Setting up the folder structure¶

Truffle¶

Truffle requires an empty folder to start, so let's initialize it first before our React Native project:

mkdir truffle-tempcd

truffle-temp truffle init You should have the following inside the project folder:

├── contracts ├── migrations ├── test └── truffle-config.js

React Native¶

1. Initialize your React Native project in its own folder, as a sibling folder of your Truffle projecttruffle-temp
2. :
3. react-native
4. init
5. DrizzleReactNativeTutorial
6. React Native and Truffle folders should be in root since React Native doesn't allow you to use symlinks yet, and cannot import from files outside the React Native project folder.
7. Copy all the files intruffle-temp
8. into the root folder of your React Native project. Your folder should look like this in the end:
9. ├── android
10. ├── contracts
11. ├── ios
12. ├── migrations
13. ├── node_modules
14. ├── test
15. ├── App.js
16. ├── app.json
17. ├── index.js
18. ├── package.json
19. ├── truffle-config.js
20. ├── truffle.js
21. └── yarn.lock

Shimming web and node libraries on React Native¶

React Native is missing some of the global objects that are available on other platforms such as the web or Node. We will have to provide our own (i.e. a shim) through imported libraries or in some cases our own code.

1. Installnode-libs-react-native
2. ,vm-browserify
3. ,Base64
4. , andreact-native-randombytes
5. :
6. yarn
7. add
8. node-libs-react-native
9. vm-browserify
10. Base64
11. react-native-randombytes
12. Link the native libraries inreact-native-randombytes
13. :
14. react-native
15. link
16. react-native-randombytes
17. Create a new fileshims.js
18. in the root folder with the following code:
19. import
20. "node-libs-react-native/globals"
21. ;
22. import
23. {
24. btoa
25. }
26. from

```
"Base64"
;
import
nodeUrl
from
'url'
;
global
.
btoa
=
btoa
;
global
.
URL
=
class
URL
{
constructor
(
url
)
{
return
nodeUrl
.
parse
(
url
)
}
}
/**
  ◦ From https://github.com/facebook/react-native/blob/1151c096dab17e5d9a6ac05b61aacecd4305f3db/Libraries/polyfills/Object.es6.js
  ◦ This on RN's master branch as of Sep 11, 2018, however it has not made it into a release yet.
*
  ◦ The only modification made in Truffle's polyfill was to remove the check for an existing implementation.
  ◦ RN 0.57.7 (and below I assume) uses the non-spec compliant Object.assign that breaks in dev RN builds
  ◦ https://github.com/facebook/react-native/issues/16814
*/
Object
.
defineProperty
(
Object
,
'assign'
,
{
value
:
function
assign
(
target
,
varArgs
)
{
'use strict'
```

```
;
if
(
target
==
null
)
{
throw
new
TypeError
(
'Cannot convert undefined or null to object'
);
}
let
to
=
Object
(
target
);
for
(
let
index
=
1
;
index
<
arguments
.
length
;
index
++
)
{
let
nextSource
=
arguments
[
index
];
if
(
nextSource
!=
null
)
{
for
(
let
nextKey
in
nextSource
)
{
if
(
Object
.
prototype
.
hasOwnProperty
```

157. .
158. call
159. (
160. nextSource
161. ,
162. nextKey
163.))
164. {
165. to
166. [
167. nextKey
168.]
169. =
170. nextSource
171. [
172. nextKey
173.];
174. }
175. }
176. }
177. }
178. return
179. to
180. ;
181. },
182. writable
183. :
184. true
185. ,
186. configurable
187. :
188. true
189. ,
190. });
191. Create a new filern-cli.config.js
192. in the root folder with the following code:
193. const
194. nodeLibs
195. =
196. require
197. (
198. "node-libs-react-native"
199.);
200. nodeLibs
201. .
202. vm
203. =
204. require
205. .
206. resolve
207. (
208. "vm-browserify"
209.);
210. module
211. .
212. exports
213. =
214. {
215. resolver
216. :
217. {
218. extraNodeModules
219. :
220. nodeLibs
221. },
222. serializer
223. :
224. {

```
// From https://github.com/facebook/react-native/blob/v0.57.7/rn-get-polyfills.js
getPolyfills
:
()
=>
[
/**

    ◦ We omit RN's Object.assign polyfill

    ◦ If we don't, then node_modules will be using RN's polyfill rather than ours.
*/
// require.resolve('react-native/Libraries/polyfills/Object.es6.js'),
require
.
resolve
(
'react-native/Libraries/polyfills/console.js'
),
require
.
resolve
(
'react-native/Libraries/polyfills/error-guard.js'
),
require
.
resolve
(
'react-native/Libraries/polyfills/Number.es6.js'
),
require
.
resolve
(
'react-native/Libraries/polyfills/String.prototype.es6.js'
),
require
.
resolve
(
'react-native/Libraries/polyfills/Array.prototype.es6.js'
),
require
.
resolve
(
'react-native/Libraries/polyfills/Array.es6.js'
),
require
.
resolve
(
'react-native/Libraries/polyfills/Object.es7.js'
),
]
}
};
```

If you're wondering why we did all that inrn-cli.config.js
, refer tothis Gist
for a great explanation.
Finally let's import our shims inindex.js
. The very first line should be the following:

```
import
"./shims"
```

We're now done with replacing all the global objects and functions that Drizzle was expecting.

