Masnad Nehith

Building A Blockchain DAPP From An Web APP

Helsinki Metropolia University of Applied Sciences

28 April 2019

Author	Masnad Nehith
Title	Building a blockchain dapp from an app
Number of Pages	62 pages
Date	28 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineer
Instructors	Antti Piironen

Blockchain is a new form of processing conceptualization that takes the distributed computing idea to a whole new level. There are a lot of advantages to it and as a result, it is now increasingly gaining popularity from all over the globe by IT industry.

This project implements the blockchain concept by using the back-end technologies such as NodeJS, Express, MySQL database and Github for keeping track of code updates, along with front-end technologies such as Axios, ReactJS and PayPal APIs for funds processing.

The idea was to develop a system that builds in NodeJS at back-end and in ReactJS in front-end and uses different APIs for communication between the layers of software to demonstrate the usage and development of basic blockchain application. The process also showcases the conversion from a traditional Web 2.0 application to a web 3.0 application.

Keywords	Axious, Express, Rinkeby, MetaMask, Web 3.0, Web 2.0, ReactJS, NodeJS, NextJS, decentralized, blockchain

Table of Contents

1 Introduction 1

1.1 What is blockchain? 1
1.2 Bitcoin 1

2 Theoretical background 3

2.1 Blockchain 3
2.2 Front-end technologies 4
2.2.1 ReactJS 4
2.2.2 Bootstrap 5
2.2.3 AXIOS 5
2.2.4 PayPal 5
2.3 Back-end technologies 6
2.3.1 NodeJS 6
2.3.2 MySQL DB 6
2.3.3 EXPRESS framework 6
2.3.4 NODEMON 7
2.3.5 POSTMAN framework 7
2.4 Working mechanism overview 7

3 BACK-END STRUCTURE AND IMPLEMENTATION 8

3.1 User interaction 8
3.2 Back-end creation overview 9
3.3 Installing the POSTMAN 11
3.4 System validation 11
3.5 Creating the Database 13
3.6 Index file 15

4 Front-end Centralized Web 2.0 16

4.1 Overview 16
4.2 Setting up the ReactJS 16
4.3 Adding Fields to View 17
4.4 POST with AXIOS to MySQL database 18
4.5 Fetching information from MySQL database 19

4.6 Setting up PayPal 19
4.7 Installing PayPal SDK 20
4.8 Creating the payment with PayPal 20
4.9 User check and Funding the App 22
4.10 Storing sessions 22
4.11 Picking the Winner Function 23
4.12 Picking the Participant 23
4.13 Winner Session 24
4.14 Deleting the participants 25
4.15 Makeover with bootstrap 25

5 App to Dapp – Decentralization and why do it? 27

5.1 Decentralized Application 27
5.2 Solidity Contract Types 28
5.3 Deploying and testing the smart contract 32
5.4 Getting Ether from Rinkeby 32
5.5 Deploying to Rinkeby TestNet 33

6 Back-end Web 3.0 Decentralized Application 33

6.1 Overview 33
6.2 Architecture of Back-end Web 3.0 33
6.3 Backbone of decentralized Application 34
6.4 Setting up NextJS for pages 34
6.5 Compile script (Compile.js) 35
6.6 Deploy script (deploy.js) 36
6.7 Deploy the contract to Rinkeby TestNet 37

7 Front-end of Web 3.0 Interactive Smart Contract Implementation 38

7.1 Index.js for ReactJS interface 38
7.2 Web 3.0 installation to front-end 38
7.3 Getting Contract Balance 39
7.4 Adding Ether to Contract 39
7.5 Pick winner function 40

8 Conclusion 40

References 1

1 Introduction

In today's world of faster communication and quick responses, users expect the quickest responses even from the toughest of computational problems and scientific problems that take even days and months to be processed. In order to process the large batches of information and big data, one might need a super computer to generate results. However, to use a super computer can be very costly so blockchain maybe a better solution.

1.1 What is blockchain?

Blockchain can be conceptualized as a chain of blocks of data that are tied together by the cryptographic hashes that contain the timestamps and address to the connected blocks. Each block contains the cryptographically secured information that can be used for different purposes and is safer to use for the scenarios, where changes to the blocks data are not expected. Blockchain differs from the distributed computing quite a bit and therefore, the implementations are very different.

In blockchain, the connection between the block nodes' is like a peer-to-peer network, where each one has the complete control, while staying connected to each other. This calls for validation protocols that can be used to manage and alter the structure of blockchain. Blocks keep adding to the chain and there is no limit to it, unless until, the development constraints of the system require it.

1.2 Bitcoin

The very first implementation of blockchain was bitcoin. An example of knowledge management system such as Wikipedia can be considered for understanding the immense number of articles and users, who keep adding data which ultimately goes to the primary database of the system. Getting closer to the structural analysis of blockchain and a distributed database, we see the differences in implementation of the system. A system such as Wikipedia uses a central or primary

database for it, whereas in blockchain, every node has the address and pointing capability to the next block and each block in itself is an ecosystem with localized database for keeping the data and addresses. This type of digital distribution changes a lot of factors of operating mechanism of systems. For the transactions to be carried out, they are broadcast and the for specific blocks, when needed, they update their hashes and events. Blockchain can also be imagined like a Merkle tree graph.

Bitcoin being the first major implementation of blockchain concept is a decentralized digital currency. It has no bank, no administrator and all the work that is being carried out is by peer-to-peer network for bitcoin. A publicly distributed ledger is a record keeper for the nodes and transactions that are carried out and is known to be the blockchain, because it contains the information about all the connected nodes and transactions going on.

Internet and the way we interact with it over the decades has evolved. When the internet was initially made available across the globe, it was a dump of information in a very static way that you could only access the information with some images and text, with no personalized information or user catered design or personal preferences. It was Web 1.0, whereas the era of web we are using in today's world is known as web 2.0; it provides a very personalized approach to most of the websites today where users can change and filter things according to their preferences, along with the systems automatically detecting and devise the content and interaction sites to the users. The application of such implementation can be seen in social media sites, where if a user searches for one particular type of content, the suggested or sponsored content may appear in their feed. This is due to the interactive nature of Web 2.0, but it is also worth remembering that the internet is also moving towards Web 3.0 which will automate many of the aspects of today's systems and will provide a whole new experience that will transform the user experience.

2 Theoretical background

2.1 Blockchain

Blockchain is a technological advancement of how decentralized a system can be. As the name indicates, it is a chain of blocks that work together by keeping the information about each proceeding block of information. As mentioned earlier, it can be conceptualized a Merkle tree graph, where blocks of information contain multiple types of information like previous block information, hashes and tx_roots (cryptocurrency). The blocks/records keep adding to the chain, where a publicly distributed ledger keeps the information of nodes, transactions and addresses. The structure of a bitcoin being a decentralized infrastructure can be seen to its roots in the following structure:

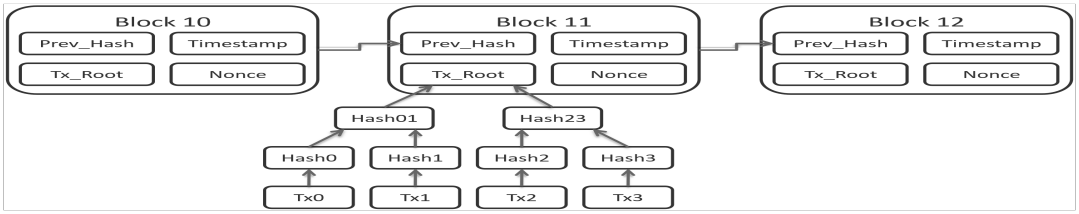

Figure 1 Bitcoin block structure [1]

As we can see from the following graph for transactions of bitcoin, the number of transactions keep increasing and adding more to the list.

