

CSE 526: Blockchain Term Project BidPoint-Dapp

Name: Nikhil Pandharinath Yadav

Person Number: 50317921

Email: nyadav2@buffalo.edu

Phase 1

Description

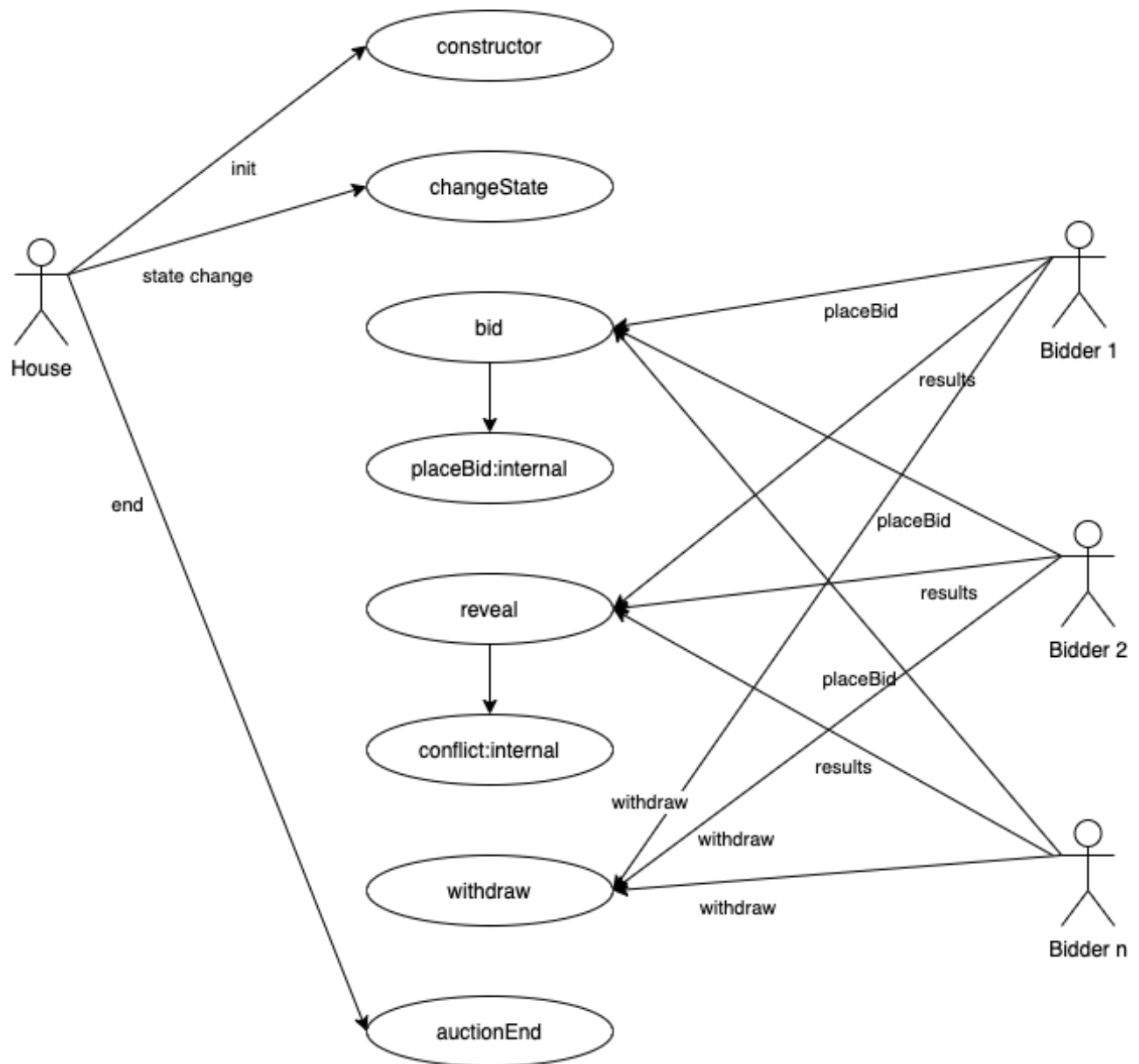
- **Major Area:** Competitive Market Place
- **Title:** Online commerce for multiple products.
- **Dapp Name:** *BidPoint-Dapp*
- **Clients:** Buyers/Sellers
- **Problem Statement:**
 - This application will be a decentralized market place for people around the world. The inspiration for this idea is the website **StockX.com**. StockX is a market place for sneakerheads where the bidding war can go beyond \$100k for a single pair of shoes. In this DAPP users across the globe can find what they want and can bid over the item. The bidding will be set up within a certain time frame. This DAPP will become a competitive market place.
 - The DAPP will establish **direct contact** between the buyers and the sellers. The transactions will take place in **real-time** and only between the concerned parties. Blockchain will ensure a **trust layer** that will verify and validate the information of the users leaving zero doubts of whether the clients are spending the right amount and check whether the transactions are going to the right party.
 - This will also be a place that will attract consumers as it will be using the principles of gamification where the competition in the bidding will attract more users since there are definite winners and losers for each product.
 - Creating a fun experience and a functional atmosphere with 100% trust for transactions in this market place is the ultimate goal of this DAPP.

Phase 2

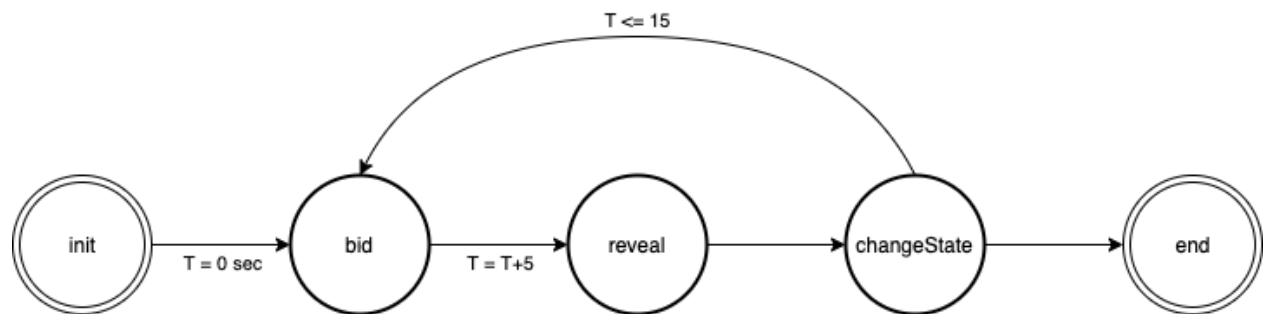
Unified Modelling Language Diagrams

Any problem can be well described using visual representation. In this phase, UML diagrams allow us to see the overall purpose and working of the *BidPoint-Dapp*. The various actors, roles, rules and work flow of the problem are described as follows.

▪ Use Case Diagram:



■ Finite State Machine Diagram:



States:

- **init**: initialization phase
- **bid**: phase to allow bidders to bid
- **reveal**: phase to allow bidders to view the interim result
- **changeState**: phase to allow House to start and stop the bidding
- **end**: phase to declare the final winner

Note:

- If there are multiple common bids then they are resolved using the time-stamp history internally by blockchain.
- '**T**' in the FSM denotes the time gap (in seconds) between the two phases (**bid** and **reveal**).

