Getting Started with Drizzle and React¶

Archived: This tutorial has been archived and may not work as expected; versions are out of date, methods and workflows may have changed.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

Drizzle is the newest member of the Truffle Suite and our first front-end development tool. At its core,Drizzle takes care of synchronizing your contract data, transaction data and more from the blockchain to a Redux store. There are also higher-level abstractions on top of the basedrizzle library; tools for React compatibility (drizzle-react) and a set of ready-to-use React components (drizzle-react-components).

We're going to focus on the lower levels today, taking you through setting up a Truffle project with React and Drizzle from scratch. This way we can gain the best understanding of how Drizzle works under the hood. With this knowledge, you can leverage the full power of Drizzle with any front-end framework of your choosing, or use the higher-level React abstractions with confidence.

This will be a very minimal tutorial focused on setting and getting a simple string stored in a contract. It's meant for those with a basic knowledge of Truffle, who have some knowledge of JavaScript and React.js, but who are new to using Drizzle.

Note : For Truffle basics, please read through the TrufflePet Shop tutorial before proceeding.

In this tutorial we will be covering:

1. Setting up the development environment
2. Creating a Truffle project from scratch
3. Writing the smart contract
4. Compiling and migrating the smart contract
5. Testing the smart contract
6. Creating our React.js project
7. Wiring up the front-end client
8. Wire up the React app with Drizzle
9. Write a component to read from Drizzle
10. Write a component to write to the smart contract

Setting up the development environment¶

There are a few technical requirements before we start. Please install the following:

- Node.js v8+ LTS and npm
- (comes with Node)

Truffle¶

Once we have those installed, we only need one command to install Truffle:

npm install -g truffle To verify that Truffle is installed properly, typetruffle version on a terminal. If you see an error, make sure that your npm modules are added to your path.

Create-React-App¶

Finally, since this is a React.js tutorial, we will be creating our React project withCreate-React-App .

You won't have to do anything if you have NPM version 5.2 or above. You can check your NPM version by runningnpm --version . If you do not, then you will need to install the tool globally with this command:

npm install -g create-react-app

Creating a Truffle project¶

1. Truffle initializes in the current directory, so first create a directory in your development folder of choice and then move inside it.

mkdir drizzle-react-tutorialcd

drizzle-react-tutorial 1. Now we're ready to spawn our empty Truffle project by running the following command:

truffle init Let's take a brief look at the directory structure that was just generated.

Directory structure¶

The default Truffle directory structure contains the following:

- contracts/
- : Contains theSolidity
- source files for our smart contracts. There is an important contract in here calledMigrations.sol
- , which we'll talk about later.
- migrations/
- : Truffle uses a migration system to handle smart contract deployments. A migration is an additional special smart contract that keeps track of changes.
- test/
- : Contains both JavaScript and Solidity tests for our smart contracts.
- truffle-config.js
- : Truffle configuration file.

Writing our smart contract¶

We'll add a simple smart contract called MyStringStore.

1. Create a new file namedMyStringStore.sol
2. in thecontracts/
3. directory.
4. Add the following content to the file:

pragma solidity

^ 0.5.0 ; contract

MyStringStore

{

string

public myString

=

"Hello World" ;

function

set (string

memory

x)

public

{

myString

x;

} } Since this isn't a Solidity tutorial, all you need to know about this is:

- We've created a public string variable namedmyString
- and initialized it to "Hello World". This automatically creates a getter (since it's a public variable) so we don't have to write one ourselves.
- We've created a setter method that simply sets themyString
- variable with whatever string is passed in.

Launching a test blockchain with Truffle Develop¶

Before we move ahead, let's first launch our test blockchain with the Truffle Develop console.

Open up a new terminal, navigate to the project directory, and run the following command:

truffle develop This will spawn a new blockchain that listens on127.0.0.1:9545 by default.

Compiling and migrating the smart contract¶

Now we are ready to compile and migrate our contract.

Compilation¶

In the Truffle Develop console, type the following command:

compile Note : If you're on Windows and encountering problems running this command, please see the documentation on resolving naming conflicts on Windows.