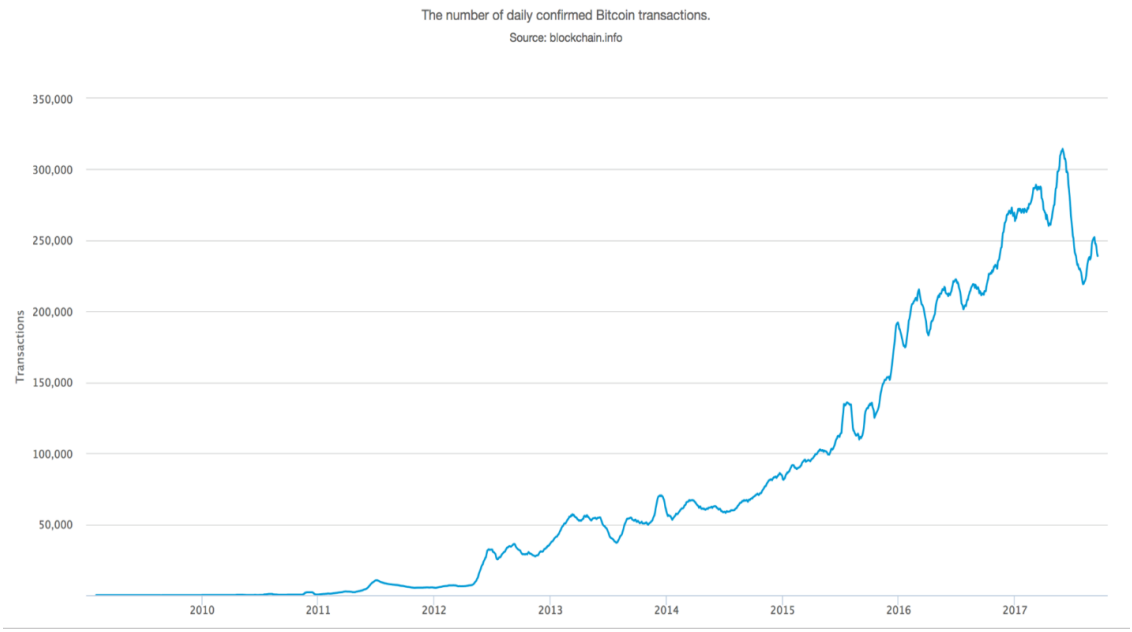

Figure 2 Bitcoin daily transactions over the years [2]

The above graph is indicative of how number of transactions for bitcoin increased over the years due to addition of blocks and due to price increase.

2.2 Front-end technologies

Front-end technology must be built in a way so that it coherently works with the back-end of the system. For example, in a PHP based system, the information is saved in MySQL database and is connected through PHP strings.
In this project, the front-end uses the technologies discussed below.

2.2.1 ReactJS

ReactJS is being used as the front-end because when playing with the crypto currencies and payment transactions that is big in numbers, the information needs to be updated very quickly and therefore the application needs to be fast and responsive. The ReactJS has an advantage of providing the features that can be used for building dynamic systems. It also has the ability to re-render a

part of the user interface when needed, which makes it faster because not a whole page needs to be loaded again for a part of it, as a result it enhances the user experience by providing fast response. It can interact with NodeJS backend without a hassle which makes it a perfect choice for the project. [3]

2.2.2 Bootstrap

Bootstrap is a fast and responsive framework for front-end. It is used for making the interface of the system look pretty. As the user interface is major part of the system, it helps users get the look and feel of it, the better looking and professional the design is, a greater impression it makes. It is also a learning part of the project to make a good interface that compliments the complex functionality of the system by helping in organization. [4]

2.2.3 AXIOS

Axios is a framework that forms as a middleware between the ReactJS and the NodeJS to carry out the transactions to MySQL database that contains the information of the system. It helps NodeJS and ReactJS interact smoothly so that they can keep interacting with one another when the information on screen needs to be updated rapidly.

2.2.4 PayPal

In order to process the payments, PayPal API is important because the system will be accepting payments from PayPal only. The interaction for PayPal API will be done between the NodeJS and PayPal, while the NodeJS is a back-end technology, but the interaction will be appearing on front-end as well in form of payment information update and users being redirected to PayPal login page for their information. [5]

2.3 Back-end technologies

Back-end technologies manage the underlying working mechanism of the system so that information could be organized and managed with user sessions and all the front-end functionality.
Following back-end technologies are being used:

2.3.1 NodeJS

NodeJS is known to be a non-block and event driven JavaScript. The file format of the NodeJS is unified with other versions of JS i.e. JSON. The rationale for using it is the two-way connection it provides to the system. That is, in real-time the systems can update the information dynamically based on the behavior of user, events driven by the users and the information being updated at the back-end of the systems, that needs to be pushed to the front-end without a delay or without requiring the page or application to be loaded again to the browser.
Along with the NodeJS itself, there are several libraries of that are also required by the system used as add-ons. [6]

2.3.2 MySQL DB

MySQL database is simple to write and provides a complete flexibility of creating a relational database. The database in this project is one of the simplest, yet most important parts because it will store the information like a distribution ledger in crypto currencies. The rationale is to develop a system that keeps things tied together by itself and for the blockchain record list as well.

2.3.3 EXPRESS framework

Express framework will help in navigation through the pages to further enhance the experience. This framework needs to be installed to ensure the flow of system. Addition of frameworks might complex the system development, but it is one of the essentials in blockchain projects.

2.3.4 NODEMON

It is a library responsible for starting and operating the server whenever it detects a change in information. The updating information is controlled at the front-end, but NODEMON helps us detect the changes and keeps the server engaged whenever needed.

2.3.5 POSTMAN framework

POSTMAN is used for testing the system's POST and GET requests. The implementation can't be directly deployed and therefore needs to be tested with dummy requests and outgoing information. The testing is very crucial part in a blockchain system, because once deployed the records will keep adding to the chain, which means no tolerances for alteration to the working mechanism.

2.4 Working mechanism overview

Starting the server on front-end with a server file (server.js). The initialization will make use of server file from front-end. This file will help us get started with the server making it up and running and this will also be used by the system for logging in and out of the system for user interactions.

But, for it to work we also need EXPRESS, which essentially a framework built on top of NodeJS that helps navigating through the pages and also helps push the results to the browser. Moreover, it helps with interaction of the APIs. The quick and responsiveness of the system requires the information to be pushed to the front-end as fast as possible and this is where EXPRESS will make it easier to keep the portion of UI updated.
We test the express first in a local environment and point it to the port of our choice for example port 3000. Testing at this stage will help make sure that we are good to start with the actual working scenario of express for making it listen to port (3000 in this example). [7]

3 BACK-END STRUCTURE AND IMPLEMENTATION

3.1 User interaction

The user interaction with the system for a user can be observed in the following use case diagram:

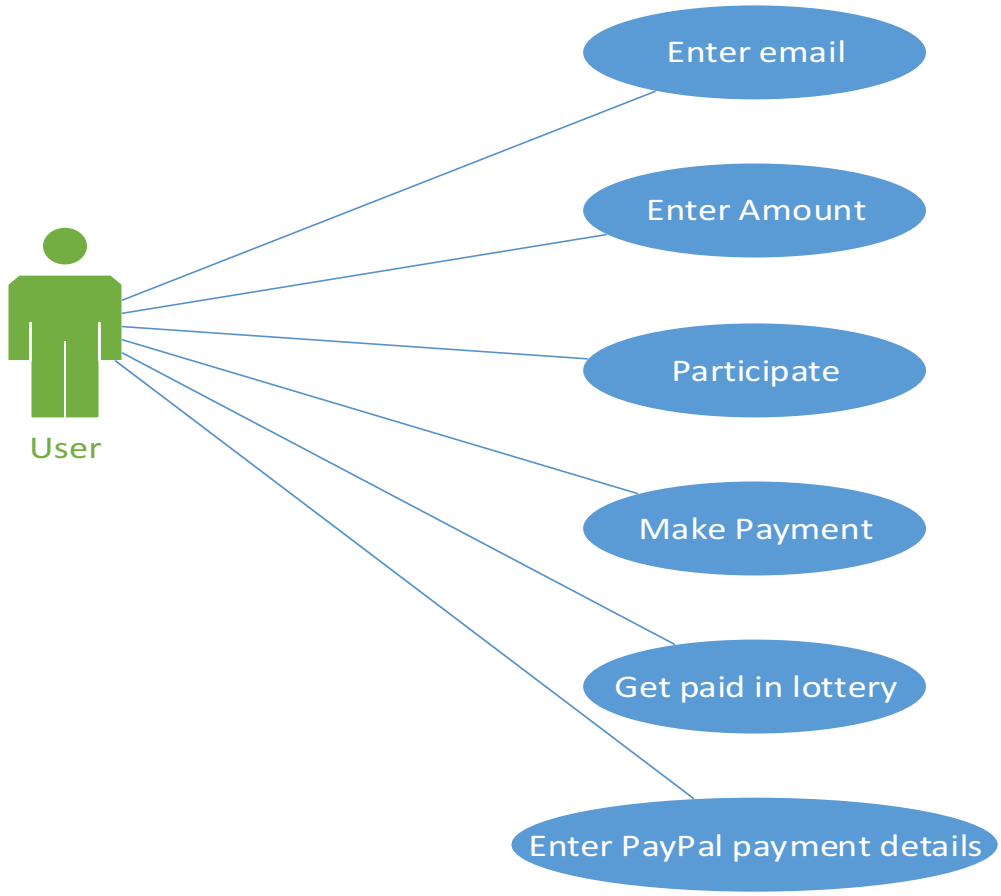

Figure 3 Use case scenario for Lottery Application

From the above use case diagram, we can see that the system has limited interaction with the user as the functions for users are kept simple. The process starts by the event called by the user, when user clicks participate after entering their amount and email address. The system cuts the total amount by 10% as a commission for the lottery.

