called on contract

Testing if function was called on the provided contract:

```
await token.balanceOf(wallet.address)

expect('balanceOf').to.be.calledOnContract(token);
```

3.4.4 Called on contract with arguments

Testing if function with certain arguments was called on provided contract:

```
await token.balanceOf(wallet.address)

expect('balanceOf').to.be.calledOnContractWith(token, [wallet.address]);
```

3.4.5 Revert

Testing if transaction was reverted:

```
await expect(token.transfer(walletTo.address, 1007)).to.be.reverted;
```

3.4.6 Revert with message

Testing if transaction was reverted with certain message:

```
await expect(token.transfer(walletTo.address, 1007))
  .to.be.revertedWith('Insufficient funds');
```

3.4.7 Change ether balance

Testing whether the transaction changes the balance of the account:

```
await expect(() => wallet.sendTransaction({to: walletTo.address, value: 200}))
  .to.changeEtherBalance(walletTo, 200);

await expect(await wallet.sendTransaction({to: walletTo.address, value: 200}))
  .to.changeEtherBalance(walletTo, 200);
```

`expect` for `changeEtherBalance` gets one of the following parameters:

- **transaction call** : () => Promise<TransactionResponse>

- **transaction response** : TransactionResponse

Note: `changeEtherBalance` won't work if there is more than one transaction mined in the block.

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

The matcher can accept numbers, strings and BigNumbers as a balance change, while the account should be specified either as a Wallet or a Contract.

`changeEtherBalance` ignores transaction fees by default:

```
// Default behavior
await expect(await wallet.sendTransaction({to: walletTo.address, value: 200}))
  .to.changeEtherBalance(wallet, -200);

// To include the transaction fee use:
await expect(await wallet.sendTransaction({to: walletTo.address, gasPrice: 1, value:␣
↪200}))
  .to.changeEtherBalance(wallet, -21200, {includeFee: true});
```

Note: `changeEtherBalance` calls should not be chained. If you need to check changes of the balance for multiple accounts, you should use the `changeEtherBalances` matcher.

3.4.8 Change ether balance (multiple accounts)

Testing whether the transaction changes balance of multiple accounts:

```
await expect(() => wallet.sendTransaction({to: walletTo.address, value: 200}))
  .to.changeEtherBalances([wallet, walletTo], [-200, 200]);

await expect(await wallet.sendTransaction({to: walletTo.address, value: 200}))
  .to.changeEtherBalances([wallet, walletTo], [-200, 200]);
```

Note: `changeEtherBalances` calls won't work if there is more than one transaction mined in the block.

3.4.9 Change token balance

Testing whether the transfer changes the balance of the account:

```
await expect(() => token.transfer(walletTo.address, 200))
  .to.changeTokenBalance(token, walletTo, 200);

await expect(() => token.transferFrom(wallet.address, walletTo.address, 200))
  .to.changeTokenBalance(token, walletTo, 200);
```

Note: The transfer call should be passed to the `expect` as a callback (we need to check the balance before the call).

The matcher can accept numbers, strings and BigNumbers as a balance change, while the account should be specified either as a Wallet or a Contract.

Note: `changeTokenBalance` calls should not be chained. If you need to check changes of the balance for multiple accounts, you should use the `changeTokenBalances` matcher.

3.4.10 Change token balance (multiple accounts)

Testing whether the transfer changes balance for multiple accounts:

```
await expect(() => token.transfer(walletTo.address, 200))
  .to.changeTokenBalances(token, [wallet, walletTo], [-200, 200]);
```

3.4.11 Proper address

Testing if a string is a proper address:

```
expect('0x28FAA621c3348823D6c6548981a19716bcDc740e').to.be.properAddress;
```

3.4.12 Proper private key

Testing if a string is a proper private key:

```
expect('0x706618637b8ca922f6290ce1ecd4c31247e9ab75cf0530a0ac95c0332173d7c5').to.be.
↪properPrivateKey;
```

3.4.13 Proper hex

Testing if a string is a proper hex value of given length:

```
expect('0x70').to.be.properHex(2);
```

3.4.14 Hex Equal

Testing if a string is a proper hex with value equal to the given hex value. Case insensitive and strips leading zeros:

```
expect('0x00012AB').to.hexEqual('0x12ab');
```

3.5 Deprecated matchers

3.5.1 Change balance

Deprecated since version 3.1.2: Use `changeEtherBalance()` instead.

