TESTING A SMART CONTRACT USING JAVASCRIPT

Truffle Introduction

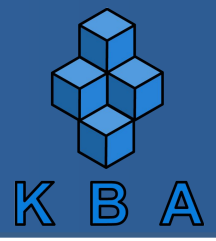
- Truffle is a framework used for building smart contracts for dApp development
- We don’t need truffle to write a smart contract. (For that we can use an editor like *remix*)
- Truffle is to coding a smart contract like Ruby on Rails is to Web Programming.

Prerequisites

- We need to install ***nodejs***
  - check for the current installed node version : *node -v*
  - Check the currently installed npm version : *npm -v*

- Check whether truffle is installed correctly
  - *truffle -v*

- Command to install truffle : ***sudo npm install -g truffle***



Setup the Repository

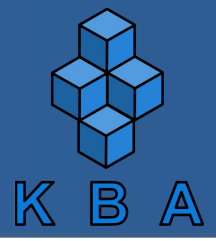
- We create a directory and initialize it with Truffle.

```
mkdir Auction
cd Auction
truffle init
```

- Inside the directory, we put the smart contract inside the contracts directory.

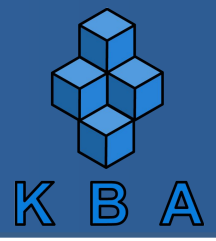
```
code contracts/Auction.sol
```

- We can replace code with the command to launch our beloved text editor, such as vim, edit, sublime, emacs, or atom.



Code

```
pragma solidity ^0.5.0;

contract Auction {
  address public manager;
  address public seller;
  uint public latestBid;
  address payable public latestBidder;

  constructor() public  {
    manager = msg.sender;
  }

  function auction(uint bid) public payable {
    latestBid = bid * 1 ether; //10000000000000000;
    seller = msg.sender;
  }

  function bid() public payable {
    require(msg.value > latestBid);

        if (latestBidder != address(0)) {
          latestBidder.transfer(latestBid);
    }
    latestBidder = msg.sender;
    latestBid = msg.value;
  }

  function finishAuction() restricted public payable {
    msg.sender.transfer(address(this).balance);
  }

  modifier restricted() {
    require(msg.sender == manager || msg.sender == seller);
    _;
  }
}
```



Continued...

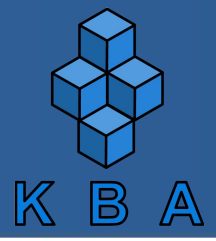
- We can compile the smart contract by launching this command:

  truffle compile

- By default the output of our compilation process is in build/contracts/Auction.json file. We can open it to see what it is inside. It is a json file. There are two keys of this json object that are very important, which are **ABI and bytecode**.

- The bytecode is the content that the Ethereum Virtual Machine understands. It's like the binary file. The ABI is the interface so we can interact with the smart contract. So if the bytecode is the house, the ABI is like the map of the house (where the doors are located).

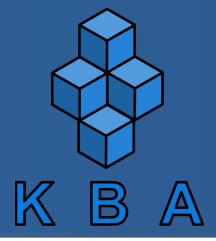
- We can not execute this bytecode just like that. It's different than when we compile the C/C++ code and be able to execute the binary directly. We have to put the smart contract in the blockchain first.



ganache-cli

- We are going to deploy this smart contract to the local blockchain.
- The way to launch the smart contract to Ethereum main network where real Ethereum lives is similar.
- But first let us launch the local virtual blockchain with Ganache.
- To install : **npm install -g ganache-cli**
- To run : **ganache-cli**

- By default it launches the service in port 8545 in localhost



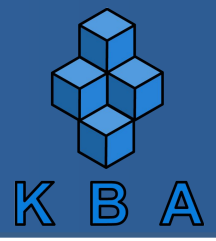
truffle-config

- Before we can deploy the smart contract, we need to tell the Truffle project where to look for the blockchain.
- Edit the truffle config file:

```
code truffle-config.js
```

- Copy this content into that file (you can uncomment these lines in the file):

```
module.exports = {
 // See <http://truffleframework.com/docs/advanced/configuration>
 // to customize your Truffle configuration!
 networks: {
    development: {
    network_id: “*”,
    host: “localhost”,
    port: 8545
   },
 }
};
```



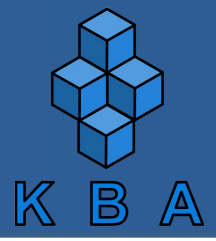
Migrations

Then we need to write the migration. To some people who are familiar with frameworks like Ruby on Rails, migration in this case is not like database migration. Here, migration means the process to deploy that particular smart contract.

```
code migrations/2_deploy_contracts.js
```

