smock

DeFi Wonderland

Nov 01, 2021

DEVELOPER DOCS

1 Documentation **3**

2 Quick Start **5**
- 2.1 Installation . . . 5
- 2.2 Basic Usage . . . 5

3 License **7**

4 Contributors **9**
- 4.1 Getting Started . . . 9
- 4.2 Fakes . . . 10
- 4.3 Mocks . . . 16
- 4.4 Development . . . 18
- 4.5 Migrating from v1 to v2 . . . 19

Smock is the **S**olidity **mock**ing library. It's a plugin for hardhat that can be used to create mock Solidity contracts entirely in JavaScript (or TypeScript!). With Smock, it's easier than ever to test your smart contracts. You'll never have to write another mock contract in Solidity again.

Smock is inspired by sinon, sinon-chai, and Python's unittest.mock. Although Smock is currently only compatible with hardhat, we plan to extend support to other testing frameworks like Truffle.

If you wanna chat about the future of Solidity Mocking, join our Discord!

- Get rid of your folder of "mock" contracts and **just use JavaScript**.
- Keep your tests **simple** with a sweet set of chai matchers.
- Fully compatible with TypeScript and TypeChain.
- Manipulate the behavior of functions on the fly with **fakes**.
- Modify functions and internal variables of a real contract with **mocks**.
- Make **assertions** about calls, call arguments, and call counts.
- We've got extensive documentation and a complete test suite.

CHAPTER

ONE

DOCUMENTATION

Detailed documentation can be found here.

CHAPTER

TWO

QUICK START

2.1 Installation

You can install Smock via npm or yarn:

```
npm install @defi-wonderland/smock
```

2.2 Basic Usage

Smock is dead simple to use. Here's a basic example of how you might use it to streamline your tests.

```
...
import { FakeContract, smock } from '@defi-wonderland/smock';

chai.should(); // if you like should syntax
chai.use(smock.matchers);

describe('MyContract', () => {
    let myContractFake: FakeContract<MyContract>;

    beforeEach(async () => {
        ...
        myContractFake = await smock.fake('MyContract');
    });

    it('some test', () => {
        myContractFake.bark.returns('woof');
        ...
        myContractFake.bark.atCall(0).should.be.calledWith('Hello World');
    });
});
```

CHAPTER

THREE

LICENSE

Smock is released under the MIT license. Feel free to use, modify, and/or redistribute this software as you see fit. See the LICENSE file for more information.

CHAPTER

FOUR

CONTRIBUTORS

Maintained with love by Optimism PBC and DeFi Wonderland. Made possible by viewers like you.

4.1 Getting Started

4.1.1 Installation

yarn

npm

```
yarn add --dev @defi-wonderland/smock
```

```
npm install --save-dev @defi-wonderland/smock
```

4.1.2 Required Config for Mocks

Mocks allow you to manipulate any variable inside of a smart contract. If you'd like to use mocks, you **must** update your `hardhat.config.<js/ts>` file to include the following:

JavaScript

TypeScript

```
// hardhat.config.js

... // your plugin imports and whatnot go here

module.exports = {
  ... // your other hardhat settings go here
  solidity: {
    ... // your other Solidity settings go here
    settings: {
      outputSelection: {
        "*": {
          "*": ["storageLayout"]
        }
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

```
// hardhat.config.js

... // your plugin imports and whatnot go here

const config = {
  ... // your other hardhat settings go here
  solidity: {
    ... // your other Solidity settings go here
    settings: {
      outputSelection: {
        "*": {
          "*": ["storageLayout"]
        }
      }
    }
  }
}

