

COBrA

Fair COntent Trade on the BlockchAin

Final Project Report



Alessandro Berti
505328

Contents

Backend	3
ContentManagement.sol.....	3
Catalog.sol.....	4
Frontend	6
setting.js.....	6
ContentManagement.js.....	9
Catalog.js.....	9
Instructions	10
Leverage payment scheme	14

Backend

The general approach and the implementation of the smart contracts do not differ much from the one already seen in the FinalTerm except for the modifications required by the extension of the specifications. Below are the differences.

ContentManagement.sol

Events:

- 1) *feedbackActivation(bytes32 indexed contentName)*. Triggered on consumed content.

News

- The constructor now allows the author to specify a price for the content.
- **ConsumeContent()** has remained unchanged except that it now enables the user to leave feedback (*canRate [msg.sender] = true*). Also, before the function terminates, it emits the *feedbackActivation* event.
- Each content keeps information about the average of the 3 different categories and about the overall of the categories (*uint8[4] rateByCategory*), which are:
 - Appreciation
 - Quality (i.e. sound quality in the case of song)
 - Price Fairness
 - Overall $\rightarrow \frac{\text{Appreciation} + \text{Quality} + \text{PriceFairness}}{3}$

RateContent(uint8 _cat0Rate, uint8 _cat1Rate, uint8 _cat2Rate)

- **Modifiers:**
 - onlyWithRange: check that the ratings left for feedback are in the range [0,100].
 - onlyIfConsume: check that *canRate[msg.sender]* is true, i.e. that the user has previously consumed the content.

This function, therefore, can only be executed if the content has been consumed. It takes 3 inputs in the range [0,100] (i.e. the rate chosen by the user for each of the categories and it adjusts the average in *rateByCategory*. Before terminating, it invokes **updateMostRated()**, a *Catalog*'s method. (Section below)

getRate(uint8 _index)

Returns the rate for the category saved in the `_index` position of `rateByCategory` [4].

Catalog.sol

Events:

- 1) *premiumEvent(address indexed user)*. Triggered on **buyPremium()/giftPremium()**
- 2) *grantedAccessEvent(address indexed user, byte32 contentName)*. Triggered on **getContent()/getContentPremium()/giftContent()**.
- 3) *newContentPublication(address indexed author, bytes32 indexed genre)*. Triggered on **addContent()**
- 4) *catalogSelfDestructed()*. Triggered on **closeContract()**.

News

Unlike the previous version of the smart contract, both in the struct *statstByAuthor* and *statsByGenre*, a *BaseContentManagement* [4] *mostRatedByCategory* array was added, in which the address of the content with the highest rate is maintained for each category.

In addition, an additional *BaseContentManagement* [4] *overallMostRated* array has been added, which keeps track of content with greater installments regardless of genre and author.

UpdateMostRated()

- **Modifiers**
 - *isInCatalog(BaseContentManagement(msg.sender))*: check that the smart contract that invoked this function is present in the Catalog.

The function is invoked by the ***rateContent()*** of the instances of *ContentManagement.sol*. It deals with checking whether the *ContentManagement* that invoked it, has become the "most Rated" compared to those stored in the previously mentioned arrays; if so, update them with the *ContentManagement* in question.

getMostRatedByGenre(string _genre, uint8 _cat)/getMostRatedByAuthor(address _author, uint8 _cat)

These two functions, through the mapping *statsByGenre* and *statsByAuthor* respectively, retrieve the "most Rated" compared to genre/author on a certain category *_cat* and return it. In the case that *_cat* is not defined, the value contained in the last position of *rateByCategory* (i.e. *rateByCategory[3]*) that identifies the content by genre/author with greater overall is returned.

getMostRated(uint8 _cat)

It relies solely on the *overallMostRated* array, allowing direct access to the array element in the *_cat* position

closeContract()

- **Modifiers**
 - *onlyCreator*: if msg.sender is the one who deployed the Catalog, then he/she can close it.

This function, as well as closing the catalog itself, also deals with closing all content belonging to the catalog. This solution was pursued in response to the request in the specifications of the project "*All deployed contracts are required to have a suicide operation to relieve the testnet after the experiments*". Since the object was to clean up the Ropsten after the experiments, I thought it useful for the *closeContract* to automatically do it for everyone without a manual operation to be performed *for each contentManagement*.

