

University of Pisa

Master's Degree in Computer Science



PEER TO PEER AND BLOCKCHAINS COURSE

COBrA DAPP

Final project report

Student:

Aldo D'Aquino

Teacher:

Prof. Laura Ricci

Anno Accademico 2017/18

Table of Contents

1	Introduction.....	4
1.1	Aim of the project	4
1.2	Project specifications	4
1.3	Implementation specifications	5
2	Contracts implementation.....	8
2.1	Style	8
2.1.1	Encoding	8
2.1.2	Line length	8
2.1.3	Indentation	8
2.1.4	Line breaks	8
2.1.5	Spaces	9
2.1.6	Components order	10
2.1.7	Modifiers.....	10
2.1.8	Naming.....	11
2.1.9	Other.....	11
2.1.10	Comments and documentation	11
2.2	Design choices	11
2.2.1	Suicide function	11
2.2.2	Parameter names.....	12
2.2.3	Contract vs storage structures.....	12
2.2.4	getContentPremium	12
2.2.5	Premium subscription.....	13
2.2.6	Murder function.....	13
2.2.7	Suicide function	13
2.2.8	Authors payment	15
2.2.9	Statistics functions	15
2.2.10	Rating functions	16
2.2.11	Prevent authors from cheating	16
2.2.12	Parameters of the system	16
2.2.13	Events.....	17
2.2.14	Content	17
2.3	Implementation choices	18
2.3.1	Assert, require, throw and revert	18

2.3.2	Strings	18
2.3.3	Arrays	19
2.3.4	Reentrancy problem	19
2.4	Gas estimation	21
2.4.1	CatalogContract	21
2.4.2	BaseContentManagementContract	24
2.4.3	GenericContentManagementContract (is BaseContentManagementContract)	24
2.4.4	DAPPContentManagementContract (is GenericContentManagementContract)	25
3	DAPP Implementation	26
3.1	Programming language and tools	26
3.2	Design choices	26
3.2.1	DAPP	26
3.2.2	GUI	26
3.2.3	Serving the content.....	27
3.3	Project structure	28
3.3.1	Contracts.....	29
3.3.2	GUI	30
3.3.3	Author server	32
3.3.4	Connections	33
3.3.5	Test	34
3.4	Execute the DAPP.....	34
3.4.1	Requirements.....	34
3.4.2	Compile sources.....	35
3.4.3	Start Ethereum client.....	36
3.4.4	Run the app.....	36
3.5	GUI snapshots.....	37
3.5.1	Starter panel	37
3.5.2	Customer panel.....	38
3.5.3	Author panel	40
4	Works Cited	41

1 Introduction

1.1 Aim of the project

COBrA is a university project that aims to implement a decentralized content publishing service.

Some authors will be able to publish their contents on a catalog and customers will be able to use them by buying them or subscribing to a premium subscription.

Contents can be songs, videos, photos, or something else created by the artists and they will be rewarded accordingly to customers' fruition.

The system will have to rely entirely on the blockchain, so must be completely decentralized and not having to depend on third-party servers that entail high running costs, are single points of failure and can often be unreliable.

To help authors and customers interact with blockchain contracts, it is necessary to develop an application (DAPP) with a graphical interface that makes the use of this service more natural.

The purpose of the entire project is to deepen the use of blockchain for decentralized and secure systems, other than the mere exchange of money, and learn how to develop applications that can interact with it.

1.2 Project specifications

Users of the system are divided into Customers and Authors. An author can also be a customer of the system as he may be interested in content from other authors.

Customers are divided into two types: Standard and Premium.

All the accounts start as Standard accounts: they must buy contents before accessing them and they are entitled to a single fruition of them. For further access they need to buy it again.

Standard account can subscribe a Premium access that last for x blocks number on the blockchain.

During the Premium period they can watch all the contents for free as many times they want.

Users interact with the blockchain through a DAPP equipped with a graphic interface. The DAPP provides two different interfaces, one for the customers and one for the authors.