3.2 Back-end creation overview

Firstly, we will start running all the nodes required for the system, then we'll be installing multiple node package model, along with which Express will EXPRESS framework will also help us in navigation of the pages that users can interact with from the back-end of the system.

Secondly, we will be using the Postman API for sending and receiving the information which will also be used for sending the data to the database. MySQL database will be used for fetching the information from the front-end and saving the required information to the back-end of the system.

NodeJS will act as intermediary app for sending the information to the browser and taking it from the database. The Postman will help in making a call for receiving the data. Postman being a very crucial part of the system will make calls when it detects any changes in the system. This is essential for keeping the interface updated with recent information. The very first thing in installing the express is making a small script and getting it run onto the server.
In this installation process, we'll be using the Web 2.0 folder that we created on GITHUB. Starting with the NPM, the package will be used for creation of the json file. Staring by entering the following command in the Node JS server
Initializing the folder with npm init. And the package name is Web 2.0. The author name will be user specific.

```
Npm init
```

For the process there will also be additional files in the system like package.json which will be initialized by the system. We will use the Atom to edit the code.
We will create the server.js file that will start the server, but before that we need to install the express using NodeJS console by writing the following command:

```
Npm install express –save.
```

After the installation of express, we will use it in server.js file. To start with, we'll be using the port 3000 for server to push information using the following commands in server.js:

```
app.listen(3000,()=>{
  console.log ('server is running on port 3000');
});
```

In the start of server.js, we need to add and import the important libraries to it so that they can be used in different sections and also to add variables.
Server.js will also be responsible for running other nodes for the systems and therefore the variables for initialization and the libraries will include the body-Parser, app variable for express library, the public link variable and the PayPal and session libraries. The start of the server.js file should look like this:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const {save_user_information, get_list_of_participants, delete_users} = re-
quire('./models/server_db');
const path = require('path');
const publicPath = path.join(__dirname, './public');
const paypal = require('paypal-rest-sdk');
const session = require('express-session');
```

Using the MySQL community download from MySQL server website, this will also require the user to enter the password for security purposes for access, but it is recommended not to use it for the initial demonstration as it may cause errors. MySQL is a database is used for storage of information.
Typing the MySQL in NodeJS terminal will allow us to check the output of the system to verify if its installed. If the installation is done correctly, it will state the access denied.

For further development, there needs to be MySQL workbench, which will make the interaction with database possible. This will use the same password which was entered while installing MySQL.

Getting back to the server, will be using the script to setup the inputs to system. We'll start by adding the packages to GIT, which will include the package-lock.json, package.json, and server.js.The commit command will be used to push the files to the server.

3.3 Installing the POSTMAN

Postman is a software where we can test our all API calls. This will require the postman API to be downloaded for the system. This will be used for sending the POST requests to the server.js. Any information we are sending towards the system needs to be parsed for execution without the error. This is the one thing where BodyParser will be required.

We will use the postman to send the raw code via localhost to port 3000.
This needs to be installed as a dev dependency and can be installed using the following script in NodeJS terminal:

```
Npm install nodemon –save-dev.
```

Another addition will also be required in package.json file. Under the scripts section that will be:

```
“start”: “nodemon server.js”
```

3.4 System validation

The server.js file will be used for validation check. The system will check that the amount is never less than otherwise the system will generate the error message. The code for message validation check will look like:

```
if(amount <= 1){
    return_info = {};
    return_info.error = true;
    return_info.message = "The amount should be greater than 1";
    return res.send(return_info);
  }
```

Whereas for testing the validation check, we'll use the POSTMAN for sending the raw json code and see if it works by entering the following script:

```
{
            "amount" : "1",
            "email" : masnad@masnad.com
}
```

Connecting the MySQL database to NodeJS app:

We'll start by creating the db.js file for connection. The script below will install the node.

```
Npm install mysql –save
```

The script in this file will create the connection and will also check for connection errors or disconnection. The timeout also needs to be included because we want the system to check the disconnection and redo the connection process.

It must be noted that this file provides the error check for db connection and will also only provide the name and password of the database. The actual database is to be created in a separate way. This approach is modular and one can easily check for the errors by simply including one file at a time to test where the error might be coming from.

The actual script that creates the connection of the DB is also the one which checks for disconnection in a recursive manner (if(err)) it'll keep checking for error until it goes away.

```
function handleDisconnect() {
  connection = mysql.createConnection(db_config);
    connection.connect(function(err) {
        if(err) {
            console.log('error when connecting to db:', err);
            setTimeout(handleDisconnect, 2000);
        }
    });
  connection.on('error', function(err) {
      console.log('db error', err);
      if(err.code === 'PROTOCOL_CONNECTION_LOST') {
```

```
            handleDisconnect();
        } else {
            throw err;
        }
    });
}
```

3.5 Creating the Database

For creation of the database, we'll use the MySQL workbench.
We will use the user interface to create the SQL queries. The creation of database should be done with the same name as the one we entered in db.js.To prevent the cluttering of information in node.js file, we create another folder called models, that will contain the server_db.js file. This file will help us keep the information organized.

The MySQL queries are used for performing the CRUD (Create, Remove, Update & Delete) operations in database. The database for the project at current stage contains one table named 'Lottery_information' for storing the email and amount. The table must have a primary key that is unique and therefore, the lottery_information table uses the 'ID' integer auto-increment, unique and primary key field for handling all the records.

For the purpose of keeping things organized, we create a new folder named models to place the queries and we create a new file called server_db.js which will interact with the server.js file for operating queries.

In this file (server_db.js) we link the db.js file to use the connection strings for database and error check, which in this way saves the effort of creating the script for connection between MySQL and NodeJS. This will be used to save and return the message using promises.

The working will start by taking the information in data variable and using the POST method for sending the information to the database and then checking it

by using promises before it is added to DB. The query works and executes directly in the .js file. This also contains the error checks and sends the output message. To send the data to the DB in a table we require the following 'INSERT' query script.

```
save_user_information = (data) => new Promise((resolve,reject)=>{
  db.query('INSERT INTO lottery_information SET ?',data, function(err,results,
fields){
    if(err){
      reject('could not insert into lottery information');
    }
    resolve('Succesful');
  });
})
```

The save_user_information method is used to perform this operation in server_db.js file therefore, we will be requiring this file in server.js. Therefore, it goes into start of server.js. For the save query function to be used in the server file, the module.exports is used with same name as function. The declaration of save_user_information also requires us to use the async and await so that the system can send the results only after it gets the return from promises, while also checking if there is a rejection. The results are pushed to the screen to validate input. This can also be validated by checking the table data from MySQL Workbench as can be seen in the figure below:

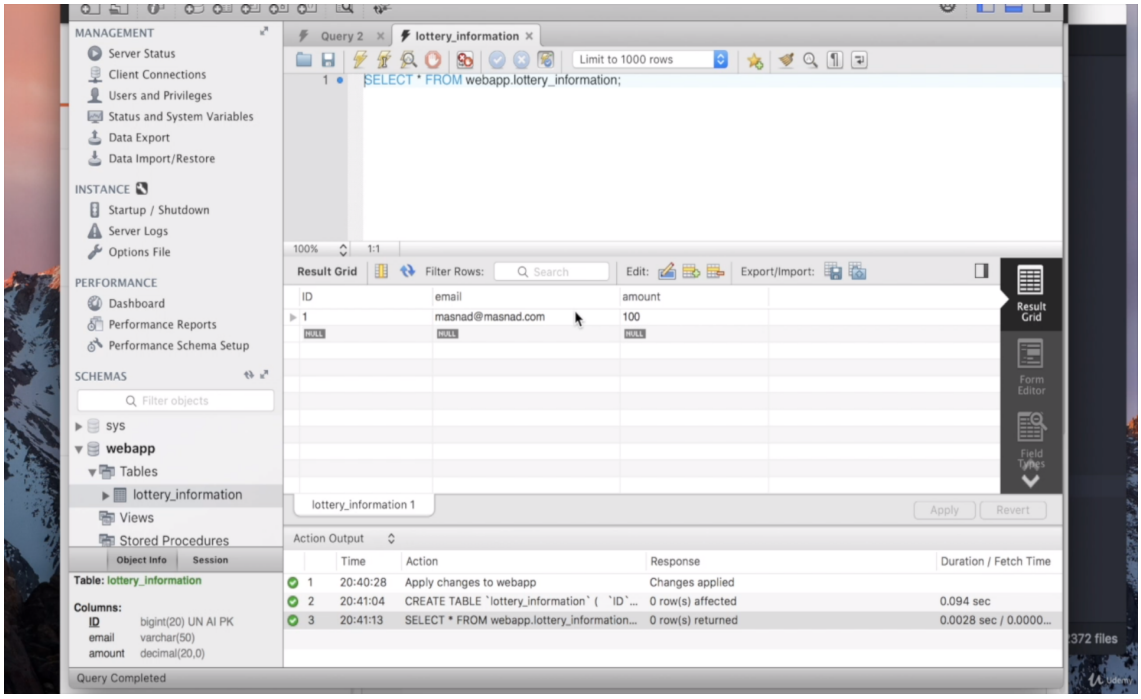

Figure 4 MySQL table check

The GET query is also used to get the information from DB and pushing it back to front-end. The code will also be added to server.js named get_total_amount. The server.js will only be calling the function, whereas the working code with MySQL query is written in server_db.js under models' folder. This also uses the await and async to wait for results from promises. Because in this return function we need the sum of all the amounts in table, we will use the 'SELECT' query with MySQL sum() function applied on column.