Testing whether the transaction changes the balance of the account:

```
await expect(() => wallet.sendTransaction({to: walletTo.address, gasPrice: 0, value:␣
↪200}))
  .to.changeBalance(walletTo, 200);

await expect(await wallet.sendTransaction({to: walletTo.address, gasPrice: 0, value:␣
↪200}))
  .to.changeBalance(walletTo, 200);
```

`expect` for `changeBalance` gets one of the following parameters:

- **transaction call** : () => Promise<TransactionResponse>
- **transaction response** : TransactionResponse

Note: `changeBalance` won't work if there is more than one transaction mined in the block.

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

The matcher can accept numbers, strings and BigNumbers as a balance change, while the account should be specified either as a Wallet or a Contract.

Note: `changeBalance` calls should not be chained. If you need to check changes of the balance for multiple accounts, you should use the `changeBalances` matcher.

3.5.2 Change balance (multiple accounts)

Deprecated since version 3.1.2: Use `changeEtherBalances()` instead.

Testing whether the transaction changes balance of multiple accounts:

```
await expect(() => wallet.sendTransaction({to: walletTo.address, gasPrice: 0, value:␣
↪200}))
  .to.changeBalances([wallet, walletTo], [-200, 200]);

await expect(await wallet.sendTransaction({to: walletTo.address, gasPrice: 0, value:␣
↪200}))
  .to.changeBalances([wallet, walletTo], [-200, 200]);
```

Note: `changeBalances` calls won't work if there is more than one transaction mined in the block.

3.6 Fixtures

When testing code dependent on smart contracts, it is often useful to have a specific scenario play out before every test. For example, when testing an ERC20 token, one might want to check whether each and every address can or cannot perform transfers. Before each of those tests however, you have to deploy the ERC20 contract and maybe transfer some funds.

The repeated deployment of contracts might slow down the test significantly. This is why Waffle allows you to create fixtures - testing scenarios that are executed once and then remembered by making snapshots of the blockchain. This speeds up the tests considerably.

Example:

Waffle 3.0.0

Waffle 2.5.0

```
import {expect} from 'chai';
import {loadFixture, deployContract} from 'ethereum-waffle';
import BasicTokenMock from './build/BasicTokenMock';

describe('Fixtures', () => {
  async function fixture([wallet, other], provider) {
    const token = await deployContract(wallet, BasicTokenMock, [
      wallet.address, 1000
    ]);
    return {token, wallet, other};
  }

  it('Assigns initial balance', async () => {
    const {token, wallet} = await loadFixture(fixture);
    expect(await token.balanceOf(wallet.address)).to.equal(1000);
  });

  it('Transfer adds amount to destination account', async () => {
    const {token, other} = await loadFixture(fixture);
    await token.transfer(other.address, 7);
    expect(await token.balanceOf(other.address)).to.equal(7);
  });
});
```

```
import {expect} from 'chai';
import {loadFixture, deployContract} from 'ethereum-waffle';
import BasicTokenMock from './build/BasicTokenMock';

describe('Fixtures', () => {
  async function fixture(provider, [wallet, other]) {
    const token = await deployContract(wallet, BasicTokenMock, [
      wallet.address, 1000
    ]);
    return {token, wallet, other};
  }

  it('Assigns initial balance', async () => {
    const {token, wallet} = await loadFixture(fixture);
    expect(await token.balanceOf(wallet.address)).to.equal(1000);
  });
```

(continues on next page)

(continued from previous page)

```
  it('Transfer adds amount to destination account', async () => {
    const {token, other} = await loadFixture(fixture);
    await token.transfer(other.address, 7);
    expect(await token.balanceOf(other.address)).to.equal(7);
  });
});
```

Fixtures receive a provider and an array of wallets as an argument. By default, the wallets are obtained by calling `getWallets` and the provider by `createMockProvider`. You can, however, override those by using a custom fixture loader.