▪ **Contract Diagram:**

BidPoint
address house address highestBidder address interimWinner struct Bid enum Phase uint highestBid uint interimWinningBid uint refund uint amount mapping(address => Bid) bids mapping(address => uint) count mapping(address => uint) depositReturns
modifier onlyBeneficiary modifier validState modifier onlyBidders
constructor() changeState(Phase x): onlyBeneficiary bid(bytes32 bidValue) payable onlyBidders placeBids(address bidder, uint value) internal submitBids(uint value, bytes32 secret) withdraw() bidEnd()
event BidEnded(address winner, uint highestBid) event BiddingStarted() event RevealStarted() event AuctionInit() event FinalPhase()

Variables:

- house: Contract deployer
- highestBidder: final address of the winner
- interimWinner: address of the winner in respective cycles
- Bid: structure to keep details of the bidders
- Phase: enum to keep track of the phase
- highestBid: final winning bid

- `interimWinningBid`: winning bid in respective cycles
- `refund`: refund amount for the non-winning bids
- `uint amount`: amount for each user
- `bids`: mapping addresses and 'bid'
- `count`: mapping to keep count of state number
- `depositReturn`: keep track of deposit returns for bidders

Modifiers:

- `onlyHouse`: used for functions only accessible to the deployer
- `validState`: used to keep check on the state during execution of `stateChange()`
- *`onlyBidders`: used for functions only accessible to bidders*

Functions:

- `constructor()`: initialization
- `changeState()`: used by the House to change the state
- `bid(bytes32 bidValue)` payable: used by the bidders to place a bid
- `placeBids(address bidder, uint value)` internal: internal function called in `bid()`
- `submitBids(uint value, bytes32 secret)`: declare winner (interim winner and final winner)
- `withdraw()` internal: used to transact after the bid is over
- `bidEnd()`: declare winner in end phase

Events:

- `BiddingStarted`: indicates bidding session has started
- `RevealStarted`: indicates the bids are in submission phase
- `AuctionInit`: indicates initialization phase
- `FinalPhase`: indicates the last phase
- `BidEnded(address winner, uint highestBid)` : indicates the termination of the smart contract

Phase 3

Smart Contract Development

This smart contract deals with competitive bidding through multiple states. Every phase reveals an interim winner so the bidders know how much to bid in the next phase. Each phase is handled by the beneficiary. Conflicts are resolved internally by the blockchain. After 3 phases the beneficiary ends the auction. The non-winners get the refund after the end of 3 phases after clicking on the '**withdraw**' button. The winners win an item and do not receive refunds.

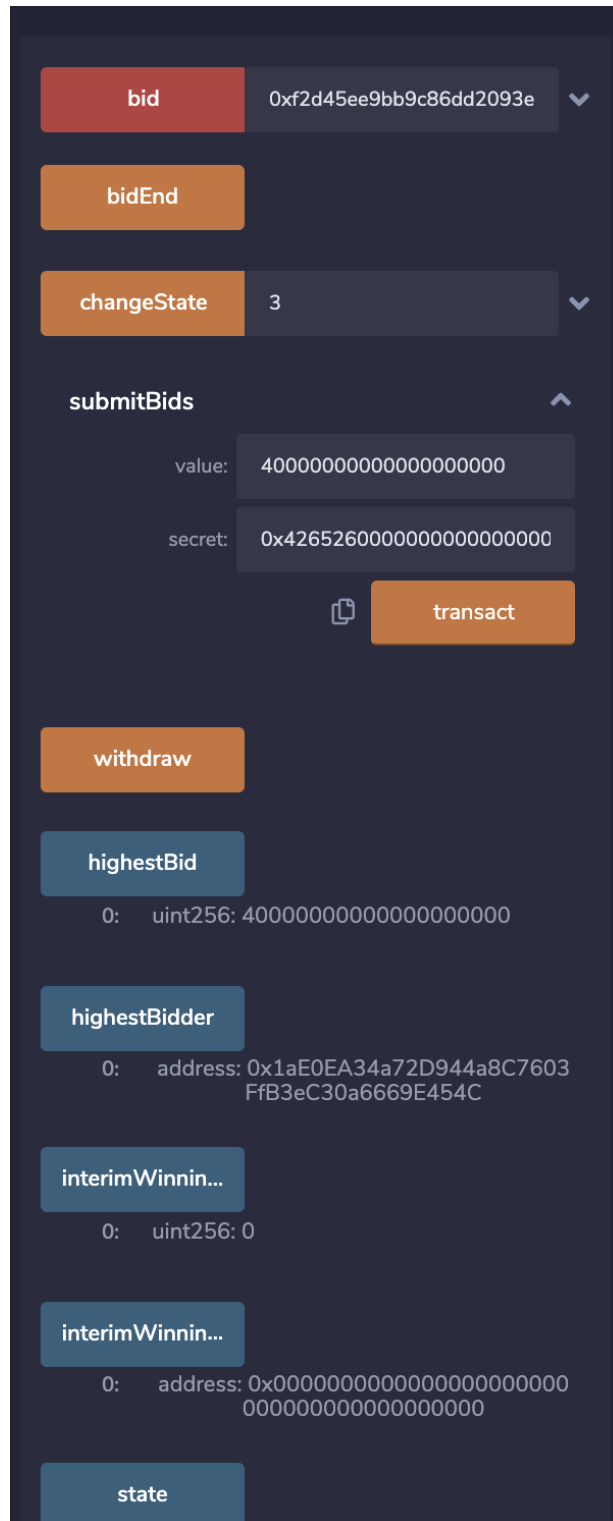
▪ Execution Flow:

- 1) Initial default phase as *bidding* phase
- 2) Select *user 1* place blind bid (repeat for all the users)
- 3) Select house and change phase to *submitBids*
- 4) Submit bids using *secret* and *bid* value (this will reveal interim winners after each submission)
- 5) Repeat steps 2 to 4 for 3 cycles
- 6) Select house and change phase to *Done*
- 7) Final winner is revealed with winning bid
- 8) Withdraw non-winning bids using the *withdraw* button for each user
- 9) Select house and end the bidding by clicking *bidEnd* button

▪ Smart Contract Development Snapshots

The following snapshots depict the working of the smart contract:

- Smart Contract Deployment UI:



A screenshot of a smart contract deployment interface. The interface is dark-themed with orange and blue buttons. It features several input fields and buttons for interacting with the contract. The 'bid' field is highlighted in red. The 'submitBids' section includes 'value' and 'secret' inputs, a 'transact' button, and a copy icon. Below this are buttons for 'withdraw', 'highestBid', 'highestBidder', 'interimWinnin...', and 'state'. The 'highestBidder' and 'interimWinnin...' sections show address information.

Function	Input/Output
bid	0xf2d45ee9bb9c86dd2093e
bidEnd	
changeState	3
submitBids	
value:	4000000000000000000
secret:	0x4265260000000000000000
transact	
withdraw	
highestBid	
0: uint256:	4000000000000000000
highestBidder	
0: address:	0x1aE0EA34a72D944a8C7603FfB3eC30a6669E454C
interimWinnin...	
0: uint256:	0
interimWinnin...	
0: address:	0x00
state	

- **Variables and data structures**

The snapshot below shows the various variables and data structures used in the contract.

```
struct Bid {
    bytes32 blindedBid;
    uint deposit;
}

// state will be set by beneficiary
enum Phase {Init, Bidding, Reveal, Done}
Phase public state = Phase.Init;

//total addresses used
address payable house;           //owner
address public highestBidder;     //final winner
address public interimWinningBidder; //interim winners in each cycle

//total mappings used
mapping(address => Bid) bids;
mapping(address => uint) count; //keeps count of number of cycles a particular user has completed
mapping(address => uint) depositReturns;

//total variables used
uint refund = 0;
uint amount = 0;
uint public interimWinningBid; //interim winner in each cycle
uint public highestBid = 0;    //final winning bid
```

All the variables and functions are explained in the contract diagram in **Phase 2**.

