CHAPTER 4
RESULT, DISCUSSION AND FINDING

This chapter will focus more on the finding from the testing phase to make sure the system work according to the development plan and satisfy all of the objective that have been stated on the previous chapter. In this chapter, the results from the testing phase and the problem faced during the development phase will be defined.

4.1 System Development and Testing

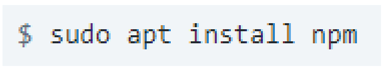
System development and testing is a phase where the development of the system will be documented in details, from the beginning of the process until the end of the development process. The topics that will be discusses in this phase are the installation of the dependencies, the programming language used, and the problem that arise.

4.1.1 Installation of the Dependencies

Before developing the system, dependencies like Node Packet Manager (NPM), Create React App, Interplanetary File System (IPFS) and Metamask (web extension) need to be installed first. This is to avoid any missing component or system error during the development process. The installation processes that are shown below are for the Linux OS because it is easier to setup and install the dependencies in Linux than Windows. Not only that, Linux also supports a wide variety of software and some that are already built-in.

a) Node Packet Manager (NPM)

NPM is package managers that were used for the installation of other dependencies, managing JavaScript programming Language and running the development server for the project. The installation commands for the NPM are shown in the figure 4.1.

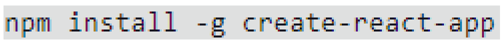


```
$ sudo apt install npm
```

Figure 4. 1 NPM Installation Command

b) Create React App

Create React App are used for the development of the front end as it is used for initiating a new React App by providing the basic HTML,JS and CSS bundle and the component needed to run the development server for the web. The installation for the Create React App is shown in figure 4.2.

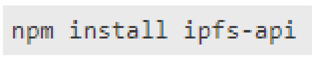


```
npm install -g create-react-app
```

Figure 4.2 Create-React-App Installation Command

c) InterPlanetary File System API (ipfs-api)

The used of ipfs-api in this project is to store the uploaded files in the IPFS and get the hash for the files without running the IPFS server or used any database to store the files. The installation command for the ipfs-api is as shown in the figure 4.3.

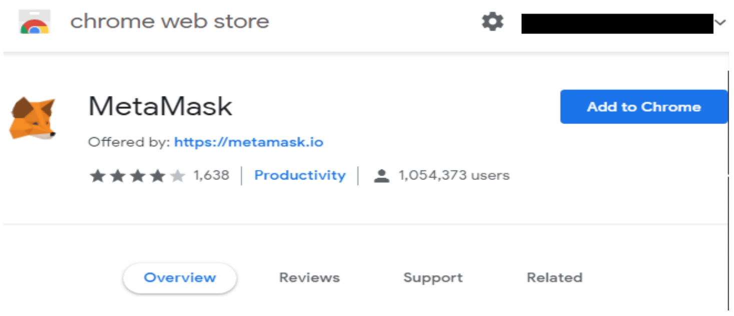


```
npm install ipfs-api
```

Figure 4.3 IPFS-API Installation Command

d) Metamask

Metamask is a web extension that can be downloaded by any browser and allows the browser to interact with distributed web and Ethereum decentralized application. Not only that, metamask also allows the user to choose the type of network we want to connect such as Etherem main network, Rinkeby test network, Ropsten test network, Kovan test network or local blockchai. The metamask also provide us with a “Seed Words” where we can use it to restore an account. Figures 4.4 and 4.5 below show the metamask extension and the “Seed Words” generated after the registration process.

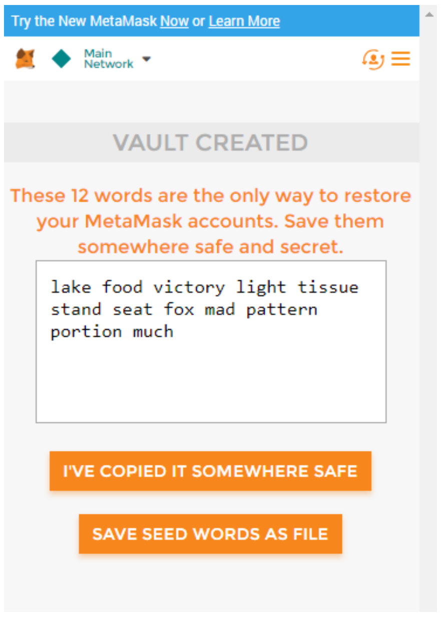

chrome web store [redacted]

MetaMask

Offered by: https://metamask.io

★★★★☆ 1,638 | Productivity | 1,054,373 users

Figure 4.4 Metamask Installation


Try the New MetaMask Now or Learn More

Main Network

VAULT CREATED

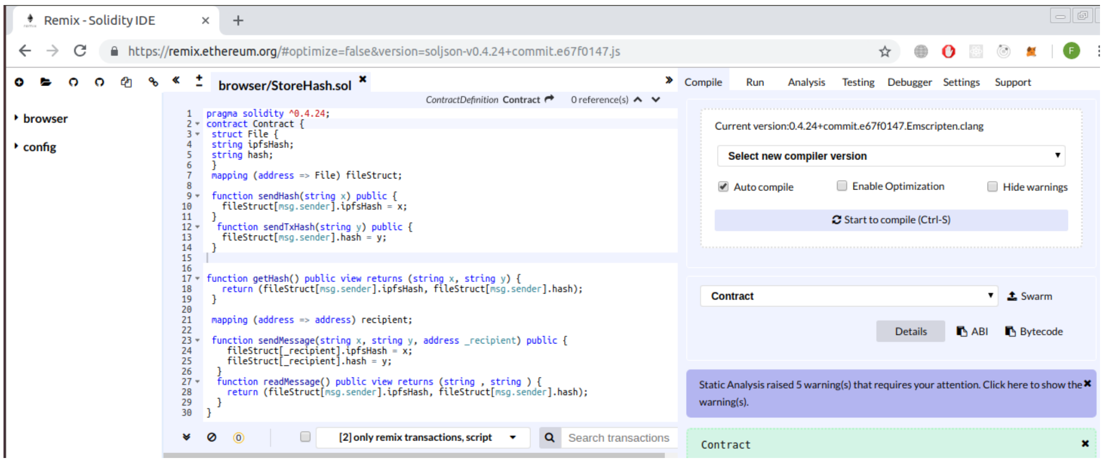
These 12 words are the only way to restore your MetaMask accounts. Save them somewhere safe and secret.

```
lake food victory light tissue
stand seat fox mad pattern
portion much
```

Figure 4.5 Metamask Seed Phrase

4.1.2 Writing the Smart Contract for Storing File Hash, Send Message and Read Message

The process of writing the smart contract was done in Remix Solidity Editor at https://remix.ethereum.org. Remix provided the platform for writing and deployment of the smart contract in various environments such as JavaScript VM, Injected Web3 and Web3 Provider. Figures 4.6 shown below are the interface of Remix Solidity IDE interface and the coding for the smart contract that is written in solidity programming language.