Copy this content to that file:

```
var Auction = artifacts.require('Auction');
module.exports = function(deployer) {
 deployer.deploy(Auction);
};
```



truffle migrate

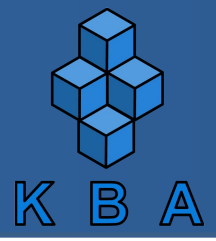
Now you can deploy the smart contract. Open a new terminal and go inside the project directory. Don't disrupt the terminal where you launched ganache-cli.

```
truffle migrate
```

You will get the output like this:

```
Using network 'development'.
Running migration: 1_initial_migration.js
Replacing Migrations...
... 0x3a8f8c4379f1d7269b9d931a088984f4ccfeec797bbb6e128582e2915c8f3ea1
Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
Replacing Auction...
... 0xf0b10908aeb415b8b13fe60aeed8ade6769679572a44cf679b5616e38670329d
Auction: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving artifacts...
```

JavaScript Testing



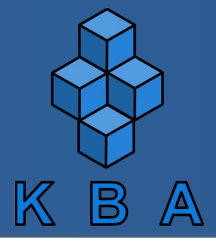
Testing

- Now, let's write unit test for our smart contract.
- For now, we just have to write the test manually.
- First install some stuff with npm.

```
npm install --save ganache-cli mocha web3@1.0.0-beta.37
```

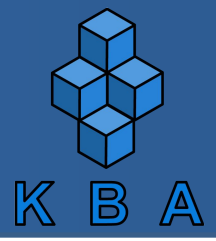
- This is local installation. It will save these libraries into this project, but not system wide. Ganache is like Truffle develop, a local blockchain for smart contract unit testing. Mocha is unit testing framework.

- The web3 is the way we connect to the deployed smart contract. Here, the web3 is not version 1.0 production ready yet.



Hooks - Mocha

```
describe('hooks', function() {
  before(function() {
    // runs before all tests in this block
  });

  after(function() {
    // runs after all tests in this block
  });
```

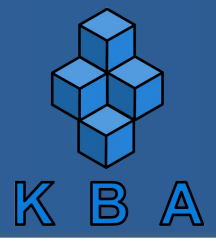


Continued...

```
 beforeEach(function() {
   // runs before each test in this block
 });

 afterEach(function() {
   // runs after each test in this block
 });

 // test cases
});
```

Assert

- The assert.equal() method tests if two values are equal, using the == operator.
- If the two values are not equal, an assertion failure is being caused, and the program is terminated.
- Syntax : **assert.equal(*value1, value2, message*);**



Testing

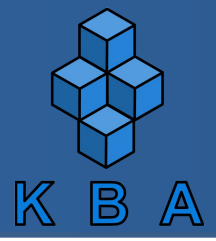
Put the test file inside the test directory and name it test.js

```
code test/test.js
```

In test.js Write these lines:

```
const assert = require('assert');
const ganache = require('ganache-cli');
const Web3 = require('web3');
const web3 = new Web3(ganache.provider());
const json = require('./../build/contracts/Auction.json');
```

Here, we are importing important stuff from libraries we just installed. assert is to check the whether the method is working properly or not. ganache-cli is our local virtual blockchain. web3 is the way we connect to this local blockchain. json is a the output of the compilation of Auction.sol file.

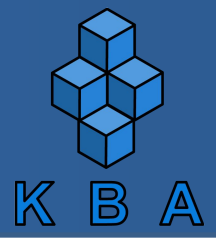


Variables

Below that, write these lines.

```
let accounts;
let auction;
let manager;
const interface = json['abi'];
const bytecode = json['bytecode'];
```

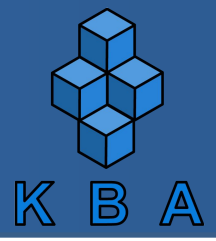
We initialize some variables. As I described above, we only care about two parts of the output json file, the bytecode and the interface (abi).



beforeEach

Go on with these lines:

```
describe('Auction', () => {
    beforeEach(async () => {
    accounts = await web3.eth.getAccounts();
    manager = accounts[0];
    auction = await new web3.eth.Contract(interface)
        .deploy({ data: bytecode })
        .send({ from: manager, gas: '1000000' });
    });
//Continue from this line from now on...
});
```



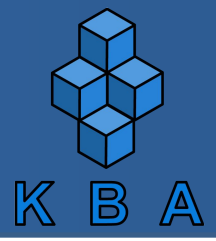
Testing

- *describe* is a way of grouping tests.

