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Archived: This tutorial has been archived and may not work as expected; versions are out of date, methods and workflows may have changed.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!

Note: This guide also applies to users upgrading from Truffle beta 3.0.0-9 to Truffle 3.0.1

Introduction¶

Truffle 3.0 offers a ton of new features. With it comes some very important breaking changes. These breaking changes make network management easier, make contract abstractions easier to use, and allow you to install third party packages from any number of sources. We consider these breaking changes helpful -- they're due to the careful maturation of Ethereum's most popular development tool, and we'll guide you through the steps needed to take advantage of all these new features.

In order to show you code samples that describe the evolution from 2.0 to 3.0, we're going to use examples like this:

v2.0:

// This is old code from Truffle 2.0 v3.0:

// This is new code for Truffle 3.0! Now let's get started!

First Things First: Configuration¶

Truffle 2.0 had an odd configuration -- we admit it. Not only can you have a default, unnamed network sitting around (therpc configuration option) but you can also have named networks like"ropsten" or"live" . A result of this default vs. named distinction came were other oddities, such as accidentally overwriting important network artifacts or deploying to the wrong network. With Truffle 3.0 we fixed all that, but it requires a change in your configuration.

In v2.0, a configuration with a named network might look like this:

```
module . exports

=

{

rpc :

{

host :

"localhost" ,

port :

8545 ,

},

networks :

{

staging :

{

host :

"localhost" ,

port :

8546 ,

network_id :
```

```
1337 ,
},
ropsten :
{
host :
"158.253.8.12" ,
port :
8545 ,
network_id :
3 ,
},
}, };
```

In v3.0, this same configuration would look like this:

```
module . exports
=
{
networks :
{
development :
{
host :
"localhost" ,
port :
8545 ,
network_id :
"*" ,
},
staging :
{
host :
"localhost" ,
port :
8546 ,
network_id :
1337 ,
},
ropsten :
{
```

host :

"158.253.8.12" ,

port :

8545 ,

network_id :

3 ,

},

}, }; What's the difference? First, there's no default network anymore: therpc configuration item has been removed. In its place is a specially nameddevelopment network. This is a network that will match any network it connects to at the specified host and port, and when running commands likemigrate Truffle will default to this network if none is specified. If you'd like to protect yourself from deploying to the wrong network completely, you canremove thedevelopment configuration entirely; if you don't specify the network viatruffle migrate --network ropsten , for example, Truffle will exit with an error, ensuring you never deploy to a network you don't intend to.

Additionally, each network needs to have a network id specified, or a"*" . This is another security measure used during migrations to ensure that evenif Truffle connects to the Ethereum client at the specified host and port, if it's not the network you intend, Truffle will error before attempting any deployments.

Importing and the Ethereum Package Manager (EthPM)¶

As Truffle 3.0 uses the newEthereum Package Manager (EthPM) we need to tell Truffle when the package manager should be used and when to look in the local directory. If you don't specify the directory to search relative to thecontracts/ directory, it will assume that EthPM will handle this.

So to import a local file:

v2.0:

import

"test.sol" v3.0:

import

"./test.sol" The./ tells Truffle that the contract is in the current directory. See theCompiling contracts page for more details.

Migrations and Test Dependencies¶

Before package management, Truffle could assume that all the contracts you wrote yourself were the contracts you wanted to interact with via your migrations and tests. Now that package management is here, this is no longer a safe assumption -- contract dependencies can come from any number of sources, and so you have to explicitly ask for those abstractions yourself. This is all in the name of reducing magic. Let's look at an example migration.

v2.0 :./migrations/2_deploy_contracts.js

module . exports

=

function

(deployer)

{

deployer . deploy (ConvertLib);

deployer . autolink ();

deployer . deploy (MetaCoin); }; v3.0 :./migrations/2_deploy_contracts.js

var

ConvertLib

=

artifacts . require ("ConvertLib.sol"); var

MetaCoin

=

artifacts . require ("MetaCoin.sol"); module . exports

=

function

(deployer)

{

deployer . deploy (ConvertLib);

deployer . link (ConvertLib ,

MetaCoin);

deployer . deploy (MetaCoin); }; Note there are a couple differences. For this section, the changes we'll focus on are the two lines which useartifacts.require() . This is a function similar to Node'srequire() statement provided by Truffle. If given a path to a solidity file local to your project, or the path of a package dependency, this function will locate the correct artifacts and make them ready for use on your current network.