```
get_total_amount = (data) => new Promise((resolve,reject)=>{
  db.query('select sum(amount) as total_amount from lottery_information
',null, function(err,results, fields){
    if(err){
      reject('Could not get total amount');
    }
    resolve(results);
  });
})
```

3.6 Index file

This requires the public path so that the browser can execute the information from back-end using the index file. For the purpose of including the public library, we need to add the path library and use it to join the public folder. The index file itself

is a html file that will provide the way for front-end interaction. Here we use the express by providing it with the publicPath variable. The index file is the starting point for a website therefore, it is important to provide a way for localhost to see the default file. The sample html code looks like:

```
<!DOCTYPE html>
<html>
            <head>Title</head>
            <body>LOTTERY APPLICATION</body>
</html>
```

This is also the point, where we start the development of Front-end as the NodeJS file is already complete.

4 Front-end Centralized Web 2.0

4.1 Overview

The application front-end will be using the ReactJS in index.js file. It will have the two fields and also the pay button which will transfer the money between the accounts. This interface will make use of AXIOUS to interact with the NodeJS application we just built and our NodeJS application will be interacting with the MySQL database for POST and GET operations. The NodeJS will also interact with PayPal SDK for acting as a payment gateway for funds transfer.
The participant's information will be sent from DB to the PayPal where they will login and transfer the funds. AXIOUS will be handling the API calls and managing the requests from browser to the NodeJS application. We'll also add the admin functionality, where manager will be responsible for handling the and paying the winner.

4.2 Setting up the ReactJS

We'll need the React CDN to help us render the information on screen, which can be taken from Reactjs site. The code needs to be placed in header of the file for it to make use of React CDN. The script looks like:

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.develop-
ment.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.develop-
ment.js"></script>
```

Now we need to create the index.js file that will add the react components and, in the render function we will be using the html tags for rendering our interface. We also need to add the babel cdn for babel-standalone inclusion as follows:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
standalone/6.26.0/babel.min.js"></script>
```

We also need to tell in our index.html file that we want to inject our index.js file into this file which can be done by adding the following code to it:

```
<script type=”text/babel” src=”index.js”> </script>
```

We also need to tell where to bring this information and therefore, in the body tag of index.html we add the div for the ReactJS to render our application.

4.3 Adding Fields to View

We need to take input from the user therefore, we need to add the fields which will take the amount and email from the participant and then a participate button that will take the information and store it in the MySQL database.
To serve the purpose we need a form with fields and a button that will send the information from the form. Here we will use the constructor to initialize the view therefore, we need to create a constructor inside our index.js file. The constructor will look like:

```
  constructor(props){
      super(props);
      this.state = {
        'total_amount' : 1000,
      }
    }
```

And to show the value in placeholder for amount we’ll write the following code:

```
<p> Total Lottery amount is {this.state.total_amount}</p>
```

4.4 POST with AXIOS to MySQL database

Now that we have the form settled, we can use it to take the information and send it to database. Previously, it was done using the POSTMAN since we didn't have the interface, but now we want the values from the users to flow to the system for which we'll be using AXIOS.

We'll use the on change event in html to make sure when the user enter the information, the event is triggered. The script will look like:

```
<input placeholder=”amount” value = {this.state.amount} onChange = {event=>
this.setstate({amount : event.target.value})} />
```

Same for the email field. After that we need to add the AXIOS library which is:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.18.0/ax-
ios.js"></script>
```

And we need to add it to index.html just after the babel CDN. The AXIOS code POST operation will look like:

```
onSubmit = async (event) =>{
    event.preventDefault();
    const response = await axios.post('/post_info',{
      amount : this.state.amount,
      email : this.state.email
    })
    window.location.href = response.data;
```

As we can see in the following figure, the data taken from the field has been posted to the MySQL DB.

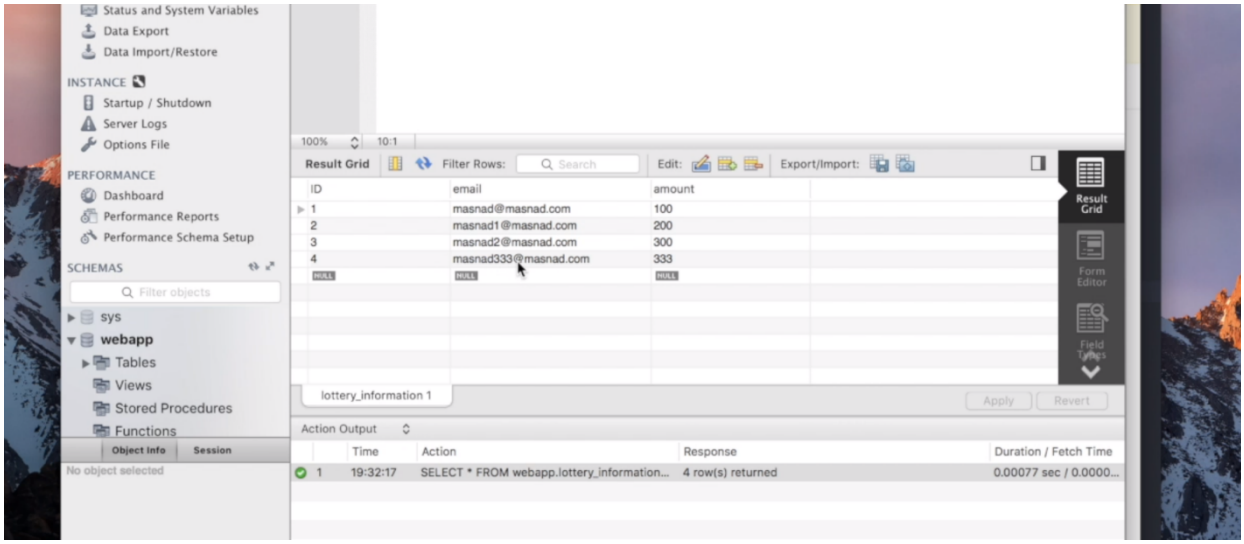

Figure 5 MySQL data insertion

4.5 Fetching information from MySQL database

We'll be using the get_total_amount function we have written in our server.js file Again, by using the ComponentDidMount function, we are getting the information using the server.js which will look like the following:

```
async componentDidMount(){
    const result = await axios.get('/get_total_amount');
    this.setState({total_amount : result.data["0"].total_amount});
  }
```

We also need the ReactJS to set the value of total_amount variable with this amount. Therefore, we'll simply add the setState for total_amount.

4.6 Setting up PayPal

It is our payment gateway for funding the application and therefore, we need PayPal to send the amount to our manager, for which we will need a manager account. We'll start from the PayPal developer. For this we will need PayPal access. We'll be creating multiple accounts in PayPal's developer section. One account for manager and other accounts for participants.

