

# Peer to Peer Systems and Blockchains

## final project

Giacomo De Liberali - 580595  
Development of a dApp for Smart Auctions  
on the Ethereum blockchain

July 21, 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Architecture</b>	<b>2</b>
2.1	Actors . . . . .	3
2.1.1	AuctionHouse . . . . .	3
2.1.2	Root user . . . . .	4
2.1.3	Owner . . . . .	4
2.1.4	Seller . . . . .	4
<b>3</b>	<b>dApp Operation flow</b>	<b>4</b>
3.1	Creation of an AuctionHouse . . . . .	5
3.2	Creation of a new dutch auction . . . . .	6
3.3	Creation of a new vickery auction . . . . .	7
3.4	Dutch auction details . . . . .	8
3.5	Vickery auction details . . . . .	10
3.6	Test the contract in a local network . . . . .	11
<b>4</b>	<b>Events</b>	<b>12</b>
<b>5</b>	<b>Deploy on real network</b>	<b>13</b>

# 1 Introduction

This project involves the implementation of a dApp to allow people to interact with the smart contracts created in the previous course assignment.

## 2 Architecture

The front end of the dApp is built in JavaScript, allowing any user to interact with the smart contracts through its web browser. Since the reactivity of the user interface is important, I chose to exploit the Angular framework to simplify the management of the asynchronous operations. Angular indeed can detect the end of such tasks and automatically update the interface according to their result. Exploiting the so-called *two-way data binding*, that connects the DOM with the data model behind it, any changes to the model will instantly be reflected, without any need to manually reload the content. Angular introduces also the benefit of *Typescript* adoption, a typed version of JavaScript, that make the life easier to developers thanks to the type-checking performed by its compiler.

To interact with the EVM the front end needs a bridge; the choose was between *Web3.js* or *ethers.js*. After some troubles with the first one, I adopted the latter which is, from my point of view, more robust and better supports Typescript ecosystem. *ether.js* allows to interact with Ethereum Blockchain and its ecosystem over Metamask which is, in turn, a bridge to the Ethereum network.

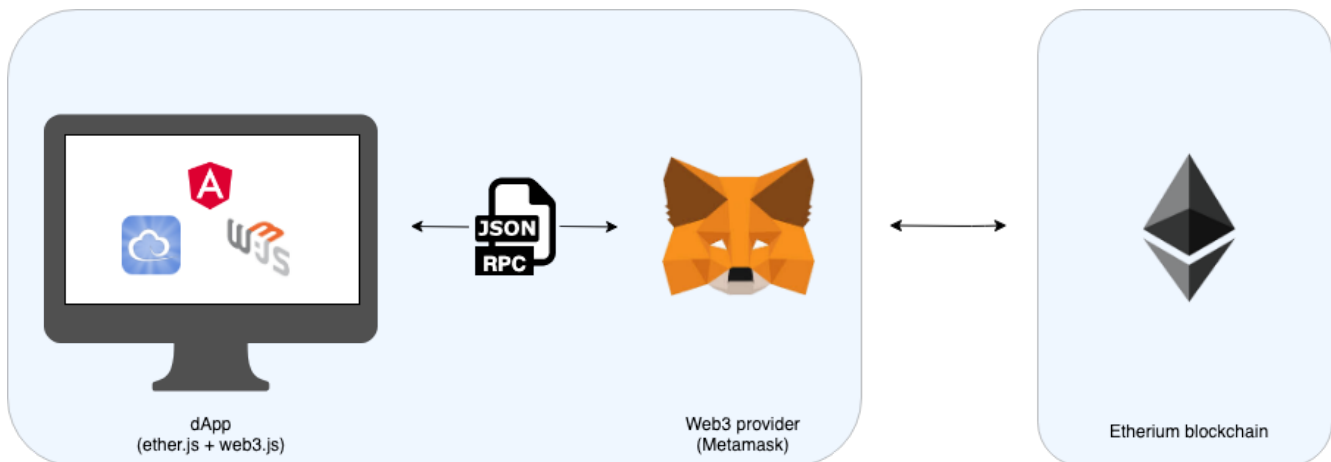


Figure 1: dApp components interaction architecture

## 2.1 Actors

The contracts from the previous assignment have been slightly modified to fit all scenarios that the dApp requires.

