**LAB 7: Building Interactive Front Ends of Lottery Contract**

**Overview**

The purpose of this lab is to start to pivot working on a web application front end that a user can make use of to interact with this lottery contract. This lab will also guide you on playing lottery from our React App.

**Prerequisites**

This lab will use the GENI environment that you set up. If you haven’t set up the environment, please refer to the document on setting up environment.

You should have completed Lab 5 – Contract Deployment, received address of the lottery contract you deployed.

**Install Sublime Text 3 Code Editor**

To install Sublime Text 3 on your GENI machine, follow these steps:

1. Update the apt package list and install the dependencies necessary to fetch packages from https sources:

sudo apt update

sudo apt install apt-transport-https ca-certificates curl software-properties-common

1. Import the repository’s GPG key using the following curl command:

curl -fsSL https://download.sublimetext.com/sublimehq-pub.gpg | sudo apt-key add -

Add the Sublime Text APT repository to your system’s software repository list by typing:

sudo add-apt-repository "deb https://download.sublimetext.com/ apt/stable/"

1. Once the repository is enabled, update apt sources and install Sublime Text 3 with the following commands:

sudo apt update

sudo apt install sublime-text

1. You can start the Sublime Text editor from the terminal by typing

subl

**Building Interactive Front-Ends Of Lottery Contract**

We're now going to start to pivot working on a web application front end that a user can make use of to interact with this lottery contract. In order to put together a front end for a user to interact with this contract we're going to be making use of the React.js viewer rendering library.

**Ethereum App Architecture**

We're going to first start off by talking about the architecture around a traditional web application. A traditional web application does not use theory at all. So below is traditional architecture that you will see in the vast majority of web applications that exist in the world right now.

A screenshot of a cell phone

Description automatically generated

We might have some server. A user can access the server. The Server will respond with an HTML document and maybe some JavaScript assets. The user will then mess around with the application and eventually they will want to modify the data that exists inside the application. For instance, they may want to click on a button that says I want to add a new to do or whatever it is they want to modify some amount of data. So when a user clicks on that button that will traditionally send some type of request to our server and we will create a new to do inside of some persistent database so that whenever a user comes back to our application again in the future they'll see a new to do added on their to do list. So the entire idea here is that any time we want to write data to a traditional application we will always send-off requests to some server and that data will be written to some database and it does not matter if writing this data or sending the request to the server takes the form of a classic HTM form submittal or takes the form of an Ajax request no matter what it always gets sent to the server and the data is persisted to the database. So, with this traditional architecture everything revolves around the server that you and I would put together.

Let's now talk about how the Ethereum world takes this approach right here and really turns it on its head. Here's a very similar diagram but with a couple of little changes to it. It's an Ethereum sort of architecture.

A screenshot of a cell phone

Description automatically generated

We can certainly still have a server, but its role is dramatically diminished. The server does far less work than it did in the past. In this sort of world, our server can still send an HTML document and some JavaScript assets down to the browser. The user might interact with our application in some fashion and then at some point in time they'll click on say a button and that button is intended to change the data that is related to our contract. For instance, they want to enter into a lottery, or they want to add a new to do to a to do list or they want to create a new blog post whatever it might be. In the Ethereum world, whenever a user tries to change some data, they do not reach back out to our server. The server is not at all involved in that process. Instead the Ethereum application running inside the browser will make use of Web3 which communicates with Metamask. Metamask creates a transaction, signs it with the user's private key and sends that transaction to the Ethereum network.

The very important and critical thing to understand here is that the only way for a user to change data is through the use of their public and private keys. Remember any time user changes data they have to send a transaction to the network and that costs some amount of money and chances are you want your users to spend that money. You don't want your application to spend that money. These public and private keys only ever exist on the user's machine. The public and private keys will never ever under any circumstance be sent to your server. You will never receive private keys from your users. So, your server is no longer going to be responsible for writing any data to a database or to the blockchain or anything similar to that unless you specifically want your server to do to do so. However, if your server is doing that it will be by using your set of private keys. It will not be done by using your users. Your user will never send you their private keys.

So essentially what we're doing with this Ethereum architecture is moving all this responsibility of writing data to some database or to some persistent database from the server over to the client. And that means that the client needs to get a lot more intelligent and it needs to have a lot more functionality built into it than it may have had in the past. In the past with a traditional web application we can build a server that sends down some silly HTML document with no JavaScript and we can make use of plain HTML form submittals to allow users to change data inside the application.

And so that meant that our web application on the browser might have been very simplistic. No JavaScript whatsoever just plain HTML and we absolutely could get away with that back with this traditional architecture. But with this new way of doing things with an Ethereum architecture because the client must be responsible for modifying any data inside of our application, it means that all the code that we write is going to be focused around running in the user's browser. So, what that means is we need to use libraries and technologies that are all focused around putting more logic and more code more business logic inside the browser than we had in the past. So that is why we are going to be using React in this course. We're going to React because it makes our lives of writing complex applications in the browser much easier and much more straightforward.

**Application Overview**

Here's a mockup of the application we're going to make:

**A screenshot of a cell phone

Description automatically generated**

This first application is going to be focused entirely on how we interact with Ethereum contracts using JavaScript code. So, this is much more focused on some of the nuts and bolts rather than making a very pretty looking application. We're going to be focused much more on the functionality side of things.

A screenshot of a cell phone

Description automatically generated

Let's take a look at some of them at the very top are going have a little bit of text that is focused around informing the user about the status of our contract. So are we going to say who the manager of the contract is. Notice that it's going to be the managers address as opposed to any name or anything like that. On the line underneath that we will then state the number of people who have entered the contract and also how much money is currently inside of the prize pool. Most of this information right here can be retrieved through calls to our contract. Remember that a call is a read only process for obtaining some information from an individual contract.

The next section one down is going to enable users to enter into the lottery so our users should be able to enter some amount of ether that they want to use to enter into the contract. They should then be able to click on the enter button and then somehow enter into the contract itself. Essentially we want to call the enter function on our contract.

Underneath that we will have the section that is intended for use by the manager of the application. So, it's going to be a button that calls the pickWinner function. Remember that the pickWinner function can be only invoked by the manager of the contract. So, in theory we really only want to show this section right here if the user who is visiting the application is the manager that would be really nice to do. But for our first time to this application we're going to keep things simple and we're going to say that no matter who you are whether you are the manager or not we're going to always show the section right here with a button that says “Pick a Winner”. In the future we'll figure out how to do some more conditional rendering to decide whether or not to show a given element.