To set the paypal with our App, we will need to create the app credentials. We'll use the REST API and will create the app account using the manager account as sandbox developer. This will yield the client ID and a Secret that will be used for connecting it to our NodeJS application. The newly created lottery_manager account can be tested in PayPal's sandbox.

4.7 Installing PayPal SDK

We need to install the PayPal SDK into our NodeJS application. To do that we need PayPal SDK for NodeJS from their GitHub repository.

We insert the paypal SDK by using the following statement

```
var paypal = require('paypal-rest-sdk');
```

We'll require this paypal_rest_sdk in our NodeJS app which is server.js
then we'll use the configuration parameters which will use the client id and client secret like this:

```
paypal.configure({
  'mode': 'sandbox', //sandbox or live
  'client_id': 'EBWKjlELKMYqRNQ6sYvFo64FtaRLRR5BdHEESmha49TM',
  'client_secret': 'EO422dn3gQLgDbuwqTjzrFgFtaRLRR5BdHEESmha49TM'
});
```

4.8 Creating the payment with PayPal

To create a payment, we'll be using the PayPal's JSON method. We'll need the following code in our server.js file:

```
var create_payment_json = {
    "intent": "sale",
    "payer": {
        "payment_method": "paypal"
    },
    "redirect_urls": {
        "return_url": "http://return.url",
        "cancel_url": "http://cancel.url"
    },
    "transactions": [{
```

```
        "item_list": {
            "items": [{
                "name": "item",
                "sku": "item",
                "price": "1.00",
                "currency": "USD",
                "quantity": 1
            }]
        },
        "amount": {
            "currency": "USD",
            "total": "1.00"
        },
        "description": "This is the payment description."
    }]
};
paypal.payment.create(create_payment_json, function (error, payment) {
    if (error) {
        throw error;
    } else {
        console.log("Create Payment Response");
        console.log(payment);
    }
});
```

We will have to configure this code to work with our app. Therefore, we start by changing the return_url and cancel_url for our localhost links. After that we change the price to our amount variable, name will be Lottery as this is a lottery app and sku will be set to funding. Currency and Quantity will remain the same. The total will be changed to amount variable.

We also need to tell the JSON file the person we are paying. Therefore, we will add the section for ‘Payee” by using the following code:

```
‘payee’ : {
            ‘email’ : ‘Lottery_manager@lotteryapp.com’
},
“description”: “Lottery purchase”
```

Other than that, we also need to modify the create payment method so that it can loop all the users through the PayPal page for depositing money. For this we have made the following code:

```
For(var i=-o; i<payment.links.lengthl i++)
{
If(payment.links[i].rel ==’approval_url’]{
Return res.send(payment.links[i].href};
}
```

4.9 User check and Funding the App

To test the system, we'll need the other accounts for participants. Consequently, we will create one personal account in PayPal's Developer section. For funding the app, we need to redirect the user to success return link. The code for doing it is following:

```
app.get('/success', async (req,res)=>{
  const payerId = req.query.PayerID;
  const paymentId = req.query.paymentId;
  var execute_payment_json = {
    "payer_id": payerId,
    "transactions": [{
        "amount": {
            "currency": "USD",
            "total": req.session.paypal_amount
        }
    }]
  };
```

We also need to put a check on it so that if there is any error in the payment processing, we'll know by throwing the error message to console.
Running the code, we have tested and transferred the $100 for lottery using PayPal and by checking the balance in PayPal sandbox.

4.10 Storing sessions

We need to store the amount that the participant is entering and need to store it, so that after redirect from the PayPal account, we know how much money is in the system. For this we need express session CDN that is compatible with express. We'll install it using console using the following line of code:

```
npm install express-session –save
```

after the session is installed. We require it in our server.js file. Using the following line of code:

```
const session = require('express-session');
the code for session will look like:
        app.use(session(
          {secret: 'my web app',
```

```
        cookie :{maxAge: 60000}
        }
      ));
```

After we have the amount, we'll be initializing our database and storing the amount there. Which will be done by writing the following statement:

```
Req.session.paypal_amount = amount;
```

4.11 Picking the Winner Function

Now that we have the participants in the system, we will need the admin/manager to have the functionality to pick the winner. We'll write the code right after get total amount. We need the get function for pick winner. For this we need to payment amount to be taken from the system using the get function., but this time we'll be taking the amount from the PayPal session for which the code is:

```
app.get('/pick_winner', async (req,res)=>{
  var result = await get_total_amount();
  var total_amount = result[0].total_amount;
  req.session.paypal_amount = total_amount;}
```

Also, we will be using the same PayPal payment JSON code we have done before with the alteration for putting the total amount we just settled above.

4.12 Picking the Participant

To do this we'll be writing a function in server_db.js for a query to fetch the data of all the participants in the system. We will parse the results and will also check if there is an error with the fetching data using the following function:

```
get_list_of_participants = (data) => new Promise((resolve,reject)=>{
  db.query('select email from lottery_information',null,function(err,re-
sults,fields){
    if(err){
      reject('Could not fetch list of participants');
    }
    resolve(results);
  });
});
```

After the query is ready in the server_db.js, we can start using it in server.js file which will get the value of participants in the following code:

```
 var list_of_participants = await get_list_of_participants();
  list_of_participants = JSON.parse(JSON.stringify(list_of_partici
pants));
  var email_array = [];
  list_of_participants.forEach(function(element){
    email_array.push(element.email);
  });
```

We also need the emails of as so we have included the code for it as well. We are taking the information in array that contains all the emails. To check it we push the results to the console to verify.

We need this list of arrays to be parsed to the variable so we use the JSON parse stringify function to achieve it. We also need to store these emails to the email array element by using the following code:

```
Email_array.push{element.email};
```

Now we need to select the random winner by using the random function in JS. To do that we use the following code:

```
Var winner= email_array{Math.floor(Math.random()* email_array.length)};
```

4.13 Winner Session

There is a last touch to the winner function, which we are doing in this section. Previously we were only getting the email and pushing the randomly selected winner to the console. Now, we need to create the winner session. We go to our success URL and redirect the user, but we need to check and delete all the users so that the new session can start. For that we have to check that the winner is already picked, and therefore, we need to see the condition to assist us. Following is the code:

```
if(req.session.winner_picked){
    var deleted = await delete_users();}
```

To make this code work, we add the winner picked and set it to true when the winner was picked. After deletion of the users, we will set this to false so that it can work over and over for checking.

4.14 Deleting the participants

This function will be used to delete the users after the winner is picked. We have already settled the placement for deleting participants, now we need to add the query to our server_db.js. The function and query in the server_db.js should look like the following:

```
delete_users = (data) => new Promise((resolve,reject)=>{
 db.query('delete from lottery_information where ID > 0', null,
           function(err,results,fields){
   if(err){
     reject("Could not delete all users");
   }
   resolve("success on  deleting all users");
 });
 });
```

It must be noted that at every function, we place the error check and resolve to send the success message.

4.15 Makeover with bootstrap

To start with, bootstrap is a framework that can be used for creating the responsive and simple websites. To include it into our front-end for makeover, we add the css for it to the header of our index.html and the JS script right after the footer. The CSS CDN is as follows:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/boot-
strap/4.1.3/css/bootstrap.min.css" integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" cros-
sorigin="anonymous">
```

And the JS script to include after the body tag is:

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integ-
rity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/pop-
per.min.js" integrity="sha384-
ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49" cros-
sorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/boot-
strap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxy-
MiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></script>
```

We also add the index.css file to add our own CSS modifications to the site. After the modifications the interface will look like the following:

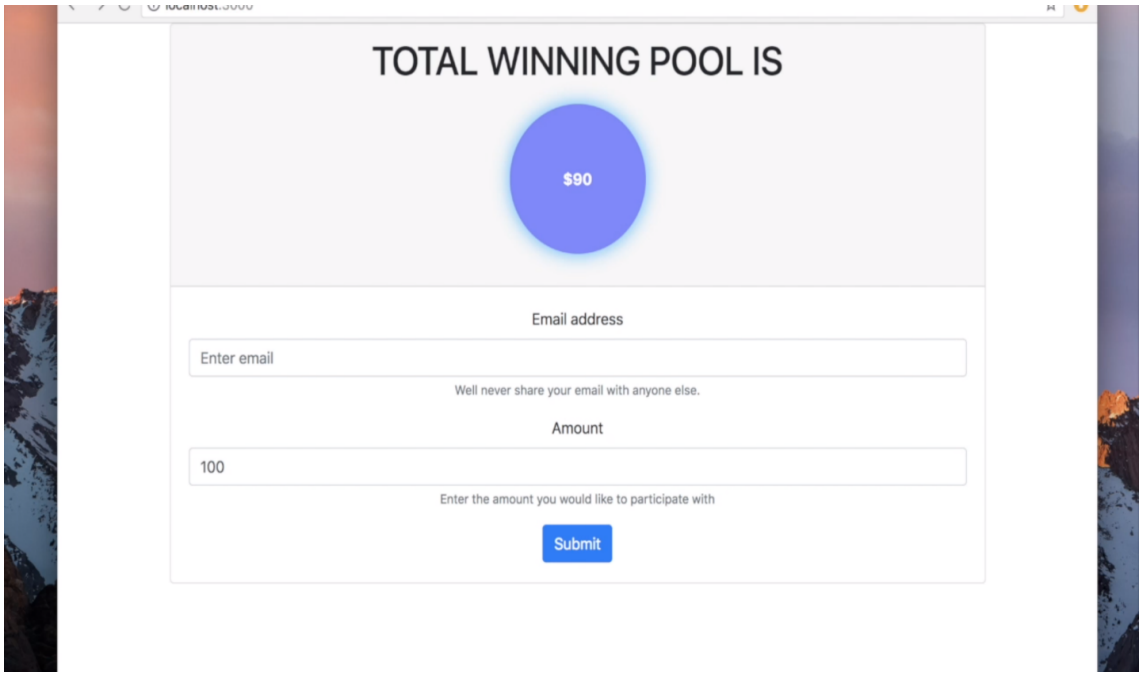

Figure 6 Centralized Lottery application Front-end/Interface

5 App to Dapp – Decentralization and why do it?

The major issues that lie with centralized servers is that they are vulnerable to hackers because the information storage (DB) stays in one place. For payments processing, we have to use third party gateways and vendors, therefore the dependency for funds transfer relays with third party vendor. The system information like the information about the exact figure of money in the system is unknown. Fees for transaction with third party payment processors apply; this applies for both the sender and receiver's end. The amount in manager's account can be checked if you have the access for email and password. Lengthy codes with unnecessary effort required. The information of the participants is gathered in one centralized database.

Web 3.0 provides solid solutions to the problems mentioned above. For a decentralized system like Ethereum to be hacked, intruder needs to have 51% control of the system, which is difficult to attain because if one or several nodes are attacked, it won't affect the overall system. Third part payments processors are not required for transactions. The payment transfer occurs directly between the user and the contract. Public information can be shared over the blockchain if contract address is open on .io link, which in case of Ethereum is etherscan.io. The funds transfer charges could be as low as 0.01 cents because there is no third-party vendor to hold them or let the application down. In etherscan.io, it can be checked for the fees to be taken by manager. Ethereum itself is a payment gateway and a database and therefore less code is required to write application. Participants information is unknown to DAPP and therefore, the privacy is guaranteed. [8]

5.1 Decentralized Application

The application will be using the multiple technologies such as ReactJs, NextJs and the Ethereum network. We will be first using Remix a JS virtual environment in browser and contains the Ethereum client. Codes and test accounts will be used for testing the new app. It'll be used for lottery funding. Rinkeby ethereum

test-net will be connecting the accounts from rinkeby to the remix for testing the working with meta mask. When the process of building the application with NodeJS will start, we will write the Ethereum contract in REMIX browser to the rinkeby network via NodeJS. Furthermore, the contract can be deployed using REMIX, but we will be deploying the contract using our own NodeJS application. The basic block of an Ethereum contract block is made up using the below mentioned list.

1. Nonce: Transactions count (in and out of contract)
2. Address: Public address for example, bank IBAN (24-character address)
3. Balance: How much ether contract stores (Amount of ether stored in account)
4. Code: Script used to carry out certain tasks on contract (programming language)
5. State: Variables declared at the start of contract

The ethereum network uses gas as a fee that is paid to use the network. The gas fee is paid when someone tries to use the payable function of our contract. The miners are there to validate the contracts and add the block to blockchains, whereas considering an example if one of the 4 projects is paying higher GAS, then it will be the first to get attached while others will stay in-line until they are validated.

5.2 Solidity Contract Types

The difference between the JS declarations and solidity declarations is that, in solidity we have to declare the type of variable first and then set its scope for example public, so that it is viewable to everyone. As the contract service is used and each time a transaction is carried out, GAS will be charged, which means we can run the loops as Ethereum simply does not allow that.

The Ethereum smart contract as stated in the previous chapter will be making use of the REMIX browser or Ethereum Remix editor, which is an online environment and tool for deploying and executing the Ethereum network.

The code below is an example of a simple Smart contract.

```
Pragma solidity ^0.4.0
Contract Lottery{
String public message;
Constructor () public {
            Message = “Hello Decentralized world”;
}
};
```

In web 2.0 the manager was our PayPal gateway who was liable to carry out the task of manager, or in other words it was the manager, however, in web 3.0 the contract itself will chose the manager and to know the manager, the contract has to have the manager’s public address instead of email or password. The script in remix can hold the value or address of the manager’s account. The address will look like the following:

```
0xca35b7d915458ef540ade6068dfe2f44e8fa733c
```

To make the contract realize who the manager is, we make an address public variable and following that we put the variable in constructor so that when the contract is initialized, it automatically gets to take the manager’s address. The system takes the manager’s address using the following code in constructor:

```
Manager= msg.sender;
```

Also, there are different types of solidity such as Booleans, integers, fixed point numbers, address, members of addresses, arrays, and address literals.

The manager’s address is already in the contract, but we also need the addresses of participants and therefore we need to store them somewhere.

To get started with it, we need to create an array that will contain the addresses of all the participants. Therefore, for the participants to enter in to the game, we create a function that will charge them the gas fee to allow them and participate in it.

The following code will push the participants into the list just like the JavaScript:

```
Participants.push(msg.sender);
```

Where it must be noted that the msg.sender will change and it will not be a manager, because the constructor is making use of it for every new instance of participant.

Payable is a part of the function in contract that uses the solidity when the function is meant to transfer the ether, this is the way the payable is used for transferring of ether and does not necessarily point towards Gas.

Now, we need the system to require at least 0.01 ether to allow the participant into system. To achieve this, we use the require statement in the function enterlottery(). The inclusion of following statement into the function will ensure that each participant must have 0.01 ether to enter. We also need to tell the contract that it is a whole ether. Therefore, the code will look like the following:

```
Function enterlottery() public payable {
Require(msg.sender > 0.01 other);
Participants.psuh(msg.sender);
}
```

In case of error, the program remix tells us to debug the code and does not tell anything about the line at which the error is occurring. The work is near completion by this stage, but we still need to add the functionality to pick a winner.

There are four major parts for picking the winner which are:

1. Manager calls the pick winner function
2. Randomly select the winner
3. Transfer the contract balance to winner's account
4. Empty the address array

To achieve the first, we use the following statement:

```
Require(msg.sender == manager)
```

We want the system to check if the current msg sender is the manager.

It is very hard in remix to create the random number, due to the nature of it. Therefore, to achieve that we use another approach. We create the following function:

```
Function random() private view returns(uint256)
{
Return uint(Keccak256(block.difficulty, now, participants));
}
```

It must be noted that this is a private function as we do not want to it to be shown to public and therefore, it will be called by pickWinner public function. The function will return the random number between the range of number of participants in the game and in pickWinner function, we call it by using the following statement:

```
Uint index = random() % participants.length;
```

i.e. we push the number to random function, then it returns an uint type 256 length number and take a modulus of it by the number of participants to select the random winner.

Now that we have the function to check manager and get the random number for selecting the winner, we can transfer the balance to that particular participant by using the following code:

```
Participants[index].transfer(this.balance);
```

It follows the footsteps of web 2.0 application of our implementation i.e. we clear the array once the winner has balance transferred. In remix, we can do this by using the following code line in pickWinner() function:

```
Participants = new address[](0);
```

The above statement will initialize the new address array with 0 elements populated in it.

5.3 Deploying and testing the smart contract

We will be using the JS VM in remix. The code needs to be saved before deploying. We start by entering the three participants in the lottery game. Then we move towards the pickWinner function to test the functioning of application.

After that we select the manager's account in remix and click the pickWinner function and we see that the second participant in our network gets the ether balance from the contract based on random selection.