- **Modifiers and Functions**

Various functions and modifiers used in the contract are depicted in the snapshot below,

```
//total modifiers used
modifier validPhase(Phase reqPhase)
{...}

modifier onlyHouse()
{...}

modifier onlyBidders(){...}

//Initialization
constructor( ) public{...}

//total Functions

//function used to change states/phases during bidding
function changeState(Phase x) public onlyHouse{...}

//function to store bids and deposit values
function bid(bytes32 blindBid) public payable validPhase(Phase.Bidding) onlyBidders{...}

//function to submit Bidders' bids
function submitBids(uint value, bytes32 secret) public validPhase(Phase.Reveal) onlyBidders{...}

//internal function to place requested bids
function placeBid(address bidder, uint value) internal returns (bool success){...}

//function withdraw a non-winning bid
function withdraw() public validPhase(Phase.Done){...}

//function to end the auction and send the highest bid to the house.
function bidEnd() public validPhase(Phase.Done){...}

}
```

- **Balance details**

The below data depicts various users and the House (a.k.a. Beneficiary). The highest bidder in this case is the user 3 with balance ~60. The non-winning bidders received the money back after withdrawing the amount (user 1 and user 2).

Multiple users can participate in the event. For this example, I have used 3.

```
0x17F...8c372 (139.9999999999998821452 ether)
0x5c6...21678 (99.9999999999999491078 ether)
0x03C...D1Ff7 (99.9999999999999626078 ether)
0x1aE...E454C (59.999999999999962449 ether)
0x0A0...C70DC (100 ether)
0xCA3...a733c (100 ether)
0x147...C160C (100 ether)
0x4B0...4D2dB (100 ether)
0x583...40225 (100 ether)
0xdD8...92148 (100 ether)
```

By default, all the users (including the House) have 100 ethers.

The first user in the above snapshot is the beneficiary holding the transaction amount of the winner after the end of the Bidding session.

The fourth user is the winner that has transferred the money to the House.

The second and the third users are the non-winning bidders, hence, they receive the refunds.

Phase 4

Security and Events

Every application must be secure and should maintain privacy.

- Security

In this Dapp, I have used the *kaccak256* hashing function in order to secure the online bidding process. The *kaccak256* hashing function hashes the bid into **256 bits** or **32 bytes** hashes. Each bid is hashed using a unique password creating a unique hash for each user, making it a secure transaction. This hashed bid is then validated in the *submitBids()* function in order to check the authenticity of the bid and know for a reason that it is from a valid user. For example:

Password

```
0x4265260000000000000000000000000000000000000000000000000000000000
```

First bid

20000000000000000000000000

```
# : 0xab7d85c7fe09d8882710e188e34e627259a9858bf3d30d6cd3db8f9ee4a81d5
```

Second bid

30000000000000000000

: 0x44b4d6f86f3a6ffc2ca7766341b64560d0b322aa2def31b44614a535c599daac

Third bid

35000000000000000000

```
# : 0x2ab6c5c4248020ae8a7fb197df6e9f352e6efe9ffd3876a0cc03978a15a86c20
```

These are the few transaction amounts used in the sample testing. These are hashed using the password provided at the top.

Syntax for hashing:

```
keccak256(abi.encodePacked(value, secret))
```

where,

- value: transaction amount
- secret: hashing password

Execution flow in security layer:

- Initially, the user bids using the hashed value of the amount.
- This hashed value is entered in the *bidding* phase in each cycle.
- After the bidding phase is over, in the *submitBids()* phase, the hashed amount is then compared with the newly generated hashed value using the actual value and the secret hash key.
- If they match, then only the transaction proceeds, else it is reverted.

This security layer allows each user to place a bid in a unique fashion and hence no other user can bid from any other account.

Following is the snapshot of usage of *keccak256* in the contract:

```
function submitBids(uint value, bytes32 secret) public validPhase(Phase.Reveal) onlyBidders{
    refund = 0;
    Bid storage bidToCheck = bids[msg.sender];

    if (bidToCheck.blindedBid == keccak256(abi.encodePacked(value, secret))){
        refund += bidToCheck.deposit;
        if(value > interimWinningBid){
            interimWinningBid = value;
            interimWinningBidder = msg.sender;
        }

        if(count[msg.sender] == 3){
            if (bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value)){
                    refund -= value;
                }
            }
        }
    }
    else revert();
    msg.sender.transfer(refund);
}
```

Validation: In the above code the *submitBids()* function will only work if a valid *keccak256* hash is sent to the function along with the transaction amount.

■ Events

Events are used to store arguments that are passed in a transaction and emitted are emitted in transaction logs. Events are used to notify the application the various changes made to the contract. Events are defined and emitted as follows:

```
//total events used
event BidEnded(address winner, uint highestBid);
event BiddingStarted();
event RevealStarted();
event AuctionInit();
event FinalPhase();
```

```
// Emit appropriate events for the respective phases
if (state == Phase.Reveal) emit RevealStarted();
if (state == Phase.Bidding) emit BiddingStarted();
if (state == Phase.Init) emit AuctionInit();
if (state == Phase.Done) emit FinalPhase();
```

Events are used to suggest the what phase is currently being executed.

All the events used in the contract are as follows,

- BiddingStarted: indicates bidding session has started
- RevealStarted: indicates the bids are in submission phase
- AuctionInit: indicates initialization phase
- FinalPhase: indicates the last phase
- BidEnded: indicates the termination of the smart contract

Few examples from testing are as follows:

1) *BiddingStarted event*

```
logs [ { "from": "0xe2dfc07f329041a05f5257f27ce01e4329fc64ef", "topic":
      "0x02c124e22ee7da1c9905bbb317cf67658c1cc3ealc0953b1633f85d0b5c281d9", "event": "BiddingStarted",
      "args": { "length": 0 } } ]
```

2) *RevealStarted event*

```
logs [ { "from": "0xe2dfc07f329041a05f5257f27ce01e4329fc64ef", "topic":
      "0x3bcc93d35219aa0512d7c0d1ca10b0231f473e3d5f9396877d1b7215ee0ab3d5", "event": "RevealStarted",
      "args": { "length": 0 } } ]
```

3) *BidEnded* event

```
logs      [ { "from": "0xe2dfc07f329041a05f5257f27ce01e4329fc64ef", "topic":  
      "0x4f5f8574ff23a6afac7e2def0fd2f109a37e9c10c58972cf2900352a8d254beb", "event": "BidEnded", "args":  
      { "0": "0x1aE0EA34a72D944a8C7603FfB3eC30a6669E454C", "1": "400000000000000000", "winner":  
      "0x1aE0EA34a72D944a8C7603FfB3eC30a6669E454C", "highestBid": "400000000000000000", "length": 2 }  
    } ]
```

These are the few events used in the smart contract for this *BidPoint-Dapp*.