Setting up the smart contract¶

To add our smart contract we'll just follow the previous tutorial on Drizzle and React.

Do the steps fromWriting our smart contract up to (and including)Migration .

Connecting your app to your Ganache testnet¶

When we're Working with React Native and mobile apps, accessing the Ganache server that's running on your machine takes a bit more work than when we are building web apps. The sections below detail how to connect to the Ganache testnet with your mobile device/emulator.

Running the app¶

1. Start React Native Metro bundler:react-native start
2. Start your emulator/plug in your device

Android (Emulator/Physical Device)¶

The main thing for Android devices is that we have to reverse the ports so that we can point tolocalhost on the Android device to the Ganache server.

Make sure you've setup theAndroid Debug Bridge (adb) before doing these steps.

1. Startganache-cli
2. :ganache-cli -b 3
3. Compile and migrate contracts:truffle compile && truffle migrate
4. Reverse ports:adb reverse tcp:8545 tcp:8545
5. Install app:react-native run-android

iOS¶

Simulator¶

The iOS simulator will see servers onlocalhost just fine.

1. Startganache-cli
2. :ganache-cli -b 3
3. Compile and migrate contracts:truffle compile && truffle migrate
4. Install app:react-native run-ios
5. (you can also do this through Xcode)

Physical device¶

iOS physical devices involve the most manual work relative to other devices. You have to look up the local IP address of your machine and manually handle it every time it changes.

1. Find yourLOCAL_MACHINE_IP
2. by checking your network settings on your Mac where Ganache is running
3. Startganache-cli
4. :ganache-cli -b 3 -h LOCAL_MACHINE_IP
5. Intruffle.js
6. fordevelopment
7. , point Truffle toLOCAL_MACHINE_IP
8. Compile and migrate contracts:truffle compile && truffle migrate
9. Inindex.js
10. , point Drizzle toLOCAL_MACHINE_IP
11. const
12. options
13. =
14. {
15. ...
16. web3
17. :
18. {
19. fallback
20. :

21. {
22. type
23. :
24. "ws"
25. ,
26. url
27. :
28. "ws://LOCAL_MACHINE_IP:8545"
29. }
30. }
31. };
32. Install: Do it through Xcode

Setting up Drizzle¶

Install Drizzle:

yarn add drizzle Set up the Drizzle store by adding the following code toindex.js :

import

React

from

"react" ; import

{

Drizzle ,

generateStore

}

from

"@drizzle/store" ; import

MyStringStore

from

"./build/contracts/MyStringStore.json" ; const

options

=

{

contracts :

[MyStringStore] }; const

drizzleStore

=

generateStore (options); const

drizzle

=

new

Drizzle (options ,

drizzleStore); AppRegistry . registerComponent (appName ,

()

=>

()

=>

< App

drizzle

{ drizzle }

/>); Yourindex.js should look like this in the end:

/* *@format* / import

"./shims" ; import

{

AppRegistry

}

from

"react-native" ; import

App

from

"./app/App" ; import

{

name

as

appName

}

from

"./app.json" ; import

React

from

"react" ; import

{

Drizzle ,

generateStore

}

from

"@drizzle/store" ; import

MyStringStore

from

"./build/contracts/MyStringStore.json" ; const options

=

{

contracts :

[MyStringStore] }; const

drizzleStore

=

generateStore (options); const

drizzle

=

new

Drizzle (options ,

drizzleStore); AppRegistry . registerComponent (appName ,

()

=>

()

=>

< App

drizzle

{ drizzle }

/>);

Wiring up the App component¶

This is pretty much the same as the web tutorial , but with React Native components instead of web ones. Refer to the web tutorial for a more in-depth explanation of what's going on.

Let's create a folder calledapp in the root of the project. Add a file calledApp.js to it.

Inapp/App.js , your code should look like this in the end:

/* *\ Sample React Native App * https://github.com/facebook/react-native * * @format * @flow*/ import

React ,

{

Component

}

from

"react" ; import

{

Platform ,

StyleSheet ,

```
Text ,
View
}
from
"react-native" ; type
Props
=
{}; export
default
class
App
extends
Component < Props
{
state
=
{
loading :
true ,
drizzleState :
null
};
componentDidMount ()
{
const
{
drizzle
}
=
this . props ;
this . unsubscribe
=
drizzle . store . subscribe (()
=>
{
const
drizzleState
```

=

drizzle . store . getState ();

if

(drizzleState . drizzleStatus . initialized)

{

this . setState ({

loading :

false ,

drizzleState

});

}

});

}

componentWillUnmount ()

{

this . unsubscribe ();

}

render ()

{

return

(

< View

style

{ styles . container }

{ this . state . loading

?