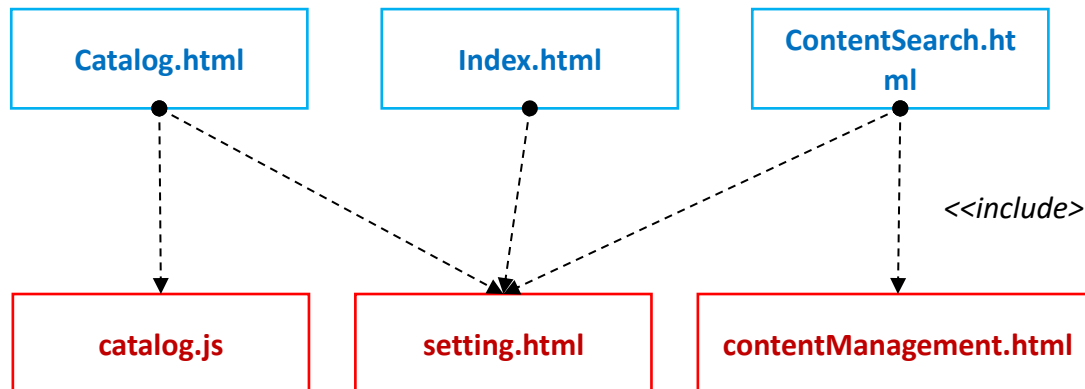
Small implementation detail: assuming a very large number of content, I considered fair against the catalog creator that, before paying the author according to the number of views by content, a cycle was performed in which all content is closed first and then with the balance left, the authors are paid. In this way the catalog creator does not have to take on additional expenses caused by the high number of iterations of the cycle [Catalog.sol - line 278].

The ***grantedAccessEvent***, compared to the Final Term version, has been moved from the *ContentManagement.sol* to the *Catalog.sol* since, otherwise, each fronted user would have to run a ***watchEvents()*** for each content in the catalog to receive the notification: solution considered extremely little practice.

Frontend

The frontend was created as a web page using web3js 0.20.6 as a library to interact with the geth client.

The main files that characterize it are the following:



Since it is a web page and the Internet is stateless, moving from one page to another, the state is lost. In a real context, the solution could have been to use cookies and/or similar mechanisms.

The main functions related to the interaction with the blockchain are as follows:

setting.js

SettingUp()

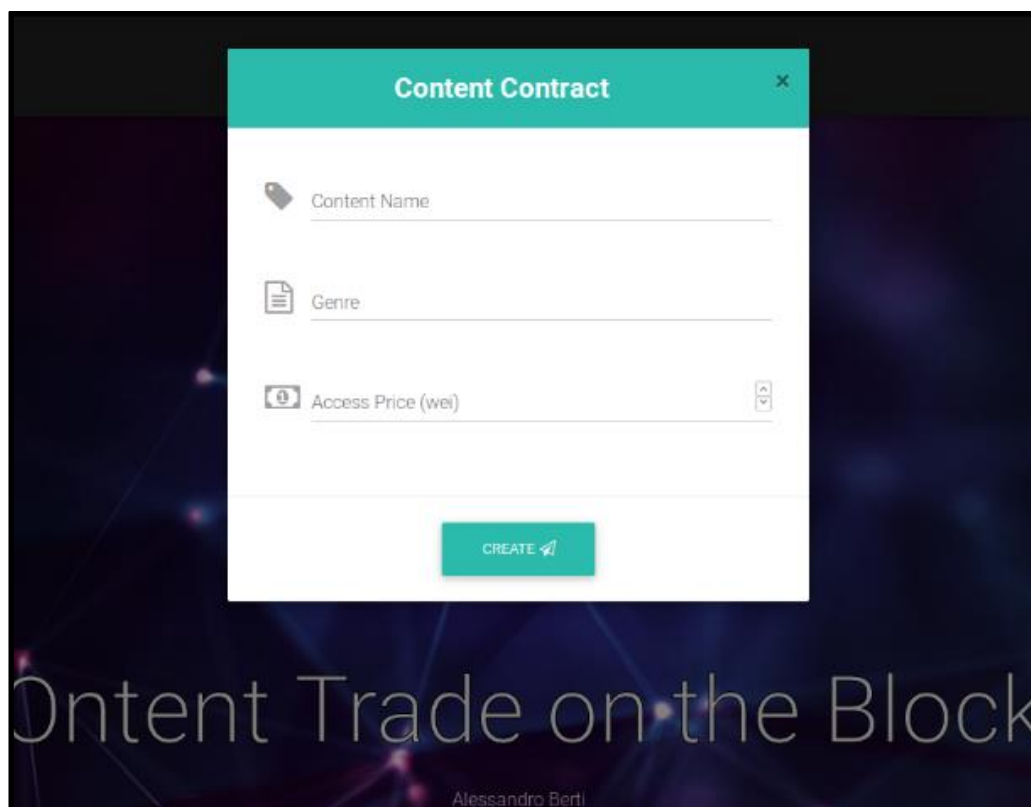
It is invoked whenever the *.html is opened. It takes care of connecting to the geth client and setting the *catalogContract* and *contentManagementContract* variables that define the ABI of *Catalog.sol* and *ContentManagement.sol* respectively.

