waffle Documentation

Marek Kirejczyk

Aug 23, 2019

Contents:

1 Philosophy: **3**

2 Features: **5**

3 Versions and ethers compatibility **7**

3.1 Getting Started 7

3.2 Features 9

3.3 Fast compilation 13

3.4 Configuration 15

Waffle

Waffle is a library for writing and testing smart contracts.

Sweeter, simpler and faster than Truffle.

Works with ethers-js.

CHAPTER 1

Philosophy:

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

CHAPTER 2

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 3

Versions and ethers compatibility

- Use version 0.2.3+ with ethers 3.* and solidity 4.*
- Use version 1.0.0+ with ethers 4.* and solidity 4.*
- Use version 2.0.5-beta with ethers 4.*; solidity 4, 5 and to use experimental native solc and dockerized solc.

3.1 Getting Started

3.1.1 Installation

To start using with npm, type:

```
npm i ethereum-waffle
```

or with Yarn:

```
yarn add ethereum-waffle
```

3.1.2 Adding a contract

Below is example contract written in Solidity. Place it in `contracts` directory of your project:

```
pragma solidity ^0.5.1;

import "openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";


// Example class - a mock class using delivering from ERC20
contract BasicTokenMock is ERC20 {
  constructor(address initialAccount, uint256 initialBalance) public {
    super._mint(initialAccount, initialBalance);
```

(continues on next page)

(continued from previous page)

```
  }
}
```

3.1.3 Writing first tests

Belows is example test written for the contract above written with Waffle. Place it in `test` directory of your project:

```
import chai from 'chai';
import {createMockProvider, deployContract, getWallets, solidity} from 'ethereum-
↪waffle';
import BasicTokenMock from './build/BasicTokenMock';
import MyLibrary from './build/MyLibrary';
import LibraryConsumer from './build/LibraryConsumer';

chai.use(solidity);
const {expect} = chai;

describe('INTEGRATION: Example', () => {
  let provider = createMockProvider();
  let [wallet, walletTo] = getWallets(provider);
  let token;

  beforeEach(async () => {
    token = await deployContract(wallet, BasicTokenMock, [wallet.address, 1000]);
  });

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

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

});
```

3.1.4 Compilation

To compile contracts simply type:

```
npx waffle
```

To compile using custom configuration file:

```
npx waffle config.json
```

Example configuration file looks like this:

```
{
  "sourcesPath": "./contracts",
  "targetPath": "./build",
  "npmPath": "./node_modules"
}
```

3.1.5 Running tests

To run test type in the console:

```
mocha
```

3.1.6 Adding a task

For convince, you can add a task to your `package.json`. In the sections `scripts`, add following line:

```
"test": "waffle && test"
```

Now you can build and test your contracts with one command:

```
npm test
```

3.2 Features

3.2.1 Basic features

Create a mock provider

To create a mock provider for running your contracts test against it, e.g.:

```
provider = createMockProvider();
```

To modify default provider behavior createMockProvider() takes optional Ganache options parameter. It can be object with specified options or absolute path to waffle.json or another config file, e.g.:

```
provider = createMockProvider({gasLimit: 0x6691b7, gasPrice: 0x77359400});

provider = createMockProvider('./waffle.json');
```

(continues on next page)

(continued from previous page)

```
waffle.json:
  {
    ...
    "ganacheOptions": {
      "gasLimit": "0x6691b7",
      "gasPrice": "0x77359400"
    }
  }
```

Get example wallets

Get wallets you can use to sign transactions:

```
[wallet, walletTo] = getWallets(provider);
```

You can get up to ten wallets.

Deploy contract

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;
}
```

Linking

Link a library:

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

Note: Note: As the second parameter of the link function, you need to use a fully qualified name (full path to file, followed by a colon and the contract name).

Import contracts from npm library

Install library:

```
npm i open-zeppelin
```

Import solidity files from project imported to npm modules:

```
import "openzeppelin-solidity/contracts/math/SafeMath.sol";
```

3.2.2 Chai matchers

A set of sweet chai matchers, makes your test easy to write and read. Below is the list of available matchers:

Bignumbers

Testing equality of big numbers:

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

Available matchers for BigNumbers are: *equal*, *eq*, *above*, *below*, *least*, *most*.

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);
```

Revert

Testing if transaction was reverted:

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

Revert with message

Testing if transaction was reverted with certain message:

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

Change balance

Testing whether the transaction changes balance of an account

```
await expect(() => myContract.transferWei(receiverWallet.address, 2))
  .to.changeBalance(receiverWallet, 2);
```

Note: transaction 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 address should be specified as a wallet.

Note: `changeBalance` calls should not be chained. If you need to chain it, you probably want to use `changeBalances` matcher.

