**Ticket Sale Using Smart Contracts dApp on an Ethereum Blockchain**

# Use Case

The objective of this project is to demonstrate the ability to develop, test and deploy a Smart Contract decentralized application (dApp) on an Ethereum Blockchain for the following Use Case:

**The Blockchain Owner, wishes to sell concert tickets securely to buyers using a Smart Contract running on an Ethereum Blockchain. Buyers can go to a web page to see how many tickets are available for purchase. Buyers can then purchase tickets using their own MetaMask wallets.**

# Scope of Project

The following are the deliverables:

* A Front End web page
  + Showing tickets available for purchase
  + Allowing buyers to select tickets to purchase
  + Updating tickets available for purchase after each transaction
* A Smart Contract using Solidity deployed on a Blockchain

# Project Architecture

Diagram

Description automatically generated

# Environment Setup

The following tools are installed for development, testing and deployment:

* **Visual Studios Codes** for code development.

Graphical user interface, text, application, chat or text message

Description automatically generated

* **Truffle framework** to provide a framework for Smart Contract development, testing and deployment.
* **Ganache** to provide a personal local blockchain for Ethereum development to deploy the Smart Contract, develop applications, and run tests.
* **Solidity** for programming the Smart Contract.
* **Node.js** as a Javascript Runtime for executing Javascript codes.
* **Web3.js** libraries to interact with the Ethereum Blockchain using HTTP, IPC or WebSocket. It can retrieve user accounts, send transactions, interact with smart contracts, and more.
* **lite-server** is local web server bundled in the Truffle framework which will allow us to run the dApp on a browser.

Text

Description automatically generated

* **Git** and **GitHub** for version control

Graphical user interface, text

Description automatically generated

* **MetaMask** extension on Chrome browser to allow buyers to connect to the dApp.

Graphical user interface, application

Description automatically generated

# Directory Structure

We can start a project by either downloading a Truffle Box or starting a Truffle project from scratch.

To download a Truffle Box, issue the following commands in a terminal:

mkdir truffle-box

cd truffle-box

truffle unbox pet-shop

Text

Description automatically generated

This will create the necessary directories and basic files. The key ones are described below:

* /build\contracts **directory**
  + **Initially, this directory is empty. But after compilation, two JSON files will be created. They are:**
    - Migrations.json **– This file contains information about the** Migrations.sol **contract in JSON format, including its Contract Application Binary Interface (ABI), which is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification.**
    - Ticket.json **– As above, but for the** Ticket.sol **contract.**
* /contracts directory
  + Migrations.sol – This Smart Contract is provided as part of the Truffle framework to observe subsequent smart contract migrations and to ensure that we don't double-migrate unchanged contracts in the future. This file has not been modified for this project.
  + Ticket.sol – This is a new Smart Contract created to implement the required business logic for this project.
* /migrations directory
  + 1\_initial\_migration.js – This JS file is provided as part of the Truffle framework to deploy the first contract (Migrations.sol). This file has not been modified for this project.
  + 2\_deploy\_contracts.js – This is a new JS file created to deploy the second contract (Ticket.sol)contract.
* /src directory
  + index.html – This is a new html file to display the web page.
  + tickets.json – This is a new JSON file to contain the tickets’ information.
* /src/images directory
  + Contains the JPEG files to be displayed on the web page.
* /src/js directory
  + bootstrap.min.js – Provided as part of the Truffle framework to integrate bootstrap libraries. This file has not been modified for this project.
  + truffle-contract.js – Provided as part of the Truffle framework to manage the Smart Contracts. This file has not been modified for this project.
  + web3.min.js – Provided as part of the Truffle framework to integrate web3 libraries. This file has not been modified for this project.
  + app.js – This is a new JS file created to implement the logic tying the Front End, web3 and Smart Contract together.
* /test directory
  + TestTicket.sol – This is a new file created to test the contract logic in Ticket.sol
* Truffle-config.js
  + Defines the Blockchain information.
* bs-config.json
  + Tells the lite-server web server where to find the files.

The source codes for the key files are attached in the Appendix.

# Compiling the Smart Contracts

First, we need to compile the Solidity codes into bytecode for the Ethereum Virtual Machine (EVM) to execute.

This is done by issuing the following command in the terminal: truffle compile

After compilation, the Migrations.json and Ticket.json files will be created in the /build\contracts directory. These files contain information about the Migrations.sol and Ticket.sol contracts in JSON format.