You should see output similar to the following:

Compiling ./contracts/Migrations.sol... Compiling ./contracts/MyStringStore.sol... Writing artifacts to ./build/contracts

Migration¶

Now that we've successfully compiled our contracts, it's time to migrate them to the blockchain!

Note : Read more about migrations in theTruffle documentation.

To create our own migration script.

1. Create a new file named2_deploy_contracts.js
2. in themigrations/
3. directory.
4. Add the following content to the2_deploy_contracts.js
5. file:

const

MyStringStore

=

artifacts . require ("MyStringStore"); module . exports

=

function (deployer)

{

deployer . deploy (MyStringStore); }; 1. Back in our Truffle Develop console, migrate the contract to the blockchain.

migrate You should see the migrations being executed in order, with the details of each migration listed.

Testing the smart contract¶

Before we proceed, we should write a couple tests to ensure that our contract works as expected.

1. Create a new file namedMyStringStore.js
2. in thetest/
3. directory.
4. Add the following content to theMyStringStore.js
5. file:

const

MyStringStore

=

artifacts . require ("./MyStringStore.sol"); contract ("MyStringStore" ,

accounts

```
=>
{
it ( "should store the string 'Hey there!'" ,
async
()
=>
{
const
myStringStore
=
await
MyStringStore . deployed ();
// Set myString to "Hey there!"
await
myStringStore . set ( "Hey there!" ,
{
from :
accounts [ 0 ]
});
// Get myString from public variable getter
const
storedString
=
await
myStringStore . myString . call ();
assert . equal ( storedString ,
"Hey there!" ,
"The string was not stored" );
}); });
```

Running the tests¶

1. Back in the Truffle Develop console, run the tests:

test 1. If all the tests pass, you'll see console output similar to this:

Using network 'develop'.

Contract: MyStringStore ✓ should store the string 'Hey there!' (98ms)

1 passing (116ms) Awesome! Now we know that the contract actually works.

Creating our React.js project¶

Now that we are done with the smart contract, we can write our front-end client with React.js! In order to do this, open

another terminal, navigate to your project directory, and simply run this command (if you have NPM version 5.2 or above):

npx create-react-app client If you have an older version of NPM , make sure Create-React-App is installed globally as per the instructions in theSetting up the development environment section and then run the following:

create-react-app client This should create aclient directory in your Truffle project and bootstrap a barebones React.js project for you to start building your front-end with.

Wiring up the front-end client¶

Since Create-React-App's default behavior disallows importing files from outside of thesrc folder, we need to bring the contracts in ourbuild folder insidesrc . We can copy and paste them every time we compile our contracts, but a better way is to simply configure Truffle to put the files there.

In thetruffle-config.js file, replace the contents with the following:

const

path

=

require ("path"); module . exports

=

{

contracts_build_directory :

path . join (__dirname ,

"client/src/contracts") }; This will make sure to output the contract build artifacts directory inside your React project. But this also means we'll have to restart our Truffle Develop console. PressCTRL + C to exit out of the Truffle Develop console and then start it again withtruffle develop .

From there, make sure you run thecompile andmigrate commands again so that the new build artifacts will be output into the new folder. If you are encountering issues, trymigrate --reset for a clean migration from scratch.

Install Drizzle¶

This is the most delicious part, we install Drizzle. Make sure you are in theclient directory and then run the following:

npm install @drizzle/store And that's it for dependencies! Note that we don't need to install Web3.js or @truffle/contract ourselves. Drizzle contains everything we need to work reactively with our smart contracts.

Wire up the React app with Drizzle¶

Before we go further, let's start our React app by running the follow command inside ourclient directory:

npm start This will serve the front-end underlocalhost:3000 , so open that up in your browser.

Note : Make sure to use an incognito window if you already have MetaMask installed (or disable MetaMask for now). Otherwise, the app will try to use the network specified in MetaMask rather than the develop network under localhost:9545.

If the default Create-React-App page loaded without any issues, you may proceed.