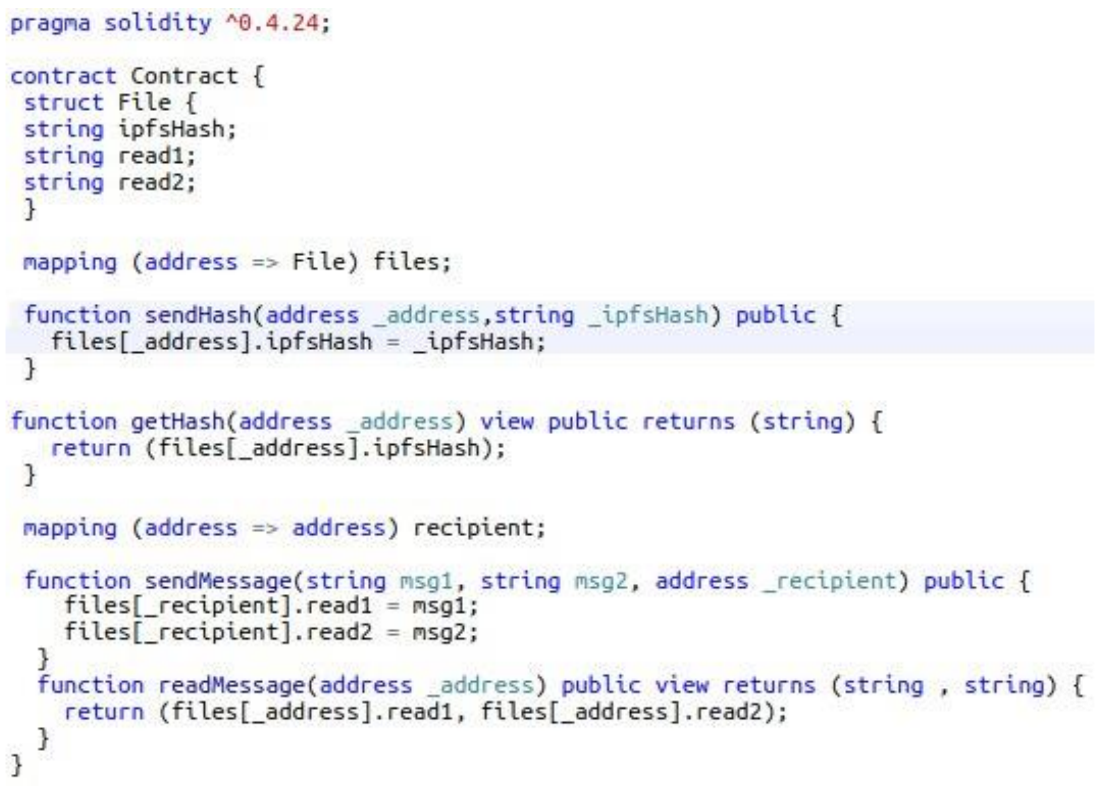
```
pragma solidity ^0.4.24;
contract Contract {
 struct File {
 string ipfsHash;
 string hash;
 }
 mapping (address => File) fileStruct;

 function sendHash(string x) public {
   fileStruct[msg.sender].ipfsHash = x;
 }
  function sendTxHash(string y) public {
   fileStruct[msg.sender].hash = y;
 }


function getHash() public view returns (string x, string y) {
   return (fileStruct[msg.sender].ipfsHash, fileStruct[msg.sender].hash);
 }

 mapping (address => address) recipient;

 function sendMessage(string x, string y, address _recipient) public {
    fileStruct[_recipient].ipfsHash = x;
    fileStruct[_recipient].hash = y;
  }
  function readMessage() public view returns (string , string ) {
    return (fileStruct[msg.sender].ipfsHash, fileStruct[msg.sender].hash);
  }
}
```

Current version:0.4.24+commit.e67f0147.Emscripten.clang

Static Analysis raised 5 warning(s) that requires your attention. Click here to show the warning(s).

Figure 4.6 Remix IDE Interface

On the right side of the figure 4.6 is where it shows the compiler version and produced the ABI, Bytecode of the smart contract and other information like the smart contract address which is essential for the next phase of the development.



```
pragma solidity ^0.4.24;

contract Contract {
 struct File {
 string ipfsHash;
 string read1;
 string read2;
 }

 mapping (address => File) files;

 function sendHash(address _address,string _ipfsHash) public {
   files[_address].ipfsHash = _ipfsHash;
 }

function getHash(address _address) view public returns (string) {
   return (files[_address].ipfsHash);
 }

 mapping (address => address) recipient;

 function sendMessage(string msg1, string msg2, address _recipient) public {
    files[_recipient].read1 = msg1;
    files[_recipient].read2 = msg2;
  }
  function readMessage(address _address) public view returns (string , string) {
    return (files[_address].read1, files[_address].read2);
  }
}
```

Figure 4.7 Source Code of the Smart Contract

The figure 4.7 shown above is the source code of the smart contract that were used to serve the function such as storing file hash, return the file hash, sending message to other user and receive message from the sender. Starting from the top, the “pragma solidity ^0.4.24” were defined to specify the type of compiler that are going to be used to compile the source code. Each new contract must be prepended with “contract” followed with the name of the contract with the first letter capitalized, followed by open/close curly brackets to contain the logic.

In this project, the name of the contract will be call “Contract”. After that, all variables are group to a custom defined type called File using struct, map the struct File to an address and called it as “files”. This will allow the system to look up for specific file with their Ethereum address. Next, the sendHash function are used to get the file hash that were passed to the smart contract, map it to the metamask account address to prevent other address from getting the file hash and stored it inside the smart contract. While the getHash function were used to simply return the stored file hash according to the metamask account address of the user.

For the sending message or share, file hash function, first the address of the recipient need to be mapped with the address input by the user instead of the current user. This is to prevent other addresses from reading the message and only the input address can read the message. In the sendMessage function, it will take the file hash, the transaction hash and the recipient address to be stored inside the smart contract. Lastly, the readMessage function will get the current metamask address and return the file hash and transaction hash that were stored for that particular address if there are any.

4.1.3 Deploying Smart Contract on Rinkeby (Ethereum Test Net)

After the completion of writing the smart contract, the next step is the deployment of the smart contract which can be done by clicking the run option in the top right of the figures 4.8 that was shown previously. As stated previously, Remix provided a few option for the deployment of the smart contract such as javascript vm, injected web3, which allows the smart contract to be, deploy on the ethereum testnet like rinkeby testnetwork through metamask account. Remix also can be connected to the local blockchain such as Ganache using the web3 provider option but it will chose the first address in Ganache to deployed the smart contract.