Your Javascript tests need the exact sameartifacts.require() statements in order to use those abstractions like before.

Migrations: No more autolink¶

Like the section above, autolinking was a boon of a very small dependency tree. With package management, Truffle can no longer autolink your contracts because it can't determine all possible dependencies you might need within your migration. Instead, you need to explicitly link your libraries yourself. Fortunately this is not hard, and you can see the changes we made in the example above.

Contract Abstractions: All JSON (goodbye .sol.js!)¶

Truffle had originally saved your contract abstractions on disk in.sol.js files. These files were Javascript files that represented specific solidity files in your project, hence the file extension, and they were originally intended to make your contract artifacts really easy to use, everywhere. Well, it turns out we got it wrong. Really wrong. Not only were these files hard to usewithin Javascript in certain cases, but they were completely unusable if you wanted to use them outside of Javascript. To overcome this limitation and make your contract artifacts available everywhere, Truffle 3.0 now saves all artifacts within the JSON format.

So what should you do if you've been using Truffle 2.0 and have important contract artifacts you'd like to save? Easy. Upgrade them! We built you a handy tool that does just that.

To upgrade your.sol.js files, first installtruffle-soljs-updater globally via npm:

npm install -g truffle-soljs-updater This will provide thesjsu command available for your use. Next, navigate to the directory where your.sol.js files are stored (usually./build/contracts), and runsjsu . Your output should look something like this:

cd

./build/contracts

sjsu Converting ConvertLib.sol.js... Converting MetaCoin.sol.js... Converting Migrations.sol.js... Files converted successfully. By defaultsjsu only creates newer,.json versions of your artifacts and leaves the original.sol.js files alone. You'll need to remove the.sol.js files from this directory in order for Truffle 3.0 to function properly. Before removing, check to ensure that the converted.json files were created properly, and perhaps as a safeguard backup your old.sol.js files somewhere else before continuing.

Once you're sure all data was copied correctly and you have a nice backup, you can runsjsu with the-f parameter. This will tellsjsu to delete the existing.sol.js files. Your output should look something like this:

cd

./build/contracts

sjsu -f Converting ConvertLib.sol.js... Converting MetaCoin.sol.js... Converting Migrations.sol.js... Files converted successfully. Successfully deleted old .sol.js files. And that's it! You're now ready to move onto the next step.

Contract Abstractions: .deployed() is now thennable¶

⚠ Warning: This change will affect your migrations, your tests, and your application code! ⚠

In Truffle 2.0, your contract abstractions managed your networks in a naive way, and added constructs like a "default network" that opened up the possibility of using the wrong network artifacts and deployed addresses at the wrong time. This provided a fancy and easy to use syntax -- i.e.,MyContract.deployed().myFunction(...) -- but it left your code open to errors. Truffle 3.0 changes this syntax, where.deployed() is now thennable, like a promise (see example below). Similarly, this makes your contract abstractions seamlessly integrate withEIP 190 , Ethereum's package management standard. All of this is for the better, but it means you'll have some changes to make all across the board.

In the general use case, here is how the new syntax differs from the old:

v2.0

```
MyContract . setProvider ( someWeb3Provider ); MyContract . deployed ()

. someFunction ()

. then ( function

( tx )

{

// Do something after the someFunction() transaction executed

});
```

v3.0

```
MyContract . setProvider ( someWeb3Provider ); MyContract . deployed ()

. then ( function

( instance )

{

return

instance . someFunction ();

})

. then ( function

( result )

{

// Do something after the someFunction() transaction executed

});
```

This syntax is a little more verbose, but it ensures the correct network artifacts are used based on the Ethereum client the abstraction is connected to.