Setup the store¶

The first thing we need to do is to setup and instantiate the Drizzle store. We are going add the following code toclient/src/index.js :

// import drizzle functions and contract artifact import

{

Drizzle

}

from

"@drizzle/store" ; import

MyStringStore

from

"./contracts/MyStringStore.json" ; // let drizzle know what contracts we want and how to access our test blockchain const

options

=

{

contracts :

[MyStringStore],

web3 :

{

fallback :

{

type :

"ws" ,

url :

"ws://127.0.0.1:9545" ,

},

}, }; // setup drizzle const

drizzle

=

new

Drizzle (options); First, we imported the tools from Drizzle as well as the contract definition.

We then built our options object for Drizzle, which in this case is just specifying the specific contract we want to be loaded by passing in the JSON build artifact.

And finally, we created thedrizzleStore and used that to create ourdrizzle instance which we will pass in as a prop to ourApp component.

Once that is complete, yourindex.js should look something like this:

import

React

from

'react' ; import

ReactDOM

from

'react-dom' ; import

'./index.css' ; import

App

from

'./App' ; import

*

as

serviceWorker

from

'./serviceWorker' ; // import drizzle functions and contract artifact import

{

Drizzle ,

generateStore

}

from

"@drizzle/store" ; import

MyStringStore

from

"./contracts/MyStringStore.json" ; // let drizzle know what contracts we want and how to access our test blockchain const

options

=

{

contracts :

[MyStringStore],

web3 :

{

fallback :

{

type :

"ws" ,

url :

"ws://127.0.0.1:9545" ,

},

}, }; // setup the drizzle store and drizzle const

drizzle

=

new

Drizzle (options); ReactDOM . render (< App

drizzle

{ drizzle }

/> ,

document . getElementById ('root')); Note again that thedrizzle instance is passed into theApp component as props.

Wire up the App component¶

Now that we have adrizzle instance to play around with, we can go intoclient/src/App.js to start working with the React API.

Adding state variables¶

First thing we will do is to add the following line inside our App component:

state

=

{

loading :

true ,

drizzleState :

null

}; We are going to be using two state variables here:

1. loading
2. — Indicates if Drizzle has finished initializing and the app is ready. The initialization process includes instantiatingweb3
3. and our smart contracts, fetching any available Ethereum accounts and listening (or, in cases where subscriptions are not supported: polling) for new blocks.
4. drizzleState
5. — This is where we will store the state of the Drizzle store in our top-level component. If we can keep this state variable up-to-date, then we can simply use simpleprops
6. andstate
7. to work with Drizzle (i.e. you don't have to use any Redux or advanced React patterns).

Adding some initialization logic¶

Next we will add in ourcomponentDidMount method into the component class so that we can run some initialization logic.

componentDidMount ()

{

const

{

drizzle

}

=

this . props ;

// subscribe to changes in the store

this . unsubscribe

=

drizzle . store . subscribe (()

=>

{

// every time the store updates, grab the state from drizzle

const

drizzleState

=

drizzle . store . getState ();

// check to see if it's ready, if so, update local component state

if

(drizzleState . drizzleStatus . initialized)

{

this . setState ({

loading :

false ,

drizzleState

});

}

}); } First, we grab thedrizzle instance from the props, then we calldrizzle.store.subscribe and pass in a callback function. This callback function is called whenever the Drizzle store is updated. Note that this store is actually a Redux store under the hood, so this might look familiar if you've used Redux previously.

Whenever the store is updated, we will try to grab the state withdrizzle.store.getState() and then if Drizzle is initialized and ready, we will setloading to false, and also update thedrizzleState state variable.

By doing this,drizzleState will always be up-to-date and we also know exactly when Drizzle is ready so we can use a loading component to let the user know.

Unsubscribing from the store¶

Note that we assign the return value of thesubscribe() to a class variablethis.unsubscribe . This is because it is always good practice to unsubscribe from any subscriptions you have when the component un-mounts. In order to do this, we save a reference to that subscription (i.e.this.unsubscribe), and insidecomponentWillUnmount , we have the following:

componentWillUnmount ()

{

this . unsubscribe (); } This will safely unsubscribe when the App component un-mounts so we can prevent any memory leaks.

Replace therender

method¶

Finally, we can replace the boilerplate render method with something that applies to us better:

render ()

{

if

(this . state . loading)

return

"Loading Drizzle..." ;

return

< div

className

"App"

Drizzle

is

ready < /div>; } In the next section, we will replace "Drizzle is ready" with an actual component that will read from the store. If you refresh your browser and run this app now, you should see "Loading Drizzle..." briefly flash on screen and then subsequently "Drizzle is ready".