Note:

- Execution flow for the code has been updated in *Phase 3*
- All the relevant diagrams have been updated
- Events have been added in the previous phases is relevant sections

Phase 5

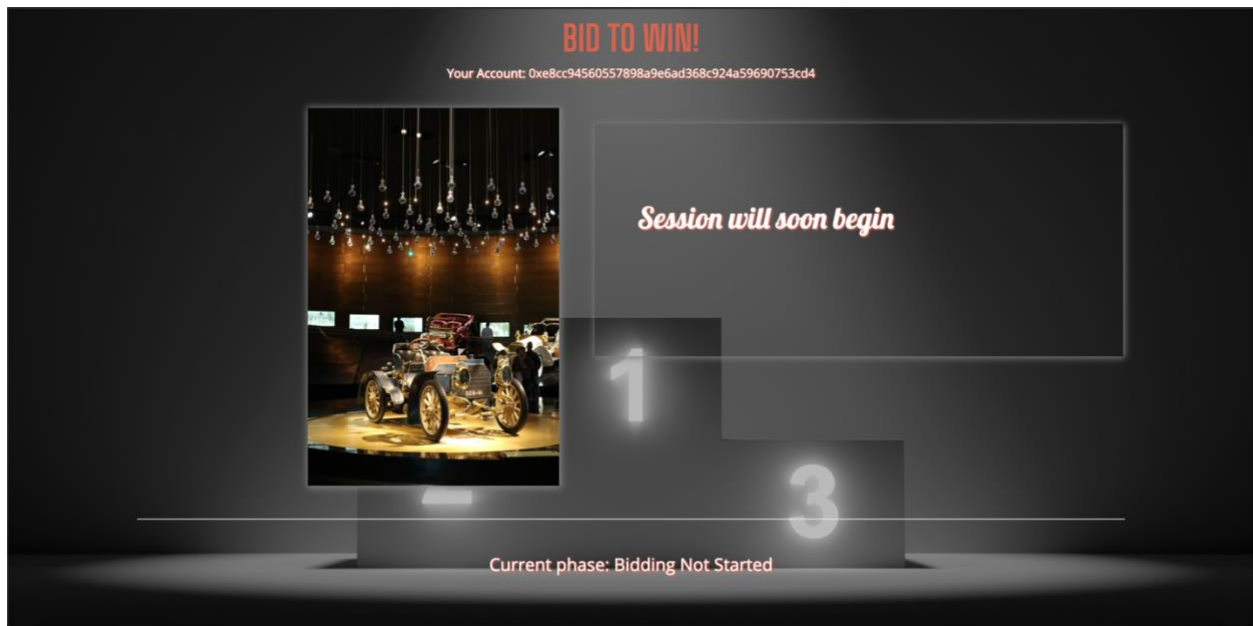
Web components of the Dapp

Every application is incomplete without the front end. This phase deals with developing the front-end and integrating it with the previously developed smart contract. The features of the smart contract are accessed using *web3Provider*. By the end of this phase I develop a full-stack Dapp.

The *BidPoint-Dapp* is a competitive market place. There are 4 phases and 3 cycles of bidding. The 4 phases are *init*, *start bidding*, *submit/reveal bids*, and *end bid*. The execution flow is as illustrated in *phase 3*.

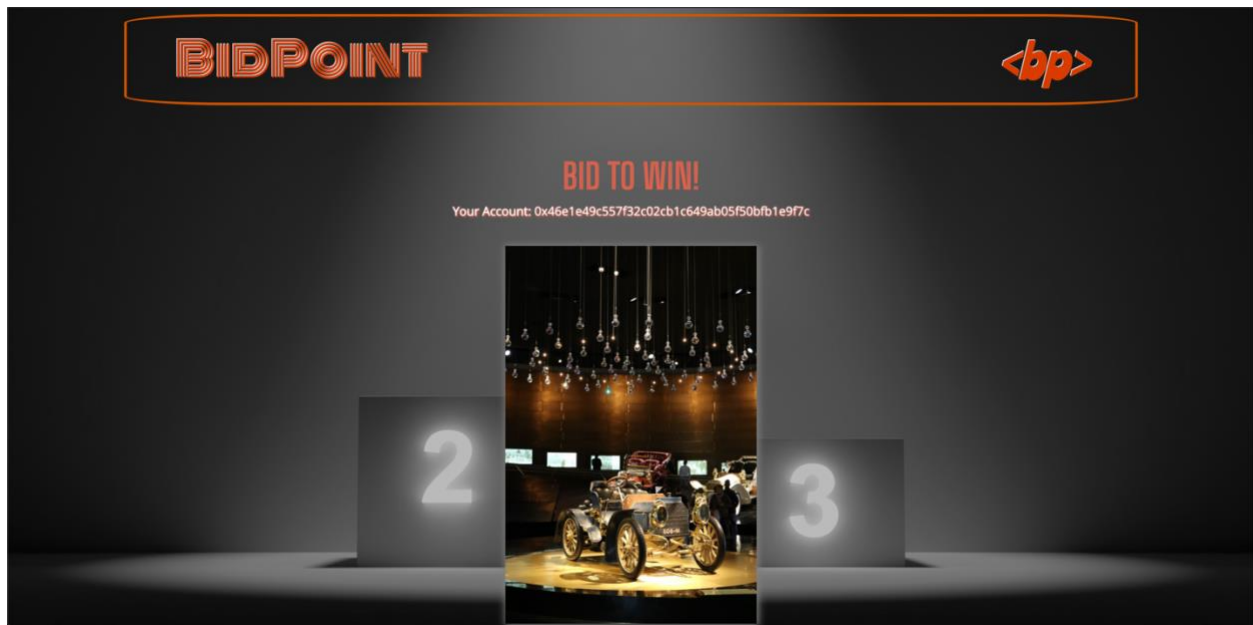
▪ UI web components:

- Home Screen (Bidders)



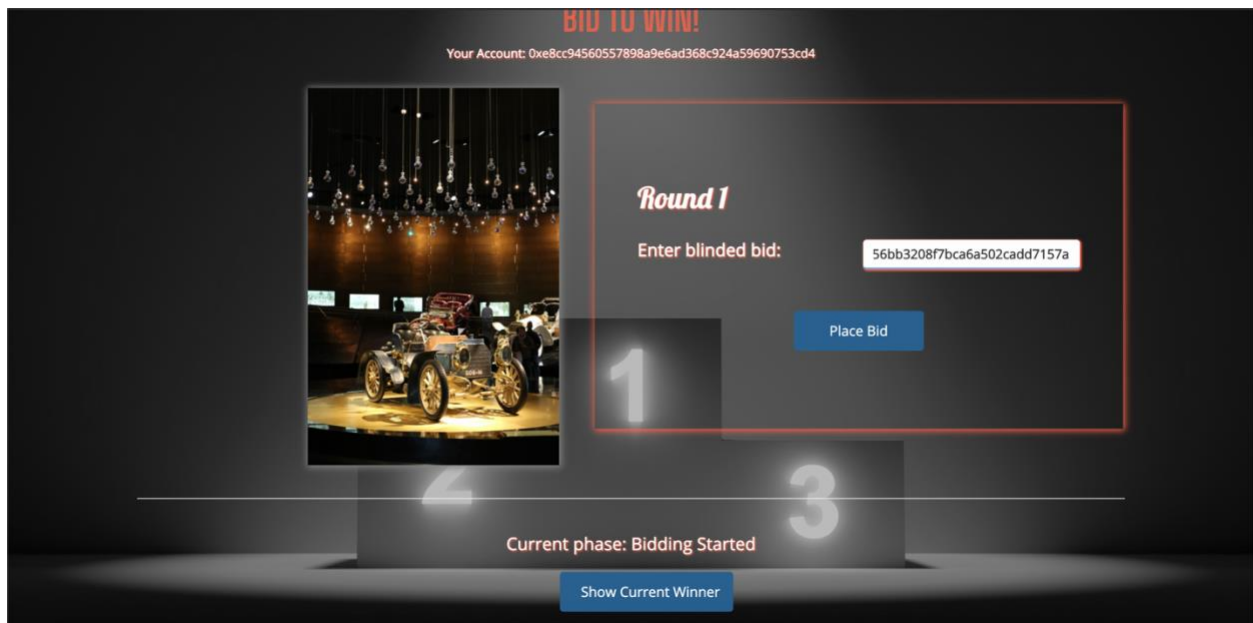
This is the first screen that the Bidders visit. The image at the center is the object for the bidding session. The bidders also can see the current phase and a message for their convenience.

