Course 3 Project: Test-driven Development of a Dapp

Title: Auction Dapp with Test Script

Learning objectives:

On completion of this course project you will be able to

1. Design and develop a decentralized application (Dapp) using Truffle Integrated Development Environment (IDE)
2. Explain and apply the concepts of test-driven development
3. List the steps in Dapp development using Truffle IDE.

Pre-requisite: You will work on a virtual machine image that runs Ubuntu operating system. This is the same VM image that you downloaded in Course 1. If you do not have it installed, download and install the virtual machine image from the Course Resources section. You will use command line interface (CLI) to navigate to the various directories and to issue the commands needed for Truffle-based development. All the code needed for the course demos and this project are available in CourseraDocs in the home directory of the VM. Make sure you can locate and view the contents. Open a terminal by cntrl-t and issue the following command to view the CourseraDocs. Then navigate back to home directory (cd).

```
>cd | cd CourseraDocs | ls
```

Problem Statement:

There are two parts to this problem. Part 1 is the development of the smart contract core logic of the Dapp from Course 2 of the Blockchain Basics specialization titled *Smart Contracts*. Part 2 is the test-driven development of the Dapp using Truffle IDE.

Part 1: (20 %) If you have completed *Smart Contracts*, you will just have move the smart contract you already created to the Truffle environment. If you have not completed Course 2, follow the project instructions given here and complete the project before you move on to Part 2. The creation of the smart contract is a pre-requisite for this project.

Make sure you submit to the project 3 grader the COMPLETED Auction.sol from Course 2 or from the code generated as directed below. Then move on to Part 2. (Part 1 continued on a new page.)

Part 1 Description (Skip this if you have completed Course 2: Smart Contract project and have the Auction.sol smart contract ready and submitted it for 20% grade)

Problem Statement:

Consider the problem of Chinese auction or penny social. We will refer to it as simple "Auction." It is a conventional approach used for fundraising for a cause. The organizers collect items to be auctioned off for raising funds. Before the auction, the items for auctions are received and arranged each with a bowl to place the bid. A chairperson is a special person among the organizers. She/he heads the effort and is the only person who can determine the winner by random drawing at the end of the auction. A set of bidders buy sheets of tickets with their money. The bidder's sheet has a stub that identifies the bidder's number, and tokens bought.

The bidders examine the items to bid, place the one or more tickets in the bowl in front of the items they desire to bid for until all the tickets are used. After the auction period ends the chairperson, collects the bowls, randomly selects a ticket from each item's bowl to determine the winning bidder for that item. The item is transferred to the winning bidder. Total money collected is the fund raised by the penny social auction.

Assumptions:

The description given above is for a general penny social auction. For the sake of our project implementation we will introduce some simplifying assumptions. Here they are:

1. Fixed number of bidders, initialized to 4. All 4 need to self-register. Funds transfer from bidder is automatically done and is not in the scope of this application.
2. Fixed number of items to be auctioned off, initialized to 3.
3. Items auctioned are indexed from 0..N-1 where N is the number of items for auction. N is 2.
4. Each bidder buys just 1 sheet of tickets or tokens; each sheet has only 5 tokens.
5. Assume simple number for the serial numbers for the sheet of tickets: 0,1,2,3. Here we show the tokens of bidder 0 and 1.