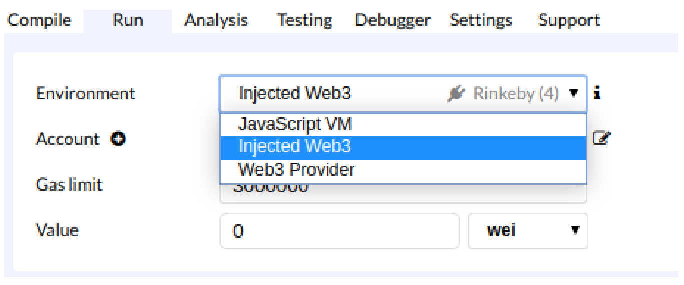
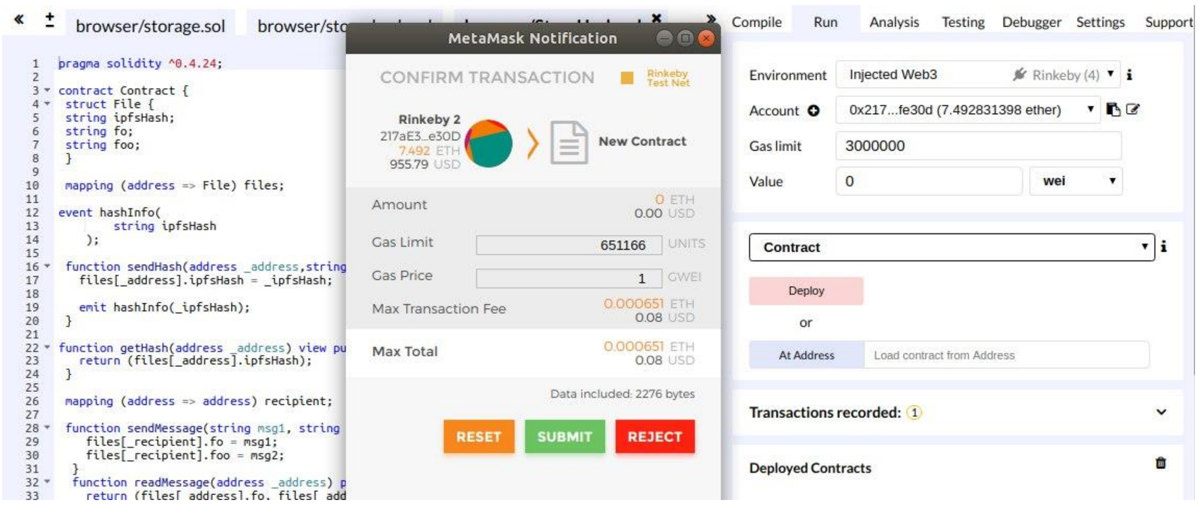


Figure 4.8 Remix IDE Environments

For the purpose of this project, the environment used are the injected web3 where the metamask account is connected to the Rinkeby (Ethereum Test Network). After choosing the environment, clicking the deploy button will pop out the transaction confirmation for deploying the smart contract as shown in figure 4.9.



```
pragma solidity ^0.4.24;

contract Contract {
 struct File {
 string ipfsHash;
 string fo;
 string foo;
 }

 mapping (address => File) files;

event hashInfo(
        string ipfsHash
    );

 function sendHash(address _address,string
   files[_address].ipfsHash = _ipfsHash;

   emit hashInfo(_ipfsHash);
 }

function getHash(address _address) view pu
   return (files[_address].ipfsHash);
 }

 mapping (address => address) recipient;

 function sendMessage(string msg1, string
    files[_recipient].fo = msg1;
    files[_recipient].foo = msg2;
  }
  function readMessage(address _address) p
    return (files[_address].fo, files[_add
```

Rinkeby 2
217aE3...e30D
7.492 ETH
955.79 USD
New Contract

Amount	0 ETH 0.00 USD
Gas Limit	651166 UNITS
Gas Price	1 GWEI
Max Transaction Fee	0.000651 ETH 0.08 USD
Max Total	0.000651 ETH 0.08 USD

Data included: 2276 bytes

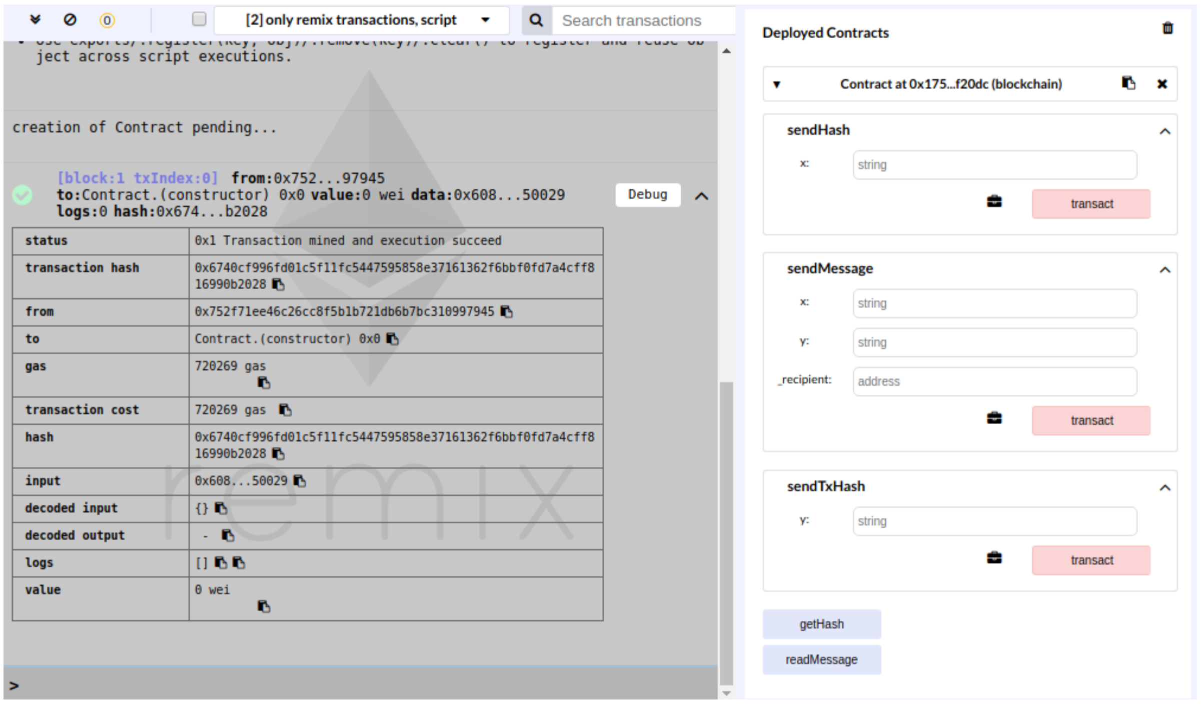
Environment	Injected Web3 — Rinkeby (4)
Account	0x217...fe30d (7.492831398 ether)
Gas limit	3000000
Value	0 wei

Contract

Transactions recorded: 1

Figure 4.9 User Confirmations for Deploying the Smart Contract on Rinkeby Test Net