- Home Screen (House)



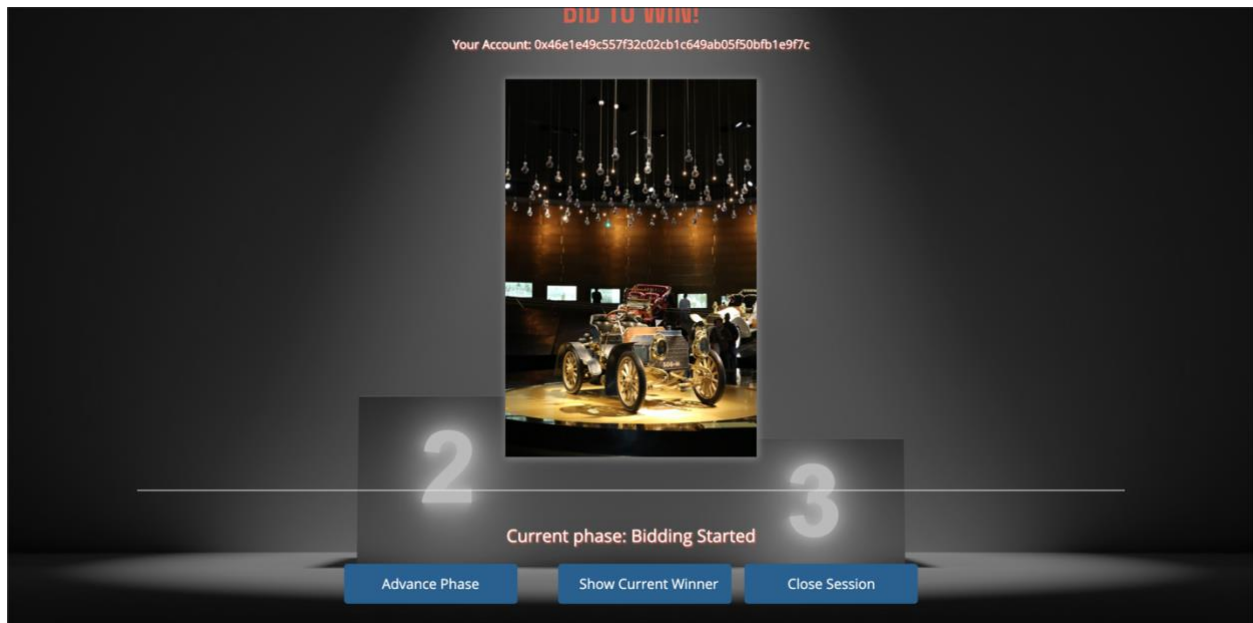
This is the first screen that the House/Beneficiary visit. The image at the center is the object for the bidding session.

- Phase 1: Start Bidding (Bidders)



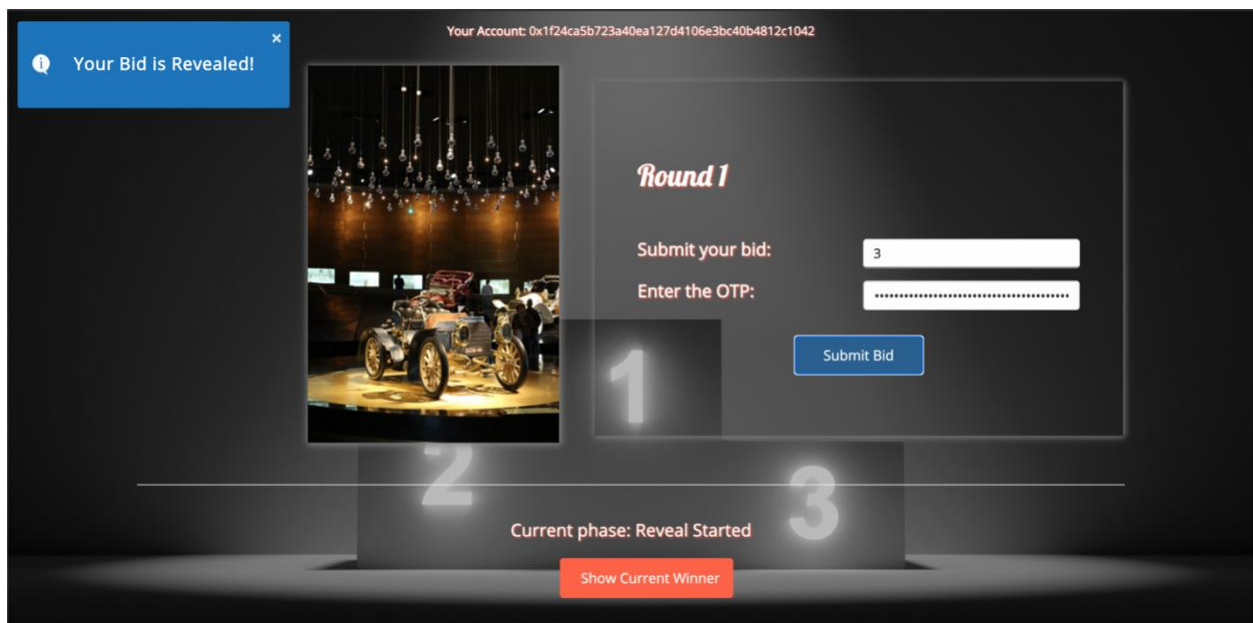
This is the bidding page for the *bidders*. The bidders can place their blinded bid for the object. Since, this is the first round there is no deposit amount and the bidders bid in order to get insights about the bidding battle. After this phase we proceed to the *submit/reveal bid* phase.