Waffle 3.0.0

Waffle 2.5.0

```
import {createFixtureLoader} from 'ethereum-waffle';

const loadFixture = createFixtureLoader(myWallets, myProvider);

// later in tests
await loadFixture((myWallets, myProvider) => {
  // fixture implementation
});
```

```
import {createFixtureLoader} from 'ethereum-waffle';

const loadFixture = createFixtureLoader(myProvider, myWallets);

// later in tests
await loadFixture((myProvider, myWallets) => {
  // fixture implementation
});
```

3.7 Mock contract

Mocking your smart contract dependencies.

3.7.1 Usage

Create an instance of a mock contract providing the `ABI` of the smart contract you want to mock:

```
import {deployMockContract} from '@ethereum-waffle/mock-contract';

...

const mockContract = await deployMockContract(wallet, contractAbi);
```

The mock contract can now be integrated into other contracts by using the `address` attribute. Return values for mocked functions can be set using:

```
await mockContract.mock.<nameOfMethod>.returns(<value>)
await mockContract.mock.<nameOfMethod>.withArgs(<arguments>).returns(<value>)
```

Methods can also be set up to be reverted using:

```
await mockContract.mock.<nameOfMethod>.reverts()
await mockContract.mock.<nameOfMethod>.revertsWithReason(<reason>)
await mockContract.mock.<nameOfMethod>.withArgs(<arguments>).reverts()
await mockContract.mock.<nameOfMethod>.withArgs(<arguments>).revertsWithReason(
↪<reason>)
```

3.7.2 Example

The example below illustrates how `mock-contract` can be used to test the very simple `AmIRichAlready` contract.

```
pragma solidity ^0.6.0;

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
}

contract AmIRichAlready {
    IERC20 private tokenContract;
    uint private constant RICHNESS = 1000000 * 10 ** 18;

    constructor (IERC20 _tokenContract) public {
        tokenContract = _tokenContract;
    }

    function check() public view returns (bool) {
        uint balance = tokenContract.balanceOf(msg.sender);
        return balance > RICHNESS;
    }
}
```

We are mostly interested in the `tokenContract.balanceOf` call. Mock contract will be used to mock exactly this call with values that are relevant for the return of the `check()` method.

```
const {use, expect} = require('chai');
const {ContractFactory, utils} = require('ethers');
const {MockProvider} = require('@ethereum-waffle/provider');
const {waffleChai} = require('@ethereum-waffle/chai');
const {deployMockContract} = require('@ethereum-waffle/mock-contract');

const IERC20 = require('../build/IERC20');
const AmIRichAlready = require('../build/AmIRichAlready');

use(waffleChai);

describe('Am I Rich Already', () => {
  async function setup() {
    const [sender, receiver] = new MockProvider().getWallets();
    const mockERC20 = await deployMockContract(sender, IERC20.abi);
    const contractFactory = new ContractFactory(AmIRichAlready.abi, AmIRichAlready.
↪bytecode, sender);
    const contract = await contractFactory.deploy(mockERC20.address);
    return {sender, receiver, contract, mockERC20};
  }
```

(continues on next page)

(continued from previous page)

```
  it('returns false if the wallet has less then 1000000 coins', async () => {
    const {contract, mockERC20} = await setup();
    await mockERC20.mock.balanceOf.returns(utils.parseEther('999999'));
    expect(await contract.check()).to.be.equal(false);
  });

  it('returns true if the wallet has at least 1000000 coins', async () => {
    const {contract, mockERC20} = await setup();
    await mockERC20.mock.balanceOf.returns(utils.parseEther('1000001'));
    expect(await contract.check()).to.equal(true);
  });

  it('reverts if the ERC20 reverts', async () => {
    const {contract, mockERC20} = await setup();
    await mockERC20.mock.balanceOf.reverts();
    await expect(contract.check()).to.be.revertedWith('Mock revert');
  });

  it('returns 1000001 coins for my address and 0 otherwise', async () => {
    const {contract, mockERC20, sender, receiver} = await setup();
    await mockERC20.mock.balanceOf.returns('0');
    await mockERC20.mock.balanceOf.withArgs(sender.address).returns(utils.parseEther(
↪'1000001'));

    expect(await contract.check()).to.equal(true);
    expect(await contract.connect(receiver.address).check()).to.equal(false);
  });
});
```

3.8 ENS

3.8.1 Creating a ENS

Creating a simple ENS for testing with `ENS`.

```
import {MockProvider} from '@ethereum-waffle/provider';
import {deployENS, ENS} from '@ethereum-waffle/ens';

const provider = new MockProvider();
const [wallet] = provider.getWallets();
const ens: ENS = await deployENS(wallet);
```

This class takes a `wallet` in the constructor. The `wallet` available in `MockProvider` class in package `@ethereum-waffle/provider`.