### 2.1.1 AuctionHouse

We need to notify all users when a new auction is deployed to allow them to participate in it. But we can not simply subscribe to a general event on the whole blockchain since this will be unsustainable on a real network. To allow this use case I created a new central actor, called *AuctionHouse*, that aggregates auctions and emits an event whenever a new auction is deployed under its jurisdiction. In this way, a user will need to know only the address of the house to receive a notification whenever a new auction is created. The house thus allows to create auctions under its control and store them in a public property readable by everyone. Ideally, a house could host only auctions regarding a certain topic, allowing a user to subscribe by interest.

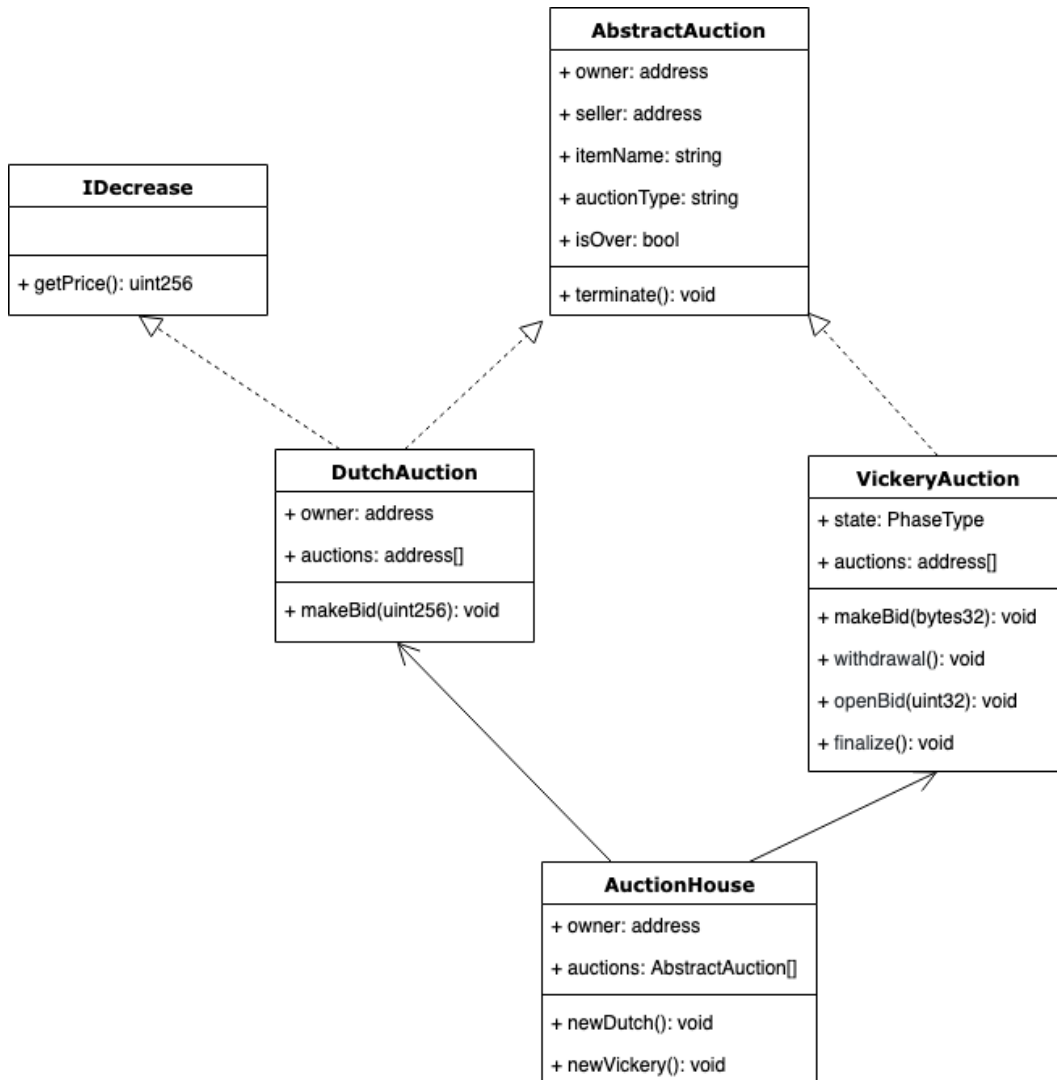


Figure 2: System's actors

From previous assignment we can also notice the presence of another new contract, *AbstractCon-*

*tract*, that contains the common metadata between the dutch and vickery auctions.