The DAPP allows an author to publish new contents and access information about previously published content, including the number of views and statistics.

Customers can see the list of contents, their views, the ranking of the most viewed and highest voted by category, and rankings by author and genre. They can also subscribe or gift a premium subscription, buy, or gift content and access the purchased content. Finally, they can vote the contents already consumed.

The DAPP also notify the user when some events occurred. The DAPP also notifies the user when events occur. Among these there must be a notification about the possibility to vote for a content, the publication of new content and access grant on content bought by the user or gifted by another one.

1.3 Implementation specifications

The blockchain based backend must be implemented as a single Catalog Contract and some Content Management Contracts.

The number of Authors, Customers and Contents is not fixed and can change dynamically.

Each Content Management Contract controls exactly one unique content.

An author submits a new content by deploying through the DAPP a new Content Management Contract – that in the implementation we have called DAPP Content Management Contract – that inherit a predefined set of functionalities by extending the Base Content Management Contract. The Content Management Contract must include the name of the content and the content data and may include other features for content management at the discretion of the author.

Once the Content Management Contract is deployed the author must submit a new content publishing request to the Catalog Contract to link the new contract to the Catalog. This linking is provided by a method call with which the Catalog can find the necessary information about the content from the Content Management Contract.

The Catalog Smart Contract acts as an intermediary between Authors and Customers.

Customers can consult through the DAPP the library of all the contents published in the Catalog by Content Management Contracts.

Customers access the Catalog Contract to request access to contents. To access the content, the Customer must send to the contract an amount of ether equal to the cost of the content, which is chose by the author. Premium accounts do not have to pay and can directly access all the contents for free until the Premium subscription expiration. When the access is granted the Customer receives a notification by the DAPP and can consume the content.

All the payments from the users are collected and redistributed among authors by the Catalog Contract, according to the number of views of each content.

Premium Account content fruitions are not considered in the view count.

To ensure decentralization of the system the Catalog Contract deployer cannot receiver any rewards at the end of the Catalog Contract the remaining budget must be redistributed among the authors according to the number of views obtained by their contents.

Both the Catalog Contract, the Base Content Management Contract and any Content Management Contract must be written in solidity and deployed on the Ropsten network.

It is mandatory to implement at least the following functions:

- Public views
 - GetStatistics(): returns the number of views for each content.
 - GetContentList(): returns the list of contents without the number of views.
 - GetNewContentsList(): returns the list of newest contents.
 - GetLatestByGenre(g): returns the most recent content with genre g.
 - GetMostPopularByGenre(g): returns the content with genre g, which has received the maximum number of views
 - GetLatestByAuthor(a): returns the most recent content of the author a.
 - GetMostPopularByAuthor(a): returns the content with most views of the author a.
 - IsPremium(x): returns true if x holds a still valid premium account, false otherwise.
 - GetMostRated(y): returns the content with highest rating for feedback category y (or highest average of all ratings if y is not specified).

- `GetMostRatedByGenre(x, y)`: returns the content with highest rating for feedback category `y` (or highest average of all ratings if `y` is not specified) with genre `x`.
- `GetMostRatedByAuthor(x, y)`: returns the content with highest rating for feedback category `y` (or highest average of all ratings if `y` is not specified) with author `x`.
- `RateContent`: must be provided a function to allow users who have consumed a content to rate it.
- Public actions (modify the state of the contracts):
 - `GetContent(x)`: pays for access to content `x`.
 - `GetContentPremium(x)`: requests access to content `x` without paying, premium accounts only.
 - `GiftContent(x,u)`: pays for granting access to content `x` to the user `u`.
 - `GiftPremium(u)`: pays for granting a Premium Account to the user `u`.
 - `BuyPremium()`: starts a new premium subscription.
 - `GetPayout`: must be provided a function to allow authors to get payed for the contents that have reach a fixed amount of views.