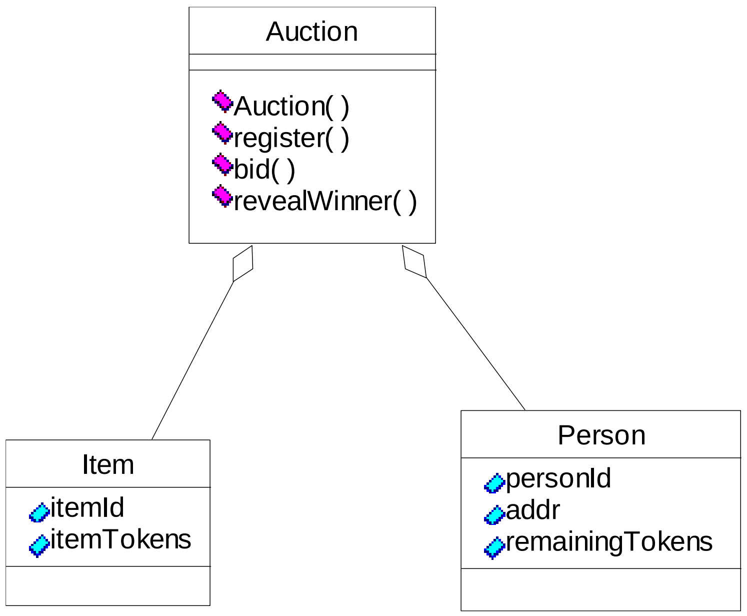
0	0	0	0	0
1	1	1	1	1
.	.	.	.	.
n-1	n-1	n-1	n-1	n-1

For n people and so on.

The Design: Let's design the smart contract for this. Visualize the situation using the screen shot given here.

These are the pictures of items, we need to define "Items" in the smart contract. We will also need to define Persons or bidders who will bidding on the Items. We will also need some supporting data variables.

The functions are similar to the Ballot of our lessons. Constructor that initializes the owner, register that allows (decentralized person) to register online to get the tokens and start bidding, bid function that lets a person bid, and finally revealWinner, to randomly choose the winner for the item auctioned. Here is our design. Always remember to design first. The Auction smart contract has Item and Person structs and other data items such as array of Items, array of Persons, array of winners, mappings, and beneficiary address.



Implementation:

You will write two versions of the implementation templates: version 1: without modifier (80%) and version 2 with modifier (20%). We have provided the templates for both versions.

1. Please understand the problems before you proceed.
2. Copy the template into your Remix IDE. Complete the code. You will need to fill in the code at *ONLY* at the locations indicated. Test it to see if it is operational.
3. Implement version of as Auction.sol and submit for grading. Then update this base version for Auction.sol for Part 2 requirements, and submit for grading.

Version 1: Auction.sol: There are 6 tasks where you will fill in the code. You need to review the code given and understand it fully before you start adding code. This partial code is available in the course resources section as Auction.sol.

Testing on Remix:

Test the completed code by compiling and running it on Remix JavaScript VM. (i) Account 0 provided by JavaScript VM is the Auction beneficiary and will not bid. (ii) Navigate to each of the other accounts, and “register.” You will register four bidders using register function. (iii) The next step is for the bidders to each bid; for test purposes, you can execute the “bid” function with {{0,1}{1,1}{2,1} } for each of the four accounts. (iv) Next, execute the “revealWinner” function to determine the winner for each item randomly, and (v) the getter of the “winners” data can be executed with {0, 1 and 2} as a parameter in sequence to reveal the winners of the draw respectively. You can do a lot of more testing and exploration with the buttons provided by the Remix web interface.