### 2.1.2 Root user

The root user is the actor that is in charge of deploy the *AuctionHouse*. It has only to deploy it the first time, then each user that wants to create a new auction under the jurisdiction of that house can simply call its methods, *newDutch()* or *newVickery()* from the dApp correspondent page.

### 2.1.3 Owner

The owner of a contract is the address of the account used to deploy the contract under a certain house.

### 2.1.4 Seller

The seller of a contract is the the address of the account that will receive the money after the action will terminate successfully.

## 3 dApp Operation flow

Before navigate to the dApp ensure to have Metamask installed on the browser or you will get a notification asking for it. Once landed on the first page, a popup asking you the permissions to connect to Ethereum account will be shown.

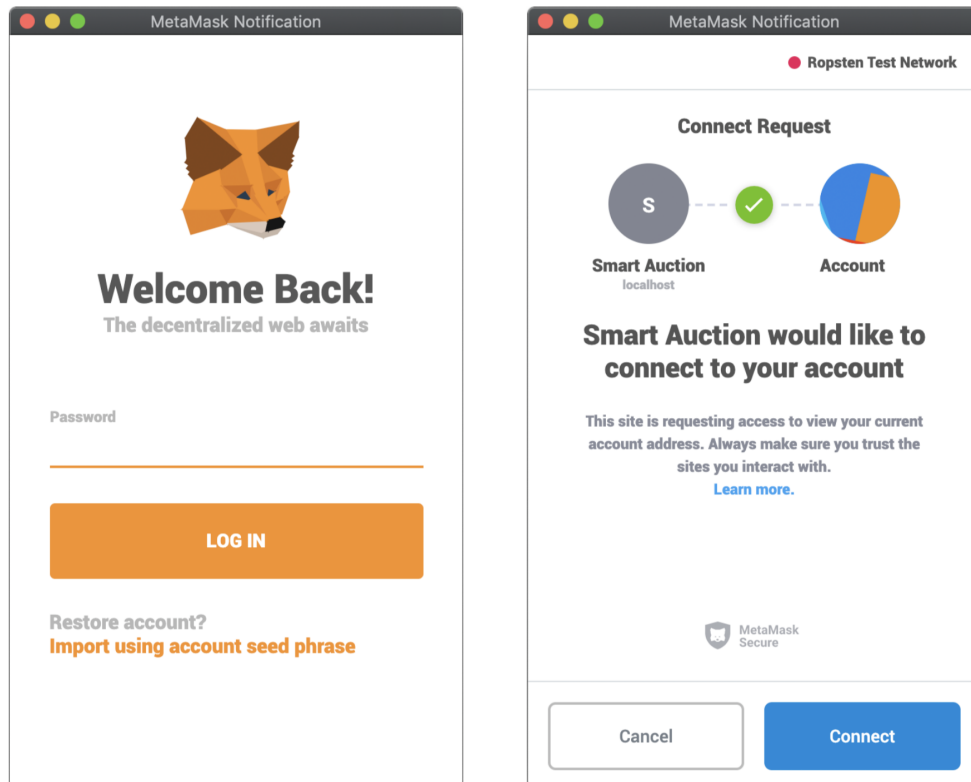


Figure 3: dApp components interaction architecture

### 3.1 Creation of an AuctionHouse

This is the first page shown, and it simply contains a input and a button. You can select an existing house and load it from the address or either create a new one. The deployment of a new one should be done by the root user.