2 Contracts implementation

2.1 Style

We choose to write all the code in compliance of the official stylish guidelines (1). In particular we refer to the Solidity version 0.4.24 (2), which is the last release at the moment. The Ethereum docs are continuously under construction, their progress as well as the latest version that still under construction (3) can be found in the *docs* path in the GitHub repository (4).

Among others, we consider in particular the following conventions.

2.1.1 Encoding

The stylish guidelines recommend UTF-8 or ASCII. We choose UTF-8.

2.1.2 Line length

Line length cannot overcome 79 characters.

2.1.3 Indentation

Indentation is 4 spaces wide as suggested; spaces are preferred instead of tabs.

2.1.4 Line breaks

There must be 2 empty lines between contracts, 1 empty line between the functions implementation and 0 between functions and variables declaration.

```
contract A {  
    function spam() public;  
    function ham() public;  
}
```

```
contract B is A {  
    uint public counter = 1;  
    bytes32 public test = "Test";  
  
    function spam() public {
```



```

        ...
    }

    function ham() public {
        ...
    }
}

```

2.1.5 Spaces

Spaces must be avoided immediately inside parenthesis, brackets or braces, and Immediately before a comma or a semicolon.

Instead there should be a single space between the control structures if, while, and for and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Parenthesis should be opened on the same line as the declaration and closed on their own line at the same indentation level as the beginning of the declaration.

Admitted:

```

spam(ham[1], Coin({name: "ham"}));

function singleLine() public { spam(); }

function() public {
    ...
}

v = 1;
long_variable = 2;

```

Not Admitted:

```

spam( ham[ 1 ], Coin( { name: "ham" } ) );

function spam(uint i , Coin coin) public ;

function () public {

```

```

    ...
}

function() public
{
    ...
}

y = 1;
long_variable = 2;

```

2.1.6 Components order

Imports must be placed at the top of the file. Then will come variable declaration, events, modifiers and finally functions.

Variables are usually ordered starting from constants, then runtime variable, the ones that changes during the contract lifecycle, and finally structs and arrays.

Functions must follow this order: constructor first, then fallback function and then, accordingly with the visibility modifier, external, public, internal and lastly private functions. For the same visibility, priority is given to the functions that modify the status, leaving the views and the pure at the end.

We choose to put the suicide function immediately after the fallback function because we believe that it is one of the special functions.

We have placed the functions with the onlyOwner modifier at the end of the public functions, as their access is not totally public but is restricted to a single person.

2.1.7 Modifiers

There is not a specific order for modifiers, but the visibility modifier for a function should come before any custom modifiers.

Admitted:

```

function kill() public onlyowner {
    selfdestruct(owner);
}

```

Not admitted:

```

function kill() onlyowner public {

```

```
        selfdestruct(owner);  
    }
```

2.1.8 Naming

Contract, struct and event names must be in CapitalizedWord style. Variable, modifiers, and functions in mixedCase.

For this reason, we choose to rename all the mandatory functions from CapitalizedWord to mixedCase, considering the coding language specifications more important than the assignment.

2.1.9 Other

Declarations of array variables should not have a space between the type and the brackets.

Strings should be quoted with double-quotes instead of single-quotes.

Admitted:

```
uint[] x;
```

Not Admitted:

```
uint [] x;
```

2.1.10 Comments and documentation

Although there are no specifications regarding the comments we have noticed that remix supports Doxygen-style (5) documentation and we have therefore decided to adopt this convention.

2.2 Design choices

2.2.1 Suicide function

The suicide function already exists, it was the deprecated version of the selfdestruct function. We consider reasonable to not override a default function, although is deprecated. For this reason, the suicide function starts with an underscore (`_suicide`).

2.2.2 Parameter names

All the mandatory functions have x as argument name except for GiftContent that has also u that stands for user. For clarity we choose to rename the parameter name choosing a for author, g for genre, leaving u for user and customer and limiting x only for content.

This convention that we adopted helps to keep the code clean and readable. In any case all the parameters of the functions have been documented in the code.