Change balance (multiple accounts)

Testing whether the transaction changes balance for multiple accounts:

```
await expect(() => myContract.transferWei(receiverWallet.address, 2))
  .to.changeBalances([senderWallet, receiverWallet], [-2, 2]);
```

Proper address

Testing if string is a proper address:

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

Proper private key

Testing if string is a proper secret:

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

Proper hex

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

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

3.2.3 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 that specific addresses 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 significantly speeds up the tests.

Example:

```
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.eq(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.eq(7);
  });
});
```

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

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

const loadFixture = createFixtureLoader(myProvider, myWallets);

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

3.3 Fast compilation

By default, Waffle uses solcjs. Solcjs is solidity complier cross-complied to javascript. It is slow, but easy to install. As an alternative, you can use the native Solidity compiler, which is faster. There are two options: 1) Native solc 2) Dockerized native solc

3.3.1 Native solc

This option is by far the fastest but requires you to install native solidity. If you need an old version that might be somewhat complicated and require you to build *solidity* from sources. Therefore it is the recommended option if you want to use latest solidity version.

You can find detailed installation instructions for native *solc* in documentation.

Note: You need to install version compatible with your sources.

If you need the latest version that is pretty straight forward, see installation instructions below.

Installation

MacOS

To install lastest versions on MacOS:

```
brew install solidity
```

To install other versions, it seems that currently, you need to build it from source:

1. Download sources from release list on GitHub
2. Follow installation instructions in the documentation

Ubuntu

To install lastest versions on Ubuntu:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

Project setup

Setup compiler in your waffle configuration file:

```
{
  ...
  "compiler": "native"
}
```

You can now run tests with native solc, eg:

```
npx waffle
```

3.3.2 Dockerized solc

This option is pretty easy to install especially if you have Docker installed. This is the recommended option if you need to use old solidity version. If you don't have docker use following instructions to install it.

Pull solc docker container tagged with the version you are interested in, for example for version 0.4.24:

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

Than setup compiler in your waffle configuration file:

```
{
  ...
  "compiler": "dockerized-solc",
  "docker-tag": "0.4.24"
}
```

Default docker tag is *latest*.

You can now run tests in docker.

3.4 Configuration

3.4.1 Waffle configuration file

Waffle takes as a first argument configuration file. The configuration file can be of type JSON, e.g.:

```
{
  "sourcesPath": "./some_custom/contracts_path",
  "targetPath": "../some_custom/build",
  "npmPath": "./other/node_modules"
}
```

Configuration can also be of type js, e.g.:

```
module.exports = {
  npmPath: "../node_modules",
  compiler: process.env.WAFFLE_COMPILER,
  legacyOutput: true
};
```

Native and dockerized solc compiler configuration is described in "Fast compilation" section.

Configuration can even be a Promise in a js, e.g.:

```
module.exports = new Promise((resolve, reject) => {
  resolve({
    "compiler": "native"
  });
});
```

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, whereas 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.

3.4.2 Solcjs and version management

Solcjs allows switching used version of solidity compiler on the fly. To set up a chosen version of solidity compiler add the following line in the Waffle configuration file:

```
{
  ...
  "solcVersion": "v0.4.24+commit.e67f0147"
}
```

Version naming is somewhat counter-intuitive. You can deduce version name from list available here.

Also Solcjs can be provided by passing path to Solcjs directory. Path have to be relative to working directory.

```
{
  ...
  "solcVersion": "./node_modules/solc"
}
```

3.4.3 Legacy / Truffle compatibility

Starting with Waffle 2.0, the format of contract output .json files in solidity standard JSON. This is not compatible with older Waffle versions and with Truffle. You can generate files that are compatible with both current and previous versions by adding "legacyOutput": "true" flag Waffle in the configuration file:

```
{
  ...
  "legacyOutput": "true"
}
```

3.4.4 Custom compiler options

To provide custom compiler options in waffle configuration file use compilerOptions section. Example below.

```
{
  "compilerOptions": {
    "evmVersion": "constantinople"
  },
  "compiler": "native"
}
```

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').

3.4.5 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 single file for each contract and second (KLAB friendly) generates one file (Combined-Json.json) combining all contracts. The second type does not meet (in contrary to the first one) all official solidity standards since KLAB requirements are slightly modified. To choice of the output is set in config file, i.e.:

```
outputType: 'combined'
```

Possible options are: - *'multiple'*: single file for each contract; - *'combined'*: one KLAB friendly file; - *'all'*: generates both above outputs.

An example of full KLAB friendly config file:

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

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

3.4.6 Monorepo

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

Lernajs + Native solc

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

If you see a message like below 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:

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

That should solve a problem.

Currently Waffle does not support similar feature for dockerized solc.

3.4.7 Human Readable Abi

Waffle supports *Human Readable Abi <https://blog.ricmoo.com/human-readable-contract-abis-in-ethers-js-141902f4d917>*.

In order to enable its output you need to specify a special flag in your config file:

```
{
  ...
  outputHumanReadableAbi: true
}
```

You will now see the following in your 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)"
  ]
}
```