Text

Description automatically generated

# Setting up Ganache

Now we will use Ganache to setup a Ethereum Blockchain so that we can deploy and test the Smart Contract. Setting up the Blockchain is simply done by clicking on the Ganache icon to run the application.

Text

Description automatically generated with medium confidence

Selecting QuickStart will create a Blockchain with 10 accounts, each with an initial balance of 100 ETH, as shown below in the Accounts tab.

This Blockchain is running locally on port 7545 (see truffle-config.js)

Table

Description automatically generated

# 

Since no Smart Contract has been deployed, there is currently no Transactions and no Blocks.

Graphical user interface, text, website

Description automatically generated

Graphical user interface, application

Description automatically generated

# Deploying the first Smart Contract

As mentioned earlier, Truffle framework has provided the Migrations.sol contract to observe subsequent smart contract migrations and to ensure that we don't double-migrate unchanged contracts in the future.

We will now migrate (ie deploy) this contract to the Blockchain (on port 7545) using the 1\_initial\_migration.js by issuing the commandtruffle migrate

Text

Description automatically generatedGraphical user interface, text

Description automatically generated

Because of this deployment, two transactions were made. This can be seen in Gananche:

Graphical user interface, text, website

Description automatically generated

The first transaction (Contract Creation) creates the contract. The Contract Address is b9286f563056C23f142F12f919f94f88D8a6596c. The owner of the contract is bADA239d37bf00f599a08f701bd2eBA60CD55124, who is also the first account listed in Ganache.

The transaction details are shown below, including the gas fees incurred, the transaction data and transaction hash.

This transaction is mined in Block 1.

Text

Description automatically generated with medium confidence

The second transaction (Contract Call) made to the same Contract Address by the same owner is mined in Block 2 and has the following details.

A picture containing table

Description automatically generated

The 2 blocks are shown as follows:

Table

Description automatically generated

# Deploying the second Smart Contract

Now, we deploy the second contract Ticket.sol contract.

The same truffle migrate command is issued. This time, only the 2\_deploy\_contracts.js script is executed.

Text

Description automatically generated

This time, two more transactions were mined in Block 3 and 4.

Graphical user interface, table

Description automatically generated with medium confidence

Table

Description automatically generated

In particular, the transaction in Block 3 is to create the Ticket.sol contract.

The Contract Address is 9E3DFdf62068F9607DF4950E1fd6AFCCEd69ddA5. The owner of the contract is bADA239d37bf00f599a08f701bd2eBA60CD55124, is the same as before.

Graphical user interface, text

Description automatically generated

The transactions have also incurred additional costs to the owner. His wallet now has a balance of 99.99 ETH.

Table

Description automatically generated

# Testing the Smart Contract

Now, we use the truffle test command to execute the TestTicket.sol script in order test the logic in Ticket.sol contract.

Text

Description automatically generated

Running the test has also created a number of transactions, which also resulted in some ETH being used up (99.74 ETH now).

Table

Description automatically generated

Graphical user interface, website

Description automatically generated

# Setting up the Metamask wallet for the Buyer

Prior to using the dApp, the buyer needs to set up his MetaMask wallet first.

First, we need to add the Ganache Blockchain network.

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Now, we import an account into Metamask.

Graphical user interface, application

Description automatically generated

For this example, let’s use the second account in Ganache.

Graphical user interface

Description automatically generated with medium confidence

Copy the private key for the account.

Graphical user interface, text, application, email

Description automatically generated

Paste it into Metamask.

Graphical user interface, text, application

Description automatically generated

A new account will be created and connected to the “Tickets” network.

Graphical user interface, text, application, email, website

Description automatically generated

# Using the dApp

First, we start the local web server by issuing this command npm run dev

Text

Description automatically generated

Text

Description automatically generated

On the browser, the following web page will appear.

Graphical user interface, website

Description automatically generated

Graphical user interface, website

Description automatically generated

Buyer can now purchase any of the tickets by clicking on any of the BUY buttons. The Metamask for the account we just imported will pop up.

Graphical user interface, text, application

Description automatically generated

Click on “Confirm” to confirm the purchase.

The ticket will be shown as “BOUGHT” and a small pop up will appear confirming the transaction.

Graphical user interface, application, website

Description automatically generated

Tickets.sol

pragma solidity ^0.5.0;

contract Ticket {

// Array containing each ticket's buyer's Ethereum Address.