Partial code for Auction.sol

```
pragma solidity ^0.4.17;
contract Auction {

// Data
    //Structure to hold details of the item
    struct Item {
        uint itemId; // id of the item
        uint[] itemTokens;  //tokens bid in favor of the item

    }

   //Structure to hold the details of a persons
    struct Person {
        uint remainingTokens; // tokens remaining with bidder
        uint personId; // it serves as tokenId as well
        address addr;//address of the bidder
    }

    mapping(address => Person) tokenDetails; //address to person
    Person [4] bidders;//Array containing 4 person objects

    Item [3] public items;//Array containing 3 item objects
    address[3] public winners;//Array for address of winners
    address public beneficiary;//owner of the smart contract

    uint bidderCount=0;//counter

    //functions

    function Auction() public payable{    //constructor

    //Version 1 Task 1. Initialize beneficiary with address of smart
contract’s owner
    //Hint. In the constructor,"msg.sender" is the address of the owner.
            // ** Start code here. 1 line approximately. **/

            //** End code here. **/
        uint[] memory emptyArray;
        items[0] = Item({itemId:0,itemTokens:emptyArray});

        //Version 1 Task 2. Initialize two items at index 1 and 2.
            // ** Start code here. 2 lines approximately. **/
        items[1] =
        items[2] =
            //** End code here**/
    }
    function register() public payable{


        bidders[bidderCount].personId = bidderCount;

        //**Version 1 Task 3.** Initialize the address of the bidder
        /*Hint. Here the bidders[bidderCount].addr should be initialized with
address of the registrant.*/

            // ** Start code here. 1 line approximately. **/

            //** End code here. **

        bidders[bidderCount].remainingTokens = 5; // only 5 tokens
        tokenDetails[msg.sender]=bidders[bidderCount];
        bidderCount++;
    }

    function bid(uint _itemId, uint _count) public payable{
        /*
            Bids tokens to a particular item.
            Arguments:
            _itemId -- uint, id of the item
            _count -- uint, count of tokens to bid for the item
        */

        /*
            **Version 1 Task 4.** Implement the three conditions below.
                  4.1 If the number of tokens remaining with the bidder is <
            count of tokens bid
                  4.2 if there are no tokens remaining with the bidder,
            revert.
                  4.3 If the id of the item for which bid is placed, is
            greater than 2, revert.

            Hint: "tokenDetails[msg.sender].remainingTokens" gives the
details of the number of tokens remaining with the bidder.
            */

        // ** Start code here. 2 lines approximately. **/


        //** End code here. **

            /***Version 1 Task 5.** Decrement the remainingTokens by the number
of tokens bid
            Hint. "tokenDetails[msg.sender].remainingTokens" should be
decremented by "_count". */

            // ** Start code here. 1 line approximately. **


            //** End code here. **
        bidders[tokenDetails[msg.sender].personId].remainingTokens=
tokenDetails[msg.sender].remainingTokens; //updating the same balance in
bidders map.
        Item storage bidItem = items[_itemId];
        for(uint i=0; i<_count;i++) {
            bidItem.itemTokens.push(tokenDetails[msg.sender].personId);
        }
    }


    function revealWinners() public {

            /*
                  Iterate over all the items present in the auction.
                  If at least on person has placed a bid, randomly select
the winner */

        for (uint id = 0; id < 3; id++) {
            Item storage currentItem=items[id];
            if(currentItem.itemTokens.length != 0){
       // generate random# from block number
            uint randomIndex = (block.number /
currentItem.itemTokens.length)% currentItem.itemTokens.length;
// Obtain the winning tokenId

                uint winnerId = currentItem.itemTokens[randomIndex];
```

/* **Version 1 Task 6**. Assign the winners.
Hint." bidders[winnerId] " will give you the person object with the winnerId.
you need to assign the address of the person obtained above to winners[id] */

```
                        // ** Start coding here *** 1 line approximately.


                        //** end code here*

            }
        }
    }

  //Miscellaneous methods: Below methods are used to assist Grading. Please
DONOT CHANGE THEM.
 function getPersonDetails(uint id) public constant
returns(uint,uint,address){
      return
(bidders[id].remainingTokens,bidders[id].personId,bidders[id].addr);
}

}
```

Version 2: After testing version 1 on Remix IDE, please submit the solution for autograding on Coursera.

For version 2, we need to add a modifier so that only the owner can invoke the function “revealWinner”.

/***Version 2 Task 1.** Create a modifier named "onlyOwner" to ensure that only owner is allowed to reveal winners.

Hint: Use require to validate if "msg.sender" is equal to the "beneficiary". */

This involves defining a modifier and using the modifier in the header of the “revealWinner” function.

Add these updates, compile and test with Remix IDE. Make sure the operations of “revealWinner” from any other account than the Owner (account{0}) results in “revert”.

Save the completed solution and submit for auto grading. (20%)

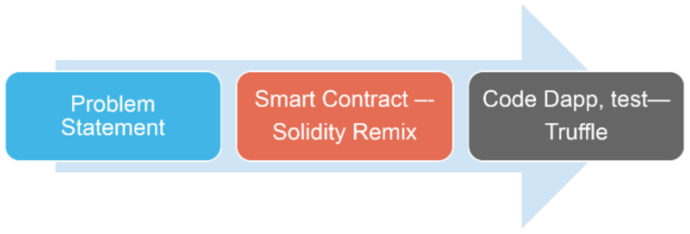
END OF PART 1 for Course3 Project

Part 2 Continued on a new page.