Finally, the **catalog** variable is set that will allow us to access the methods offered by the Catalog. The address of the catalog is hardcoded in the code, solution, in my opinion, in line with the hypothetical real use of the application.

DeployContent()

Performs checks on the fields required by the modal and deploys an instance of *ContentManagement.sol*, invoking the constructor with the values entered in the previously mentioned fields.

The callback of the function **new** (= deploy) I wait for the smart contract to be deployed on the blockchain and, after that, invoke **catalog.addContent()** automatically adding the content to the catalog. In this way, as well as reinforcing the logical link between catalog and content, it improves usability by non-navigated users.

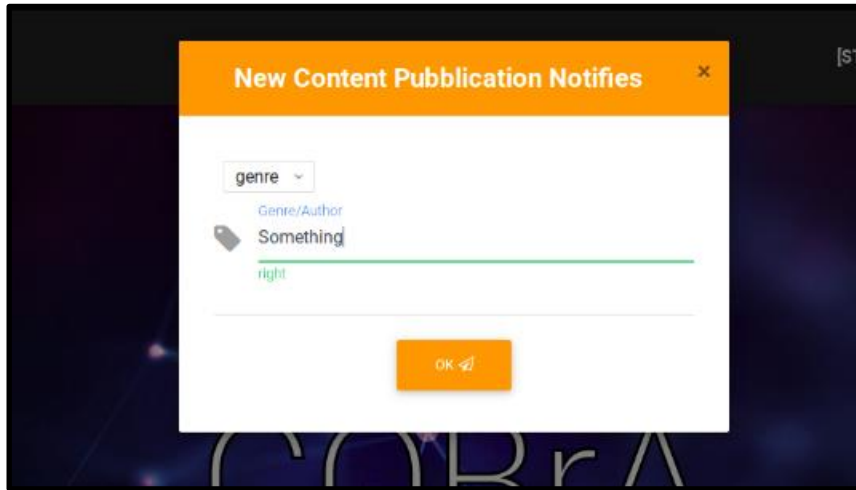


WatchingEvents()/StopWatchingEvent()

They are invoked whenever the *defaultAddress* changes so as to have customized events for each user.

SetPreference()

Start a watch on the ***newContentPublication event***. From the graphical interface it is possible to specify for which author or genre to listen. More than one author/genre can be specified.



AwaitBlock()

One problem is the managing the asynchronous blockchain responses. Web3js 0.20.6, offers a simple callback mechanism but that is not sufficient to know the status of the transaction sent. Therefore it was necessary to implement ***AwaitBlock()***, an asynchronous function which, every 5 seconds for 100 times (a value deemed reasonable), checks whether the transaction has been mined. When it is included in a block, it checks whether the transaction was successful or not, notifying the user. In the case that (5x100) secs last, an on-screen writing is shown informing the user that the testnet is potentially congested and, therefore, to be patient.

AwaitBlock() was added to each javascript function that encapsulates a method that creates a transaction.

Elenco di Checks utilizzati nei vari metodi:

- *isInCatalog(name)*: check that “name” is in the catalog.
- *isValidAddress(address)*: check the validity of address’s format.
- *nullContentAddress()*: check that the address is different from 0x00000000000000000000..
- *enoughBalance(price)*: check that the defaultAddress has a *balance* \geq *price*.
- *isAddressSelected()*: check that the defaultAddress has been correctly setted.
- *enoughValue(value, price)*: check that the *value* inserted by the user is \geq requested *price*.