3.8.2 Setup ENS

Note: The feature was introduced in Waffle 3.0.0

Also, if you use `MockProvider`, you can use `setupENS()` function in `MockProvider`, to create and setup simple `ENS`.

```
import {MockProvider} from '@ethereum-waffle/provider';

const provider = new MockProvider();
await provider.setupENS();
await provider.ens.createTopLevelDomain('test');
```

`setupENS()` method employs the last of the provider's `wallets` by default, but you can pass your own `wallet` as an argument for `setupENS(wallet)`.

Also `setupENS()` method saves ENS address in the provider's networks.

3.8.3 Creating top level domain

Use `createTopLevelDomain` function to create a top level domain:

```
await ens.createTopLevelDomain('test');
```

3.8.4 Creating sub domain

Use `createSubDomain` function for creating a sub domain:

```
await ens.createSubDomain('ethworks.test');
```

3.8.5 Creating sub domain with options

Note: The feature was introduced in Waffle 3.0.0

It is also possible to create a sub domain recursively, if the top domain doesn't exist, by specifying the appropriate option:

```
await ens.createSubDomain('waffle.ethworks.tld', {recursive: true});
```

3.8.6 Setting address

Use `setAddress` function for setting address for the domain:

```
await ensBuilder.setAddress('vlad.ethworks.test', '0x001...03');
```

3.8.7 Setting address with options

Note: The feature was introduced in Waffle 3.0.0

It is also possible to set an address for domain recursively, if the domain doesn't exist, by specifying the appropriate option:

```
await ens.setAddress('vlad.waffle.ethworks.tld', '0x001...03', {recursive: true});
```

Use `setAddressWithReverse` function for setting address for the domain and make this domain reverse. Add recursive option if the domain doesn't exist:

```
await ens.setAddressWithReverse('vlad.ethworks.tld', wallet, {recursive: true});
```

3.9 Configuration

3.9.1 Configuration file

While Waffle works well enough without any configurations, advanced users might want to excert more control over what happens when they use Waffle in their projects.

This is why we made it very easy to configure Waffle to meet your needs. All you need to do is create a `waffle.json` file inside your project and point waffle to it.

First create your `waffle.json` configuration file:

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

Note: Each configuration is optional.

Afterwards update your `package.json` build script:

Note: Since Waffle 3.0.0 it recognises `waffle.json` as the default configuration file. If your configuration file is called `waffle.json`, it's possible to use just `waffle` to build contracts.

Waffle 3.0.0

Waffle 2.5.0

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

{

“scripts”: { “build”: “waffle waffle.json”

}

}

Configuration options:

- *sourceDirectory*
- *outputDirectory*
- *nodeModulesDirectory*
- *cacheDirectory*
- *compilerType*
- *compilerVersion*
- *compilerAllowedPaths*
- *compilerOptions*
- *outputHumanReadableAbi*
- *outputType*

sourceDirectory

You can specify a custom path to the directory containing your smart contracts. Waffle uses `./contracts` as the default value for `sourceDirectory`. The path you provide will be resolved relative to the current working directory.

Example:

```
{
  "sourceDirectory": "./custom/path/to/contracts"
}
```

outputDirectory

You can specify a custom path to the directory to which Waffle saves the compilation output. Waffle uses `./build` as the default value for `outputDirectory`. The path you provide will be resolved relative to the current working directory.

Example:

```
{
  "outputDirectory": "./custom/path/to/output"
}
```

nodeModulesDirectory

You can specify a custom path to the `node_modules` folder which Waffle will use to resolve third party dependencies. Waffle uses `./node_modules` as the default value for `nodeModulesDirectory`. The path you provide will be resolved relative to the current working directory.

For more information about third party libraries, see *Using third party libraries*.