Full component code¶

When you are done this section, yourApp component should look like the following:

class

App

extends

Component

{

state

=

{

loading :

true ,

drizzleState :

null

};

componentDidMount ()

{

const

{

drizzle

}

=

this . props ;

// subscribe to changes in the store

this . unsubscribe

=

drizzle . store . subscribe (()

=>

{

// every time the store updates, grab the state from drizzle

const

drizzleState

=

drizzle . store . getState ();

// check to see if it's ready, if so, update local component state

if

(drizzleState . drizzleStatus . initialized)

{

this . setState ({

loading :

false ,

drizzleState

});

}

});

}

componentWillUnmount ()

{

this . unsubscribe ();

}

render ()

{

if

(this . state . loading)

return

"Loading Drizzle..." ;

return

< div

className

"App"

Drizzle

is

ready < /div>;

} }

Write a component to read from Drizzle¶

First, let's create a new file atclient/src/ReadString.js and paste in the following:

import

React

from

"react" ; class

ReadString

extends

React . Component

{

componentDidMount ()

{

const

{

drizzle ,

drizzleState

}

=

this . props ;

console . log (drizzle);

console . log (drizzleState);

}

render ()

{

return

< div

ReadString

Component < /div>;

} } export

default

ReadString ; And then insideApp.js , import the new component with this statement:

import

ReadString

from

"./ReadString" ; Now modify yourApp.js render method so that we pass in thedrizzle instance from props as well as thedrizzleState from the component state:

render ()

{

if

(this . state . loading)

return

"Loading Drizzle..." ;

return

(

< div

className

"App"

< ReadString

drizzle

{ this . props . drizzle }

drizzleState

{ this . state . drizzleState }

/>

< /div>

); } Go back to the browser and open up your console. You should see that the twoconsole.log statements are working and they are displaying both thedrizzle instance as well as adrizzleState that is fully initialized.

What this tells us is that thedrizzleState we get in this component will always be fully ready once this component mounts. At this point, you can take some time to explore thedrizzle instance object as well as thedrizzleState object.

drizzle

instance anddrizzleState ¶

For the most part,drizzleState is there for you to read information from (i.e. contract state variables, return values, transaction status, account data, etc.), whereas thedrizzle instance is what you will use to actually get stuff done (i.e. call contract methods, the Web3 instance, etc.).

Wiring up theReadString

component¶

Now that we have access to ourdrizzle instance and thedrizzleState , we can put in the logic that allows us read the smart contract variable we are interested in. Here is what the full code ofReadString.js should look like:

import

React

from

"react" ; 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

< p

My

stored

string :

{ myString

&&

myString . value } < /p>;

} } export

default

ReadString ; If everything is working, your app should display "Hello World". But first, let's walk through what we did here.

When the component mounts¶

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

}); } When the component mounts, we first grab a reference to the contract we are interested in and assign it tocontract .

We then need to tell Drizzle to keep track of a variable we are interested in. In order to do that, we call the.cacheCall() function on themyString getter method.

What we get in return is adataKey that allows us to reference this variable. We save this to the component's state so we can use it later.

Therender

method¶

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

< p

My

stored

string :

{ myString

&&

myString . value } < /p>; } From thedrizzleState , we grab the slice of the state we are interested in, which in this case is theMyStringStore contract. From there, we use thedataKey we saved from before in order to access themyString variable.

Finally, we writemyString && myString.value to show the value of the variable if it exists, or nothing otherwise. And in this case, it should show "Hello World" since that is the string the contract is initialized with.

Quick Recap¶

The most important thing to get out of this section here is that there are two steps to reading a value with Drizzle:

1. First, you need to let Drizzle know what variable you want to watch for. Drizzle will give you adataKey
2. in return and you need to save it for later reference.
3. Second, due to the asynchronous nature of how Drizzle works, you should be watching for changes indrizzleState
4. . Once the variable accessed by thedataKey
5. exists, you will be able to get the value you are interested in.