Metamask is a chrome extension that gets us into the Ethereum network. It can be downloaded and added to the chrome extensions directly from the browser. After installation of MetaMask, we get back to the remix editor. After we accept the license agreement and EULAs for the MetaMask extension, we create an account that gives us the 'seed words. We will need these 12 words for connecting our NodeJS app to the Ethereum network. As we are not working with the original Ethereum coins, we need to switch to Rinkeby network to continue working with the fake ethers for building and testing the application.

We create the Manager's account, Participant 1 and Participant 2 accounts.
The inclusion of MetaMask also pushes the Web 3.0 library which we will use in remix editor by selecting the environment as 'Injected Web 3.0'. We can also check this in the console of our Google chrome browser.

5.4 Getting Ether from Rinkeby

We go to the faucet.rinkeby.io site. Then it tells us to copy the meta mast address and post it as public on our social media account and then paste the link of that social media account into this link to get the ether. After pasting the link of our post on social media to faucet.rinkeby.io, we can click on give me ether and we select the option of 18.75 ether/3 days.

5.5 Deploying to Rinkeby TestNet

In order to deploy the Lottery application, we have built in remix, we need to make sure that we have received the ether in our MetaMask Rinkeby TestNet account, as in the previous section we connected the two and requested for the 18.75 ether from faucet.rinkeby.io.
We check to verify that we have received the ether and afterwards, we go the remix editor and select the Injected Web 3.0 environment to connect our app to MetaMask and Rinkeby.
We click deploy and the MetaMask asks us for confirmation of transaction. After execution, it provides us with the etherscan link that can be used to verify that the block is being created. The TX status will turn to success after its been verified and the address can be used to point to the manager's ether account.

6 Back-end Web 3.0 Decentralized Application

6.1 Overview

As in the previous chapter, we developed an application in Remix editor that was deployed into Rinkeby TestNet, we can observe that the process is simply difficult for the lay person and therefore, we need to build a decentralized Web 3.0 application in a way so that even a lay person can interact with it. Therefore, we are using the NodeJS and ReactJS for the purpose and we want our NodeJS app to deploy the contract instead of using the remix editor.

6.2 Architecture of Back-end Web 3.0

We will use the npm Sol library to compile our Lottery application we just built which will produce the Bytecode and ABI after compilation. We will be using the Bytecode for deploying the Lottery.sol app to the Rinkeby TestNet.
In this process, we'll build two files, the compile.js that will execute the npm sol statement and feed the Lottery.sol file for compilation and the Deploy.js file that will use the bycode from previous step and deploy the Lottery.sol to Rinkeby

TestNet. Furthermore, we'll be building a NodeJS application that will take the ABI and we will install the Web 3.0 in it so that it can interact with the contract deployed in Rinkeby TestNet. The Web 3.0 will be later replaced by Injected Web 3.0.

6.3 Backbone of decentralized Application

To start with and keep things organized, we need to create a few files and folders. We create the folder named 'decentralized_lottery_app' in which we create the files required for Web 3.0 backend.
The folder will contain the package.json file that will have the basic information of the project so that the system and github can be used to commit changes.

Inside that we create contracts folder in which we create the Lottery.sol file.
Lottery.sol file will contain the complete Remix editor code for contract that we built and deployed before.
After that we'll need all the Node modules required for supporting the Back-end. It can all be done by single line of code in terminal:

```
Npm install –save react react-dom next solc truffle-hdwallet-provider web3
```

All these are the libraries and technologies we mentioned earlier to be used for build Web 3.0 app.

6.4 Setting up NextJS for pages

As this is the new project, we have to create a new folder called pages so that the NextJS can locate it, similar to that of Express that used the public folder name to locate the pages. Therefore, we create a folder named 'pages' inside the 'decentralized_lottery_app' folder. The first file will be called index.js. The code is shown in appendix 1.

The explanation of different parts of this code will be done in later sections, but it is important to note that we have used NextJS for producing the output and we use the required react and web3.0 libraries for the process. As in the previous

implementation (Web 2.0), we also push the files of the decentralized implementation to GitHub using commits in Web3.0 repository.

6.5 Compile script (Compile.js)

We create this file in the root directory. The rationale of creating the compile.js is to bring the code from Lottery.sol and compile it using the solc library.
We insert the path of the Lottery.sol in the compile.js code and we also import the solc library by using the following command

```
Const solc = require{'solc'};
```

Afterwards, we compile the code by using the following command:

```
Console.log(solc.compile(source,1).contracts{':Lottery'});
```

We send the ouput to the console, because we want to see the bytecode the from the compilation and we also want to see the ABI, which we will need
Lottery.js file will also need to be created for working with ABI, which will be used later on with NodeJS. The file will have the following code:

```
import web3 from './web3.js';

const address = '0x6d3B5ec926A87224681dDe826cd16735d4709920';

const abi = [{"constant":true,"in-
puts":[{"name":"","type":"uint256"}],"name":"participants","out-
puts":[{"name":"","type":"address"}],"payable":false,"stateMutabil-
ity":"view","type":"function"},{"constant":true,"inputs":[],"name":"man-
ager","outputs":[{"name":"","type":"address"}],"paya-
ble":false,"stateMutability":"view","type":"function"},{"con-
stant":false,"inputs":[],"name":"pickWinner","outputs":[],"paya-
ble":false,"stateMutability":"nonpayable","type":"function"},{"con-
stant":false,"inputs":[],"name":"enterLottery","outputs":[],"paya-
ble":true,"stateMutability":"payable","type":"function"},{"in-
puts":[],"payable":false,"stateMutability":"nonpayable","type":"construc-
tor"}];
export default new web3.eth.Contract(abi,address);
```

The export default statement above is used to export the code so that it can be required by other files that will carry out the process.

6.6 Deploy script (deploy.js)

This will use the truffle hd wallet provider and the seed words from MetaMask and also the address from the Rinkeby TestNet.
We start by adding and requiring the truffle by using the following command:

```
Const HDWalletProvider = require(‘truffle-hdwallet-provider’);
```

After this we add the web3 library and require the compile.js file by mentioning in the command that we need interface and bytecode. Then we use another variable to create the instance of HDWalletProvider so that we can send the 12 seed words to it.

To proceed further, we use the infura that will provide is the Ethereum network link which in this case is the Rinkeby TestNet. To get the link, we signup with the infura and create a new project named Lottery and then select the Rinkeby Test-Net in it to get its link. This link is then passed to the deploy.js so that the truffle wallet can interact with our Rinkeby TestNet. The code will look like:

```
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3');
const {interface,bytecode} = require('./compile.js');


const provider = new HDWalletProvider(
  'slogan tobacco angry capable scene senior rabbit drop camera rip-
ple income swing',
  'https://rinkeby.infura.io/v3/f8ad46d404124919926cf5d925a939a6'
);
```

6.7 Deploy the contract to Rinkeby TestNet

We have deployed the contract to Rinkeby TestNet before, but the difference here is evident this time as we are pushing the code from our NodeJS application to the Rinkeby network so that we can execute it from our own project with remix editor or external systems.
The code for deployment will go under the deploy.js file of course.
By deploying this to Rinkeby TestNet, we will also get the address of the account 0 which was indicated in the code as a manager's account. This address will be used in the blank space for address we left in Lottery.js file for ABI. Therefore, the code for deploy.js will look like:

```
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3');
const {interface,bytecode} = require('./compile.js');


const provider = new HDWalletProvider(
  'slogan tobacco angry capable scene senior rabbit drop camera ripple income
swing',
  'https://rinkeby.infura.io/v3/f8ad46d404124919926cf5d925a939a6'
);


const web3 = new Web3(provider);


const deploy = async () =>{
  const accounts = await web3.eth.getAccounts();
  console.log(accounts);
  console.log('Contract is deployed by the manager with address ', accounts[0]);
  const result = await new web3.eth.Contract(JSON.parse(interface))
  .deploy({data : '0x' + bytecode})
  .send({gas : '2000000', from : accounts[0]})
  console.log('Contract deployed to address', result.options.address);
  // 0x6d3B5ec926A87224681dDe826cd16735d4709920
 }
 deploy();
```

The commented address is the one we are also using in the Lottery.js to make the back-end complete, because the Lottery.js will now be used in the front-end ReactJS for makeover of our decentralized application.

7 Front-end of Web 3.0 Interactive Smart Contract Implementation

7.1 Index.js for ReactJS interface

To start with development of front-end, we know that the index.js is the first file that will be for front-end, as for any site the initial point is index file. We will write the react.js code in it to form the interface. The working and code structure are similar to that of Web 2.0. We insert the message and two input fields. One is the input value and the other is submitting button for participation. We are also showing the address of the manager in p tag of html and we also add the button to the system for Pick winner.