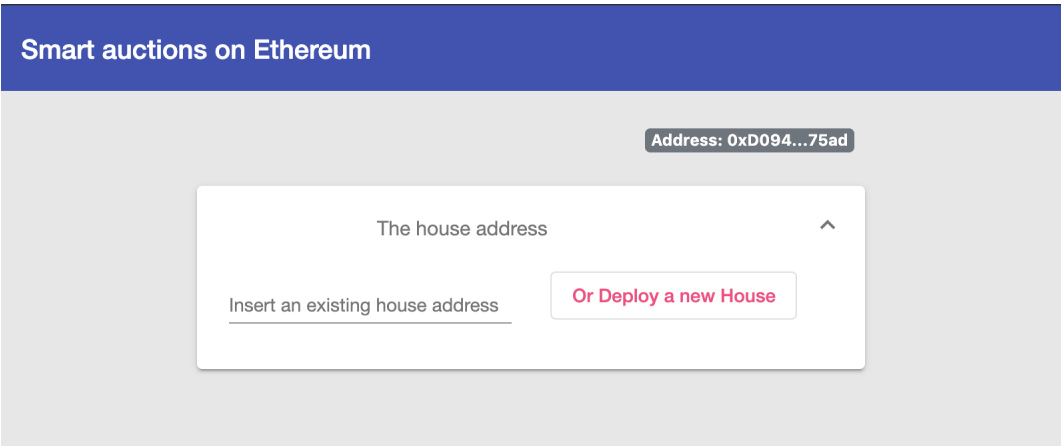


Figure 4: Creation of a new *AuctionHouse*

Once deployed or loaded a house from address, a list of all auctions under the jurisdiction of the house will be shown, together with two buttons that will allow the creation of new contracts under the current house.

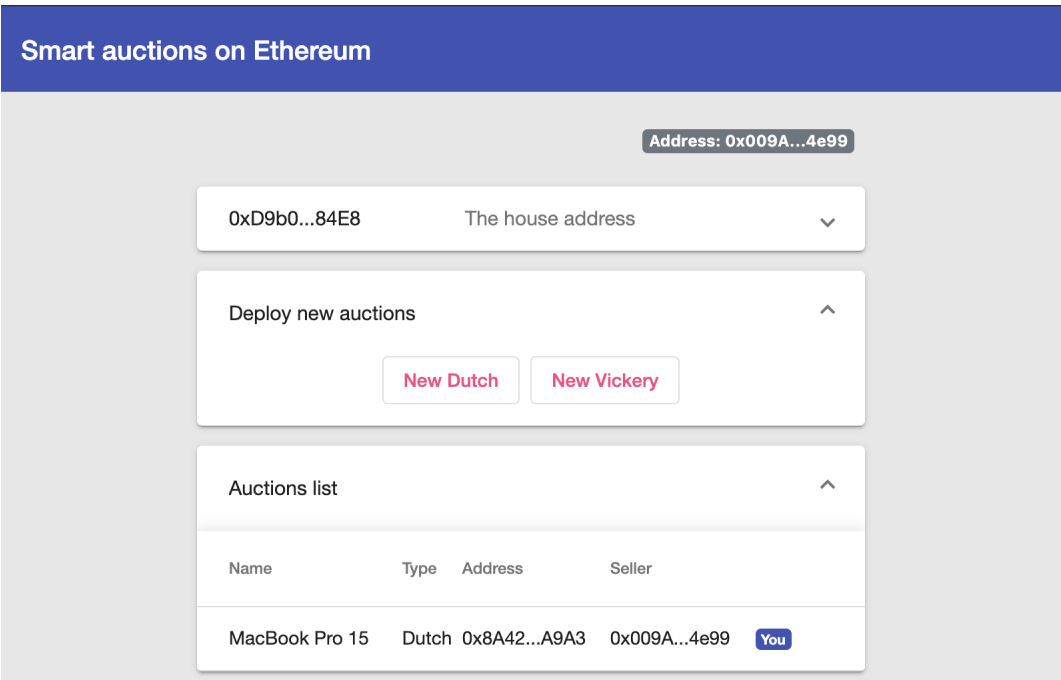


Figure 5: State once selected or deployed a house

In the table you can see a list of auctions, with the possibility to click one of them to see more details and interact with it.