- *beforeEach* in unit testing means you run this method every time you execute a unit test. Connecting to blockchain is an asynchronous process so we have to declare our function async. The first thing we do is to get all accounts.

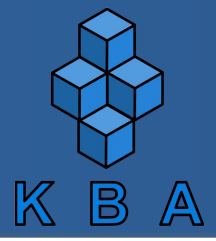
- Ganache gives us some accounts to play for. But we need these accounts before we go to other steps so we use await here to make the statement synchronous. We use the first account as manager. Then we deploy the contract.

- To deploy the contract, we need the interface, the bytecode, the address and some gas (deploying contract is not free).



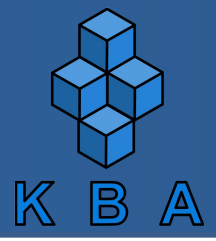
Testing : Example 1

```
it('deploys a contract', async () => {
  const auctionManager = await auction.methods.manager().call();
  assert.equal(manager, auctionManager, "The manager is the one who launches the smart contract.");
});
```



Describe

- The first test is to check the address which deploys the contract is saved to manager variable inside the smart contract.
- Here, we are interacting with the smart contract with Node.js not Solidity.
- Hence, there are some differences of accessing the methods or properties of the smart contract.



Testing : Example 2

To access the manager property of the smart contract, we use this statement:

```
auction.methods.manager().call()
```

We continue our second test.

```
it('auctions the item', async () => {
seller = accounts[1];
await auction.methods.auction(2).send({ from: seller });
auctionSeller = await auction.methods.seller().call();
assert.equal(auctionSeller, seller,
"The seller is the one who called the auction method.");
auctionBid = await auction.methods.latestBid().call();
assert.equal(auctionBid, web3.utils.toWei('2', 'ether'),
"The latest bid is the argument sent to auction method converted into wei.");
});
```

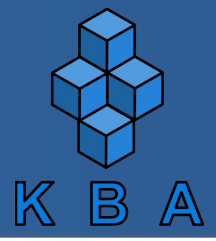
Testing : Example 3

- Here, in the previous test, we use different account for seller. To execute the method of the smart contract with a specific address, use this statement:

  ```
  auction.methods.auction(2).send({ from: seller });
  ```

We continue our journey with the third test:

```
it('bids the item', async () => {
  bidder = accounts[2];
  try {
  await auction.methods.bid().send({ from: bidder, value: web3.utils.toWei('3', 'ether') });
  auctionBid = await auction.methods.latestBid().call();
  assert.equal(auctionBid, web3.utils.toWei('3', 'ether'), 'The latest bid is the payment sent to bid method converted into wei.');
  } catch (err){
    assert(err);
  }
});
```

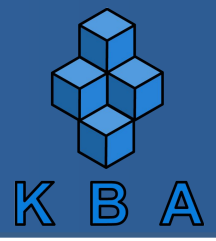


Continued...

- Here, in the previous test, we test the sending payment when we execute the method of the smart contract. We use value keyword argument. By default, it accepts wei number. But we are working with ether number so we have to convert it with web3 helper function.

```
auction.methods.bid().send({ from: bidder, value: web3.utils.toWei('3', 'ether') });
```

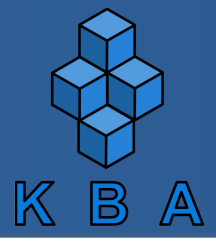
- Command to run test : **truffle test**



Exercise 1

- Write a test to check that the bidding amount is greater than the latest amount.
- Here, in this test, we test the exception thrown when we fail the require statement in the smart contract.
- We have to send the payment greater than the last bid. So in this test we want to test the fail case. We use try catch to test the fail case .

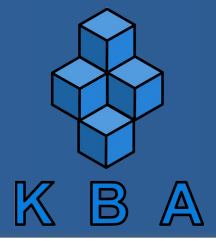
```
it('must bid above the latest bid amount', async () => {
  //initialize the bidder
  try {
      //write code here
  } catch (err) {
    //catch error
  }
});
```



Exercise 2

- Write a test to check the modifier restriction that only manager can finish the auction.

```
it('only manager can finish the auction', async () => {
  //initialize non-manager
  try {
    //write code here
   } catch (err) {
    //catch errror
  }
});
```



Exercise 3

- Write a test to finish the auction as manager.

```
it('finishes the auction as manager', async () => {
  //initialize manager
  //call finishAuction() method asynchronously
  //write assert
 });
```

THANK YOU