- **Phase 1: Start Bidding (House)**



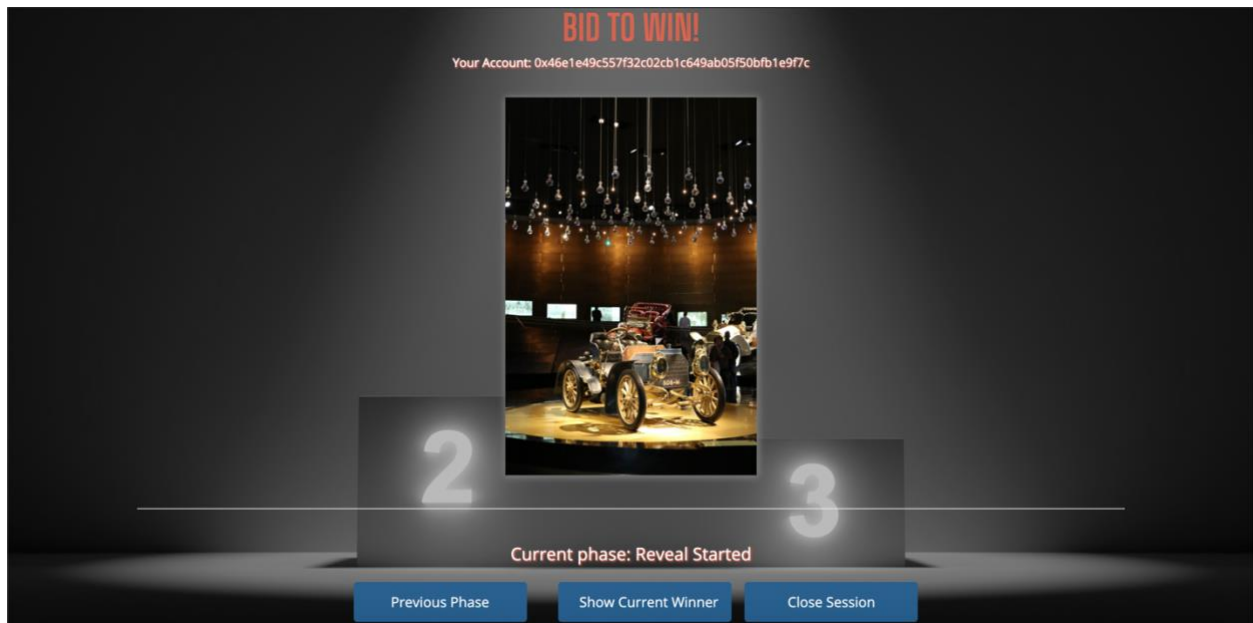
This is the bidding page for the House. Here, the House can change phases, reveal the current highest bid and close the session. On the top the account number of the current user is displayed and at the bottom the current phase.

- **Phase 2: Submit bids (bidders)**



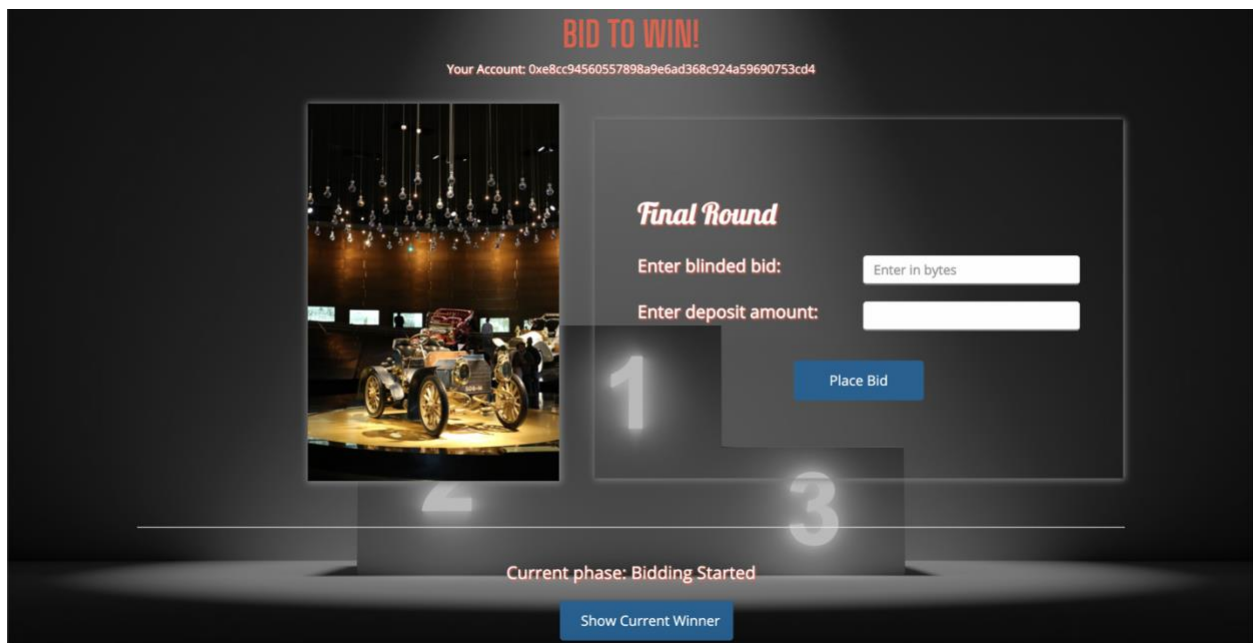
In this phase the bidders can submit their bids for the auction after the validation is done by the smart contract. The bidders can check the temporary winner at any point in the reveal phase using the highlighted button at the bottom. This gives the bidders the idea about the stakes. After the reveal is complete a notification pops up at the top left corner.

- **Phase 2: Submit bids (House)**



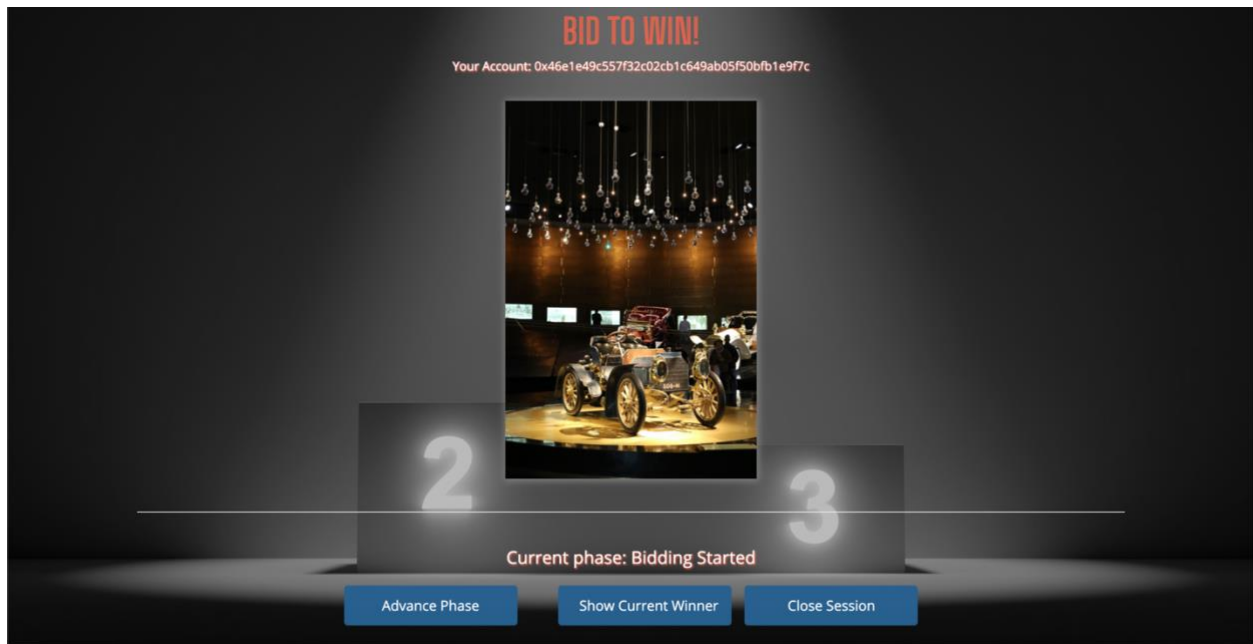
This is the submit/reveal bid phase for the House. The house can now only go to the previous stage. The advance phase button appears and continues to stay after the 3 cycles of the bidding session are complete.