Write a component to write to the smart contract¶

Of course, simply reading a pre-initialized variable is no fun at all; we want something that we can interact with. In this section, we will create an input box where you can type a string of your choice and have it save to the blockchain forever!

First, let's create a new fileclient/src/SetString.js and paste in the following:

```
import

React

from

"react" ; class

SetString

extends

React . Component

{

state

=

{

stackId :

null

};

handleKeyDown

=

e

=>

{

// if the enter key is pressed, set the value with the string

if

( e . keyCode

===

13 )

{

this . setValue ( e . target . value );

}

};

setValue

=

value

=>

{

const

{
```

```
drizzle ,
drizzleState
}
=
this . props ;
const
contract
=
drizzle . contracts . MyStringStore ;
// let drizzle know we want to call the set 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 saved stackId
```

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

return

`Transaction status: { transactions [txHash]

&&

transactions [txHash]. status } ` ;

};

render ()

{

return

(

< div

< input

type

"text"

onKeyDown

{ this . handleKeyDown }

/>

< div

{ this . getTxStatus ()} < /div>

< /div>

);

} } export

default

SetString ; At this point, import and include it insideApp.js just like you did with theReadString component:

import

SetString

from

"./SetString" ; // ...

render ()

{

if

(this . state . loading)

return

"Loading Drizzle..." ;

return

(

< div

className

"App"

< ReadString

drizzle

{ this . props . drizzle }

drizzleState

{ this . state . drizzleState }

/>

< SetString

drizzle

{ this . props . drizzle }

drizzleState

{ this . state . drizzleState }

/>

< /div>

);

} At this point, the app should work and you should try it out. You should be able to type something into the input text box, hit Enter, and Drizzle's react store will automatically display the new string.

Next, let's go throughSetString.js step-by-step.

General structure¶

First let's take a look at the general React.js boilerplate that we need.

class

SetString

```
extends
React . Component
{
state
=
{
stackId :
null
};
handleKeyDown
=
e
=>
{
// if the enter key is pressed, set the value with the string
if
( e . keyCode
===
13 )
{
this . setValue ( e . target . value );
}
};
setValue
=
value
=>
{
...
};
getTxStatus
=
()
=>
{
...
};
```

render ()

{

return

(

< div

< input

type

"text"

onKeyDown

{ this . handleKeyDown }

/>

< div

{ this . getTxStatus ()} < /div>

< /div>

);

} } In this component, we will have an input text box for the user to type in a string, and when the Enter key is pressed, thesetValue method will be called with the string as a parameter.

Also, we will display the status of the transaction. ThegetTxStatus method will return a string displaying the status of the transaction by referencing astackId state variable (more on this later).

Submitting the transaction¶

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

}); }; We first assign the contract from thedrizzle instance intocontract , and then we callcacheSend() on the method we are interested in (i.e.set). Then we pass in the string we want to set (i.e.value) as well as our transaction options (in this case, just thefrom field). Note that we can get our current account address fromdrizzleState.accounts[0] .

What we get in return is astackId , which is a reference to the transaction that we want to execute. Ethereum transactions don't receive a hash until they're broadcast to the network. In case an error occurs before broadcast, Drizzle keeps track of these transactions by giving each it's own ID. Once successfully broadcasted, thestackId will point to the transaction hash, so we save it in our local component state for later usage.

Tracking transaction status¶

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

return

`Transaction status: { transactions [txHash]

&&

transactions [txHash]. status } ` ; }; Now that we have astackId saved into our local component state, we can use this to check the status of our transaction. First, we need thetransactions andtransactionStack slices of state fromdrizzleState .

Then, we can get the transaction hash (assigned totxHash) viatransactionStack[stackId] . If the hash does not exist, then we know that the transaction has not been broadcasted yet and we return null.

Otherwise, we display a string to show the status of our transaction. Usually, this will either be "pending" or "success".

The End¶

Congratulations! You have taken a huge step to understanding how Drizzle works. Of course, this is only the beginning, you can use tools likedrizzle-react to help you integrate Drizzle into your dapp, reducing the necessary boilerplate that you would have to write.

Alternatively, you could also bootstrap your Drizzle dapp with ourTruffle box .