### 3.2 Creation of a new dutch auction

To create a new dutch auction we need first to select a price strategy, either by selecting an existing address or by deploying it. We can then provide all needed information. The owner of the auction will be the current Metamask's selected account, while the seller must be specified in the relative form field, also if it is pre-populated with the current account.

Smart auctions on Ethereum

Address: 0x009A...4e99

New Linear strategy

Create and deploy a new linear strategy for the Dutch auction

Strategy address

Deploy strategy

New Dutch auction

Create and deploy a new dutch auction

Item name

MacBook Pro 15

Seller address

0x009A92E55409F12672fcC078E3

Reserve price

1000

Initial price

2000

Last for blocks

100

Cancel

Deploy

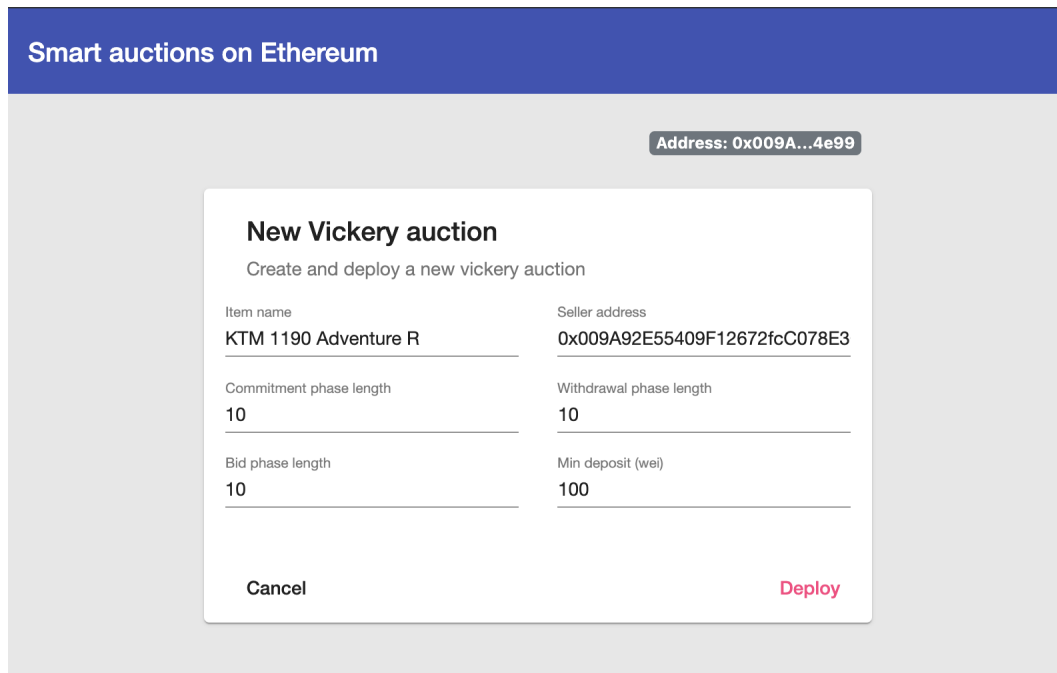
Figure 6: Creation of a dutch auction

Once deployed, all users that have the same house populated will receive a notification and the table items in figure 5 will be updated.

6

### 3.3 Creation of a new vickery auction

To create a new vickery auction we need only to complete the form, since that it has not other dependencies like the dutch one.



