waffle Documentation

Marek Kirejczyk

Feb 05, 2020

Contents:

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

3.1 Getting Started

3.1.1 Installation

To get started install `ethereum-waffle` with yarn:

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

Or if you prefer using npm:

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

3.1.2 Writing a contract

Below is example contract written in Solidity. Save it as `Counter.sol` inside the `contracts` directory of your project.

```
pragma solidity ^0.5.0;

contract Counter {
  event Increment (uint256 by);

  uint256 public value;

```

(continues on next page)

(continued from previous page)

```
  constructor (uint256 initialValue) public {
    value = initialValue;
  }

  function increment (uint256 by) public {
    // NOTE: You should use SafeMath in production code
    value += by;
    emit Increment(by);
  }
}
```

3.1.3 Compiling the contract

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

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

Then run the following command::

```
yarn build
```

Or if you prefer npm:

```
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*.

3.1.4 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 alongide with Chai. You can use a different test environment, but Waffle matchers only work with `chai`.

Run:

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

Or:

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

Belows is an example test file for the contract above written with Waffle. You can save the file as `Counter.test.js` in the `test` directory of your project.

```
const {use, expect} = require('chai');
const {solidity, createMockProvider, getWallets, deployContract} = require('ethereum-
↪waffle');
const Counter = require('../build/Counter.json');

use(solidity);

describe('Counter smart contract', () => {
  const provider = createMockProvider();
  const [wallet] = getWallets(provider);

  async function deployCounter (initialValue) {
    const counter = await deployContract(
      wallet, // a wallet to sign transactions
      Counter, // the compiled output
      [initialValue], // arguments to the smart contract constructor
    );
    return counter; // an ethers 'Contract' class instance
  }

  it('sets initial value in the constructor', async () => {
    const counter = await deployCounter(200);
    expect(await counter.value()).to.equal(200);
  });

  it('can increment the value', async () => {
    const counter = await deployCounter(200);
    await counter.increment(42);
    expect(await counter.value()).to.equal(242);
  });

  it('emits the Increment event', async () => {
    const counter = await deployCounter(200);
    await expect(counter.increment(42))
      .to.emit(counter, 'Increment')
      .withArgs(42);
  });
});
```

3.1.5 Running tests

Update your `package.json` file to include:

```
{
  "scripts": {
    "build": "waffle",
    "test": "mocha"
  }
}
```

And run:

```
yarn test
```

Or:

```
npm test
```

You should see the following output:

```
Counter smart contract
  ✓ sets initial value in the constructor (140ms)
  ✓ can increment the value (142ms)
  ✓ emits the Increment event (114ms)

3 passing (426ms)
```

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 add @openzeppelin/contracts
```

Or if you prefer npm:

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

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

```
pragma solidity ^0.5.0;

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 `npmPath` setting in your `.waffle.json` file:

```
{
  "npmPath": "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. A 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 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 `compiler` setting in your `.waffle.json` file:

```
{
  "compiler": "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 `compiler` setting in your `.waffle.json` file:

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

The default value for `docker-tag` is `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 options parameter in the constructor. The options are then passed to the underlying ganache-core implementation. You can read more about the options here.

Note: Prior to Waffle `2.3.0` provider was created using `createMockProvider(options?)`. It behaved exactly like `new MockProvider` but it returned `providers.Web3Provider`, which is the parent class of `MockProvider`.

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.

```
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();
```

Note: Prior to Waffle `2.3.0` wallets were obtained using `getWallets(provider)`.

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: Note: As the second parameter of the link function, you need to use a fully qualified name (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. 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*, *below*, *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);
```

3.4.3 Revert

Testing if transaction was reverted:

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

3.4.4 Revert with message

Testing if transaction was reverted with certain message:

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

3.4.5 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.

3.4.6 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]);
```

3.4.7 Proper address

Testing if string is a proper address:

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

3.4.8 Proper private key

Testing if string is a proper secret:

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

3.4.9 Proper hex

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

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

3.5 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.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);
  });
});
```

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
});
```

3.6 Configuration

3.6.1 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 match 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:

```
{}
```

Note: All of the configuration options are optional.

Afterwards update your `package.json` build script:

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

Next, you will learn about the configuration options:

- *sourcesPath*
- *targetPath*
- *npmPath*
- *compiler*
- *docker-tag*
- *solcVersion*
- *legacyOutput*
- *allowedPaths*
- *compilerOptions*
- *outputType*
- *outputHumanReadableAbi*
- *ganacheOptions*

sourcesPath

You can specify a custom path to the directory containing your smart contracts. Waffle uses `./contracts` as the default value for `./sourcesPath`.

Example:

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

targetPath

You can specify a custom path to the directory to which Waffle saves the compilation output. Waffle uses `./build` as the default value for `./targetPath`.

Example:

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

npmPath

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 `./npmPath`.

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

Example:

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

compiler

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

- `solcjs` (default)
- `native`
- `dockerized-solc`

Example:

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

docker-tag

Specifies the docker tag. Only use alongside `"compiler": "dockerized-solc"`. For more information, see: *Using dockerized solc*.

Example:

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

solcVersion

solc-js allows setting the version of the solidity compiler on the fly. To change the version of the solidity compiler update the value of the `solcVersion` field in your config file:

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

To find an appropriate version name please consult the list of available solc versions.

Instead of specifying a version tag you can pass the path to the solc-js package.

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

legacyOutput

Starting with Waffle 2.0, the format of contract output JSON files is the 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 setting `"legacyOutput": "true"` in the configuration file:

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

allowedPaths

The `solc` compiler has restrictions on paths it can access for security reasons. The value of `allowedPaths` 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:

```
{
  "allowedPaths": ["../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)"
  ]
}
```

ganacheOptions

Values specified here will be read by `createMockProvider` if passed the path to the config file.

Example:

```
{
  "ganacheOptions": {
    "gasLimit": 50,
    "gasPrice": 1
  }
}
```

3.6.2 Other configuration file formats

Waffle supports the following configuration file formats:

JSON:

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

JavaScript:

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

The configuration can even be a promise

```
module.exports = Promise.resolve({
  sourcesPath: './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, 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.

3.6.3 Setting Solidity compiler version

See *solcVersion*.

3.6.4 Usage with Truffle

See *legacyOutput*.

3.6.5 Usage with old Waffle versions

See *legacyOutput*.

3.6.6 Custom compiler options

See *compilerOptions*.

3.6.7 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" ]
      },
    }
  }
};
```

3.6.8 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.

3.6.9 Usage with Lernajs

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:

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

That should solve a problem.

Currently Waffle does not support similar feature for dockerized solc.