At the very bottom of the page we're going to have a little status text area. So, for example after clicking on the pick winner button right here we might show “A winner has been picked!”. The status text area is going to be used to inform our users about what is currently going on with the contract. Remember whenever a user tries to modify data inside of our contract that issues a transaction to the network and it can take anywhere from 15 seconds to 30 seconds or so for that transaction to be processed and cleared. So, we need to make sure that our users are reasonably aware of what is going on inside the contract as they start interacting with it. And for that we want to show them some status information and say okay like you know we're currently entering into the contract or maybe after clicking on the pick winner we then say OK a winner has been picked and here is who they are or whatever it might be. Essentially we just want some practice of informing our users about when a given operation has been completed for this first application. We're going to take this very simplistic approach of showing some text status. But in the future we will instead use spinner's and more complex messages to give our users a better idea of what is going on and make our application a little bit more dynamic and lively.

Now the last thing we're going to be using the Rinkeby test network for just about everything from here on out. We're using Rinkeby so that we can get a very realistic idea of how long these transactions take to be processed. A lot of people tend to develop these applications like these web front ends using test networks that run locally and those local networks always process transactions instantaneously. It makes the development process a lot smoother and a little bit faster because your transactions clear instantly. But at the end of the day that usually means that you just end up developing your applications under the assumption that your users will see transactions clear instantly. And so, it leads to some poor development practices or poor user interfaces because you're not really using the application as your users will use it. So that's why we're going to use Rinkeby because we really want to see those delays in understand how they affect how users use our application.

**Install Create-React-App Tool**

This tool right here create-react-app is a command line application that is used to generate a new react application. And we're going to use it to generate a new little project that is going to serve as the front end for our application.

In your terminal, type in command:

sudo npm install -g create-react-app

If you are not familiar with React or if you feel like you need a little bit of a refresher with React, please go to the <https://reactjs.org/tutorial/tutorial.html> and feel free to learn.

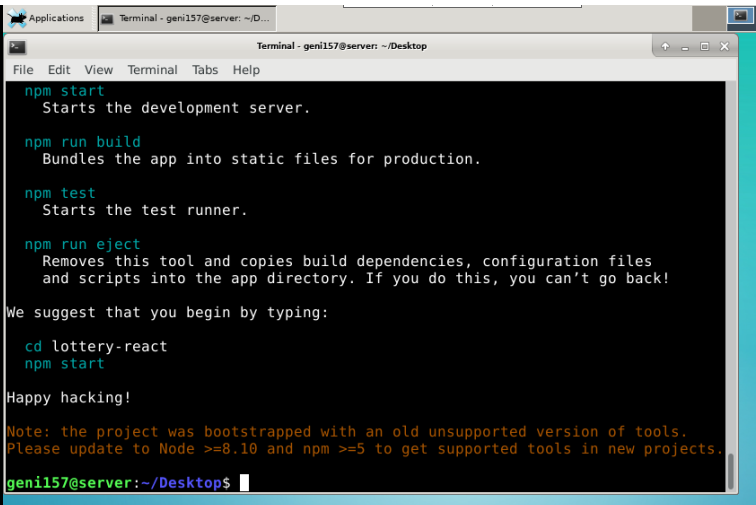
**Generate a new React Project**

We're going to generate a new react application by using the create-react tool. In your terminal, type in command

create-react-app lottery-react

The “lottery-react” on the above command is the name of our project. You can feel free to choose any name you want. But, don’t forget to add -react at the because this is how create-react-app works.

After you run this command and you should see some text about creating a new re-act up in this given folder. Once you are done creating a new react project, you will see screen like below:



**Getting Started with Create-React-App**

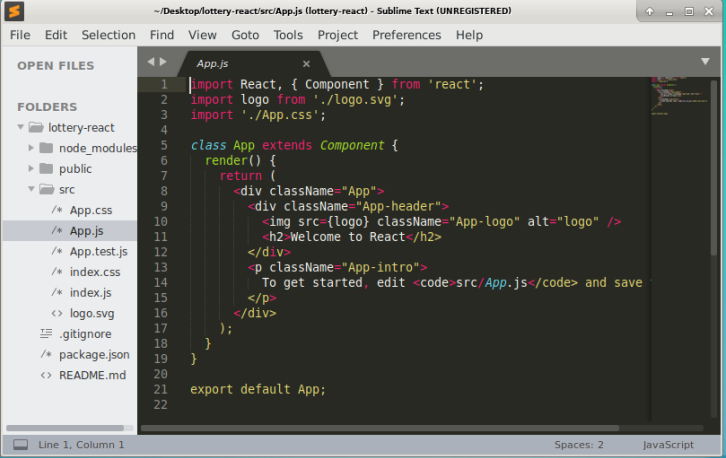
Let’s change our directory into lottery-react project directory which we just created using create-react-app tool. In your terminal, type in command:

cd lottery-react

We’ll start up our code editor inside of this directory with the following command:

subl

Here is our code editor and we're going to take a look at some of the different project files and folders that were created inside of here.



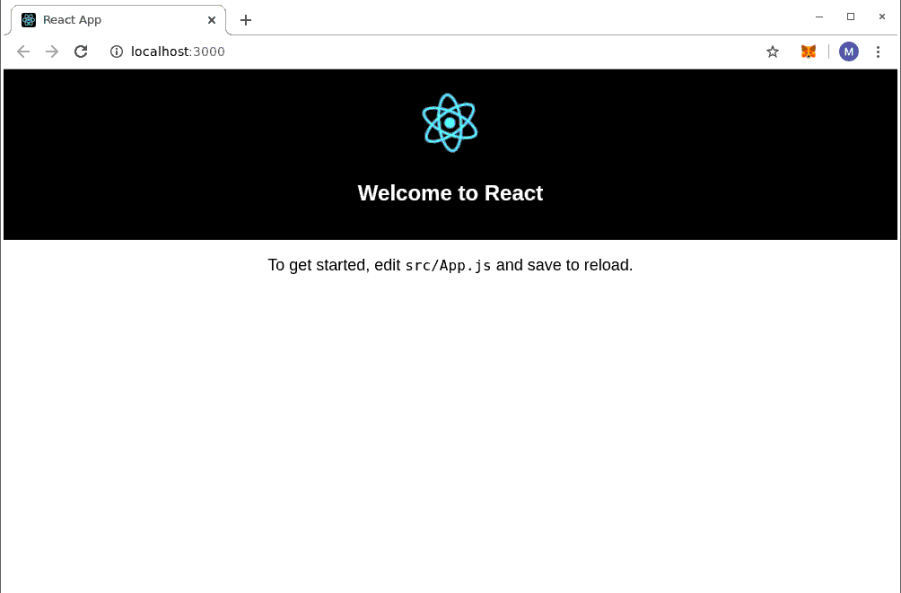
You'll notice two folders right away the public directory and the src directory. In the public directory, there's only one file that we're going to care about for the purpose of this application i.e. index.html file. This is the root HTML document that is going to be loaded up any time someone attempts to visit our application. We won't do too much inside this file, but I do want you to know that it exists.

