Truffle Suite

This is a beta document and refers to thebeta version of Truffle. The following features will not work unless you are using Truffle Beta.

Get the Beta Version¶

As this feature is in beta, you must first get the beta version of Truffle. To make sure the beta version doesn't conflict with the released version, first remove the Truffle version you have currently installed:

npm uninstall -g truffle And then install the beta version:

```
npm install -g truffle@beta
```

Overview¶

Solidity test contracts live alongside Javascript tests as.sol files. Whentruffle test is run, they will be included in the mocha run with a separate test suite per test contract. These contracts maintain all the benefits of the Javascript tests: namely a clean slate per test suite, access to deployed contracts via migrations, runnable on any Ethereum client, and usage of snapshot/revert features (if supported by your Ethereum client) for increased speed. An example solidity unit test looks like this:

```
import

"truffle/Assert.sol" ; import

"truffle/DeployedAddresses.sol" ; import

"../contracts/MetaCoin.sol" ; contract

TestMetacoin

{

function

testInitialBalanceUsingDeployedContract ()

{

MetaCoin meta =

MetaCoin( DeployedAddresses. MetaCoin());

uint

expected

=

10000 ;

Assert. equal( meta. getBalance( tx.origin ),

expected,

"Owner should have 10000 MetaCoin initially" );

}

function

testInitialBalanceWithNewMetaCoin ()

{

MetaCoin meta =

new

MetaCoin();
```

uint

expected

=

10000 ;

Assert. equal(meta. getBalance(tx.origin),

expected,

"Owner should have 10000 MetaCoin initially");

} } And when running the test:

truffle test Compiling ConvertLib.sol... Compiling MetaCoin.sol... Compiling ../test/TestMetacoin.sol... TestMetacoin ✓ testInitialBalanceUsingDeployedContract (61ms)

✓ testInitialBalanceWithNewMetaCoin (69ms)

2

passing (3s) There are some very important structural items to discuss:

- First, your assertion functions are provided by thetruffle/Assert.sol
- library. This is the default assertion library, however you can include your own assertion library so long as the library outputs the correct events to the test runner. You can find all available assertion functions inAssert.sol
- . These functions are meant to mimic those available in your Javascript tests, and more documentation will be written about each function in the future.
- The addresses of your deployed contracts (contracts that were deployed in your migration) are available through thetruffle/DeployedAddresses.sol
- library. This is provided by Truffle and is recompiled and relinked before each suite is run to provide your tests with a clean slate deployment. This library provides functions for all of your deployed contracts, in the form of:

DeployedAddresses. < contract

name

    (); This will return an address that you can then use to access that contract.

- In order to use the deployed contract, you'll have to import the contract code first. Noticeimport "../contracts/MetaCoin.sol";
- in the example. This import is relative to the test contract, which exists in the./test
- directory, and it goes outside of the test directory in order to find the MetaCoin contract. It then uses that contract in a test function below to cast the address to theMetaCoin
- type.
- All test contracts must start withTest
- , using an uppercaseT
- . This distinguishes this contract apart from test helpers and project contracts, letting the test runner know it represents a test contract.
- Like test contract names, all test functions must start withtest
- , all lowercase. Each test function is executed as a single transaction, in order of appearance in the test file (like your Javascript tests). Assertion functions provided bytruffle/Assert.sol
- trigger events that the test runner evaluates to determine the result of the test. Assertion functions return a boolean representing the outcome of the assertion which you can use to return from the test early to prevent execution errors.
- You are provided many test hooks, shown in the example below. These hooks arebeforeAll
- ,beforeEach
- ,afterAll
- andafterEach
- , which are the same hooks provided by mocha in your Javascript tests. You can use these hooks to perform setup and teardown actions before and after each test, or before and after each suite is run. Like test functions, each hook is executed as a single transaction. Note that some complex tests will need to perform a significant amount of setup that might overflow the gas limit of a single transaction; you can get around this limitation by creating many hooks with different suffixes, like in the example test below:

import

"truffle/Assert.sol" ; contract

TestHooks

{

uint

someValue ;

function

beforeEach ()

{

someValue

5 ;

}

function

beforeEachAgain ()

{

someValue +=

1 ;

}

function

testSomeValueIsSix ()

{

uint

expected

=

6 ;

Assert. equal(someValue,

expected,

"someValue should have been 6");

} } This test contract also shows that your test functions and hook functions all share the same contract state. You can setup contract data before the test, use that data during the test, and reset it afterward in preparation for the next one. Note that just like your Javascript tests, your next test function will continue from the state of the previous test function that ran.

- Youdo not
- need to extend from anyTest
- contract like in other solidity testing frameworks. This is to make writing solidity unit tests easier, and should allow for more extensibility in the future with less hassle.
- Like Javascript tests, failing tests return the events that were fired (minus assertion events) so you can get a better sense of how your contracts-under-test performed.