ContentManagement.js

searchByContentAddress()

Search for the specified content and display the main information, such as: Name, Author, Genre, No. Tot Views, Appreciation, Quality, Price Fairness, Overall rate.

Before terminating, start a watch on any ***feedbackActivationEvent*** for the address that will consume the searched content.

ConsumeContent()/SendFeedBack

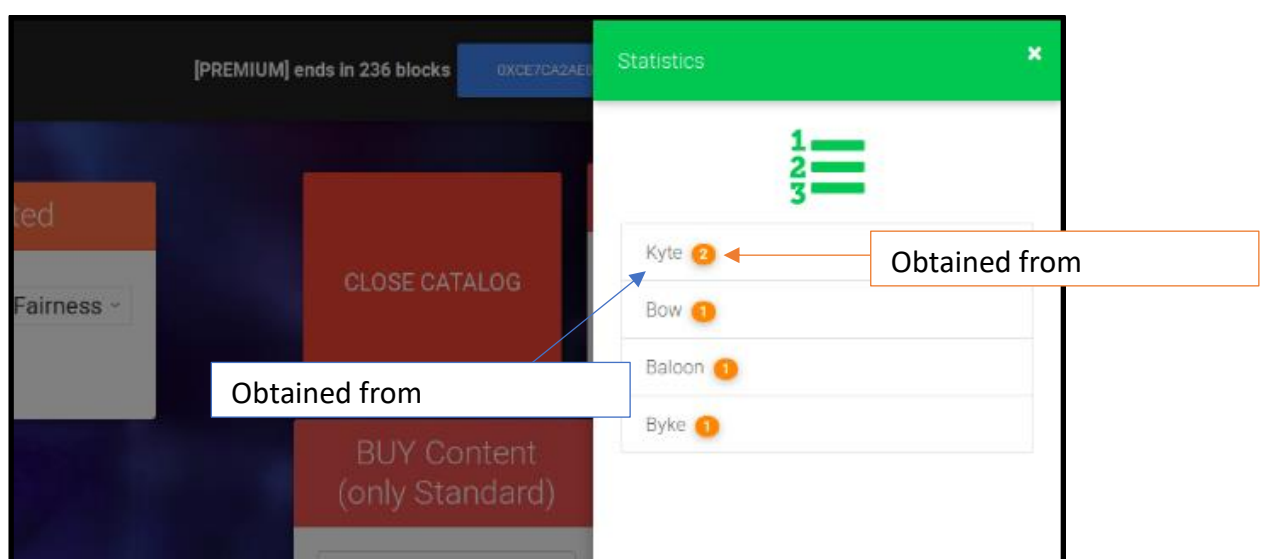
They encapsulate the corresponding methods in the Catalog.sol (***consumeContent()*** and ***rateContent()***) with some of the previously mentioned checks.

Catalog.js

The functions implemented here are responsible for encapsulating, together with the appropriate checks, the methods of the Catalog.sol instance.

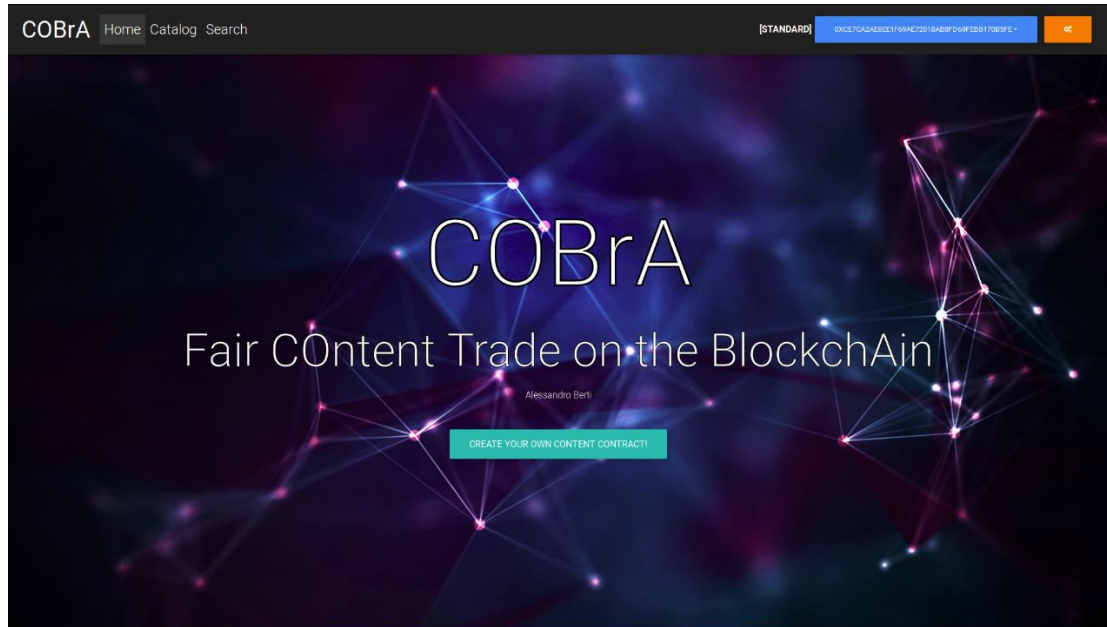
I avoid inserting the list of all the functions as they would not bring any additional information.