Inside the src directory, you'll see a collection of JavaScript files and CSS files as well. The most relevant file that we're going to be doing a lot of work on is the App.js file. If you open this up you'll see a component and hopefully at this point this component doesn't look too scary. You'll see that it declares a class that extends the root component. There is a render method and that returns some amount of JSX which will eventually be displayed on the screen of our browser when we run this application.

Let's flip back over to our command line and start up this application and just make sure that we can access it inside of our browser. So back at the terminal, type in command

npm run start

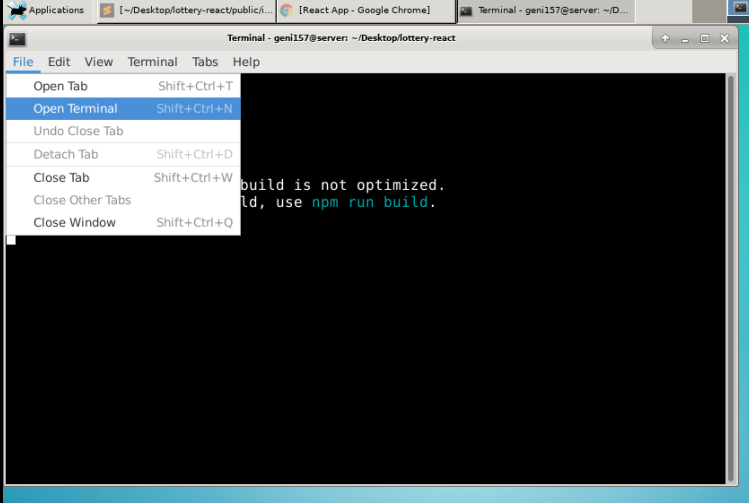
After a moment or two you'll see a message appear on the screen that says “Starting the development server” and then your web browser will automatically kick up and open a new tab at localhost:3000 after another couple of seconds. The application will finally boot up and you'll see a screen like below appear:



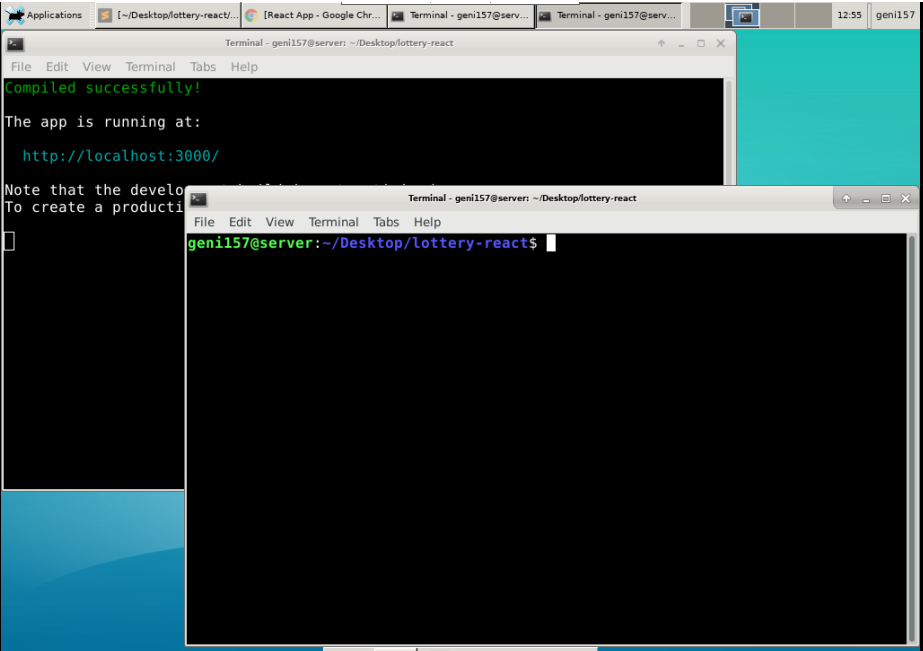
This is the App.js component that we were just looking at a moment ago inside of our code editor. We're not going to use any of this boilerplate that you see right here. In just a moment, we're going to delete a lot of the code inside that App.js file but I just want you to know and this is what you should be seen by default.

**Installing Web3 Library:**

Let's continue by installing the Web3 library. Installing the web 3 library is an absolute core pinnacle thing of every Ethereum application that we are going to put together. So over at the terminal I'm going to open up a second window. This is still inside of the same project directory of lottery-react. You can click on File and choose Open Terminal to do so.



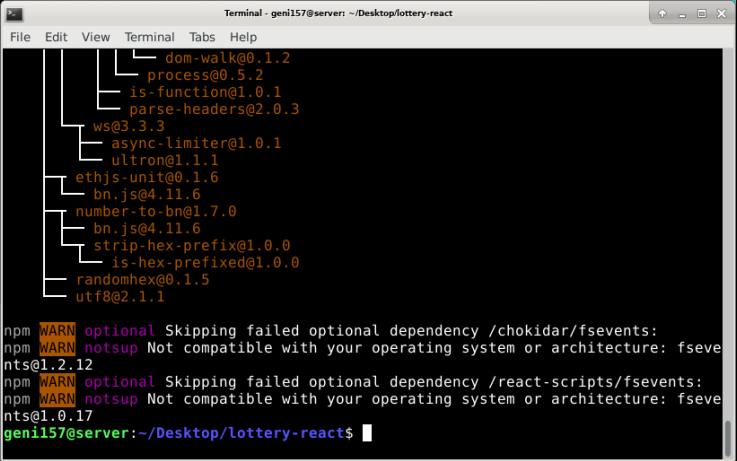
You’ll see a new terminal with lottery-react directory like below:



Inside the new terminal, type in command:

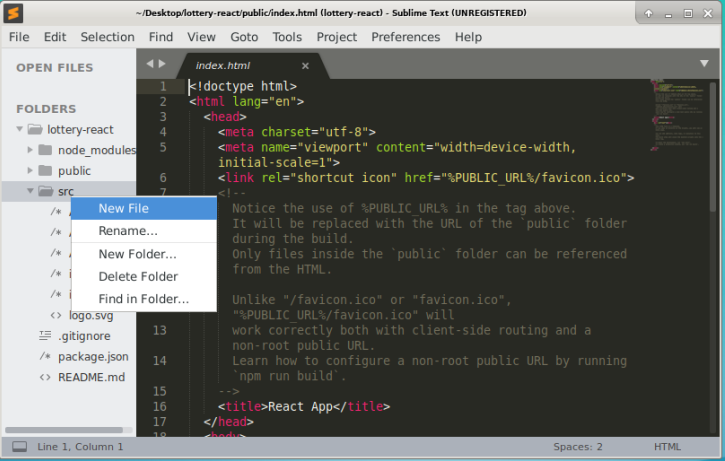
npm install –save web3@1.0.0-beta.35

It will take few minutes to install Web3 library on your project directory. Once you are done, you will see screen like below:



We're going to flip on over to our code editor and write some code that is going to set up our own local instance of Web3 and rip out the provider from the injected copy that is coming from metamask. And this is what is going to allow our instance of Web3 to automatically connect to the test network and make use of all of the accounts that have been assigned to the metamask extension.

Back inside our code editor, we’re going to create a new file inside of src directory. This file is going to contain all of our configuration code around the Web3 library. Right click on the src folder and select New File.



After that you will see empty file. Copy the following code:

import Web3 from 'web3';

const web3 = new Web3(window.ethereum)

             window.ethereum.enable().catch(error => {

             // User denied account access

             console.log(error)

             });

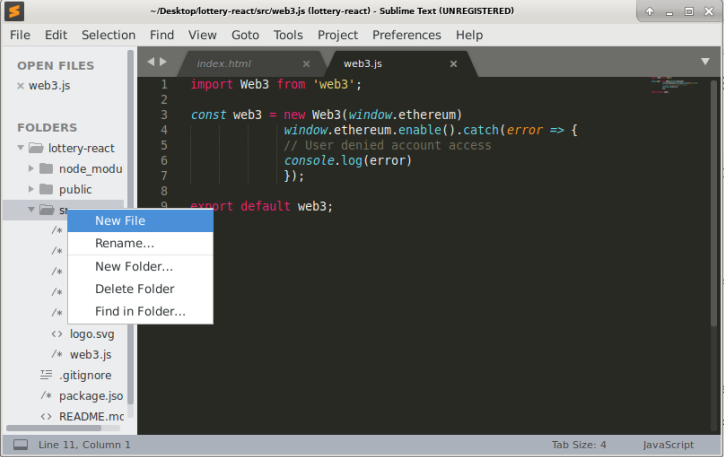
export default web3;

Click on File->Save. Enter ‘web3.js’ on filename. Click Save.

**Local Copy of Lottery Contract**

Back inside our code editor, we’re going to create a new file inside of src directory. The entire goal of this file is to set up a local copy of the contract that refers to the correct

address on the Rinkeby test network.Right click on the src folder and select New File.



After that you will see empty file. Copy the following code:

import web3 from './web3';

const address = '0x8DDB5D5F062C235A9eF6C65b789A3D3951134F0b';

const abi = [

{

constant: true,

inputs: [],

name: 'manager',

outputs: [{ name: '', type: 'address' }],

payable: false,

stateMutability: 'view',

type: 'function'

},

{

constant: false,

inputs: [],

name: 'pickWinner',

outputs: [],

payable: false,

stateMutability: 'nonpayable',

type: 'function'

},

{

constant: true,

inputs: [],

name: 'getPlayers',

outputs: [{ name: '', type: 'address[]' }],

payable: false,

stateMutability: 'view',

type: 'function'

},

{

constant: false,

inputs: [],

name: 'enter',

outputs: [],

payable: true,

stateMutability: 'payable',

type: 'function'

},

{

constant: true,

inputs: [{ name: '', type: 'uint256' }],

name: 'players',

outputs: [{ name: '', type: 'address' }],

payable: false,

stateMutability: 'view',

type: 'function'

},

{

inputs: [],

payable: false,

stateMutability: 'nonpayable',

type: 'constructor'

}

];

export default new web3.eth.Contract(abi, address);

**Note: The value of const address is the address of your deployed contract from your Lab 5: Contract Deployment. Also, the value of const abi is the ABI code of your lottery contract from same lab. Make sure you change it within your code editor.**

Click on File->Save. Enter ‘lottery.js’ on filename. Click Save.

**Rendering Lottery Contract Data:**

Once we setup the Web3 library and have local copy of our lottery contract, we can begin to change our app component which means App.js file. Now we are going to setup our form and perform specific tasks creating our lottery form like entering lottery and picking a winner as we discussed earlier on our application overview. We will call methods of our lottery contract within App.js and perform some tasks to create our lottery form.

Let’s change the App.js file code. Delete all the code present in the current file. Copy the following code into App.js file:

import React, { Component } from 'react';

import './App.css';

import web3 from './web3'; //importing Web3 instance

import lottery from './lottery'; // importing local copy of lottery contract

class App extends Component {

state = {

manager: '',

players: [],

balance: '',

value: '',

message: ''

};

async componentDidMount() {

//calling manager method of the lottery contract

const manager = await lottery.methods.manager().call();

//calling getPlayers method of the lottery contract

const players = await lottery.methods.getPlayers().call();

//calling getBalance of the lottery contract for prize pool balance

const balance = await web3.eth.getBalance(lottery.options.address);

// set the state for manager, players and balance once app is loaded

this.setState({ manager, players, balance });

}

//handles the enter button event of the lottery form

onSubmit = async event => {

event.preventDefault();

const accounts = await web3.eth.getAccounts();

this.setState({ message: 'Waiting on transaction success...' });

await lottery.methods.enter().send({

from: accounts[0],

value: web3.utils.toWei(this.state.value, 'ether')

});

this.setState({ message: 'You have been entered!' });

};

//handles the pick a winner button of our lottery form

onClick = async () => {

const accounts = await web3.eth.getAccounts();

this.setState({ message: 'Waiting on transaction success...' });

await lottery.methods.pickWinner().send({

from: accounts[0]

});

this.setState({ message: 'A winner has been picked!' });

};

//setting up the lottery form retrieving information from our lottery

//contract

render() {

return (

<div>

<h2>Lottery Contract</h2>

<p>

This contract is managed by {this.state.manager}. There are currently{' '}

{this.state.players.length} people entered, competing to win{' '}

{web3.utils.fromWei(this.state.balance, 'ether')} ether!