After confirming the transaction to deployed the smart contract it will output the function in the smart contract where the testing can be done to identify whether the smart contract are working correctly as intended. The figure 4.10 below shows the output of the deployed smart contract.



```
ject across script executions.

creation of Contract pending...

[block:1 txIndex:0] from:0x752...97945
to:Contract.(constructor) 0x0 value:0 wei data:0x608...50029
logs:0 hash:0x674...b2028

status              0x1 Transaction mined and execution succeed
transaction hash    0x6740cf996fd01c5f11fc5447595858e37161362f6bbf0fd7a4cff8
                    16990b2028
from                0x752f71ee46c26cc8f5b1b721db6b7bc310997945
to                  Contract.(constructor) 0x0
gas                 720269 gas
transaction cost    720269 gas
hash                0x6740cf996fd01c5f11fc5447595858e37161362f6bbf0fd7a4cff8
                    16990b2028
input               0x608...50029
decoded input       {}
decoded output       -
logs                []
value               0 wei

>
```

Contract at 0x175...f20dc (blockchain)

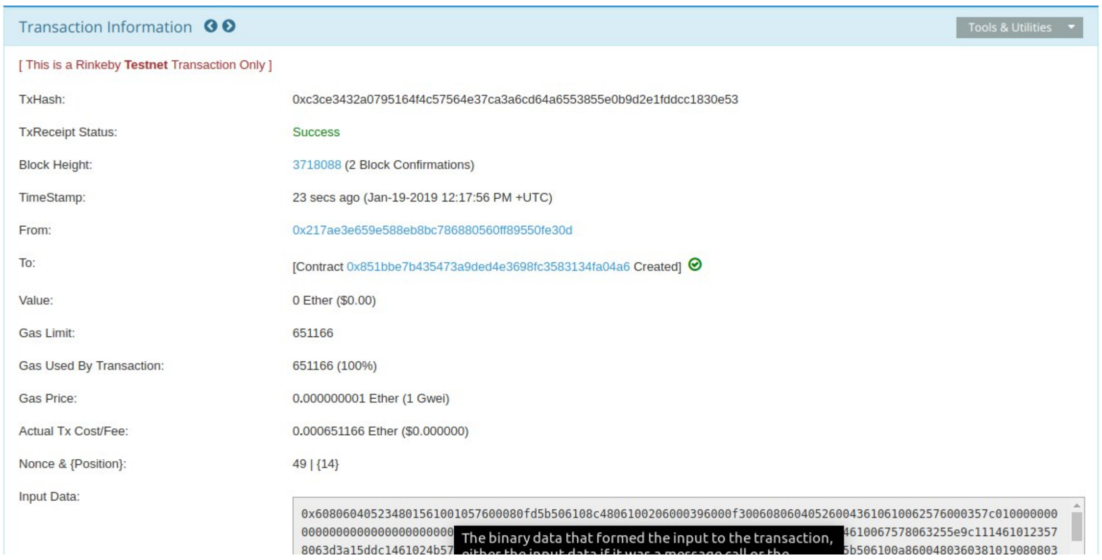
sendHash
x: string

sendMessage
x: string
y: string
_recipient: address

sendTxHash
y: string

Figure 4.10 Output of the Deployed Smart Contract on Remix IDE Terminal

The left sight of the figure 4.10 shows the transaction receipt that are generated once the contract have been successfully deployed and mined in the Rinkeby Testnet while the right side of the figures shows the output list of all the function that were set in the smart contract and also the smart contract address.



[This is a Rinkeby **Testnet** Transaction Only]

TxHash:	0xc3ce3432a0795164f4c57564e37ca3a6cd64a6553855e0b9d2e1fddcc1830e53
TxReceipt Status:	Success
Block Height:	3718088 (2 Block Confirmations)
TimeStamp:	23 secs ago (Jan-19-2019 12:17:56 PM +UTC)
From:	0x217ae3e659e588eb8bc786880560ff89550fe30d
To:	[Contract 0x851bbe7b435473a9ded4e3698fc3583134fa04a6 Created]
Value:	0 Ether ($0.00)
Gas Limit:	651166
Gas Used By Transaction:	651166 (100%)
Gas Price:	0.000000001 Ether (1 Gwei)
Actual Tx Cost/Fee:	0.000651166 Ether ($0.000000)
Nonce & {Position}:	49 | {14}
Input Data:	0x608060405234801561001057600080fd5b506108c4806100206000396000f3006080604052600436106100625760003 57c010000000 000000000000000000000 … 4610067578063255e9c111461012357 8063d3a15ddc1461024b57 … 5b506100a8600480360381019080803

The binary data that formed the input to the transaction,

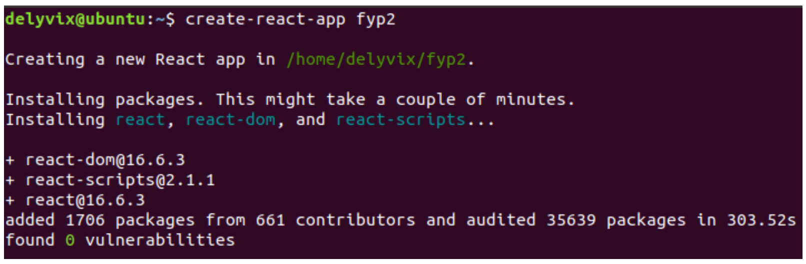
Figure 4.11 Transaction of the Deployed Smart Contract in Etherscan Website

As shown in figure 4.11 above, the successful transaction of the contract creation can be seen in Etherscan Website by providing either the smart contract address, metamask account address or the transaction hash that were generated by Remix in its terminal.

4.1.4 Development of the Frontend of the Web Application Using React, CSS, Web3 and IPFS Infura

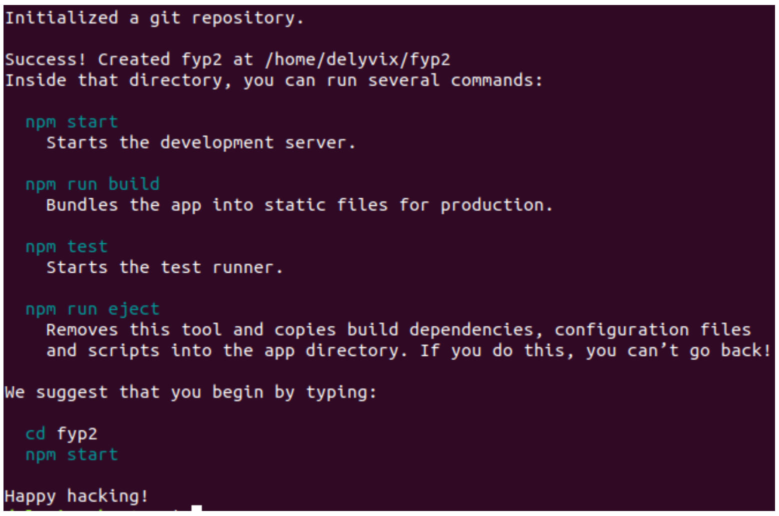
After the deployment of the smart contract, the next step is to write the graphical user interface or the frontend of the system to make it user friendly and interactive using react where it already provide us with the basic page that we need such as app.js, index.js and the css of the web application. Even though the Web3 and IPFS infura are still considered as the backend but it will be easier to understand the development of the system as both Web3 and IPFS infura are used to connect the frontend of the application to the local blockchain and IPFS node as it take the user input to be able to functionalized.