A small note: the *AppendToStatistics* function combines the results of the ***catalog.getContentList()*** and ***catalog.getStatistics()*** methods as shown in the following image.

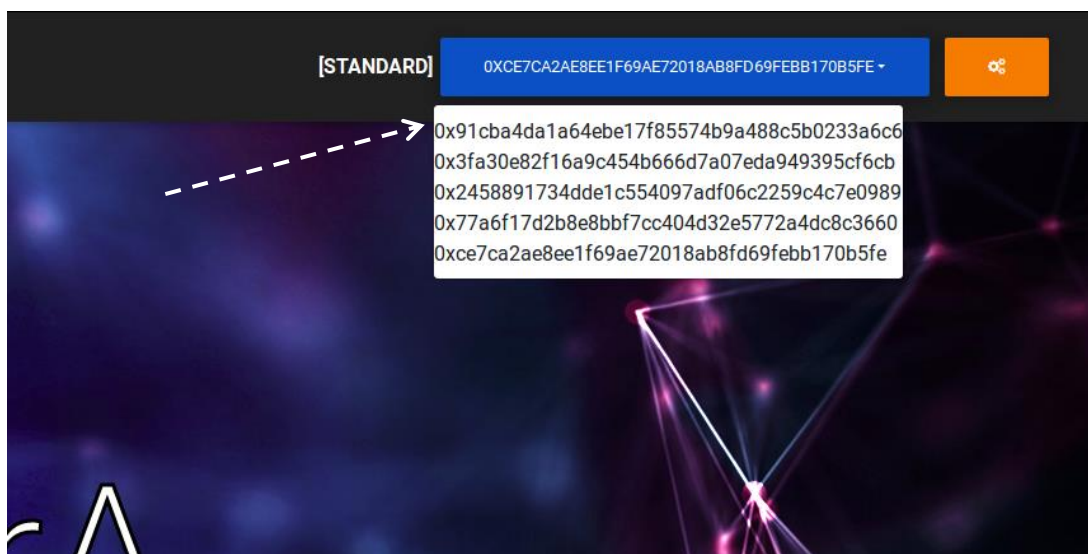


Instructions

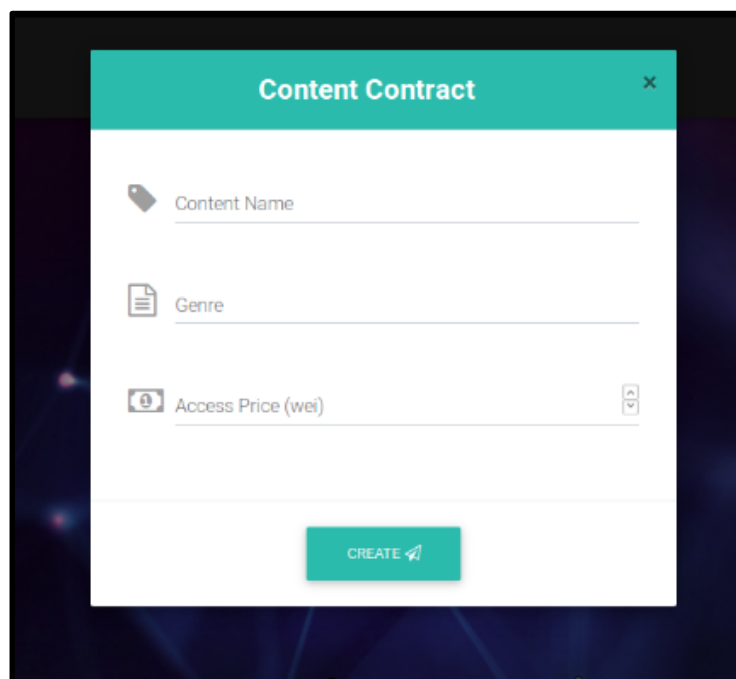
- 1) **Open** *COBrA_FinalProject_AlessandroBerti/COBrA_Dapps/index.html* with a browser (preferably Firefox)