</p>

<hr />

<form onSubmit={this.onSubmit}>

<h4>Want to try your luck?</h4>

<div>

<label>Amount of ether to enter</label>

<input

value={this.state.value}

onChange={event => this.setState({ value: event.target.value })}

/>

</div>

<button>Enter</button>

</form>

<hr />

<h4>Ready to pick a winner?</h4>

<button onClick={this.onClick}>Pick a winner!</button>

<hr />

<h1>{this.state.message}</h1>

</div>

);

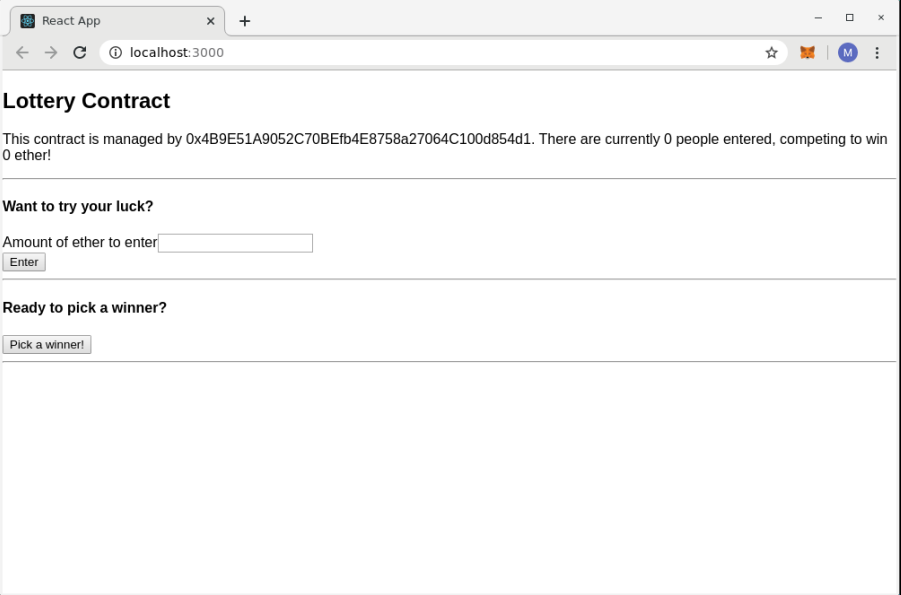
}

}

export default App;

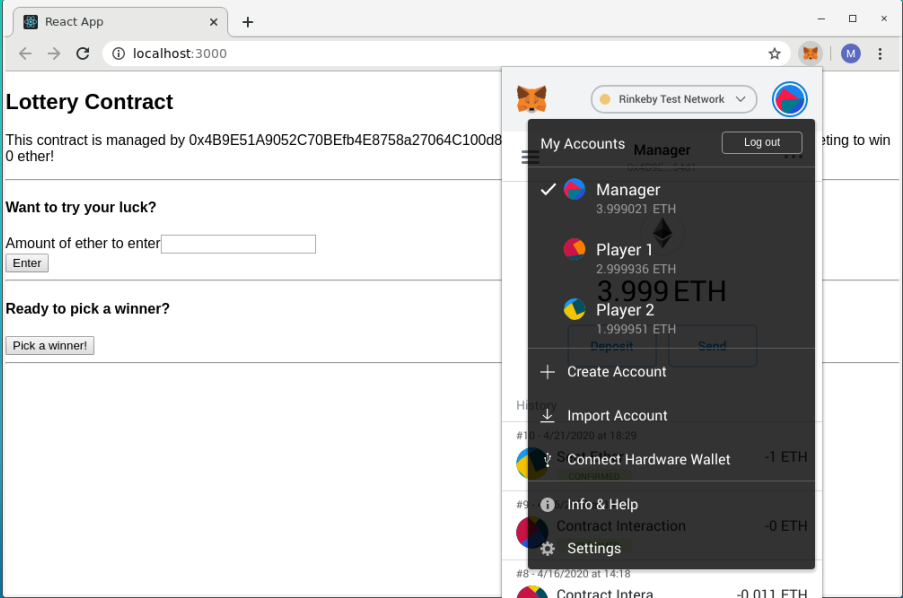
Click File->Save.

Go back to your browser and reload the localhost:3000 page which was open earlier when we run react. You will screen like below:



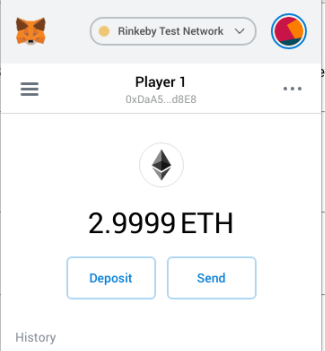
**Playing Lottery from our React App:**

Let’s at the beginning we have 0 people on our lottery contract and the prize pool amount is 0 ether. I assume that you have three accounts on your metamask wallet like below:

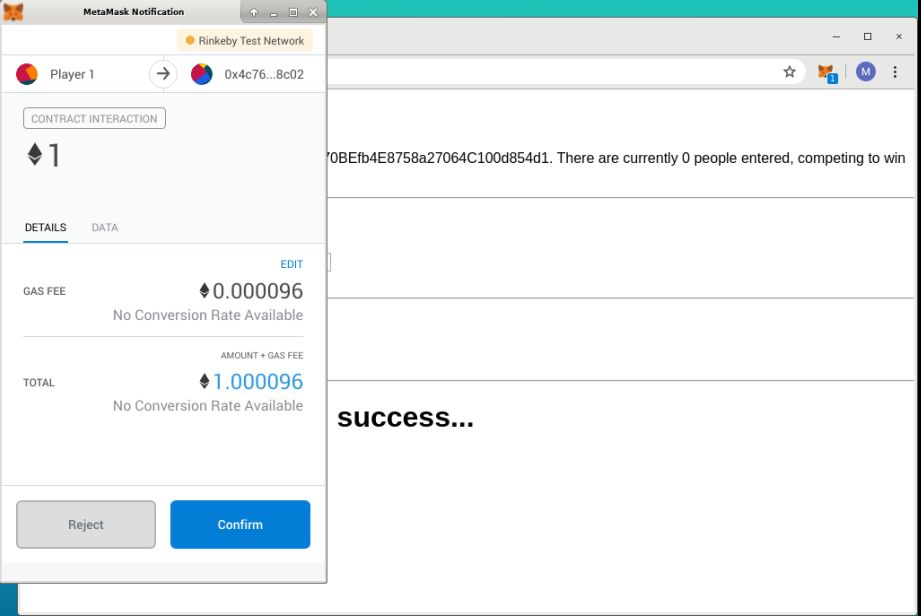


I have named the account that I used to deploy the contract as Manager and others two account as Player 1 and Player 2.