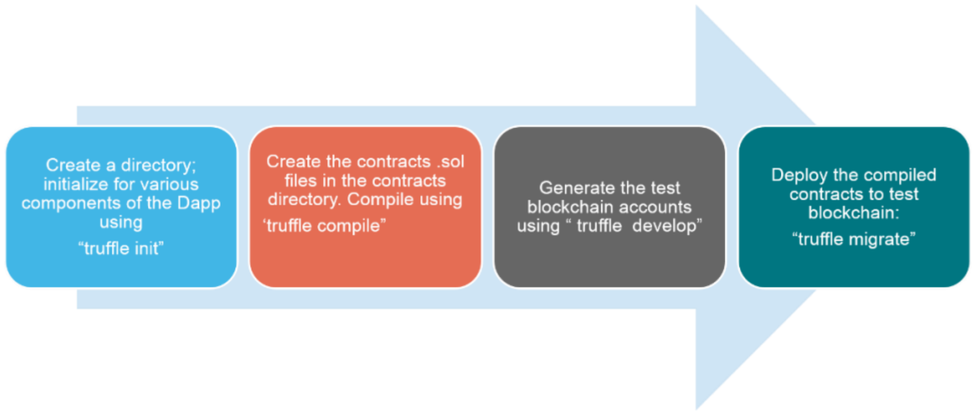
Part 2: (80%) Your objectives are to:

1. **Create a truffle project for the smart contract Auction.sol from Part 1**
2. **INCREMENTALLY develop a test.js script to test the Auction.sol**

Step 1: All the files you need, except Auction.sol and test.js, are provided for you in the course Resources. Recall the steps from your lessons, which are repeated here. Make sure you have access to these files/code, deploy scripts and other assets. Recall the design process.



Step 2: Follow the steps in creating a Truffle project that we learned in the lessons. Make sure you are able to initialize the project and deploy the Auction project on the test chain (truffle init, truffle compile, truffle develop, truffle migrate --reset)


Create a directory; initialize for various components of the Dapp using "truffle init"
Create the contracts .sol files in the contracts directory. Compile using 'truffle compile"
Generate the test blockchain accounts using " truffle develop"
Deploy the compiled contracts to test blockchain: "truffle migrate"

Instructions for making the Auction truffle project:
cd
cp CourseraDocs/Auction.zip .
unzip Auction.zip
In the contracts directory replace Auction.sol with the Auction.sol you completed in Part 1.
Navigate to the Auction base directory, "truffle compile". It should compile without any error.

Step 3: Review the template test.js in the CourseraDocs or from the course resources. This template has **five** test scripts out of which the code for first two are given to you complete. Review and understand them. The five tests are:

Test 1: It checks if the contract has been deployed. If yes, it returns the instance in the callback function. We just save the instance in a global variable for future use and also assert it to check if it is not null.

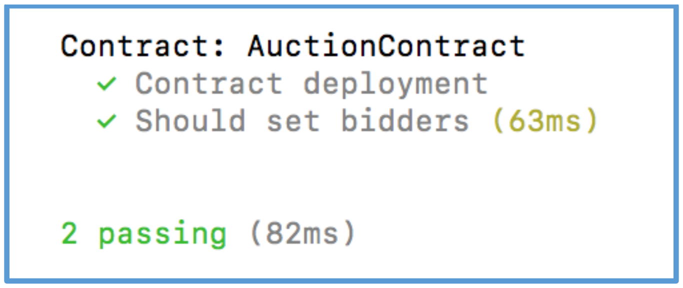
Test 2: This makes a call to the register function. Then, we call the **getPersonDetails** function to fetch the person's details. The person's details will be present in the callback function, and we are asserting to see if it is equal to the address we had registered.

Test 3: This a failure or negative test case (where revert should be ideally executed). We call the bid function with more than five tokens, which throws a revert error.