export default config
```

4.2 Fakes

4.2.1 What are fakes?

Fakes are JavaScript objects that emulate the interface of a given Solidity contract. You can use fakes to customize the behavior of any public method or variable that a smart contract exposes.

4.2.2 When should I use a fake?

Fakes are a powerful tool when you want to test how a smart contract will interact with other contracts. Instead of initalizing a full-fledged smart contract to interact with, you can simply create a fake that can provide pre-programmed responses.

Fakes are especially useful when the contracts that you need to interact with are relatively complex. For example, imagine that you're testing a contract that needs to interact with another (very stateful) contract. Without `smock`, you'll probably have to:

1. Deploy the contract you need to interact with.
2. Perform a series of transactions to get the contract into the relevant state.
3. Run the test.
4. Do this all over again for each test.

This is annoying, slow, and brittle. You might have to update a bunch of tests if the behavior of the other contract ever changes. Developers usually end up using tricks like state snapshots and complex test fixtures to get around this problem. Instead, you can use `smock`:

1. Create a `fake`.

2. Make your `fake` return the value you want it to return.
3. Run the test.

4.2.3 Using fakes

Initialization

Initialize with a contract name

```
const myFake = await smock.fake('MyContract');
```

Initialize with a contract ABI

```
const myFake = await smock.fake([ { ... } ]);
```

Initialize with a contract factory

```
const myContractFactory = await hre.ethers.getContractFactory('MyContract');
const myFake = await smock.fake(myContractFactory);
```

Initialize with a contract instance

```
const myContractFactory = await hre.ethers.getContractFactory('MyContract');
const myContract = await myContractFactory.deploy();
const myFake = await smock.fake(myContract);
```

Take full advantage of typescript and typechain

```
const myFake = await smock.fake<MyContract>('MyContract');
```

Options

```
await smock.fake('MyContract', { ... }); // how to use

// options
{
  address?: string; // initialize fake at a specific address
  provider?: Provider; // initialize fake with a custom provider
}
```

Signing transactions

Every fake comes with a `wallet` property in order to make easy to sign transactions

```
myContract.connect(myFake.wallet).doSomething();
```

Making a function return

Returning with the default value

```
myFake.myFunction.returns();
```

Returning a fixed value

```
myFake.myFunction.returns(42);
```

Returning a struct

```
myFake.getStruct.returns({
  valueA: 1234,
  valueB: false,
});
```

Returning an array

```
myFake.myFunctionArray.returns([1, 2, 3]);
```

Returning a dynamic value

```
myFake.myFunction.returns(() => {
  if (Math.random() < 0.5) {
    return 0;
  } else {
    return 1;
  }
});
```

Returning a value based on arguments

```
myFake.myFunction.whenCalledWith(123).returns(456);

await myFake.myFunction(123); // returns 456
```

Returning a value with custom logic

```
myFake.getDynamicInput.returns(arg1 => arg1 * 10);

await myFake.getDynamicInput(123); // returns 1230
```

Returning at a specific call count

```
myFake.myFunction.returnsAtCall(0, 5678);
myFake.myFunction.returnsAtCall(1, 1234);

await myFake.myFunction(); // returns 5678
await myFake.myFunction(); // returns 1234
```

Making a function revert

Reverting with no data

```
myFake.myFunction.reverts();
```

Reverting with a string message

```
myFake.myFunction.reverts('Something went wrong');
```

Reverting with bytes data

```
myFake.myFunction.reverts('0x12341234');
```

Reverting at a specific call count

```
myFake.myFunction.returns(1234);
myFake.myFunction.revertsAtCall(1, 'Something went wrong');

await myFake.myFunction(); // returns 1234
await myFake.myFunction(); // reverts with 'Something went wrong'
await myFake.myFunction(); // returns 1234
```

Reverting based on arguments

```
myFake.myFunction.returns(1);
myFake.myFunction.whenCalledWith(123).reverts('Something went wrong');

await myFake.myFunction(); // returns 1
await myFake.myFunction(123); // reverts with 'Something went wrong'
```

Resetting function behavior

Resetting a function to original behavior

```
myFake.myFunction().reverts();

await myFake.myFunction(); // reverts

myFake.reset();

await myFake.myFunction(); // returns 0
```

Asserting call count

Any number of calls

```
expect(myFake.myFunction).to.have.been.called;
```

Called once

```
expect(myFake.myFunction).to.have.been.calledOnce;
```

Called twice

```
expect(myFake.myFunction).to.have.been.calledTwice;
```

Called three times

```
expect(myFake.myFunction).to.have.been.calledThrice;
```

Called N times

```
expect(myFake.myFunction).to.have.callCount(123);
```

Asserting call arguments

Called with specific arguments