The screenshot shows a web interface titled "Smart auctions on Ethereum". At the top right, there is a button labeled "Address: 0x009A...4e99". Below this, a form titled "New Vickery auction" is displayed. The form includes the instruction "Create and deploy a new vickery auction" and several input fields:

Item name	Seller address
KTM 1190 Adventure R	0x009A92E55409F12672fcC078E3
Commitment phase length	Withdrawal phase length
10	10
Bid phase length	Min deposit (wei)
10	100

At the bottom of the form, there are two buttons: "Cancel" and "Deploy".

Figure 7: Creation of a vickery auction

Also in this case, once the auction has been deployed all the users that have the same auction house will receive a notification and the table items will be updated.

### 3.4 Dutch auction details

Once clicked a row in the figure 5 table that correspond to a dutch auction, the user will be redirected on this page. In this case he is a bidder, since that the current address is different both from the owner and the seller. He can indeed make a bid to this auction trying to win it.

Smart auctions on Ethereum

Address: 0x90D4...E7eE

Dutch auction

You can interact with this contract

Contract address

0x8A42e478aC72474497E307c67C

Item name

MacBook Pro 15

State

Open

Current price

1990 wei

Seller address

0x009A92E55409F12672fcC078E3

Owner

0xD9b0e0cBE6b8758d60B8766111

Make a bid

From account

0x90D472dFA5C9f667c2d8d64a7

Value in wei

1990

Make bid

Figure 8: Dutch details from a bidder point of view



If the same user opens again this auction after having submitted a valid bid, the following interface is present, indicating near the title with a green badge, that the auction is closed and that the current account is the winner.

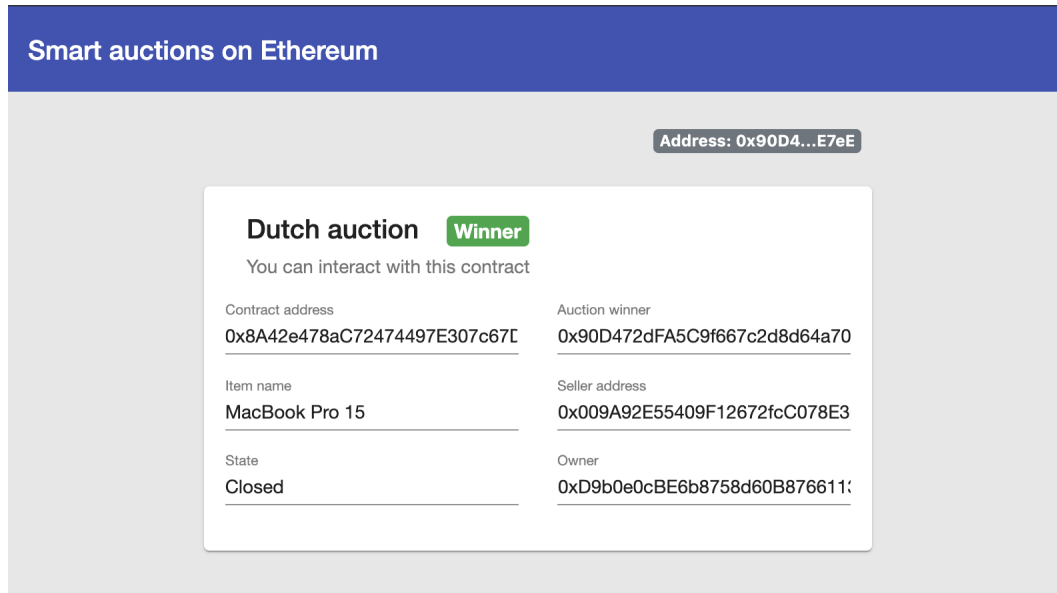


Figure 9: Dutch details from a winner point of view

If the current account is the owner/seller he can also terminate a non-closed auction, as shown in below screen shot.