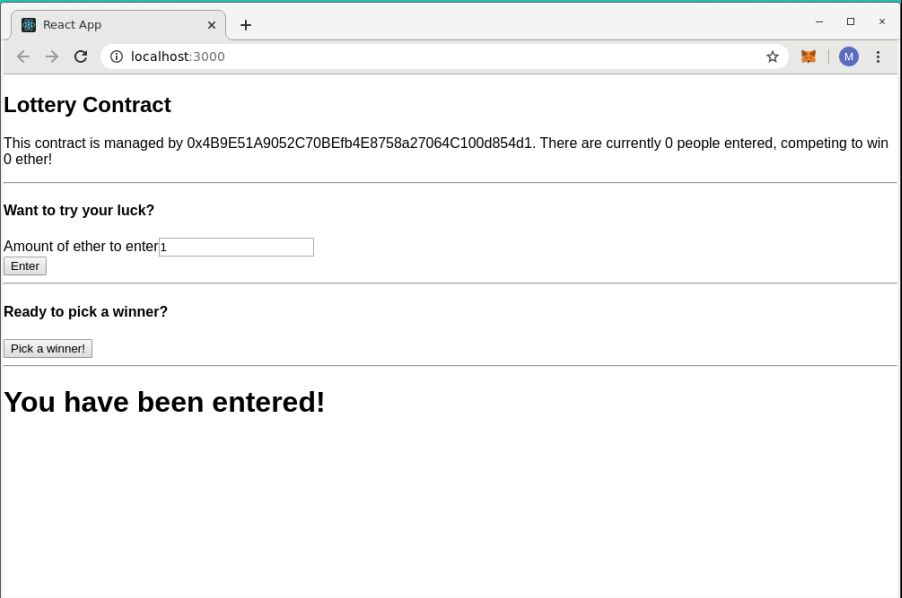
Select Player 1 account as your current metamask account like below:



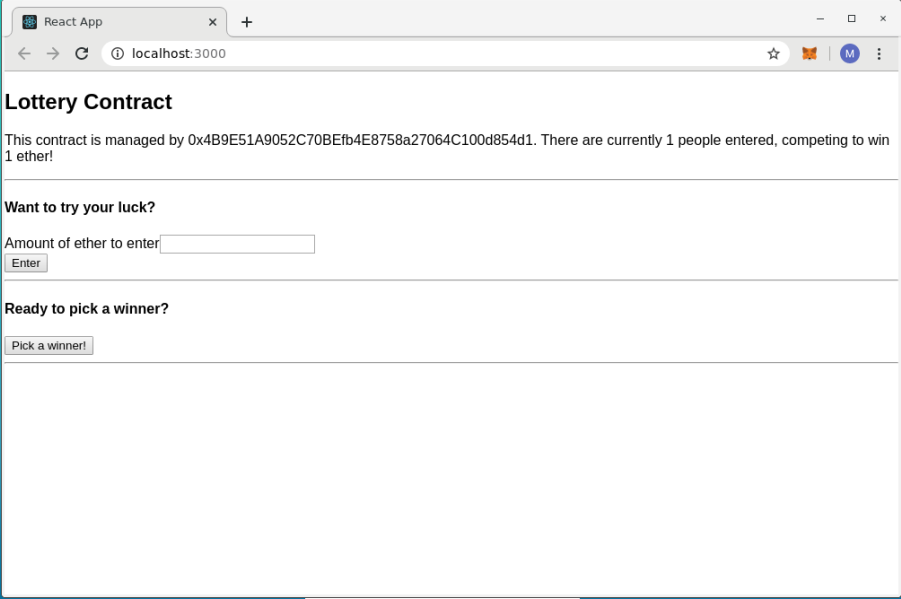
Let’s go back to the React App page. On the Amount of ether to enter box, type 1 and click Enter. You will see metamask popup to confirm the transaction like below:



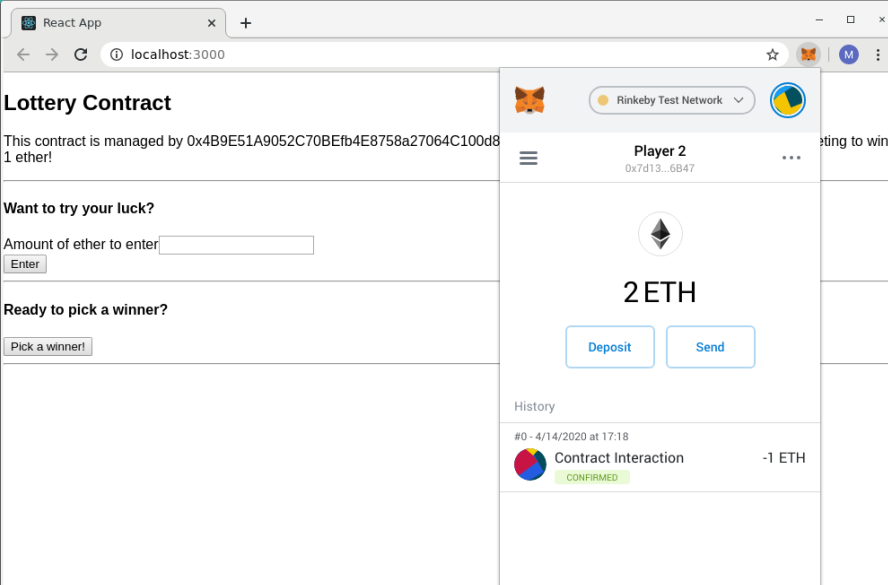
Click Confirm. It will take 10 to 15 seconds to confirm the transaction. Once your transaction completed, you will see message like below:



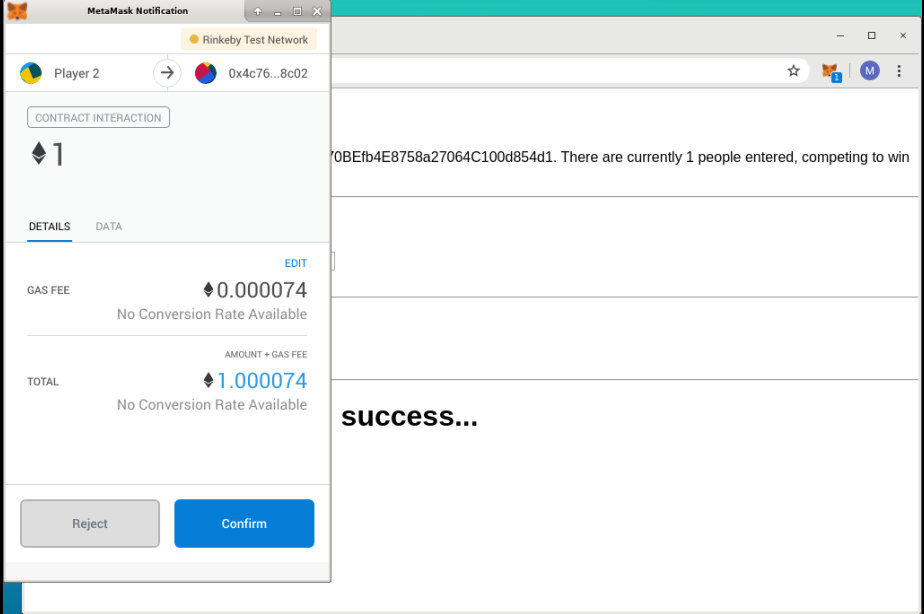
But, you may still notice it says 0 people and 0 ether on lottery contract details. You need to reload the page to see the current details of the lottery contract. Once you reload, you will see screen like below:



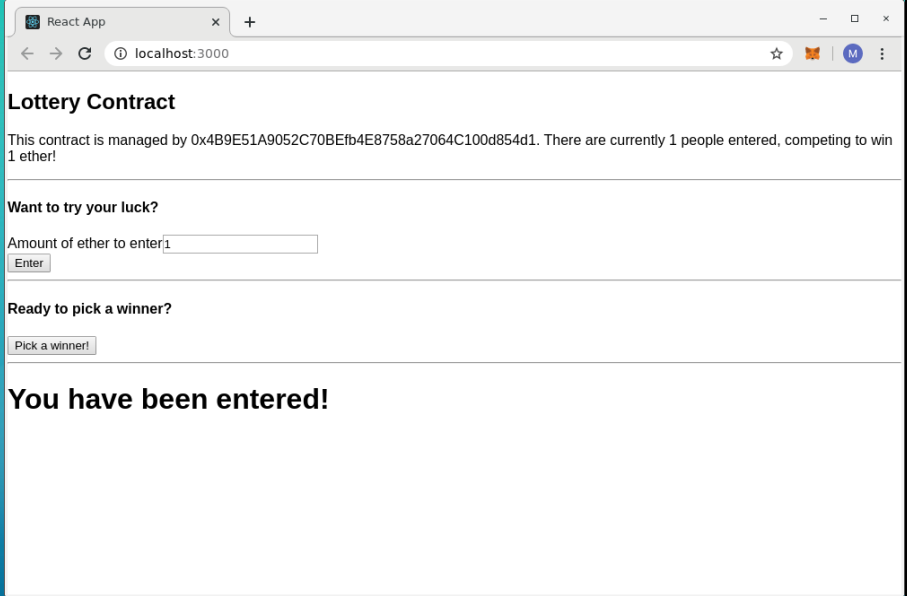
Let’s repeat the same process selecting Player 2 on metamask wallet. Select Player 2 on metamask wallet like below:



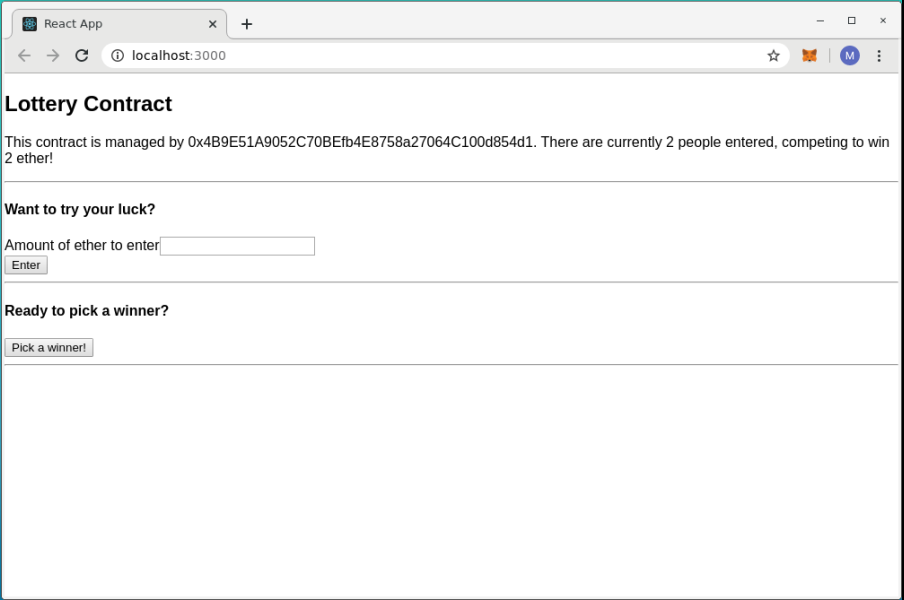
Go back to React App page. On the Amount of ether to enter box, type 1 and click Enter. You will see metamask popup to confirm the transaction like below:



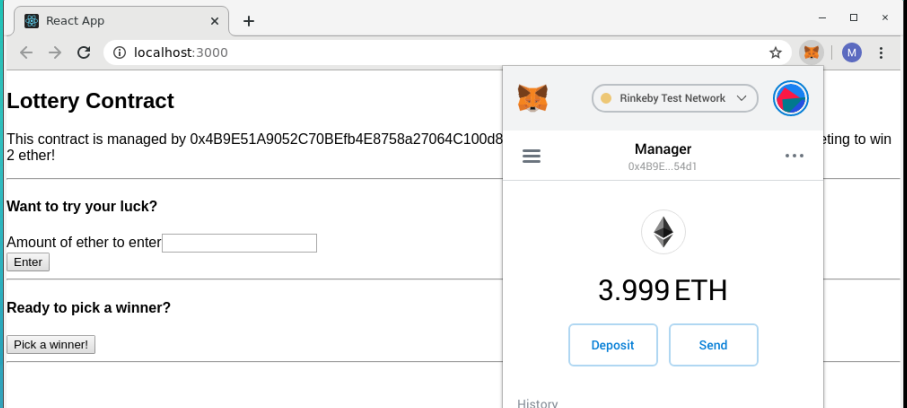
Click Confirm. It will take 10 to 15 seconds to confirm the transaction. Once your transaction completed, you will see message like below:



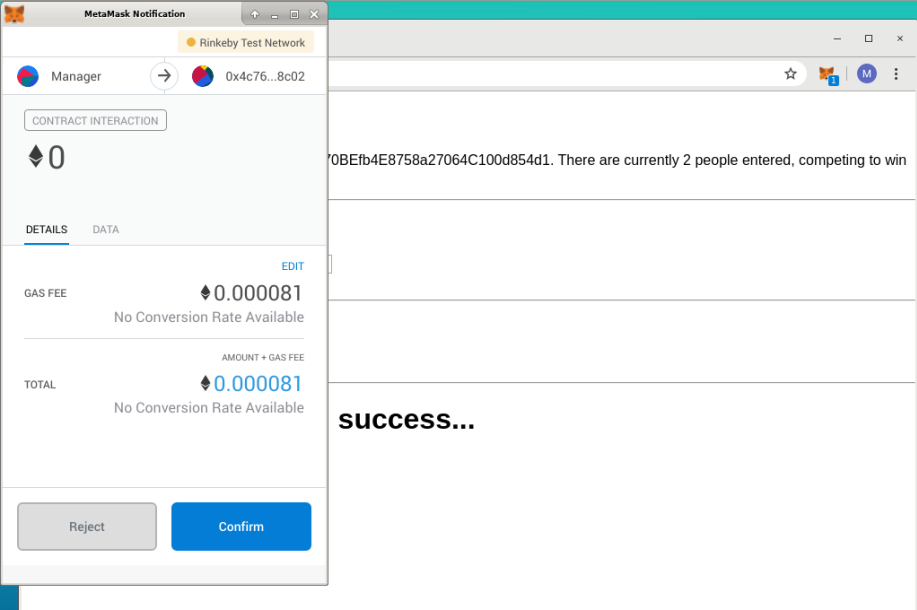
But, you may still notice it says 1 people and 1 ether on lottery contract details. You need to reload the page to see the current details of the lottery contract. Once you reload, you will see screen like below:



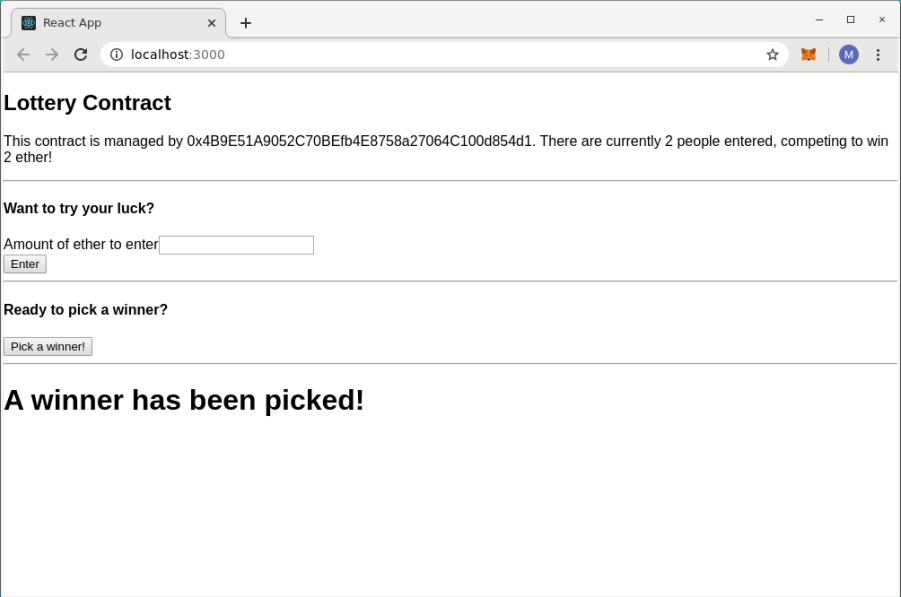
Let’s select Manager account on our metamask wallet as below:



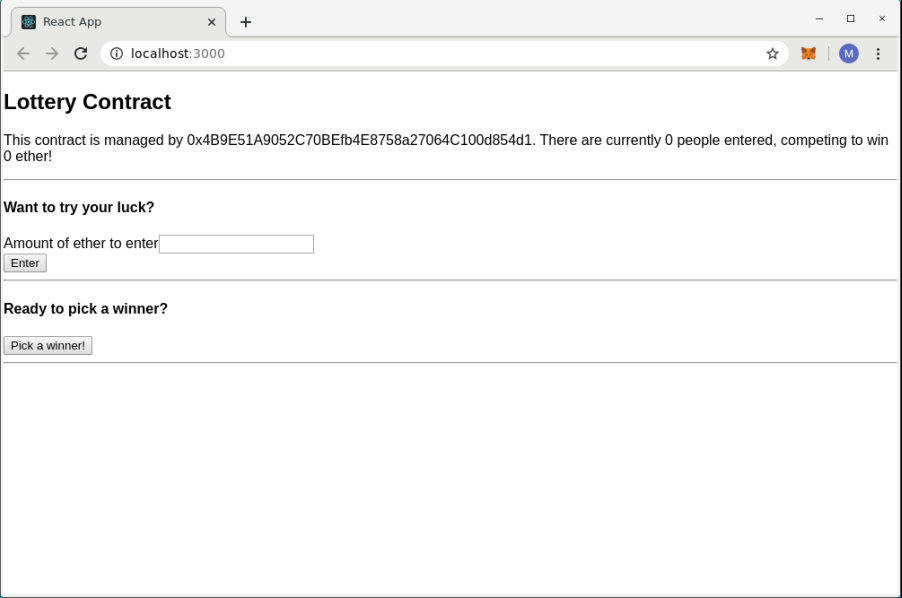
According to our lottery contract code, you know that only manager is able to pick a winner on our lottery contract. You can still click Pick a winner button using Player 1 and Player 2 account on our react app. But, you transaction will not be completed, and you will get an error. Let’s click Pick a winner! button selecting Manager account on our metamask wallet. Once you click the button, you will see metamask popup for transaction confirmation.



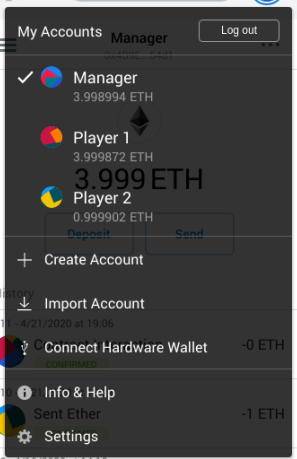
Click Confirm. It will take 10 to 15 seconds to confirm transaction. Once you transaction is completed, you will see message like below:



Reload the React App page. You will notice that it will say 0 ether and 0 player like below:



If you remember earlier writing our lottery contract, we write our code so that our contract prize pool and players array will be empty once manager pick a winner. You can check the balance of the players on metamask wallet.



On our case, Player 1 won the lottery. If you see the initial balance and new balance, you can find who won the lottery.