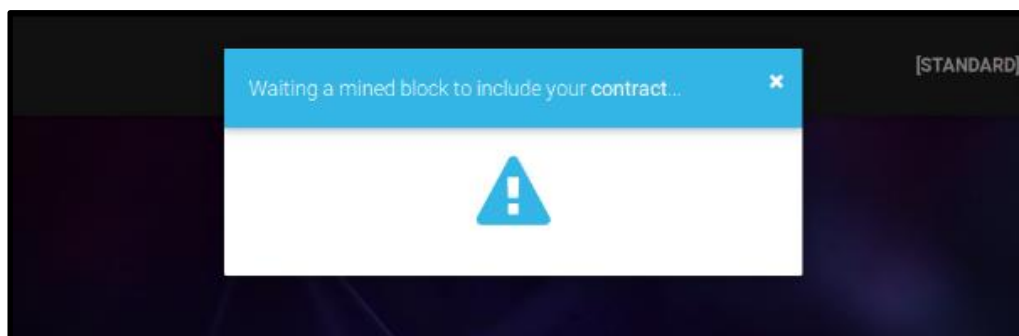
- 2) **Select and unlocked Address**



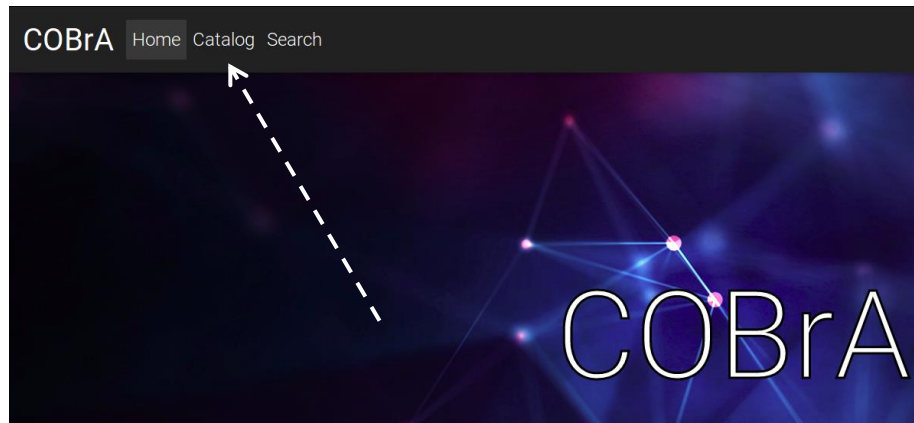
3) Click on “Create your content contract” and fill the fields.

The image shows a "Content Contract" form with a teal header and a close button (X). The form contains three input fields: "Content Name" with a tag icon, "Genre" with a document icon, and "Access Price (wei)" with a coin icon and a decimal separator icon. A teal "CREATE" button with a checkmark icon is at the bottom.

4) Wait for transaction to be mined.