Upon completion of the installation of React App, the React App can finally be created by using the command “create-react-app <project name>”. It will take a while for the files to be downloaded, as it will install all the essential files for the React App. Once it have downloaded all the files it will produce an output shown as figures 4.12 below and the user manual to run the development server.



```
delyvix@ubuntu:~$ create-react-app fyp2

Creating a new React app in /home/delyvix/fyp2.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

+ react-dom@16.6.3
+ react-scripts@2.1.1
+ react@16.6.3
added 1706 packages from 661 contributors and audited 35639 packages in 303.52s
found 0 vulnerabilities
```

Figure 4.12 Creating New Project With React

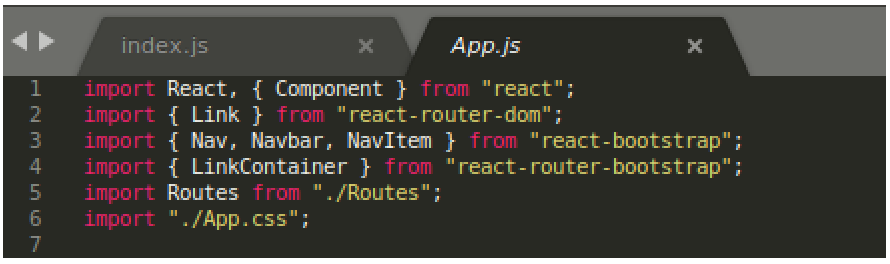


```
Initialized a git repository.

Success! Created fyp2 at /home/delyvix/fyp2
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd fyp2
  npm start

Happy hacking!
```

Figure 4.13 React App User Manual

a) App.js

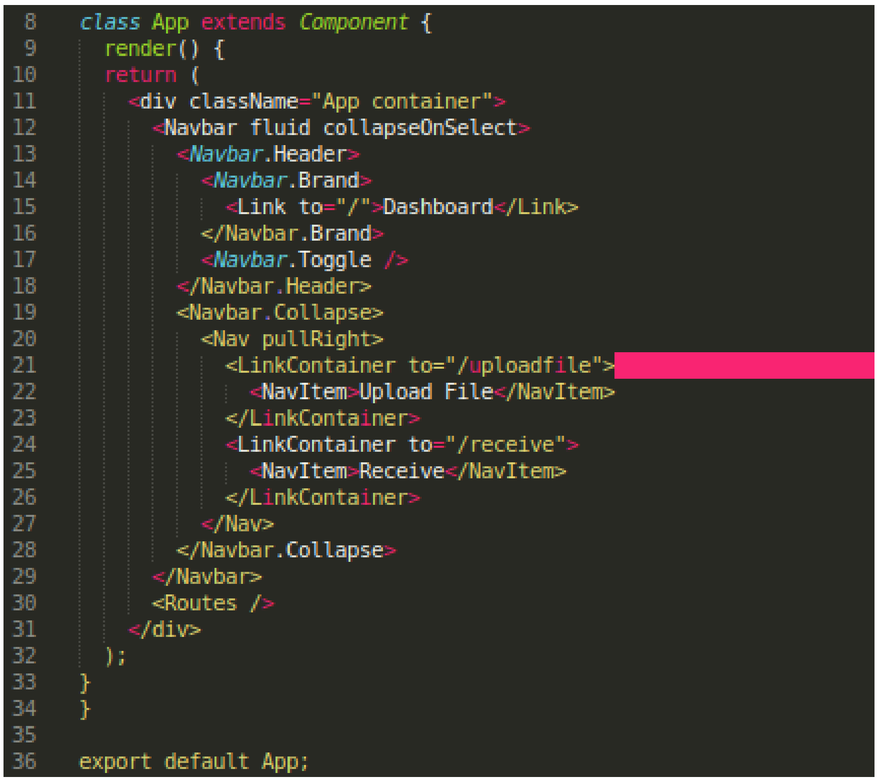
The App.js keeps all page navigation within the session of the webpage. In the figure 4.14 below, the first line of code is where we import the entire library from react bootstrap for the user interface (UI), react-router-dom from page navigation, Routes is where we store all the routes for the navigation bar and finally the css for the navigation bar.



```
import React, { Component } from "react";
import { Link } from "react-router-dom";
import { Nav, Navbar, NavItem } from "react-bootstrap";
import { LinkContainer } from "react-router-bootstrap";
import Routes from "./Routes";
import "./App.css";
```

Figure 4.14 Importing Library for App.js File

After all the libraries and related pages have been imported, the next step is to code the navigation bar for the App, which as shown in figure 4.15.



```
class App extends Component {
  render() {
  return (
    <div className="App container">
      <Navbar fluid collapseOnSelect>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">Dashboard</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
        <Navbar.Collapse>
          <Nav pullRight>
            <LinkContainer to="/uploadfile">
              <NavItem>Upload File</NavItem>
            </LinkContainer>
            <LinkContainer to="/receive">
              <NavItem>Receive</NavItem>
            </LinkContainer>
          </Nav>
        </Navbar.Collapse>
      </Navbar>
      <Routes />
    </div>
  );
}
}

export default App;
```

Figure 4.15 Source Code for Navigation Bar

b) Index.js

Index.js loads all the files in the directory synchronously and the imported serviceWorker.js was used to add offline caching support for offline development. In the figure 4.16, it shows that first the ReactDOM will render the App.js and unregister method of the serviceWorker was used to prevent the site from crashing incase the service worker is facing a technical problems.

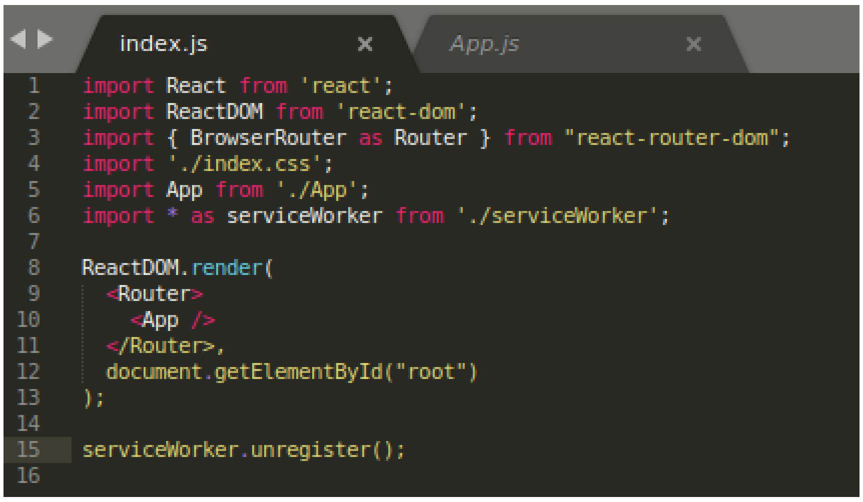

index.js App.js
```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from "react-router-dom";
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);

serviceWorker.unregister();

```

Figure 4.16 Source Code for Index.js

c) Routes.js

Routes.js is where all the routes for the navigation bar are defined. In order to route the user to the specific page in React, Route and Switch was imported from the react-router-dom library as shown in figure 4.17 below. With the Route and Switch, the pages for the web can finally be add for routing purposes.