2.2.3 Contract vs storage structures

A widespread practice is to guarantee access to a resource by deploying a contract. Sometimes contracts are used instead of structs as a method of storing data.

We have found these policies to be improper because they create many contracts and transactions on the blockchain and this helps to create spam on the Ropsten network. In addition, the amount of gas to deploy a contract is much higher than that needed to save a piece of storage: according to the Appendix G of the Yellow Paper (6), a store operation costs 20000 units of gas, a contract deployment 32000.

2.2.4 getContentPremium

We have arbitrarily decided to not implement this function. The reason is that a user could pay only a premium cycle and access all content in the future. even if the premium account is no longer active. The only limitation would be that he can watch it once only. In this case we thought that the premium account would have turned into a "bundle" of content rather than a subscription. Since this is not the expected behavior, we have abolished the function and now a premium user can consume all content without having to first request access to it as long as his premium subscription is still valid. When the Content Manager Contract checks if a user has the permission to consume a content, the Catalog will always return true if the user has a Premium subscription.

Access to content from a premium account will not affect previously bought content. The user can still consume the purchased content (once only) when the subscription has ended, even if that content has been accessed by this user multiple times during his premium account. In addition, a Premium account can also purchase content. They will be consumable (once only) when the premium subscription has ended.

We design the consumable content of a user as an array. We can put elements in the array even if the user is Premium, as long as he pays them, but we remove elements from the array only if the user has not a Premium subscription.

2.2.5 Premium subscription

We decide to use block heights as measuring meter for Premium subscription time.

Every time that a user buys a Premium period we save in a mapping the number of the block after which the Premium subscription expires. When the `isPremium` function is called we just check if the current block number is greater than the expiration block.

A Premium user can buy another Premium cycle also before the current one is expired, simply the new cycle will start after the current one. This is implemented by adding another Premium cycle length to the mapping value corresponding to the user's address.

2.2.6 Murder function

We implement on the Base Content Management Contract a murder function, that has the same behavior of the suicide function but can be called from the catalog.

The catalog will use this function only when it is closed to delete all contents. Any balance will be transferred to the author. Users will be informed with an event.

2.2.7 Suicide function

When the owner calls the suicide function, the balance is split between the authors.

The amount that belongs to each author is saved in a structure.

First of all, we save the amount due to unpaid views. The amount is calculated using the same algorithm as the payment function but does not consider the minimum number of views required. In other words, the number of uncollected views is multiplied by the price and then by the average rating divided by 5 that is the maximum rate. If there are no votes, we consider the rating of the content to be 5, because otherwise the negligence of the users would be disadvantageous for the content creator.

Subsequently the remaining balance, which comes from Premium subscriptions in addition to the part of the cost of the content not paid to the publisher, is divided among all the authors. This operation is done according to the content cost, to the number of views and to the ratings.

It is possible that after this operation the balance is not yet zero. This is because we do not consider the remains of the divisions. We thought fair to transfer this little amount to the owner not as an earning but as refund for the big amount of gas used for this operation.

We also took some precautions to limit the consumption of gas. First of all, we call the murder function on all the contents: this generates negative gas to be used for operations. Then we do two different computations to calculate the amount to be transferred to the authors, but we perform one single transaction for each author. Finally, the Catalog is deleted and that generates a lot of negative gas: the owner must provide a big quantity of gas to not run out of gas but not all of this will be spent thanks to the negative gas that lowers the worn one.

Another important aspect of the suicide function is that it leaves the callers of functions on the suicide contract in an undefined state, creating many problems.

To avoid this, we have provided a removesMe function on the Catalog that is called by the Content when committing suicide. The Catalog instead, as already described above, will murder all contents before committing suicide.

During the tests we found that the consumption of gas, despite the optimizations is very high, because of the complexity of the operations.