- **Phase 1: Continue bids (Bidders) (Final Cycle)**



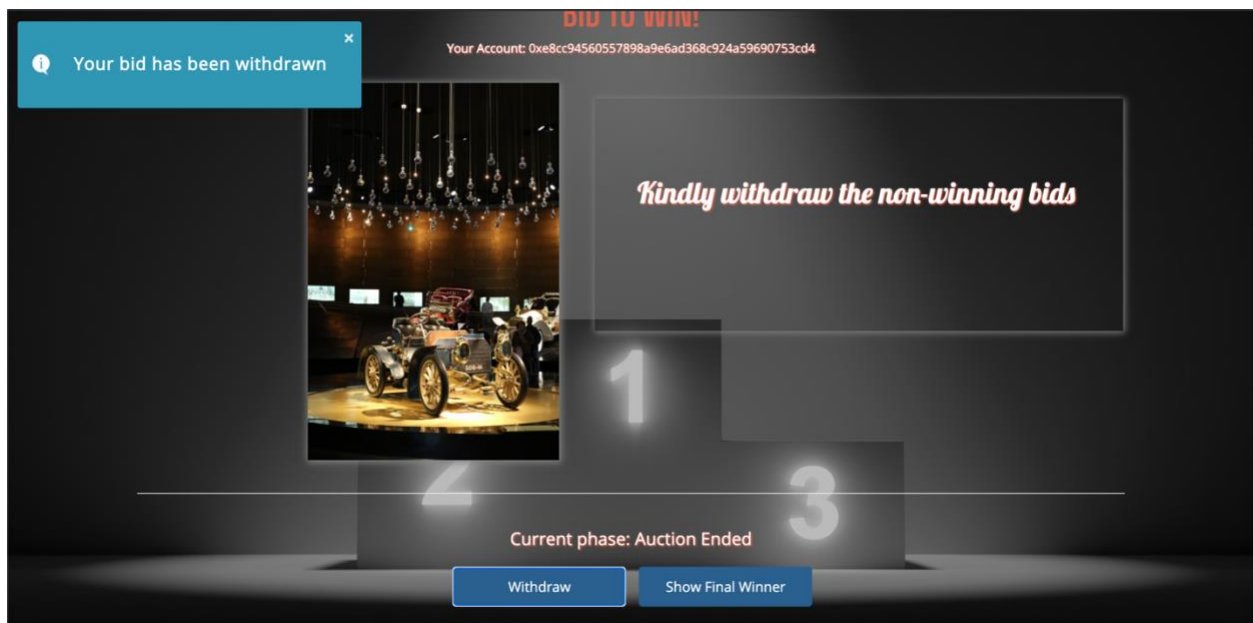
The bidders have to place a deposit amount in the final phase of the bidding session. This text field will only appear in the last cycle of the session. This amount will be deducted using the smart contract.

- **Phase 1: Continue bids (House) (Final Cycle)**



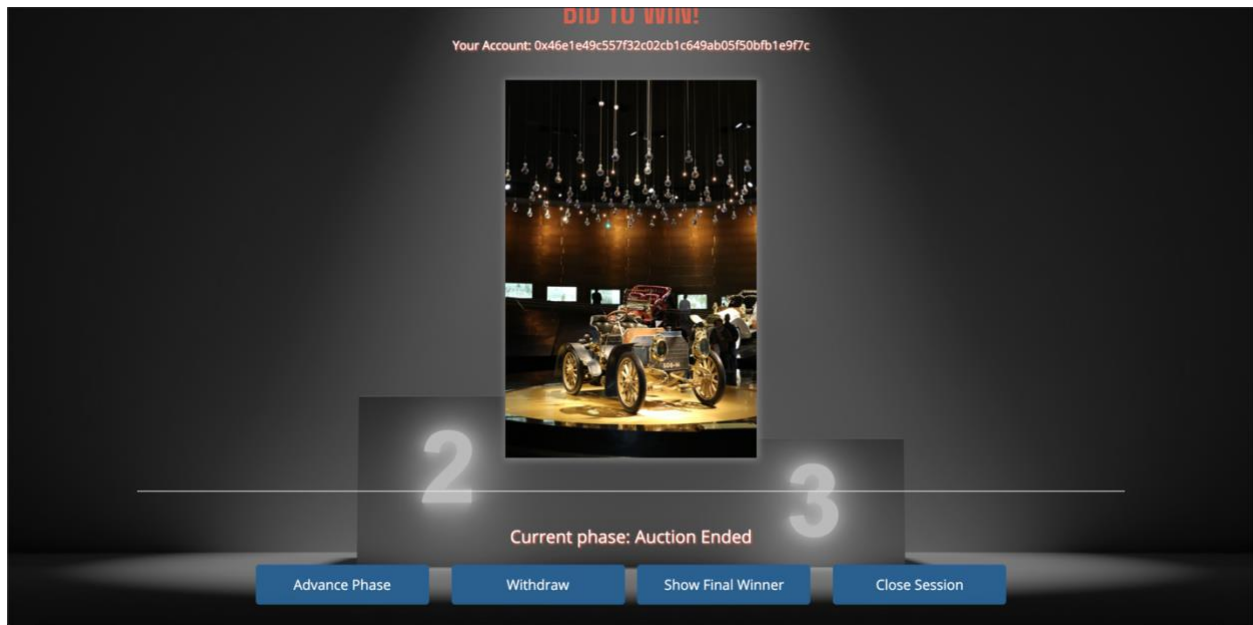
During the final cycle of the bidding session the House can only move forward to the next session by clicking the *Advance Phase* button or close the bidding session.