// Index to array is 0 to 15 (16 tickets)

address[16] public buyers;

// Buy ticket

function buyTicket(uint \_ticketID) public returns (uint) {

// Make sure that the \_ticketID is between 0 and 15

require(\_ticketID >= 0 && \_ticketID <= 16);

// Write the buyer's Address into the array

buyers[\_ticketID] = msg.sender;

// Return the \_ticketID for confirmation

return \_ticketID;

}

// Get all the buyers

function getBuyers() public view returns (address[16] memory) {

// Return the array of buyers

return buyers;

}

}

2\_deploy\_contracts.js

// This is the script to deploy the Ticket.sol contract.

// Get the contract information from Ticket.json

var Ticket = artifacts.require("Ticket");

// Deploy the contract.

module.exports = function(deployer) {

deployer.deploy(Ticket);

};

style.css

.panel-title {

background-color: rgb(151, 147, 147);

padding: 15px;

border-radius: 1px;

font-size: 20px;

font-weight: bold;

}

.btn-buy{

background-color: rgb(96, 255, 4);

padding: 10px;

border-radius: 1px;

font-size: 15px;

font-weight: bold;

float: right;

}

.price{

font-weight: bold;

}

app.js

// This JS file ties in the Front End, web3 and Smart Contract together.

App = {

// Initialize some variables first

web3Provider: null,

contracts: {},

// init function to get tickets information from the JSON file and write to the HTML (ID = ticketTemplate)

init: async function() {

// Get from JSON file

$.getJSON('../tickets.json', function(data) {

var ticketsRow = $('#ticketsRow');

var ticketTemplate = $('#ticketTemplate');

for (i = 0; i < data.length; i ++) {

ticketTemplate.find('.panel-title').text(data[i].name);

ticketTemplate.find('img').attr('src', data[i].picture);

ticketTemplate.find('.price').text(data[i].price);

ticketTemplate.find('.date').text(data[i].date);

ticketTemplate.find('.location').text(data[i].location);

ticketTemplate.find('.btn-buy').attr('data-id', data[i].id);

ticketsRow.append(ticketTemplate.html());

}

});

// Once done, call initWeb3()

return await App.initWeb3();

},

// initweb3 function to initialize the web3 module accordingly.

initWeb3: async function() {

// If it is a modern dApp browser

// or more recent versions of MetaMask where an Ethereum provider is injected into the window object.

if (window.ethereum) {

App.web3Provider = window.ethereum;

try {

// Request access to the account

await window.ethereum.enable();

} catch (error) {

// User denied account access...

console.error("User denied account access")

}

}

// Else if it is a legacy dApp browsers

// ie an older dapp browser (like Mist or an older version of MetaMask)

else if (window.web3) {

App.web3Provider = window.web3.currentProvider;

}

// Else if no injected web3 instance is detected, fall back to Ganache

// This fallback is fine for development environments, but insecure and not suitable for production.

else {

App.web3Provider = new Web3.providers.HttpProvider('http://localhost:7545');

}

// Create instance of web3 object here

web3 = new Web3(App.web3Provider);

// Now call initContract() to instantiate the contract

return App.initContract();

},

// initContract function to instantiate our smart contract so web3 knows where to find it and how it works.

initContract: function() {

// Get the contract JSON file data

$.getJSON('Ticket.json', function(data) {

// We first retrieve the artifact file for our smart contract.

// Artifacts are information about our contract such as its deployed address and

// Application Binary Interface (ABI).

// The ABI is a JavaScript object defining how to interact with the contract including

// its variables, functions and their parameters.

var TicketArtifact = data;

// Once we have the artifacts in our callback, we pass them to TruffleContract().

// This creates an instance of the contract we can interact with.

App.contracts.Ticket = TruffleContract(TicketArtifact);

// Set the provider for our contract

App.contracts.Ticket.setProvider(App.web3Provider);

// Use our contract to retrieve and mark the relevant tickets as bought

return App.markBought();

});

return App.bindEvents();

},

bindEvents: function() {

$(document).on('click', '.btn-buy', App.handleBuy);

},

// markBought function to mark tickets that were already bought previously.

// We've encapsulated this in a separate function since we'll need to update the UI

// any time we make a change to the smart contract's data.

markBought: function() {

// We first declare the variable ticketInstance outside of the smart contract calls so we can

// access the instance after initially retrieving it.

var ticketInstance;

// We access the deployed Tickets contract, then call getBuyers() on that instance.

// Using call() allows us to read data from the blockchain without having to send a full

// transaction, meaning we won't have to spend any ether.