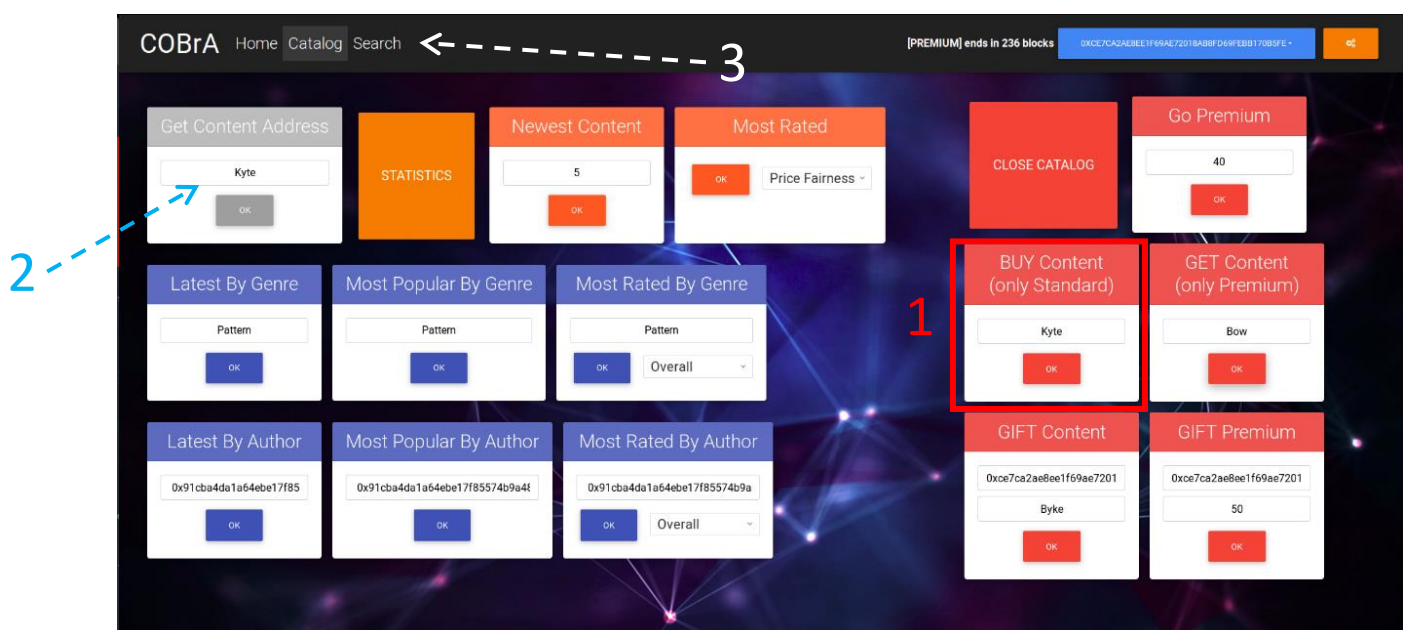


5) **Once the transaction has been mined**, click on **Catalog**.