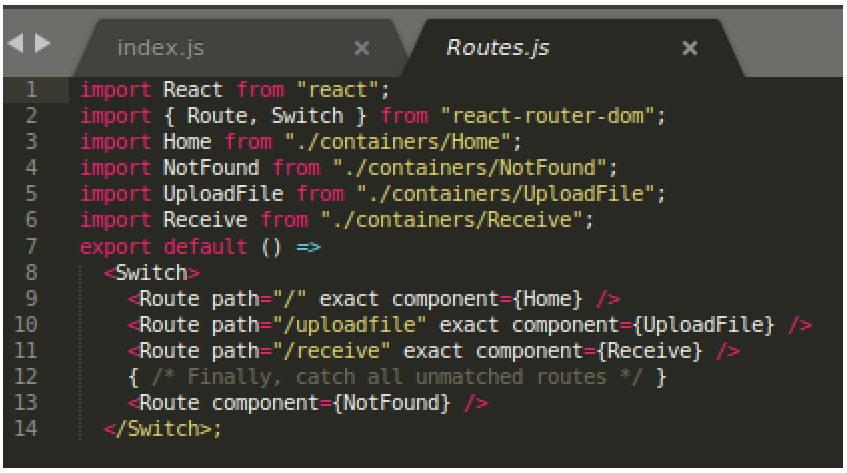
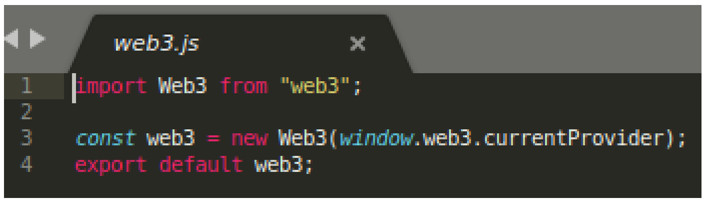

index.js Routes.js
```
import React from "react";
import { Route, Switch } from "react-router-dom";
import Home from "./containers/Home";
import NotFound from "./containers/NotFound";
import UploadFile from "./containers/UploadFile";
import Receive from "./containers/Receive";
export default () =>
  <Switch>
    <Route path="/" exact component={Home} />
    <Route path="/uploadfile" exact component={UploadFile} />
    <Route path="/receive" exact component={Receive} />
    { /* Finally, catch all unmatched routes */ }
    <Route component={NotFound} />
  </Switch>;
```

Figure 4.17 Source Code for Routes.js

d) Web3.js

Web3 is a library, which contains a collection of modules needed for the ethereum environment and allows the browser to connect with the ethereum. In figure 4.18 below, the source code, specify the environment of the ethereum from the metamask account.

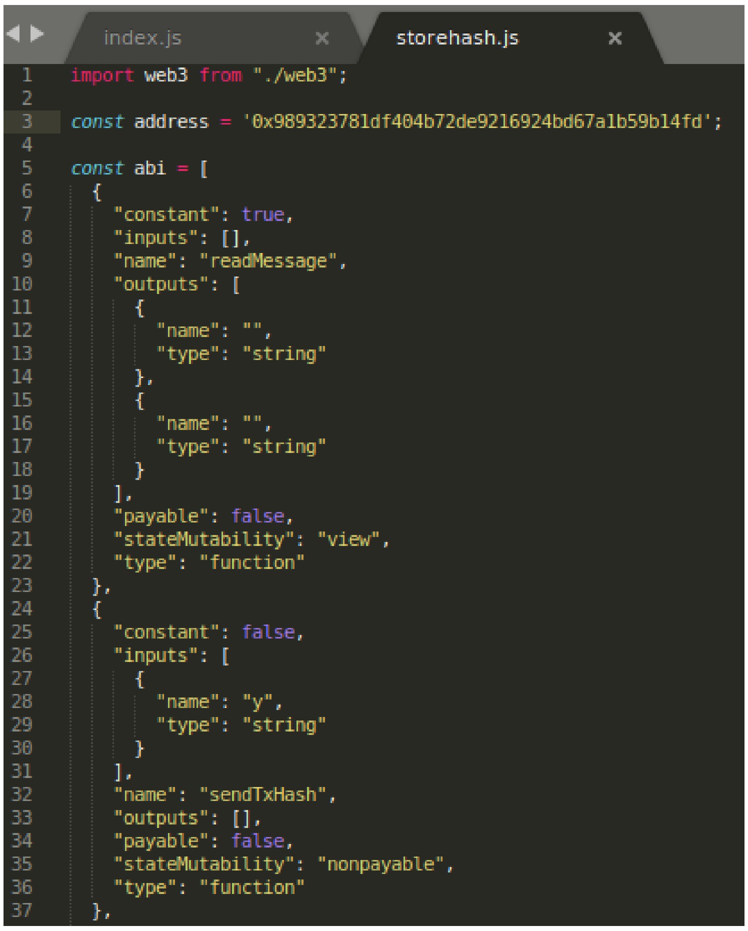


```javascript
import Web3 from "web3";

const web3 = new Web3(window.web3.currentProvider);
export default web3;
```

Figure 4.18 Source Code for Web3.js

e) StoreHash.js

In storehash.js, the address and the abi of the smart contract that were previously deployed were defined to enable the App to connect to the smart contract and use all the function in the smart contract.

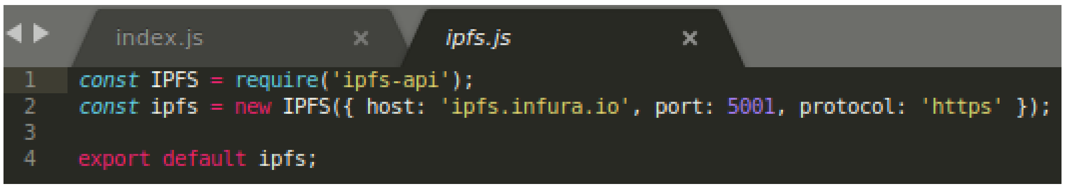


```
import web3 from "./web3";

const address = '0x989323781df404b72de9216924bd67a1b59b14fd';

const abi = [
  {
    "constant": true,
    "inputs": [],
    "name": "readMessage",
    "outputs": [
      {
        "name": "",
        "type": "string"
      },
      {
        "name": "",
        "type": "string"
      }
    ],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
  },
  {
    "constant": false,
    "inputs": [
      {
        "name": "y",
        "type": "string"
      }
    ],
    "name": "sendTxHash",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
```

Figure 4.19 Source Code for StoreHash.js

f) IPFS.js

In order to connect to ipfs node, the lines of codes that were shown in figure 4.20 below are essential and to ease the user to use the ipfs function without running the ipfs daemon, ipfs infura were used.