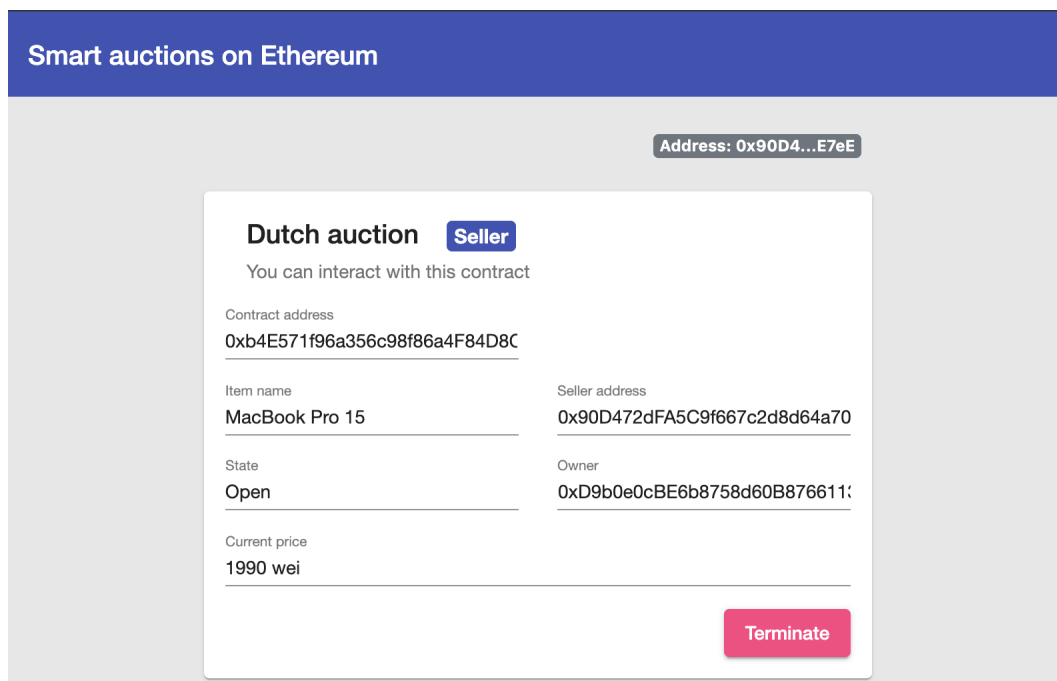


Figure 10: Dutch details from a seller point of view

### 3.5 Vickery auction details

If the row clicked in the figure 5 table correspond instead to a vickery auction, the user will be redirected on this page.

The screenshot displays a web interface titled "Smart auctions on Ethereum" with a dark blue header. Below the header, a grey bar shows the user's address: "Address: 0x90D4...E7eE". The main content area is white and contains two sections. The top section, titled "Vickery auction" with a "Seller" badge and a refresh icon, states "You can interact with this contract". It lists the following details: Contract address (0x614484DCe6D894A25F697b28B), Seller (0x90D472dFA5C9f667c2d8d64a70), Item name (KTM 1190 Adventure R), State (Commitment), and Deposit (100 wei). A pink "Terminate" button is located at the bottom right of this section. The bottom section, titled "Bid", has an upward arrow icon and contains input fields for "Value of your bid (wei)" and "Nonce", with a pink "Make bid" button at the bottom right.

Figure 11: Vickery details from a seller point of view

If the current Metamask's account is the same as the seller of the auction, the badge near the title and the terminate button will be shown. Instead, if the user is simply a bidder, he will have the possibility to submit a blind bid indicating the value and a nonce. The corresponding hash is in-browser generated, as shown in the code snippet present in figure 12. The bidding interface is present only if the contract is in commitment state, otherwise, the corresponding interface is shown to the user (eg. the interface to withdrawal the deposit and open a previous bid indicating the used nonce). After the opening phase, the contract is closed and the owner/seller can finalize it to elect the winner and start the refund process to all other bidders.

```

1  public async makeBid() {
2    try {
3      const hash = ethers.utils.keccak256(
4        ethers.utils.formatBytes32String(this.bid.value + this.bid.nonce)
5      );
6      const tx = await this.contractInstance.makeBid(hash, {
7        value: ethers.utils.bigNumberify(this.vickery.deposit),
8        gasLimit: await this.contractHelper.getEstimateGasFor(this.contractInstance.makeBid)
9      });
10     await this.provider.waitForTransaction(tx.hash);
11     this.snackBar.open("Bid sent", "Ok", { duration: 5000 });
12   } catch (ex) {
13     this.notifyError(ex);
14   }
15 }