(

< Text

Loading

Drizzle ... < /Text>

)

:

(

< View

< Text

Drizzle

is

ready < /Text>

< /View>

)}

< /View>

);

} } const

styles

=

StyleSheet . create ({

container :

{

flex :

1 ,

justifyContent :

"center" ,

alignItems :

"center" ,

backgroundColor :

"#F5FCFF"

} }); Run the app, and you should see the stringLoading Drizzle... while you wait for Drizzle to initialize. Once initialization is complete, the stringDrizzle is ready should be visible.

Writing a component to read from Drizzle¶

Once again, this is very similar to theweb tutorial , just with React Native components.

AddReadString.js to the folderapp .app/ReadString.js should look like this:

import

React

from

"react" ; import

{

Text

}

from

"react-native" ; class

ReadString

extends

React . Component

{

```
state
=
{
dataKey :
null
};
componentDidMount ()
{
const
{
drizzle
}
=
this . props ;
const
contract
=
drizzle . contracts . MyStringStore ;
// let drizzle know we want to watch themyString method
const
dataKey
=
contract . methods [ "myString" ]. cacheCall ();
// save the dataKey to local component state for later reference
this . setState ({
dataKey
});
}
render ()
{
// get the contract state from drizzleState
const
{
MyStringStore
}
=
this . props . drizzleState . contracts ;
```

// using the saved dataKey, get the variable we're interested in

const

myString

=

MyStringStore . myString [this . state . dataKey];

// if it exists, then we display its value

return

< Text

    My

stored

string :

{ myString

&&

myString . value } < /Text>;

} } export

default

ReadString ; Add it toApp.js by modifying therender method:

import

ReadString

from

"./ReadString" ; // ... render ()

{

return

(

< View

style

{ styles . container }

{ this . state . loading

?

(

< Text

    Loading ... < /Text>

)

:

(

< View

< ReadString

drizzle

{ this . props . drizzle }

drizzleState

{ this . state . drizzleState }

/>

< /View>

)}

< /View>

); } You should now see the stringHello World being rendered after Drizzle has finished loading.

Writing a component to write to the smart contract¶

Once again, this is very similar to theweb tutorial , just with React Native components.

AddSetString.js to the folderapp .app/SetString.js should look like this:

import

React

from

"react" ; import

{

Text ,

View ,

Button ,

TextInput ,

StyleSheet

}

from

"react-native" ; class

SetString

extends

React . Component

{

state

=

{

stackId :

null ,

text :

```
""
};
submit
=
()
=>
{
this . setValue ( this . state . text );
};
setValue
=
value
=>
{
const
{
drizzle ,
drizzleState
}
=
this . props ;
const
contract
=
drizzle . contracts . MyStringStore ;
// let drizzle know we want to call theset method with value
const
stackId
=
contract . methods [ "set" ]. cacheSend ( value ,
{
from :
drizzleState . accounts [ 0 ]
});
// save the stackId for later reference
this . setState ({
stackId
});
};
getTxStatus
=
()
=>
{
// get the transaction states from the drizzle state
const
{
transactions ,
transactionStack
}
=
this . props . drizzleState ;
// get the transaction hash using our savedstackId
const
txHash
=
transactionStack [ this . state . stackId ];
// if transaction hash does not exist, don't display anything
if
( ! txHash )
return
null ;
// otherwise, return the transaction status
if
( transactions [ txHash ]
&&
transactions [ txHash ]. status )
return
Transaction status: { transactions [ txHash ]. status }  ;
return
null ;
};
render ()
{
```

return

(

< View

< TextInput

style

{ styles . input }

onChangeText

{ text

=>

this . setState ({

text

})}

value

{ this . state . text }

placeholder

"Enter some text"

/>

< Button

title

"Submit"

onPress

{ this . submit }

/>

< Text

```
        { this . getTxStatus ()} < /Text>
```

< /View>

);

} } const

styles

=

StyleSheet . create ({

input :

{

height :

40 ,

borderColor :

"gray" ,

borderWidth :

1

} }); export

default

SetString ; Add it toApp.js by modifying therender function

import

ReadString

from

"./ReadString" ; import

SetString

from

"./SetString" ; // ... render ()

{

return

(

< View

style

{ styles . container }

{ this . state . loading

?

(

< Text

Loading ... < /Text>

)

:

(

< View

< ReadString

drizzle

{ this . props . drizzle }

drizzleState

{ this . state . drizzleState }

/>

< SetString

drizzle

{ this . props . drizzle }

drizzleState

{ this . state . drizzleState }

/>

< /View>

)}

< /View>

); } Run the app, enter a new string, and press the Submit button. A transaction status ofpending will show and change tosuccess on completion. This string will persist through reloads of the app since it is connected to your Ganache testnet.

The Finish Line¶

Congratulations, you've just successfully integrated the full suite of Drizzle, Truffle, and Ganache tooling into your React Native dapp!