- **Phase 3: Bid End (Bidders)**



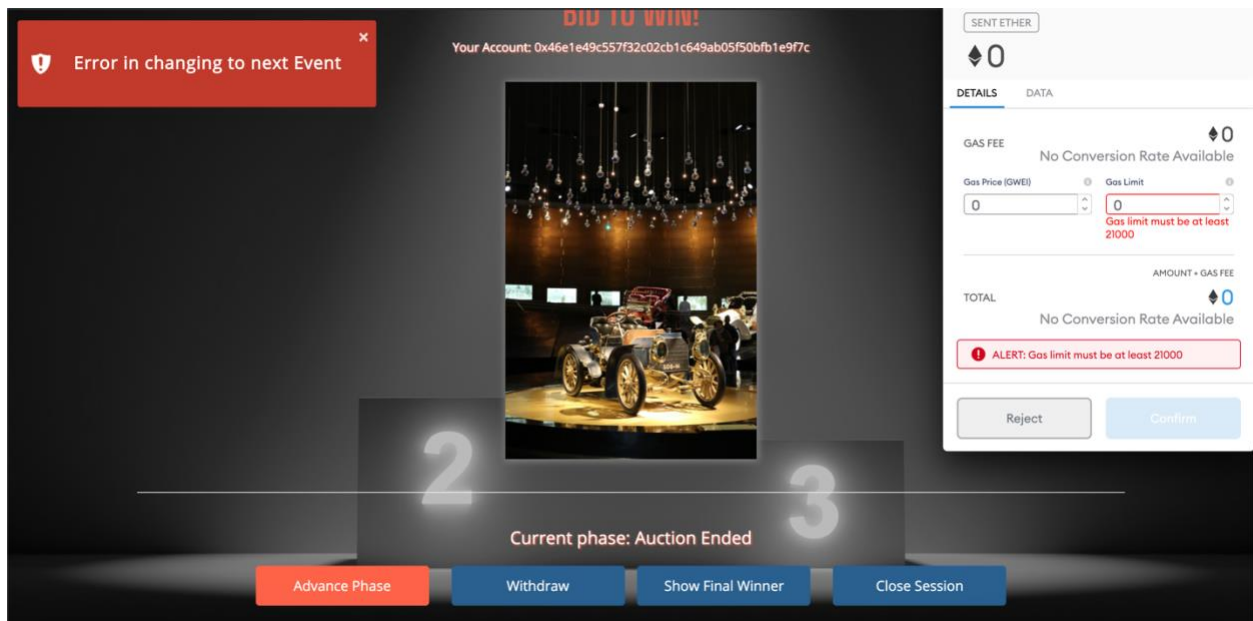
After the end of the bidding session the bidders get a message asking the non-winning bids to withdraw the escrow before the smart contract is either destroyed or a new session begins.

- **Phase 3: Bid End (Bidders)**



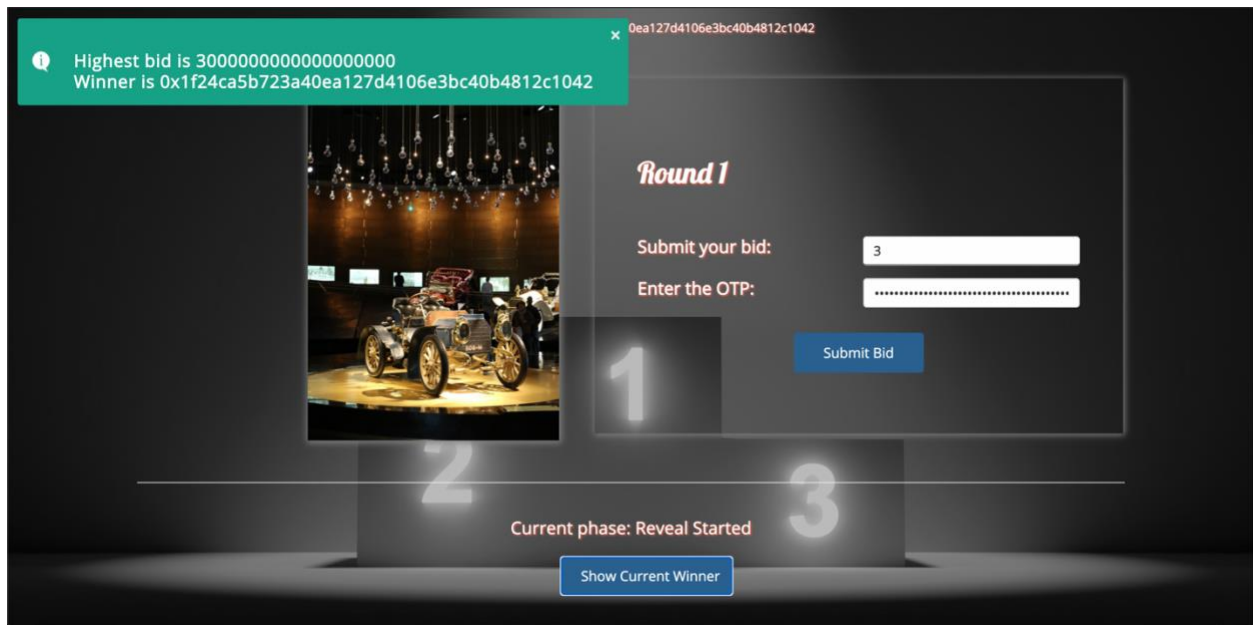
This is the final cycle page for the House. The house can *Advance Phase* to a new session or *Close Session* and destroy the smart contract.

- **Event After Session End:**



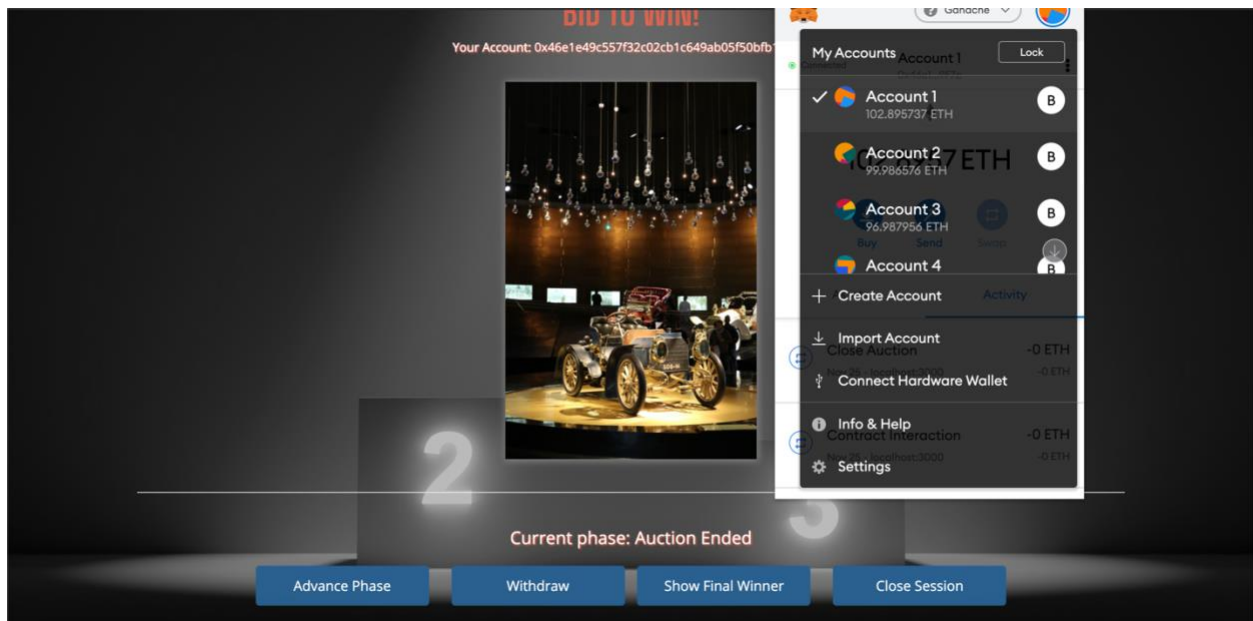
Error cases are handled using *Toast*. Majority of the error cases are taken care by hiding and disabling the buttons as per the phases. The edge cases are also handled in the smart contract as a safety net.

- **Interim Winner Toast:**



The notification on the top left corner is the winner of the interim cycles. A similar notification appears for the final phase as well.

- **Final Values in Accounts:**



The first account is the House that has received the ethers from the winning bidder. The second account is the non-winning bidder and the third account is the winner. The non-winners use the *Withdraw* button to get the ethers refunded.

Note: Execution flow for the code present in *Phase 3*