```

Figure 12: The function associated to the *makeBid()* function in vickery auctions

### 3.6 Test the contract in a local network

In order to properly test the dApp I needed to await that blocks are mined, but this is quite annoying during testing. To avoid this scenario I created an interface, visible only if a local network is detected, that sends to the EVM the events needed to virtually mine more blocks, as shown in figure 13.

The screenshot shows a web interface titled "Smart auctions on Ethereum". At the top right, there is a user address: "Address: 0x90D4...E7eE". Below this, there are several interactive elements: a dropdown menu showing "0xD9b0...84E8" with the label "The house address", a "Deploy new auctions" button, and an "Auctions list" dropdown. At the bottom, there is a section titled "Manage mined block in network" which contains two input fields: "Current block number" with the value "7" and "Mine this number of blocks" with the value "10". To the right of these fields is a red button labeled "Mine blocks".

Figure 13: Advance blocks if local network detected

## 4 Events

As first event I mention something that is not related to the blockchain but rather to Metamask. Since that the application UI must reflect the current Metamask account, we need to know when this changes in order to properly update the interface. This is performed at application root component:

```
1  this._window.ethereum.on('accountsChanged', accounts => {
2    this.ngZone.run(() => {
3      // https://github.com/MetaMask/metamask-extension/issues/5826
4      this.accountService.currentAccount = this._window.web3.toChecksumAddress(accounts[0]);
5      this.changeDetector.detectChanges();
6      this.snackBar.open("The account has been changed", "Ok", { duration: 5000 });
7    });
8  });
```

Figure 14: The Metamask account change event handler

To properly notify users of new created auction instead we must interact with the blockchain. In the *HouseComponent*, events are caught by the following code:

```
1  private registerListeners() {
2    this.houseInstance.on("NewAuction", () => this.fetchHouseEvents());
3  }
4
5  private async fetchHouseEvents() {
6    const auctionsAddresses: Array<string> = await this.houseInstance.getAuctions();
7    this.dataSource = await Promise.all(auctionsAddresses.map(async address => {
8      const contract = await this.abstractAuctionFactory.attach(address).deployed();
9      return {
10        name: await contract.itemName(),
11        type: await contract.auctionType(),
12        seller: await contract.seller(),
13        address: contract.address
14      } as AbstractContract;
15    }));
16  }
```

Figure 15: The subscription to the creation of new auction event

The dutch auction has only a single event indicating when a bidder has won and thus the auction is closed, while the vickery has different events. For example, when executing an operation on the contract the following flow is followed:

1. A transaction is created and sent
2. The transaction is awaited until mined
3. Event listeners previously subscribed are eventually called, indicating the result of the operation to the user

An example of this flow is shown in figure 12, while the event subscription is shown in figure 16.

```
1 private listenEvents() {
2   this.contractInstance.on("OpenBidEvent", (sender: string,
3                                     nonce: ethers.utils.BigNumber,
4                                     value: ethers.utils.BigNumber) => {
5     if (sender == this.accountService.currentAccount)
6       this.snackBar.open(`You have opened your bid (${value.toString()} wei) successfully`, "Ok");
7   });
8   // ...
9 }
```

Figure 16: The subscription to the creation of new auction event

## 5 Deploy on real network

Since that the provider is injected by Metamask and that no contract needs to be deployed before the front end, it's sufficient to open Metamask and switch network to the desired one. At this point, since contracts ABI are bundled with the front end, it is capable of deploying new contracts with the same operation from the interface shown in the previous chapters. For example I created on Ropsten network the *AuctionHouse* at address 0x64bdeB82261f78d363a4814668F6E4dAf194E850 and the *LinearStrategy* for a dutch auction at address 0x5805f5172ab1214cc6ceb803ff6571885276bc17.