Creating an Ethereum-enabled command line tool with Truffle 3.0

Archived: This tutorial has been archived and may not work as expected; versions are out of date, methods and workflows may have changed.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

Truffle 3is out , and it switched to a less opinionated build process. In Truffle 2, the default app fromtruffle init included a frontend example with build process. Now, there's nothing other than abuild folder for your JSON contract artifacts. This opens up the door for testing and building other (cough command linecough) types of applications!

Intended Audience¶

This is written for those familiar with Truffle and Ethereum, who want to learn how to create a testable, Ethereum-enabled command line application with Truffle.

For this tutorial, we'll be building a command line tool that interacts with theEthereum Name Service (ENS).

Getting Started¶

If you were using Truffle beta 3.0.0-9 or below,do not immediately upgrade . Readhese release notes and theupgrade guide first. Next, make a new folder and run the following command:

truffle init You should see thetest ,build , andmigrations directories were created for you -- but noapp . Additionally, the build step in thetruffle.js file is mysteriously absent. That's ok! This means Truffle is getting out of your way and letting you control the build process.

Let's Look at the Example¶

Check out the example appas a reference. I'll be referring to it often so give it a look over if you haven't already.

The first thing to look at is the build process. Since Truffle now puts us in control of the build, I've added some custom scripts withinpackage.json to handle building for us:

... "description" :

"CLI for ENS deployment" , "scripts" :

{

"ens" :

"babel-node ./bin/ensa.js" ,

"lint" :

"eslint ./" }, "author" :

"Douglas von Kohorn" , "license" :

"MIT" , ... I've defined two scripts here:lint for linting my Javascript to keep my codebase so fresh and so clean, andens for transpiling my Javascript command line tool (using Babel) and running it. That's it! That's the build process. You can run it vianpm run lint andnpm run ens . Now that that's out of the way, let's dig into how and why I structured my app this way.

Building our App: Separation of concerns¶

In order to take advantage of all parts of Truffle while building this tool, we need to separate our code into two distinct pieces. First, we need to write a library which will constitute the bulk of the app, and allow us to perform actions against the ENS contracts quickly and easily from Javascript. Next is the CLI; the CLI will take advantage of the library itself, and will be the user's interface to our application.

The Library¶

The library islib/ens_registrar.js . Let's take a look at the constructor:

... constructor

(AuctionRegistrar ,

Deed ,

registrarAddress ,

provider ,

fromAddress)

{

this . web3

=

new

Web3 (provider)

this . Deed

=

Deed

this . Deed . setProvider (provider)

AuctionRegistrar . setProvider (provider)

AuctionRegistrar . defaults ({

from :

fromAddress ,

gas :

400000

})

this . registrar

=

AuctionRegistrar . at (registrarAddress) } ... The library constructor requires a few things:

1. the contract interfaces (AuctionRegistrar
2. &Deed
3.)
4. the address on the network of the registrar
5. a web3 provider that will be used to connect to the desired Ethereum network
6. and the account address that will provide gas for transactions.

It's crucial that the library remain ignorant of the creation of these variables if we want to take advantage ofboth Truffle's testing pipeline and a CLI. As I see it, there are two ways to use the library: -Through Truffle , which manages contract addresses for testing against different networks (e.g. local, private, ropsten). Truffle makes testing our library easy. - Once we've convinced ourselves that the library is well tested and works properly, we'll want to tell the library where to find our own contracts, provider, and account on the mainnetthrough the CLI .

Let's take a look at how the library is used in both cases.

Use #1: Truffle Tests¶

Here'stest/ENS.js , a Truffle test that uses the library:

import

{

default

```
as
ENSAuctionLib
}
from
'../lib/ens_registrar' const
Registrar
=
artifacts . require ( './Registrar.sol' ) const
Deed
=
artifacts . require ( './Deed.sol' ) contract ( 'ENS integration' ,
( accounts )
=>
{
let
auctionRegistrar
before ( 'set up auction registrar' ,
( done )
=>
{
Registrar . deployed (). then (( instance )
=>
{
auctionRegistrar
=
new
ENSAuctionLib (
Registrar ,
Deed ,
instance . address ,
web3 . currentProvider ,
accounts [ 0 ]
)
}). then (()
=>
done ())
})
```

it ('demonstrates that the domain name is available' ,

(done)

=>

{

auctionRegistrar . available ('test')

. then ((isAvailable)

=>

{

assert . isTrue (isAvailable)

done ()

})

}) ... } Truffle injects a globalartifacts.require function, a helper for finding the right compiled contract artifacts within the test environment. The test then finds a deployed instance of the Registrar on the test network viaRegistrar.deployed() . Now, with the addition ofaccounts , which is passed in via thecontract wrapper(see here) , we have enough to instantiate the library and use it to test that the domain name'test' is available for auction.

Use #2: the CLI¶

Here'sindex.js , which provides our command line tool, using the library:

import

{

default

as

ENSAuctionLib

}

from

'./lib/ens_registrar' import

{

default

as

Web3

}

from

'web3' import

{

default

as

contract

}

from

'@truffle/contract' const

AuctionRegistrar

=

contract (require ('./build/contracts/Registrar.json')) const

Deed

=

contract (require ('./build/contracts/Deed.json')) export

default

function

(host ,

port ,

registrarAddress ,

fromAddress)

{

let

provider

=

new

Web3 . providers . HttpProvider (http:\/\/ { host } : { port })

return

new

ENSAuctionLib (

AuctionRegistrar ,

Deed ,

registrarAddress ,

provider ,

fromAddress

) } Here I'm using the same library,@truffle/contract , thatartifacts.require uses under the hood. Because I can't rely on the Truffle Suite within the CLI, I have to include the compiled contract artifacts manually. The rest is passed in through the CLI inbin/ensa.js :import

{

default

as

initializeLib

}

from

'../index' ... let

command

=

argv . _ [0] if

(command

===

'bid')

{

let

{

name ,

host ,

max ,

port ,

registrar ,

account ,

secret

}

=

argv

let

auctionRegistrar

=

initializeLib (host ,

port ,

registrar ,

account)

auctionRegistrar . createBid (name ,

account ,

max ,

secret)

. then (()

=>

console . log ('Created bid for '

+

name)) }

Usage¶

You can use the command line tool against any network that has an ENS registrar deployed. First, choose a command:

npm run -s ens Usage: bin/ensa.js [command]

[options] Commands: winner Current winner of bid bid Place a bid on a domain name reveal Reveal your bid on a domain

name

Options: --help Show help Then specify the correct options, including theaccount andregistrar :

npm run -s ens -- winner -n 'NewDomain' bin/ensa.js winner

Options: --help Show help

[boolean]

--host, -h HTTP host of Ethereum node [default: "testrpc"]

--port, -p HTTP port [default: "8545"]

--registrar, -r The address of the registrar [string]

[required]

--name, -n The name you want to register [string]

[required]

--account, -a The address to register the domain name [string]

[required] Missing required arguments: account, registrar

Recap¶

You might be thinking to yourself, "That was a short tutorial". That's because it doesn't need to be much longer: Creating a command line application with Truffle is very similar to creating a web application, but you have to do things a bit differently. For instance, instead of a build process, your command line tool needs to grab your contract artifacts and make them ready for use. Additionally, if you want to take advantage of Truffle's tests, you need to separate your code into a command line interface and a library -- a good practice anyway -- so you can use Truffle's testing framework to test your code.

Stay Tuned¶

Truffle 3.0 now makes it easier than ever to writeany Ethereum-enabled application, and not just web apps. Stay tuned for more examples in the future where we explore Desktop and Mobile applications, too. Cheers!