Note that in v3.0, you may need to correctly scope your deployed instance if you want to perform multiple actions with it in the same promise chain:

```
MyContract . setProvider ( someWeb3Provider ); var

deployed ; MyContract . deployed ()

. then ( function

( instance )

{

deployed

=
instance ;
return
deployed . someFunction ();
})
. then ( function
( result )
{
return
deployed . anotherFunction ();
})
. then ( function
( result )
{
// etc.
});
```

Contract Abstractions: Transaction result objects¶

People have long complained that events are hard to watch for in Web3. Similarly, in most use cases, events aren't used via a contract observer pattern; instead, in practice, events are used to check specific results of a specific transaction. Though the contract observer pattern for events is still available, we made the latter use case much easier, which required breaking changes to the return value of transactions.

In v2.0, a transaction simply returned the transaction hash. In v3.0, transactions return a result object with a wealth of information about that transaction.

v2.0

```
MyContract . deployed ()
. someFunction ()
. then ( function
( tx )
{
// tx is the transaction id (hash) of the transaction executed
}); v3.0
MyContract . deployed ()
. then ( function
( instance )
{
deployed
=
instance ;
return
deployed . someFunction ();
```

```
})

. then ( function

( result )

{

// result is an object with the following values:

//

// result.tx => transaction hash, string

// result.logs => array of decoded events that were triggered within this transaction

// result.receipt => transaction receipt object, which includes gas used
```

}); In Truffle 3.0, it's now much easier to detect if an event was fired as a result of your transaction. Here's a Javascript test for a hypothetical PackageIndex contract and ReleasePublished event:

```
var

assert

=

require ( "assert" ); var

PackageIndex

=

artifacts . require ( "PackageIndex.sol" ); contract ( "PackageIndex" ,

function

( accounts )

{

it ( "publishes a release correctly" ,

function

()

{

return

PackageIndex . deployed ()

. then ( function

( deployed )

{

return

deployed . publish ( "v2.0.0" );

})

. then ( function

( result )

{

// This result object is what provides us with the information we need.

// result.logs, specifically for this example.
var

found_published_event

=

false ;

for

( var

i

=

0 ;

i

<

result . logs . length ;

i ++ )

{

var

log

=

result . logs [ i ];

if

( log . event

==

"ReleasePublished" )

{

found_published_event

=

true ;

break ;

}

}

assert (

found_published_event ,

"Uh oh! We didn't find the published event!"

);

});
```

}); }); We have to search through all returned events to ensure we found the one we wanted, but this is certainly better than usingPackageIndex.ReleasePublished.watch(...) , and we can keep event processing within our application's (or test's) normal control flow.

Build Pipeline: No more builder, by default¶

In Truffle 1.0 and 2.0, things were heavily geared toward building web applications, and so Truffle shipped with a default build pipeline that could get your dapp up and running quickly. This build pipeline was magical: It did everything for you, and you didn't have to lift a finger. This was good for some use cases, but it became evidently clear that in any other use case the pipeline was very brittle.

Over the last year, Ethereum-enabled applications have only been growing. What started out as solely a platform for web application, now dapps can be written in native languages and run as standalone applications on mobile and the desktop. Truffle has always intended to support these use cases, and so that's why weremoved the default build pipeline from Truffle. You can stillwrite your own custom build pipeline if you'd like to tightly integrate it with Truffle, but by default Truffle will focus on continuing to be the best tool for smart contracts around. We'll let the better build pipelines -- likewebpack ,browserify ,Grunt ,Metalsmith -- do the job of, well, building.

Now, just because the build pipeline has been removed by default doesn't mean you don't have options. At Truffle, we care about your developer experience and so would never leave you hanging. In general there are two options for you to choose from, but for the latter option what you'll do is heavily dependent on which build tool (i.e., webpack) you choose to use. Let's go over the options below.