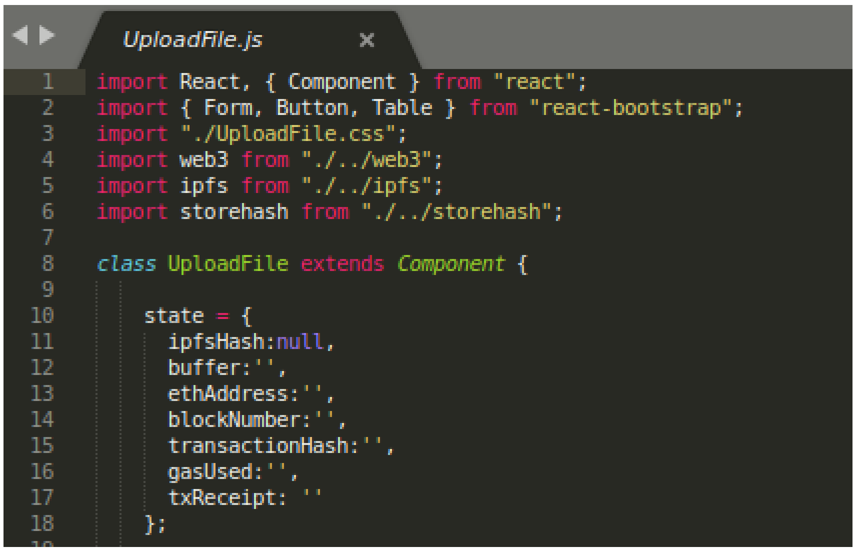


```
const IPFS = require('ipfs-api');
const ipfs = new IPFS({ host: 'ipfs.infura.io', port: 5001, protocol: 'https' });

export default ipfs;
```

Figure 4.20 Source Code for IPFS.js

g) UploadFile.js

As figure 4.21 below shows, the first six lines of the code are where all the related component, UI and files that were previously mentioned such as web3.js, ipfs.js and storehash.js were imported. The next step is to declare the state that is going to be used.

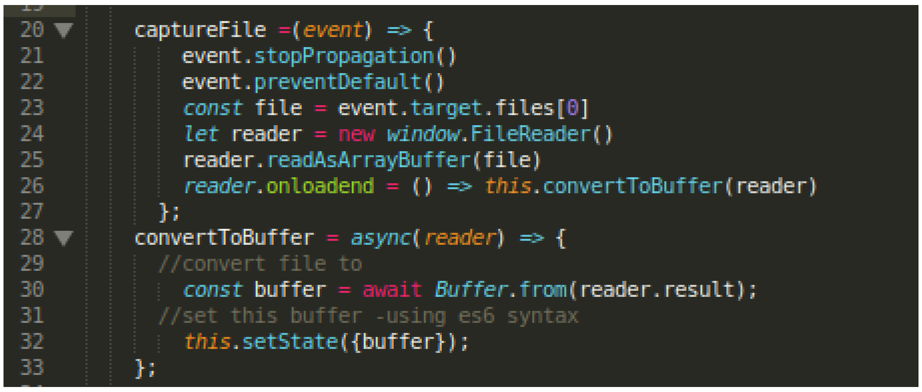


UploadFile.js

```
import React, { Component } from "react";
import { Form, Button, Table } from "react-bootstrap";
import "./UploadFile.css";
import web3 from "./../web3";
import ipfs from "./../ipfs";
import storehash from "./../storehash";

class UploadFile extends Component {

    state = {
      ipfsHash:null,
      buffer:'',
      ethAddress:'',
      blockNumber:'',
      transactionHash:'',
      gasUsed:'',
      txReceipt: ''
    };
```

Figure 4.21 Importing Component and Defining State for Upload.js

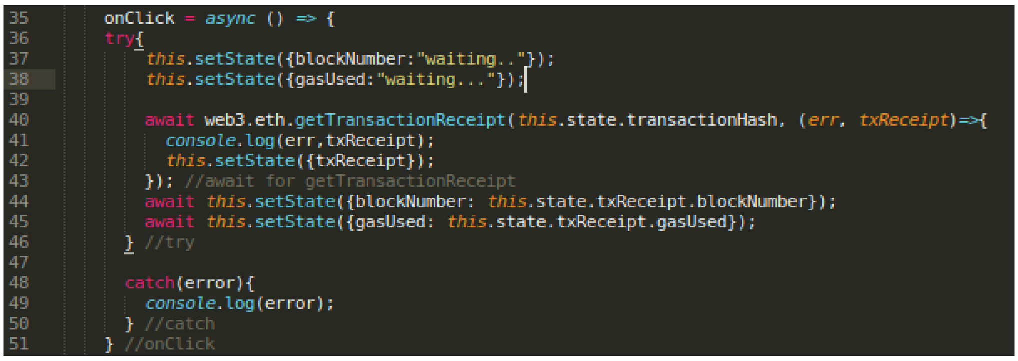
In the next line of code which is capture file, the selected file were send to convertToBuffer and converted the selected file to buffer and stored in the state that were declared previously as shown in the figure 4.22 below.



```
    captureFile =(event) => {
        event.stopPropagation()
        event.preventDefault()
        const file = event.target.files[0]
        let reader = new window.FileReader()
        reader.readAsArrayBuffer(file)
        reader.onloadend = () => this.convertToBuffer(reader)
      };
    convertToBuffer = async(reader) => {
      //convert file to
        const buffer = await Buffer.from(reader.result);
      //set this buffer -using es6 syntax
        this.setState({buffer});
    };
```