Example:

```
{
  "nodeModulesDirectory": "./custom/path/to/node_modules"
}
```

cacheDirectory

When compiling using `solcjs` and using a non-default `compilerVersion` Waffle downloads the necessary `solcjs` binary from a remote server. This file is cached to speed up subsequent runs. You can specify a custom path to the directory in which caches are saved. Waffle uses `./cache` as the default value for `cacheDirectory`. The path you provide will be resolved relative to the current working directory.

Example:

```
{
  "cacheDirectory": "./custom/path/to/cache"
}
```

compilerType

Specifies the compiler to use. For more information see: *Reducing compile times*. Allowed values:

- `solcjs`
- `native`
- `dockerized-solc`
- `dockerized-vyper`

Waffle uses `solcjs` as the default value for `compilerType`.

Example:

```
{
  "compilerType": "dockerized-solc"
}
```

compilerVersion

Specifies the version of the compiler. Should be a semver string like `0.5.9`. You can use it with `"compilerType": "solcjs"` or `"compilerType": "dockerized-solc"`.

When using `"compilerType": "solcjs"` you can also specify the exact commit that will be used or a path to a specific solc module dependency.

To find a specific commit please consult the list of available solc versions.

Examples:

```
{
  "compilerType": "dockerized-solc",
  "compilerVersion": "0.4.24"
}
```

```
{
  "compilerType": "solcjs",
  "compilerVersion": "v0.4.24+commit.e67f0147"
}
```

```
{
  "compilerType": "solcjs",
  "compilerVersion": "./node_modules/solc"
}
```

compilerAllowedPaths

The `solc` compiler has restrictions on paths it can access for security reasons. The value of `compilerAllowedPaths` will be passed as a command line argument: `solc --allow-paths <VALUE>`.

This is especially useful if you are doing a monorepo setup with Lerna, see: *Usage with Lernajs*.

Example:

```
{
  "compilerAllowedPaths": ["../contracts"]
}
```

compilerOptions

You can customize the behaviour of `solc` by providing custom settings for it. All of the information is provided in the Solidity documentation. Value of the `compilerOptions` configuration setting will be passed to `solc` as `settings`.

For detailed list of options go to solidity documentation (sections: 'Setting the EVM version to target', 'Target options' and 'Compiler Input and Output JSON Description').

Example:

```
{
  "compilerOptions": {
    "evmVersion": "constantinople"
  }
}
```

outputType

See: *KLAB compatibility*.

outputHumanReadableAbi

Waffle supports Human Readable Abi.

In order to enable its output, you need to set `outputHumanReadableAbi` to `true` in your config file:

```
{
  "outputHumanReadableAbi": true
}
```

For the compiled contracts you will now see the following in the output:

```
{
  "humanReadableAbi": [
    "constructor(uint256 argOne)",
    "event Bar(bool argOne, uint256 indexed argTwo)",
    "event FooEvent()",
    "function noArgs() view returns(uint200)",
    "function oneArg(bool argOne)",
    "function threeArgs(string argOne, bool argTwo, uint256[] argThree) view␣
↪returns(bool, uint256)",
    "function twoReturns(bool argOne) view returns(bool, uint256)"
  ]
}
```

3.9.2 Other configuration file formats

Waffle supports the following configuration file formats:

JSON:

```
{
  "sourceDirectory": "./src/contracts",
}
```

JavaScript:

```
module.exports = {
  sourceDirectory: './src/contracts'
}
```

The configuration can even be a promise

```
module.exports = Promise.resolve({
  sourceDirectory: './src/contracts'
})
```

Hint: This is a powerful feature if you want to asynchronously load different compliation configurations in different environments. For example, you can use native solc in CI for faster compilation, while deciding the exact solc-js

version locally based on the contract versions being used. Since many of those operations are asynchronous, you'll most likely be returning a Promise to waffle to handle.

3.9.3 Setting Solidity compiler version

See *compilerVersion*.

3.9.4 Usage with Truffle

Waffle output should be compatible with Truffle by default.

3.9.5 Custom compiler options

See *compilerOptions*.