Use the old pipeline in Truffle 3.0¶

If you're using the default build pipeline in Truffle 2.0 and would like to upgrade to 3.0, you're not out of luck. We've upgraded thetruffle-default-builder so it can work seamlessly with Truffle 3.0. This will be the last time we update the default builder though as engineering for all possible use cases is too complex. So we recommend you eventually choose a different build system later down the line.

The default builder no longer ships with Truffle by default. So for Truffle 3.0, you first need to maketruffle-default-builder a dependency of your project by running the following command within your project's folder:

npm install truffle-default-builder --save Once installed, you can use the default builder within yourtruffle.js configuration file. Let's have a look at how your configuration file changes from v2.0 to v3.0, using a very simple build configuration:

v2.0 :truffle.js

```
module . exports

=

{

build :

{

"index.html" :

"index.html" ,

"app.js" :

[ "javascripts/app.js" ],

"app.css" :

[ "stylesheets/app.css" ],

"images/" :

"images/" ,

},

rpc :

{

host :

"localhost" ,

port :

8545 ,
}, }; v3.0 :truffle.js
var
DefaultBuilder
=
require ( "truffle-default-builder" ); module . exports
=
{
build :
new
DefaultBuilder ({
"index.html" :
"index.html" ,
"app.js" :
[ "javascripts/app.js" ],
"app.css" :
[ "stylesheets/app.css" ],
"images/" :
"images/" ,
}),
networks :
{
development :
{
host :
"localhost" ,
port :
8545 ,
network_id :
"*" ,
// Match any network id
},
```

}, }; You'll notice that in version 3.0 werequire 'd the default builder as a dependency and then passed that object into our configuration. Otherwise, the configuration was the same!

Now, be aware that the default builder does use the latest contract abstractions (truffle-contract), so you will still need to edit your application to account for the breaking changes mentioned above.

Use a custom build process / build tool¶

Custom build processes are not hard to write. Instead, what's hard is writing a build process that fits all shapes and sizes. We now recommend you look into the many build tools available to you that best fits your application. We've already mentionedwebpack ,browserify ,Grunt , andMetalsmith , but there are many others, and their features run the gamut based on the context of the application being built and the features that you need.

Whether you're building an application to run in the browser, or a command line tool, a Javascript library or a native mobile application, bootstrapping your contracts is the same, and using your deployed contract artifacts follows the same general process no matter the app you're building.

When configuring your build tool or application, you'll need to perform the following steps;

1. Get all your contract artifacts into your build pipeline / application. This includes all of the.json
2. files within the./build/contracts
3. directory.
4. Turn those.json
5. contract artifacts into contract abstractions that are easy to use, viatruffle-contract
6. .
7. Provision those contract abstractions with a Web3 provider. In the browser, this provider might come fromMetamask
8. orMist
9. , but it could also be a custom provider you've configured to point toInfura
10. or any other Ethereum client.
11. Use your contracts!

In Node, this is very easy to do. Let's take a look at an example that shows off the "purest" way of performing the above steps, since it exists outside of any build process or tool.

// Step 1: Get a contract into my application var

json

=

require ("./build/contracts/MyContract.json"); // Step 2: Turn that contract into an abstraction I can use var

contract

=

require ("truffle-contract"); var

MyContract

=

contract (json); // Step 3: Provision the contract with a web3 provider MyContract . setProvider (

new

Web3 . providers . HttpProvider ("http://localhost:8545")); // Step 4: Use the contract! MyContract . deployed (). then (function

(deployed)

{

return

deployed . someFunction (); }); All build processes and contract bootstrapping will follow this pattern. The key when setting up your own custom build process is to ensure you're consuming all of your contract artifacts and provisioning your abstractions correctly.

Fin¶

That's it! That's all you need to know to upgrade from Truffle 2.0 to 3.0. It might require a bit of work, but the changes are surely worth it. If you have questions, Truffle has avibrant community of Trufflers available 24/7to help you with any issues you may have. Don't hesitate to ask for help, and we wish you a happy 3.0!

-- The Truffle Team