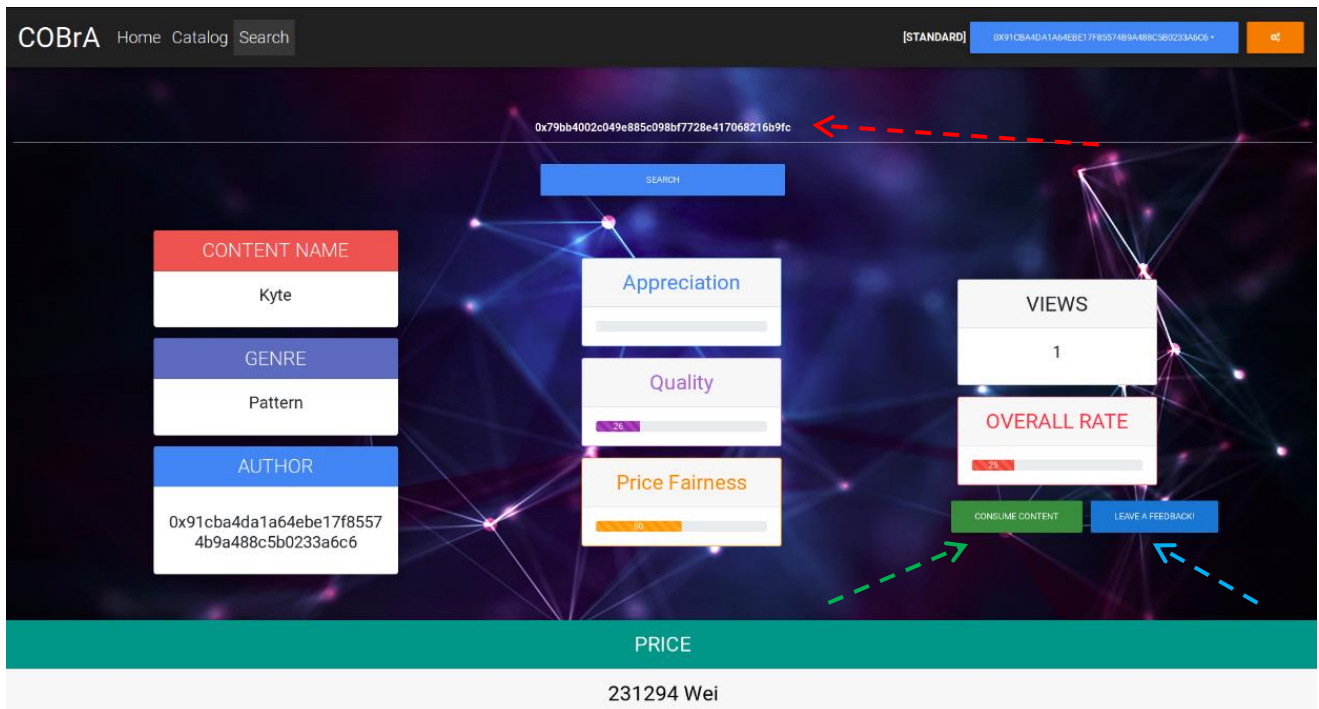


6) Here you have the set of methods offered by the catalog that you can try. The “red methods” require a payment, thus that an address has been selected. The remaining methods are “view” so they do *not* need to be invoked with an *address*.

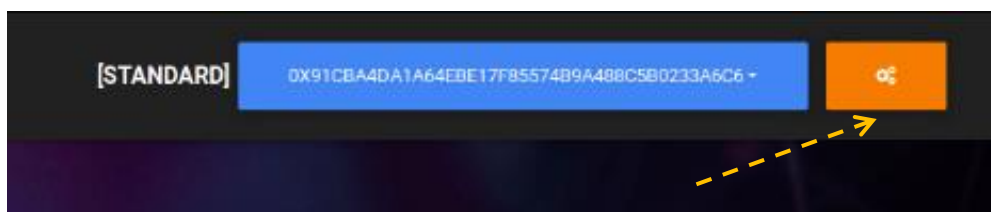
7) **Once you bought** a content (1), insert the **content name** in **Get Content Address** (2), click “OK”, *copy the address* shown and then go to **Search** (3).



- 8) Here you can paste the copied address and click on **“Search”** (1); informations about the content will be shown. If you have properly paid this content, you can also click on **“Consume Content”** (2) and then on **“Leave a Feedback”** (3).



- 9) Last thing, you can set is the **newContentPublication listener** by clicking on the gears on the right of navigation bar.



Leverage payment scheme

The catalog, in a certain sense, turns out to be a caveat that, if particularly active and lasting over time, can become extremely attractive to the authors; but in what sense?

Since on the closing of the catalog, the balance of this "caveat" is divided to the authors of the contents based on the number of views, it is reasonable to ask if any wrong behavior by an author is possible.

Suppose that the catalog has 1000 ETHs and that each content (added by honest authors) has a certain number of views. What a dishonest author could do before the catalog is closed, is to add to the catalog a content with price = 0 in order to allow a boost on the views that holds 99% of the views of all content. In this way, the dishonest author would take over 999 ETHs, having in fact not contributed in any way to the service offered by the catalog to users. This operation would however be long in terms of time, since:

$$\frac{BlockGasLimit}{BuyContentGas + ConsumeContentGas} = \frac{4.700.000}{53068 + 110085} = 28 \text{ views/block}$$

(A thought: It is interesting to note how the gas sets an upper limit on the number of transactions in a block for the smart contract; it may make sense to implement functions that perform expensive operations only to ensure that a certain maximum of operations can be done in the same block.)

Despite this, honest authors would be destined to receive an incorrect payout against them once the contract was closed.

One consequence could be the transformation of honest authors into dishonest ones by publishing their contents at price 0 to try to counter the "original" dishonest author. This necessarily puts in bad light the service offered by the catalog to which will not be added more new content of honest authors and seeing a complete affirmation only by dishonest authors within it.

A trivial solution to the problem is to put a minimum cost for each content necessarily $>> 0$. This may not be enough if the total balance accumulated in the catalog is so high as to eclipse the minimum cost. Since the minimum cost cannot be changed and the accumulated total increases constantly, with the current implementation, the incorrect behavior of authors is not hindered in the long term.

The "definitive" solution is to modify the current implementation by adding a specific function that can only be invoked by the catalog creator who takes care of emptying the total accumulated balance (paying the authors) without necessarily closing the catalog.