App.contracts.Ticket.deployed().then(function(instance) {

ticketInstance = instance;

return ticketInstance.getBuyers.call();

}).then(function(buyers) {

// After calling getBuyers(), we then loop through all of them, checking to see if an address is

// stored for each ticket.

// Since the array contains address types, Ethereum initializes the array with 16 empty addresses.

// This is why we check for an empty address string rather than null or other false value.

// If the ticket is NOT 0x0000000..... it means it has been bought.

for (i = 0; i < buyers.length; i++) {

if (buyers[i] !== '0x0000000000000000000000000000000000000000') {

// We disable its BUY button and change the button text to "BOUGHT", so the user gets some feedback.

$('.panel-ticket').eq(i).find('button').text('BOUGHT').attr({'disabled':true,'background-color':'#808080'});

}

}

}).catch(function(err) {

console.log(err.message);

});

},

// handleBuy function handles the ticket purchase operation

handleBuy: function(event) {

event.preventDefault();

var ticketID = parseInt($(event.target).data('id'));

var ticketInstance;

// We use web3 to get the user's accounts. In the callback after an error check,

// we then select the first account.

web3.eth.getAccounts(function(error, accounts) {

if (error) {

console.log(error);

}

var account = accounts[0];

// We get the deployed contract as we did above and store the instance in ticketInstance.

App.contracts.Ticket.deployed().then(function(instance) {

ticketInstance = instance;

// This time we will send a transaction instead of a call.

// Transactions require a "from" address and have an associated cost.

// This cost, paid in ether, is called gas.

// The gas cost is the fee for performing computation and or storing data in a smart contract.

// We send the transaction by executing the buyTicket() function with both the ticket ID and

// an object containing the account address, which we stored earlier in account.

return ticketInstance.buyTicket(ticketID, {from: account});

}).then(function(result) {

// The result of sending a transaction is the transaction object.

// If there are no errors, we proceed to call our markBought() function to sync the UI with

// our newly stored data.

return App.markBought();

}).catch(function(err) {

console.log(err.message);

});

});

}

};

$(function() {

$(window).load(function() {

App.init();

});

});

index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- The above 3 meta tags \*must\* come first in the head; any other head content must come \*after\* these tags -->

<title>Cheap Tix</title>

<!-- Bootstrap -->

<link href="css/bootstrap.min.css" rel="stylesheet">

<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->

<!-- WARNING: Respond.js doesn't work if you view the page via file:// -->

<!--[if lt IE 9]>

<script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></script>

<script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>

<![endif]-->

</head>

<body>

<div class="container">

<div class="row">

<div class="col-xs-12 col-sm-8 col-sm-push-2">

<h1 class="text-center">Cheap Tix</h1>

<hr/>

<br/>

</div>

</div>

<div id="ticketsRow" class="row">

<!-- TICKETS LOAD HERE -->

</div>

</div>

<div id="ticketTemplate" style="display: none;">

<div class="col-sm-6 col-md-4 col-lg-3">

<div class="panel panel-default panel-ticket">

<div class="panel-heading">

<h3 class="panel-title">Name</h3>

</div>

<div class="panel-body">

<img alt="140x140" data-src="holder.js/140x140" class="img-rounded img-center" width="220" height="140" src="https://animalso.com/wp-content/uploads/2017/01/Golden-Retriever\_6.jpg" data-holder-rendered="true">

<br/><br/>

<strong>Price</strong>: <span class="price">$$$</span><br/>

<strong>Date</strong>: <span class="date">January 2023</span><br/>

<strong>Location</strong>: <span class="location">Singapore</span><br/><br/>

<button class="btn btn-default btn-buy" type="button" data-id="0">BUY</button>

</div>

</div>

</div>

</div>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>

<!-- Include all compiled plugins (below), or include individual files as needed -->

<script src="js/bootstrap.min.js"></script>

<script src="js/web3.min.js"></script>

<script src="js/truffle-contract.js"></script>

<link rel="stylesheet" href="css/style.css" />

<script src="js/app.js"></script>

</body>

</html>

tickets.json

[

{

"id": 0,

"name": "Adele",

"picture": "images/adele.jpeg",

"price": "$150",

"date": "January 2023",

"location": "Singapore"

},

{

"id": 1,

"name": "Andy Lau",

"picture": "images/andylau.jpeg",

"price": "$100",

"date": "March 2023",

"location": "Singapore"

},

{

"id": 2,

"name": "Ariana Grande",

"picture": "images/arianagrande.jpeg",

"price": "$150",

"date": "July 2023",

"location": "Singapore"