Test 4: This a failure or negative test case (where revert should be ideally executed). We call the **revealWinners** function from a non-owner account, which throws a revert error.

Test 5: This is the proper working flow of the contract. This is a positive test. We register three accounts, bid items using them, and call **revealWinners** from the authorized account. After which, we assert the winner of each item to see if the address is set.

Review the original copy of the test.js given to you. Identify the first two test cases in the test script and understand the script. Navigate to test directory. Create a file test.js in the test directory, but only with the first two tests (scripts for which are given). Save it. Navigate back to your Auction base directory. Execute the command "truffle test", and you should see the two tests pass if your Auction.sol was correct and the Truffle commands were correct. You will see the two test passing as shown below:



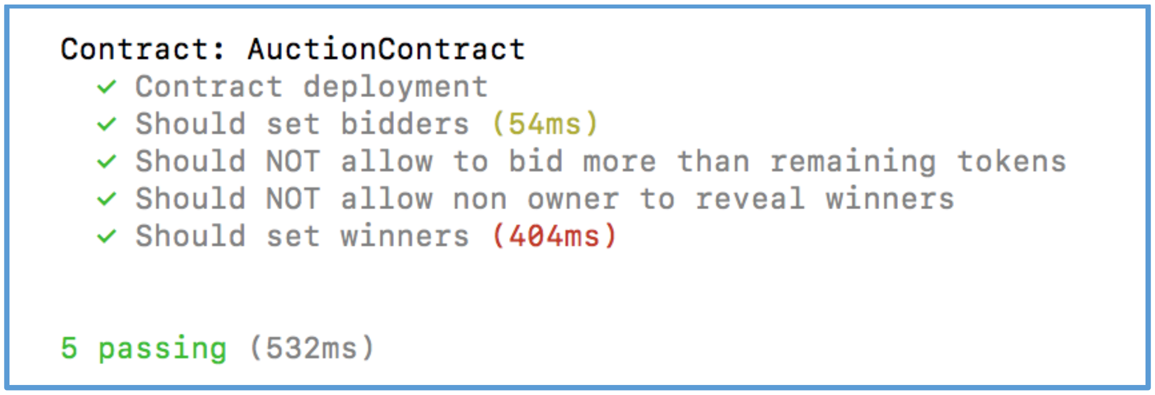
```
Contract: AuctionContract
  ✓ Contract deployment
  ✓ Should set bidders (63ms)


2 passing (82ms)
```

If this does not happen, you will have to make sure your Auction.sol is correct, or your directory structure of the Dapp is as discussed in the lesson videos and demos.

Step 4: After passing the first two tests, examine the other three test scripts provided. You can now work on the three test cases incrementally, one by one. In all, the three test cases require about 19 lines of code. If you develop the script for the test case incrementally, you will focus only on that test case and therefore less code.

Now, add template code for the three test cases but comment out the last two so that you will execute only the first of the three test cases added. Apply the knowledge from the lessons and the patterns from the test code for Ballot.sol, add scripts/code at the indicated points. Save the test.js. Then, execute "truffle test" to see one more test passing if the code you added is correct. Repeat this for the other two test cases.

If all of the tests are included in the test.js and the code you entered is correct, when you execute "truffle test" you should see the following output: (Note that the timing may be different.)



```
Contract: AuctionContract
  ✓ Contract deployment
  ✓ Should set bidders (54ms)
  ✓ Should NOT allow to bid more than remaining tokens
  ✓ Should NOT allow non owner to reveal winners
  ✓ Should set winners (404ms)


5 passing (532ms)
```

Step 5: Now you are ready to submit. Do not change the template we have given so that the grader can assess the completeness of your test script. Submit **test.js** only for the grader to evaluate it and assign your grade.

Summary: We went through the process of testing a smart contract that is at the core of a Dapp in a blockchain-based system. The overarching goal of this course project is to emphasize thorough testing of the smart contract and to highlight the importance of testing. Never deploy any code without testing.

Armed with the knowledge of what we learned in this course and this project we are ready to design, develop and test blockchain projects outside this course.