3.9.6 KLAB compatibility

The default compilation process is not compatible with KLAB (a formal verification tool, see: https://github.com/dapphub/klab). To compile contracts to work with KLAB one must:

1. Set appropriate compiler options, i.e.:

```
compilerOptions: {
  outputSelection: {
    "*": {
      "*": [ "evm.bytecode.object", "evm.deployedBytecode.object",
             "abi" ,
             "evm.bytecode.sourceMap", "evm.deployedBytecode.sourceMap" ],

      "": [ "ast" ]
    },
  }
}
```

2. Set appropriate output type. We support two types: one (default) generates a single file for each contract and the other (KLAB friendly) generates one file (Combined-Json.json) combining all contracts. The latter type does not meet (in contrary to the first one) all official solidity standards since KLAB requirements are slightly modified. Set the output in the config file:

Possible options are:

- *'multiple'*: a single file for each contract;
- *'combined'*: one KLAB friendly file;
- *'all'*: generates both above outputs;
- *'minimal'*: a single file for each contract with minimal information (just abi and bytecode).

Note: *'minimal'* option was introduced in Waffle 3.0.0.

```
{
  "outputType": "combined"
}
```

An example of full KLAB friendly config file:

```
module.exports = {
  compilerType: process.env.WAFFLE_COMPILER,
  outputType: 'all',
  compilerOptions: {
    outputSelection: {
      "*": {
        "*": [ "evm.bytecode.object", "evm.deployedBytecode.object",
               "abi" ,
               "evm.bytecode.sourceMap", "evm.deployedBytecode.sourceMap" ],

        "": [ "ast" ]
      },
    }
  }
};
```

3.9.7 Monorepo

Waffle works well with mono-repositories. It is enough to set up a common `nodeModulesDirectory` in the configuration file to make it work. We recommend using yarn workspaces and wsrun for monorepo management.

3.9.8 Usage with Lernajs

Waffle works with lerna, but requires additional configuration. When lerna cross-links npm packages in monorepo, it creates symbolic links to the original catalog. They lead to sources files located beyond allowed paths. This process breaks compilation with native solc.

If you see the following message in your monorepo setup:

```
contracts/Contract.sol:4:1: ParserError: Source ".../monorepo/node_modules/
↪YourProjectContracts/contracts/Contract.sol" not found: File outside of allowed␣
↪directories.
import "YourProjectContracts/contracts/Contract.sol";
```

you probably need to add allowedPath to your waffle configuration.

Assuming you have the following setup:

```
/monorepo
  /YourProjectContracts
    /contracts
  /YourProjectDapp
    /contracts
```

Add to waffle configuration in YourProjectDapp:

```
{
  "compilerAllowedPaths": ["../YourProjectContracts"]
}
```

That should solve a problem.

Currently Waffle does not support a similar feature for dockerized solc.

3.10 Migration guides

3.10.1 Migration from Waffle 2.2.0 to Waffle 2.3.0

Created monorepo

Waffle was internally migrated to a monorepo. Thanks to this, you can now use parts of waffle individually. We provide the following packages:

- `ethereum-waffle` - core package exporting everything
- `ethereum-compiler` - compile your contracts programmatically
- `ethereum-chai` - chai matchers for better unit testing
- `ethereum-provider` - mock provider to interact with an in-memory blockchain

Created MockProvider class

We added MockProvider class. It changed the creation of the provider.

Waffle 2.2.0

```
await createMockProvider(options);
```

Waffle 2.3.0

```
provider = new MockProvider();
```

Reorganise getWallets() method

Waffle 2.2.0

```
await getWallets(provider);
```

Waffle 2.3.0

```
new MockProvider().getWallets()
```

3.10.2 Migration from Waffle 2.3.0 to Waffle 2.4.0

Renamed configuration options

We renamed configuration options to compile contracts:

- `sourcesPath` - renamed to *sourceDirectory*
- `targetPath` - renamed to *outputDirectory*
- `npmPath` - renamed to *nodeModulesDirectory*

- `compiler` - renamed to *compilerType*
- `docker-tag` - replaced by *compilerVersion*
- `solcVersion` - replaced by *compilerVersion*
- `legacyOutput` - removed, setting it to false gave no effect
- `allowedPaths` - renamed to *compilerAllowedPaths*
- `ganacheOptions` - removed, wasn't used by the compiler

3.10.3 Migration from Waffle 2.5.* to Waffle 3.0.0

There are some new functionality and some slight refactoring and improved paradigms in Waffle v3.