```
expect(myFake.myFunction).to.have.been.calledWith(123, true, 'abcd');
```

Called with struct arguments

```
expect(myFake.myFunction).to.have.been.calledWith({
  myData: [1, 2, 3, 4],
  myNestedStruct: {
    otherValue: 5678
  }
});
```

Called at a specific call index with arguments

```
expect(myFake.myFunction.atCall(2)).to.have.been.calledWith(1234, false);
```

Called once with specific arguments

```
expect(myFake.myFunction).to.have.been.calledOnceWith(1234, false);
```

Asserting call order

Called before other function

```
expect(myFake.myFunction).to.have.been.calledBefore(myFake.myOtherFunction);
```

Called after other function

```
expect(myFake.myFunction).to.have.been.calledAfter(myFake.myOtherFunction);
```

Called immediately before other function

```
expect(myFake.myFunction).to.have.been.calledImmediatelyBefore(myFake.myOtherFunction);
```

Called immediately after other function

```
expect(myFake.myFunction).to.have.been.calledImmediatelyAfter(myFake.myOtherFunction);
```

Querying call arguments

Getting arguments at a specific call index

```
expect(myFake.myFunction.getCall(0).args[0]).to.be.gt(50);
```

Manipulating fallback functions

Modifying the `fallback` function

```
myFake.fallback.returns();
```

Modifying the `receive` function

```
myFake.receive.returns();
```

4.3 Mocks

4.3.1 What are mocks?

Mocks are extensions to smart contracts that have all of the functionality of a fake with some extra goodies. Behind every mock is a real smart contract (with actual Solidity code!) of your choosing. You can **modify the behavior of functions like a fake**, or you can leave the functions alone and calls will pass through to your actual contract code. And, with a little bit of smock magic, you can even **modify the value of variables within your contract**!

4.3.2 When should I use a mock?

Generally speaking, mocks are more advanced versions of fakes. Mocks are most effectively used when you need *some* behavior of a real smart contract but still want the ability to modify things on the fly.

One powerful feature of a mock is that you can modify the value of variables within the smart contract. You could, for example, use this feature to test the behavior of a function that changes behavior depending on the value of a variable.

4.3.3 Using mocks

Initialization

Initialize with a contract name

```
const myContractFactory = await smock.mock('MyContract');
const myContract = await myContractFactory.deploy(...);
```

Take full advantage of typescript and typechain

```
await smock.mock<MyContract__factory>('MyContract');
```

Options

```
await smock.mock('MyContract', { ... }); // how to use

// options
{
  provider?: Provider; // initialize mock with a custom provider
}
```

Using features of fakes

Mocks can use any feature available to fakes. See the documentation of fakes for more information.

Call through

Calls go through to contract by default

```
await myMock.add(10);
await myMock.count(); // returns 10

myMock.count.returns(1);
await myMock.count(); // returns 1
```

Manipulating variables

> **Warning:** This is an experimental feature and it is subject to API changes in the near future

Setting the value of a variable

```
await myMock.setVariable('myVariableName', 1234);
```

Setting the value of a struct

```
await myMock.setVariable('myStruct', {
  valueA: 1234,
  valueB: true,
});
```

Setting the value of a mapping (won't affect other keys)

```
await myMock.setVariable('myMapping', {
  myKey: 1234
});
```

Setting the value of a nested mapping

```
await myMock.setVariable('myMapping', {
  myChildMapping: {
    myKey: 1234
  }
});
```

4.4 Development

4.4.1 Code

Open an issue or a PR, we will try to see it asap.

4.4.2 Docs

In order to continue developing the docs, you will first need to install the needed dependencies locally by running:

yarn

npm

```
yarn docs:install
```

```
npm run docs:install
```

Then you can run the sphinx autobuild to see your changes live:

yarn

npm