},

{

"id": 3,

"name": "Black Eye Pea",

"picture": "images/blackeyepea.jpeg",

"price": "$160",

"date": "February 2023",

"location": "Singapore"

},

{

"id": 4,

"name": "Black Pink",

"picture": "images/blackpink.jpeg",

"price": "$120",

"date": "March 2023",

"location": "Singapore"

},

{

"id": 5,

"name": "Dua Lipa",

"picture": "images/dualipa.jpeg",

"price": "$160",

"date": "August 2023",

"location": "Singapore"

},

{

"id": 6,

"name": "Ed Sheeran",

"picture": "images/edsheeran.jpeg",

"price": "$180",

"date": "April 2023",

"location": "Singapore"

},

{

"id": 7,

"name": "Eric Clapton",

"picture": "images/ericclapton.jpeg",

"price": "$140",

"date": "May 2023",

"location": "Singapore"

},

{

"id": 8,

"name": "Harry Styles",

"picture": "images/harrystyles.jpeg",

"price": "$150",

"date": "June 2023",

"location": "Singapore"

},

{

"id": 9,

"name": "Jay Chou",

"picture": "images/jaychou.jpeg",

"price": "$150",

"date": "January 2023",

"location": "Singapore"

},

{

"id": 10,

"name": "JJ Lin",

"picture": "images/jj.jpeg",

"price": "$160",

"date": "October 2023",

"location": "Singapore"

},

{

"id": 11,

"name": "Jolin Tsai",

"picture": "images/jolintsai.jpeg",

"price": "$120",

"date": "March 2023",

"location": "Singapore"

},

{

"id": 12,

"name": "Karen Mok",

"picture": "images/karenmok.jpeg",

"price": "$120",

"date": "June 2023",

"location": "Singapore"

},

{

"id": 13,

"name": "Shakira",

"picture": "images/shakira.jpeg",

"price": "$180",

"date": "September 2023",

"location": "Singapore"

},

{

"id": 14,

"name": "Stephanie Sun",

"picture": "images/stephaniesun.jpeg",

"price": "$130",

"date": "November 2023",

"location": "Singapore"

},

{

"id": 15,

"name": "Taylor Swift",

"picture": "images/taylorswift.jpeg",

"price": "$160",

"date": "June 2023",

"location": "Singapore"

}

]

TestTicket.sol

// This Solidity script is used to test the Tickets.sol contract.

pragma solidity ^0.5.0;

// The Assert helper allows us to use various Asserts (eg Equality, inequality etc) to

// conduct our tests.

import "truffle/Assert.sol";

// When running tests, Truffle will deploy a fresh instance of the contract being tested

// to the blockchain. This smart contract gets the address of the deployed contract.

import "truffle/DeployedAddresses.sol";

import "../contracts/Ticket.sol";

contract TestTicket {

// Get the address of the contract to be tested

Ticket ticket = Ticket(DeployedAddresses.Ticket());

// We will use this as the ticket ID to test the contract

uint expectedTicketID = 5;

// We will use this as the buyer of the ticket (ID = 5)

address expectedBuyer = address(this);

// Function to test the buyTicket() function in Tickets.sol

function testBuyTicket() public {

// Call the buyTicket() function with the ticket ID of 5

uint returnedId = ticket.buyTicket(expectedTicketID);

// Assert the returned value to be equal to 5.

// If test fail, the print the message.

Assert.equal(returnedId, expectedTicketID,

"Purchased Ticket ID should match what is returned.");

}

// Function to test the retrieval of a buyer information

function testGetBuyerAddressByTicketID() public {

// We get the Address of the buyer by supplying the ticket ID = 5

// NOTE -- The "buyers" array is declared as a "public" array in Tickets.sol.

// So, it is possible to just call ticket.buyers to access the array directly.

address buyer = ticket.buyers(expectedTicketID);

// Assert the returned value to be equal to buyer.

// If test fail, the print the message.

Assert.equal(buyer, expectedBuyer,

"Buyer of the ticket does not match");

}

// Function to test the getBuyers() function in Tickets.sol

function testGetBuyerAddressByTicketIDInArray() public {

// Store buyers in memory rather than contract's storage

address[16] memory buyers = ticket.getBuyers();

// Assert the returned array to be equal to buyer.

// If test fail, the print the message.

Assert.equal(buyers[expectedTicketID], expectedBuyer,

"Buyer of the ticket does not match");

}

}