Removed deprecated APIs from the provider

In `Waffle 3.0.0` we remove deprecated APIs from the provider, such as `createMockProvider` and `getGanacheOptions`.

Swapped arguments for Fixture

In `Waffle 3.0.0` we swapped arguments for Fixture, because the provider argument is very rarely used compared to wallets. So such implementation should be more convenient for users.

Waffle 2.5.0

```
function createFixtureLoader(overrideProvider?: MockProvider, overrideWallets?:␣
↪Wallet[]);
```

Waffle 3.0.0

```
function createFixtureLoader(overrideWallets?: Wallet[], overrideProvider?:␣
↪MockProvider);
```

Waffle 2.5.0

```
import {expect} from 'chai';
import {loadFixture, deployContract} from 'ethereum-waffle';
import BasicTokenMock from './build/BasicTokenMock';

describe('Fixtures', () => {
  async function fixture(provider, [wallet, other]) {
    const token = await deployContract(wallet, BasicTokenMock, [
      wallet.address, 1000
    ]);
    return {token, wallet, other};
  }

  it('Assigns initial balance', async () => {
    const {token, wallet} = await loadFixture(fixture);
    expect(await token.balanceOf(wallet.address)).to.equal(1000);
  });
});
```

Waffle 3.0.0

```
import {expect} from 'chai';
import {loadFixture, deployContract} from 'ethereum-waffle';
import BasicTokenMock from './build/BasicTokenMock';

describe('Fixtures', () => {
  async function fixture([wallet, other], provider) {
    const token = await deployContract(wallet, BasicTokenMock, [
      wallet.address, 1000
    ]);
    return {token, wallet, other};
  }

  it('Assigns initial balance', async () => {
    const {token, wallet} = await loadFixture(fixture);
    expect(await token.balanceOf(wallet.address)).to.equal(1000);
  });
});
```

Added automatic recognising waffle.json config without cli argument

Waffle recognises `waffle.json` as the default configuration file. If your configuration file is called `waffle.json`, it's possible to use just `waffle` to build contracts.

In Waffle 2.5.0, If the argument has not been provided, the Waffle uses the default configuration.

Waffle 2.5.0

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

Waffle 3.0.0

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

Introduced MockProviderOptions

We added MockProviderOptions. It will be convenient in the future, when the provider may need some options other than `ganacheOptions`.

Waffle 2.5.0

```
import {expect} from 'chai';
import {Wallet} from 'ethers';
import {MockProvider} from 'ethereum-waffle';
import {deployToken} from './BasicToken';

describe('INTEGRATION: MockProvider', () => {
  it('accepts options', () => {
```

(continues on next page)

(continued from previous page)

```
    const original = Wallet.createRandom();
    const provider = new MockProvider({
        accounts: [{balance: '100', secretKey: original.privateKey}]
    });
    const wallets = provider.getWallets();
    expect(wallets.length).to.equal(1);
    expect(wallets[0].address).to.equal(original.address);
  });
});
```

Waffle 3.0.0

```
import {expect} from 'chai';
import {Wallet} from 'ethers';
import {MockProvider} from 'ethereum-waffle';
import {deployToken} from './BasicToken';

describe('INTEGRATION: MockProvider', () => {
  it('accepts options', () => {
    const original = Wallet.createRandom();
    const provider = new MockProvider({
      ganacheOptions: {
        accounts: [{balance: '100', secretKey: original.privateKey}]
      }
    });
    const wallets = provider.getWallets();
    expect(wallets.length).to.equal(1);
    expect(wallets[0].address).to.equal(original.address);
  });
});
```

Dropped support for contract interface

We dropped support for contract interface because it duplicated contract ABI. Also `interface` is a keyword in typescript, so we decided not to use this field. Now we support just `contract.abi`.

Waffle 2.5.0

```
{
  "abi": [
    ...
  ],
  "interface: [
    ...
  ],
  "evm": {
    ...
  },
  "bytecode": "..."
}
```

Waffle 3.0.0

```
{
  "abi": [
```

(continues on next page)

(continued from previous page)

```
    {...}
  ],
  "evm": {
    ...
  },
  "bytecode": "..."
}
```