7.2 Web 3.0 installation to front-end

To do this we add the constructor to the Lottery class in our index.js file. This constructor will be used displaying the participation amount and also the manager's address. It will also check if the manager field is empty so that it can fetch the information from DB.

To get the information we need to call the async function componentdidMount(), which will fetch the information for displaying on front. To achieve that we need to add the lottery file. The lottery will provide the web3 libraries. In the previous section in Lottery.js we forgot to import the web3 libraries and therefore we create another .js file named web3 so that we can add the installed web3.0 library to it using the following code:

```
import Web3 from 'web3';
let web3;
if(typeof window !== 'undefined' && typeof window.web3 !== 'undefined'){
// We are in the browser and metamask is running
web3 = new Web3(window.web3.currentProvider);
}else{
const provider = new Web3.providers.HttpProvider(
  'https://rinkeby.infura.io/v3/f8ad46d404124919926cf5d925a939a6'
);
web3 = new Web3(provider);
}
```

```
export default web3;
```

In the above code we have added the web3 library and we have also defined the code to automatically pick the libraries. We also need the MetaMask running. For the web3.0 library to work, we also need to provide it an address to our Rinkeby deployment that got the address using Infura.

7.3 Getting Contract Balance

The process of getting the contract balance is simple once we have the addresses, we already have the address of the manager. Now we need to get the balance of the smart contract, for which we need to pass the address of our smart contract which can be done by using the following script:

```
const total_amount = await web3.eth.getBalance(lottery.options.address);
this.setState({total_amount : total_amount})
```

It can be verified at this step that the balance is zero in current contract.

7.4 Adding Ether to Contract

To start with, we'll fund the contract using the fake ether we received from Rinkeby and we'll be using MetaMask for the purpose to connect to contract. For this we add the onsubmit tag of html which will push the amount from participants so that we can change the amount in their account and add it to our contract. We will need the async event here to check the account ether balance and also await to wait for the account to reply to our ReactJS. The code can be found in appendix 2.

All the work goes into index file. We use the web3.utils library to convert the amount to ether. The goal was to make a function that will take the amount and add to the contract. The function of enter lottery takes the amount from accounts[0] to the contract.

In the process we have encountered the errors but it must be noted that browser tells us about all the errors and we can eradicate them instantly. Till this point, the decentralized lottery application is almost complete with few bits remaining.

7.5 Pick winner function

The last peace in puzzle. Pick the winner function will randomly pick the winner from pool of participants. We start off by adding the onclick function to the html code:

```
<button onClick={this.onClick}> Pick Winner </button>
```

And using the reactjs for calling the onlick function we have made below. As usual we need the async to keep things synchronized. This function will show the message ‘please wait…’ and after that it’ll get the accounts from web3 library and use the lottery function pickWinner() in our contract to select the random participant.

```
onClick = async () =>{
  this.setState({message : "Please wait ....."})
  const accounts = await web3.eth.getAccounts();
  const winner = await lottery.methods.pickWinner().send({
    from : accounts[0]
  });
  this.setState({message : "Payment sent to winner"});

}
```

8 Conclusion

The introduction of decentralized application is meant to overcome the problems we face in Web 2.0 applications. The rationale of this thesis was to learn the transition of applications from Web 2.0 to Web 3.0 to see what kind of improvements and changes it brings to the table. The implementation of blockchain systems still has a long way to go and as we can see, each operation we are doing is performed at different nodes of the system. The development of blockchain has also pointed towards the fact that we need less coding effort. Also, the functions and methods are straightforward when compared to the centralized application where we need to add a lot more code which seems wasteful as compared to our former implementation.

References

1. Blockgeeks, Become A Bitcoin Developer: Basic 101. [online] Available at: https://blockgeeks.com/guides/bitcoin-developer/

2. Blockgeeks, Become A Bitcoin Developer: Basic 101. [online] Available at: https://blockgeeks.com/guides/bitcoin-developer/

3. International Journal of Latest Trends in Engineering and Technology, vol. 7, no. 4. 2016, Comparative analysis of angularjs and react js [online] Available at: ijltet.org/journal/148051944230.1245.pdf

4. Bootstrap. Bootstrap. [online] Available at: https://getbootstrap.com/

5. N. Tomic, "Characteristics and functioning of PayPal system", *Megatrend revija*, vol. 11, no. 2, pp. 255-270, 2014. [online] Available at: researchgate.net/publication/277887742_Characteristics_and_functioning_of_PayPal_system

6. NodeJs, Intro to nodejs
   [online] Available at: w3school.com/nodejs/nodejs_intro.as

7. M. Greaves and P. Mika, "Semantic Web and Web 2.0", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 1-3, 2008.

8. Kambria Network, Why We Need A Decentralized Internet. [online] Available at:https://medium.com/kambria-network/why-we-need-a-decentralized-internet-f46151d376a2

Appendix 1

```
import React ,{Component} from 'react';
import lottery from '../lottery';
import web3 from '../web3';


class Lottery extends Component{


  constructor(props){
    super(props);
    this.state = {
      manager : '',
      participate_amount : '0.01',
      message : '',
      total_amount : ''
    }
  }
  async componentDidMount(){
    // get the public address of the managers
    const manager  = await lottery.methods.manager().call();
    console.log(manager);
    this.setState({manager : manager});
    const total_amount = await web3.eth.getBalance(lottery.op-
tions.address);
    this.setState({total_amount : total_amount})
  }


  onSubmit = async (event) => {
    event.preventDefault();
    const accounts = await web3.eth.getAccounts();
    if(this.state.participate_amount < 0.01){
      return alert("Amount is less than 0.01 pleas eneter a bigger
amount");
    }
    this.setState({message : 'Please wait .....'});
    const enter_lotery = await lottery.methods.enterLot-
tery().send({
      from : accounts[0],
      value : web3.utils.toWei(this.state.participate_amount,
'ether')
    });
    this.setState({message: "You have been added to the lot-
tery!"});
  }
  onClick = async () =>{
    this.setState({message : "Please wait ....."})
    const accounts = await web3.eth.getAccounts();
    const winner = await lottery.methods.pickWinner().send({
      from : accounts[0]
    });
    this.setState({message : "Payment sent to winner"});
  }
  render(){
    return (
       <div>
          <h1> Total lottery pool is {web3.utils.from-
Wei(this.state.total_amount,'ether')} </h1>
          <form onSubmit={this.onSubmit}>
            <input value={this.state.participate_amount} onChange =
{event => this.setState({
               participate_amount : event.target.value
```

```
              })}
              />
            <button type="submit">Participate </button>
          </form>
          <p> {this.state.message} </p>
          <hr /> <br /> <hr />
          <p> The manager of the lottery decentralized app is
{this.state.manager}</p>
          <button onClick={this.onClick}> Pick Winner </button>
       </div>
    )
  }
}
export default Lottery;
```

Appendix 2

```
import React ,{Component} from 'react';
import lottery from '../lottery';
import web3 from '../web3';
class Lottery extends Component{
  constructor(props){
    super(props);
    this.state = {
      manager : '',
      participate_amount : '0.01',
      message : '',
      total_amount : ''
    }
  }
  async componentDidMount(){
    // get the public address of the managers
    const manager  = await lottery.methods.manager().call();
    console.log(manager);
    this.setState({manager : manager});
    const total_amount = await web3.eth.getBalance(lottery.op-
tions.address);
    this.setState({total_amount : total_amount})
  }
  onSubmit = async (event) => {
    event.preventDefault();
    const accounts = await web3.eth.getAccounts();
    if(this.state.participate_amount < 0.01){
      return alert("Amount is less than 0.01 pleas eneter a bigger
amount");
    }
    this.setState({message : 'Please wait .....'});
    const enter_lotery = await lottery.methods.enterLottery().send({
      from : accounts[0],
      value : web3.utils.toWei(this.state.participate_amount,
'ether')
    });
    this.setState({message: "You have been added to the lottery!"});
  }
  render(){
    return (
       <div>
          <h1> Total lottery pool is {web3.utils.from-
Wei(this.state.total_amount,'ether')} </h1>
          <form onSubmit={this.onSubmit}>
            <input value={this.state.participate_amount} onChange =
{event => this.setState({
               participate_amount : event.target.value
               })}
               />
            <button type="submit">Participate </button>
          </form>
          <p> {this.state.message} </p>
          <hr /> <br /> <hr />
          <p> The manager of the lottery decentralized app is
{this.state.manager}</p>
       </div>
    )
  }
}
export default Lottery;
```