Figure 4.22 Source Code for Converting File to Buffer

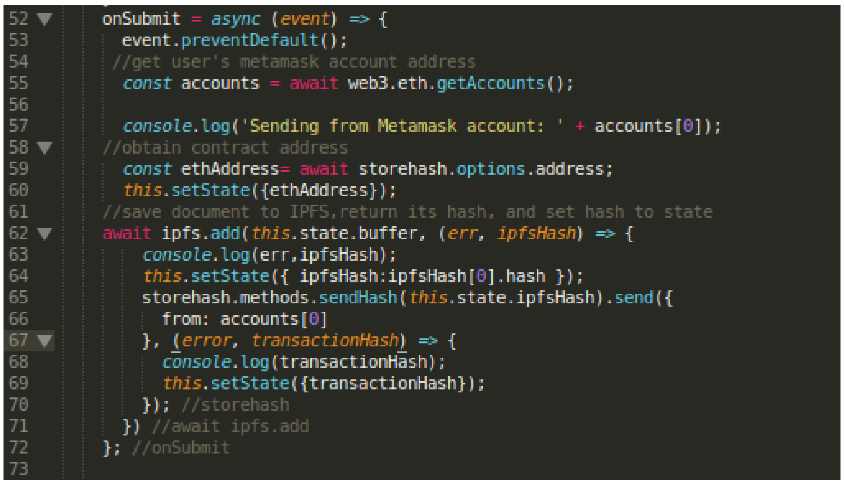
In figure 4.23 below, the onClick function are derived from the user input as the user click the "Get Transaction Receipt" which will generate the block number and gas used once the block from the transaction hash that have been mined and display it on user browser.



```
onClick = async () => {
try{
    this.setState({blockNumber:"waiting.."});
    this.setState({gasUsed:"waiting..."});

    await web3.eth.getTransactionReceipt(this.state.transactionHash, (err, txReceipt)=>{
      console.log(err,txReceipt);
      this.setState({txReceipt});
    }); //await for getTransactionReceipt
    await this.setState({blockNumber: this.state.txReceipt.blockNumber});
    await this.setState({gasUsed: this.state.txReceipt.gasUsed});
  } //try

  catch(error){
    console.log(error);
  } //catch
} //onClick
```

Figure 4.23 Source Code for Getting the Transaction Receipt

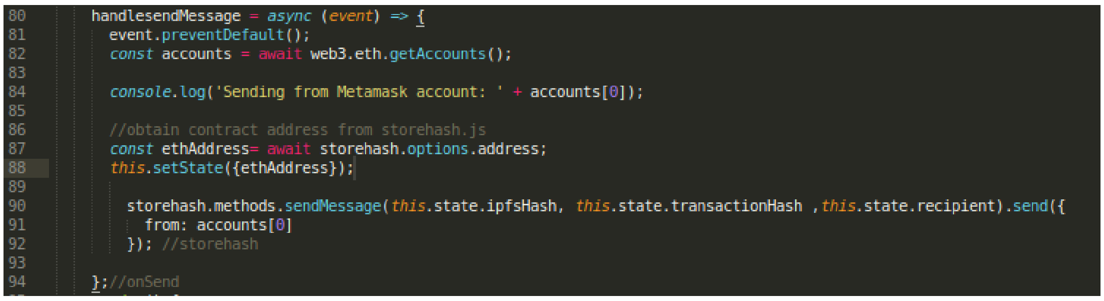
The onSubmit function will first get the user's metamask account to pay for the transaction and get the address of the smart contract address from the storehash file, while the ipfs.add() will send the buffered file to IPFS. IPFS will then generate the hash of the file to make it viewable and the hash of the file was stored inside the blockchain using the sendHash method. The user needs to confirm the transaction with their metamask account for the transaction hash to be generated.



```
onSubmit = async (event) => {
  event.preventDefault();
 //get user's metamask account address
  const accounts = await web3.eth.getAccounts();

  console.log('Sending from Metamask account: ' + accounts[0]);
//obtain contract address
  const ethAddress= await storehash.options.address;
  this.setState({ethAddress});
//save document to IPFS,return its hash, and set hash to state
await ipfs.add(this.state.buffer, (err, ipfsHash) => {
    console.log(err,ipfsHash);
    this.setState({ ipfsHash:ipfsHash[0].hash });
    storehash.methods.sendHash(this.state.ipfsHash).send({
      from: accounts[0]
    }, (error, transactionHash) => {
      console.log(transactionHash);
      this.setState({transactionHash});
    }); //storehash
  }) //await ipfs.add
}; //onSubmit
```

Figure 4.24 Source Code for Storing the File to IPFS and File Hash to Blockchain

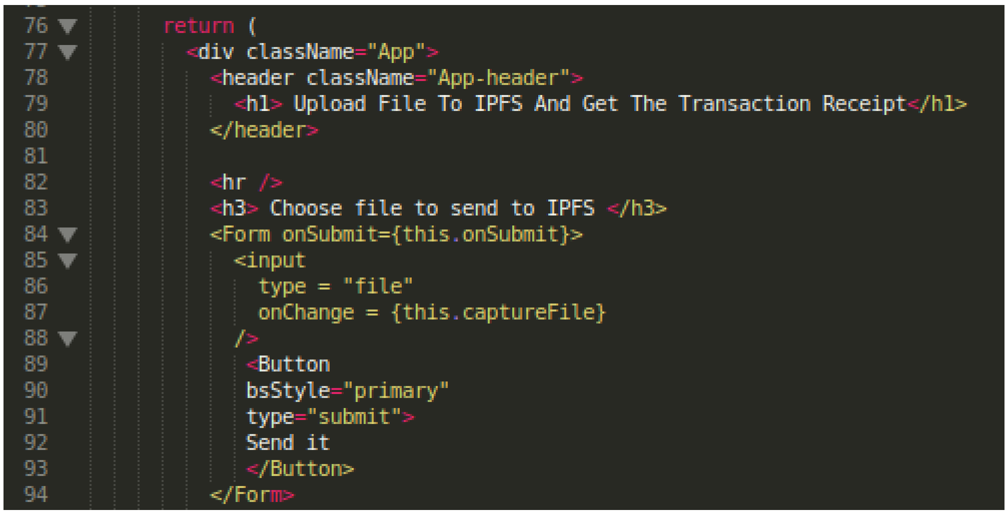
The next function is the handleMessage function as shown in figure 4.25 below, which it will act the same as onSubmit function where it will get the user's metamask account and smart contract address. After that it will call the sendMessage method to send the ipfsHash and transactionHash to the recipient ethereum address from the user input.



```
handlesendMessage = async (event) => {
  event.preventDefault();
  const accounts = await web3.eth.getAccounts();

  console.log('Sending from Metamask account: ' + accounts[0]);

  //obtain contract address from storehash.js
  const ethAddress= await storehash.options.address;
  this.setState({ethAddress});

    storehash.methods.sendMessage(this.state.ipfsHash, this.state.transactionHash ,this.state.recipient).send({
      from: accounts[0]
    }); //storehash

};//onSend
```

Figure 4.25 Source Code for Sending the File Hash and Transaction Hash to Other User

For the user interface (UI) of the UploadFile.js, the first thing is to create the form for an upload file using the HTML as shown in figure 4.26 below.



```
return (
  <div className="App">
    <header className="App-header">
      <h1> Upload File To IPFS And Get The Transaction Receipt</h1>
    </header>

    <hr />
    <h3> Choose file to send to IPFS </h3>
    <Form onSubmit={this.onSubmit}>
      <input
        type = "file"
        onChange = {this.captureFile}
      />
       <Button
       bsStyle="primary"
       type="submit">
       Send it
       </Button>
    </Form>
```

Figure 4.26 Source Code for Upload Form UI