Truffle Suite

Archived: This tutorial has been archived and may not work as expected; versions are out of date, methods and workflows may have changed. We leave these up for historical context and for any universally useful information contained. Use at your own risk!

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¶

Truffle Beta comes standard withnpm integration, and is aware of thenode_modules directory in your project if it exists. This means you can use and distribute contracts, dapps and Ethereum-enabled libraries vianpm , making your code available to others and other's code available to you.

Package Layout¶

Projects created with Truffle have a specific layout by default which enables them to be used as packages. This layout isn't required, but if used as a common convention -- or "de-facto standard" -- then distributing contracts and dapps through package management will become much easier.

The most important directories in a Truffle package are the following:

- /contracts
- /build
- (which includes/build/contracts
- , created by Truffle)

The first directory is your contracts directory, which includes your raw Solidity contracts. The second directory is the build directory, and more specifically/build/contracts , which holds build artifacts in the form of.sol.js files. Including raw contracts in your package will allow others to import those contracts within their own solidity code. Similarly, including your.sol.js build artifacts in your package will allow others to seamlessly interact with your contracts from Javascript, which can be used in dapps, scripts and migrations.

Using a Package¶

When using a package within your own project, it is important to note that there are two places where you might be interested in using other's contract code: within your contracts (.sol files) and within your Javascript code (.sol.js files). The following provides an example of each case, and discusses techniques for making the most of other's contracts and build artifacts.

Installing¶

For this example, we're going to use theExample Truffle Library , which provides a simple name registry that is deployed to the Morden test network. In order to use it as a dependency, we must first install it within our project throughnpm :

cd

my_project

npm install example-truffle-library Note that the last command above downloads the package and places it inmy_project/node_modules directory, which is important for the examples below. See thenpm documentation for help usingnpm to install packages.

Within Your Contracts¶

To use a package's contracts within your contracts, this can be as simple as Solidity's import statement. When your import

path isn't explicitly relative or absolute, this signifies to Truffle that you're looking for a file from a specific named package. Consider this example using the Example Truffle Library mentioned above:

import

"example-truffle-library/contracts/SimpleNameRegistry.sol" ; Since the path didn't start with./ , Truffle knows to look in your project'snode_modules directory for theexample-truffle-library folder. From there, it resolves the path to provide you the contract you requested.

Within Javascript Code¶

To interact with package's contracts within Javascript code, you simply need torequire that package's.sol.js files like so:

var

SimpleNameRegistry

=

require ("example-truffle-library/build/contracts/SimpleNameRegistry.sol.js"); These files are provided by EtherPudding , which Truffle uses internally. See EtherPudding's documentation for more information.

Package's Deployed Addresses¶

Sometimes you want your contracts to interact with the package's previously deployed contracts. Since the deployed addresses exist within the package's.sol.js files, you must perform an extra step to get those addresses into your contracts. To do so, make your contract accept the address of the dependency contract, and then use migrations. The following is an example contract that exists within your project as well as an example migration:

Contract: MyContract.sol

import

"example-truffle-library/contracts/SimpleNameRegistry.sol" ; contract

MyContract

{

SimpleNameRegistry registry;

address

public owner ;

function

MyContract

{

owner

msg.sender ;

}

// Simple example that uses the deployed registry from the package.

function

getModule (bytes32

name)

returns

(address)

{

return

registry. names(name);

}

// Set the registry if you're the owner.

function

setRegistry (address

addr)

{

require (msg.sender

==

owner);

registry

SimpleNameRegistry(addr);

} } Migration: 3_hook_up_example_library.js

var

SimpleNameRegistry

=

require ("example-truffle-library/build/contracts/SimpleNameRegistry.sol.js"); module . exports

=

function (deployer)

{

// Deploy our contract, then set the address of the registry.

deployer . deploy (MyContract). then (function ()

{

MyContract . deployed (). setRegistry (SimpleNameRegistry . address);

}); };

Recommendation: Use the TestRPC¶

Update: Since this tutorial was published, we have releasedGanache a personal blockchain and a replacement to the TestRPC. We have left this tutorial unaltered, but we highly recommend checking out our Ganache Quickstart page.

Theethereumjs-testrpc is wildly useful for packages that were previously deployed to the live network. The TestRPC includes a--fork feature which allows you to fork from the main chain while developing your application. What this means for you is that you can develop and test against packages that are deployed live, usingreal data , without having to worry about deploying your dependency's contracts yourself.

The Future: Where do we go from here?¶

Node'snpm is, of course, a centralized service, and the packages that exist there are on a centralized network. As Ethereum becomes more widespread, so will the urge to decentralize all of our tools and packages. The ideal package management system for Ethereum is one where the package registry exists on the live chain and the package data exists on a decentralized file storage network like IPFS or Swarm; however, that ideal is going to take time to develop and reach critical mass. Until that time, we can usenpm to distribute our contracts and applications like the rest of the Javascript community.