```
yarn docs:watch
```

```
npm run docs:watch
```

4.5 Migrating from v1 to v2

DeFi Wonderland and Optimism have decided to join forces with our shady-super-coder's magic to launch a new and improved version of the mocking library you

We know the breaking changes on the API will make you do some leg work, but we promise it is going to be totally worth it!

Also, special thanks to the Optimism team for recognizing our work and allowing us to host the new library on our Github organization (this marks our first public release)

Smock V2 focuses mainly on:

- API improvements
- Call arguments expectations
- Custom chai matchers
- Type extensions with generics
- Fakes and Mocks division
- Documentation

4.5.1 Before upgrading

If using Typescript, we highly recommend using Typechain in order to take full advantage of the type extensions we provide. If you decide not to, you can still follow along by using the type `Contract` from `ethers` or `any`.

With Typechain:

```
import { FakeContract } from '@defi-wonderland/smock';
import { CookieEater } from '@typechained';

let cookieEater: FakeContract<CookieEater>; // will extend all of the CookieEater method␣
↪types
```

Without Typechain:

```
import { FakeContract } from '@defi-wonderland/smock';
import { Contract } from 'ethers';

let cookieEater: FakeContract<Contract>; // will extend all of the CookieEater method␣
↪types
```

4.5.2 Installation

Uninstall the old package

yarn

npm

```
yarn remove @eth-optimism/smock
```

```
npm uninstall @eth-optimism/smock
```

Install the new one

yarn

npm

```
yarn add --dev @defi-wonderland/smock
```

```
npm install --save-dev @defi-wonderland/smock
```

4.5.3 New concepts

Instead of having Mock and Smod objects, now we use Fakes and Mocks.

- **Fakes** are empty contracts that emulate a given interface.
  All of their functions can be watched and pre-programmed. When calling a function of a fake, by default, it will return the return type zero-state.
- **Mocks** are deployed contract wrappers that have all of the fake's functionality and even more.
  Because they are actually deployed contract, they can have actual logic inside that can be called through. And because they have a storage, internal variable values can be overwritten

---

4.5.4 API changes

Smockit initialization

Before:

```
import { ethers } from 'hardhat';
import { smockit } from '@eth-optimism/smock';

const myContractFactory = await ethers.getContractFactory('MyContract');
const myContract = await myContractFactory.deploy(...);
const myMockContract = await smockit(myContract);
```

After:

```
import { smock } from '@defi-wonderland/smock';
import { MyContract } from '@typechained';

const myFakeContract = await smock.fake<MyContract>('MyContract');
```

Returns

Before:

```
myMockContract.smocked.myFunction.will.return.with('Some return value!');
```

After:

```
myFakeContract.myFunction.returns('Some return value!');
```

Asserting call count

Before:

```
expect(myMockContract.smocked.myFunction.calls.length).to.equal(1);
```

After:

```
expect(myFakeContract.myFunction).to.be.calledOnce;
```

Asserting call data

Before:

```
expect(MyMockContract.smocked.myFunction.calls.length).to.equal(1);
expect(MyMockContract.smocked.myFunction.calls[0]).to.deep.equal(['Something', 123]);
```

After:

```
expect(myFakeContract.myFunction).to.be.calledOnceWith('Something', 123);
```

Reverting

Before:

```
myMockContract.smocked.myFunction.will.revert();
myMockContract.smocked.myOtherFunction.will.revert.with('Some error');
```

After:

```
myFakeContract.myFunction.reverts();
myFakeContract.myOtherFunction.reverts('Some error');
```

Creating a modifiable contract

Before:

```
import { ethers } from 'hardhat';
import { smoddit } from '@eth-optimism/smock';

const myModifiableContractFactory = await smoddit('MyContract');
const myModifiableContract = await MyModifiableContractFactory.deploy(...);
```

After:

```
import { MyContract } from '@typechained';
import { MockContract, MockContractFactory, smock } from '@defi-wonderland/smock';

const myMockContractFactory: MockContractFactory<MyContract> = await smock.mock(
↪'MyContract');
const myMockContract: MockContract<MyContract> = await myMockContractFactory.deploy(...);
```

Modifying a contract variable value

Before:

```
await myModifiableContract.smodify.put({
  _myInternalVariable: 1234
});
```

After:

```
await myMockContract.setVariable('_myInternalVariable', 1234);
```

4.5.5 And more...

Smock V2 contains plenty of new features, you can check them all out in the docs!