The consequence is that at full capacity, suppose thousands of contents, the amount of gas needed to invoke the suicide function exceeds the block gas limit, so it is not possible to delete the contract. A possible solution is to reduce the number of operations required by issuing an unfair payment. Another solution is to freeze the contract by preventing customers from interacting with it, calculating the payout for each author, and leaving some time for each of them to withdraw their payout. In this way each transaction is in a separate method call and the gas consumption is below the threshold. When the time expires all the remaining balance will be given to the contract owner, that we thought is unfair.

The problem could be solved by breaking up the suicide function into smaller functions, for example by calculating the payout for a few authors at a time, to be called sequentially. The sequential call could be made by the DAPP itself, but in any case, the catalog owner could call the functions manually on the blockchain and manipulate the behavior.

Since all the solutions described above provide to trust the catalog owner, but the specifications require to not trust it, we preferred to leave the function unchanged, knowing that if we wanted to use this catalog for a real application it would be necessary to review some policies.

2.2.8 Authors payment

We have decided to not trigger the transaction as soon as v views are reached but only emit an event that notice the author that it can withdraw his reward. This is because otherwise when the v -th user consume the content, the transaction starts, and it cost to that user 21000 gas and 0 to the other users. In this case instead we have a fairer solution in which the author pays the gas needed for his withdraw.

Furthermore, the author can decide to not withdraw immediately his reward but to wait to do a bigger transaction instead of lots of small transaction, paying less gas.

2.2.9 Statistics functions

For all the statistic functions, like `getLatestBy_` and `getMostPopularBy_`, there are two opposite kinds of approach. We can keep chart lists updated storing additional information or we can store only needed information and generates the charts each time they are requested. The first solution increases the gas expenses, the second one leaves all the burden to the consumer calling the methods.

We also have to consider who pays for this operation: in the first case the gas cost falls on the customers that consume the content, that may be not interested in charts and statistics, in the second case there is no extra cost and the burden is up to the user that requested it.

For this reason, we have preferred the second approach, which avoids unnecessary costs in terms of gas and also lightens the blockchain from unnecessary data.

2.2.10 Rating functions

For the rating functions too, there are two opposite kinds of approach. The most intuitive one is to save all the ratings of the customers and calculate the average each time. We discarded it because it needs more storage on the blockchain and represent a big burden for the user who will call the function. We prefer the second approach, which is to save only the average, with a little modification. We do not directly save the average, but we save the sum of all the votes and the number of votes. This prevents we from carrying rounding errors that can become very large.

2.2.11 Prevent authors from cheating

We have studied some strategies that malicious content publishers could adopt to leverage this payment scheme either to decrease other authors' revenues or to increase theirs.

The only way to prevent authors from decrease other authors' revenues is to forbid authors to rate a content. However, if we consider a real situation with lots of authors and that to rate a content is necessary to buy it, probably is more the trouble than it's worth. For this reason, we decide to not take actions to not limit too much authors, considering that a user can be an author and a customer at the same time.

Instead, to prevent authors from increase their revenues we decide that an author cannot buy their own contents. We thought this dumb, because the content creator has his content, so this policy does not limit the author. On the other hand, we got that if an author cannot consume his content it also cannot rate for it, and that prevent the cheat.

2.2.12 Parameters of the system

The cost of the premium subscription is a rather generic and difficult to determine, as it depends on the type of content offered, so we are limited to giving an order of magnitude rather than an exact price.

To do this we started from the actual value of the ether, which is around 400€ (7).

Usually prices of premium subscription range from 10 to 40 euros per month. We therefore considered that 0.1 ether is the right price and we set the premium time to 172800 blocks that are about a month considering that currently mining a block needs 14.99 seconds (8).

As far as the redemption of the rights by the authors, we thought it was right to make the withdrawal possible after 10 views. This number is a trade-off to allow authors with few visits not to wait too much and at the same time to ensure that the cost of gas does not affect too much the value taken. Ten views could seem a small number, but we preferred not to increase the number too much considering that an author can choose to wait more time to withdraw a bigger amount later, paying only one transaction.

2.2.13 Events

The only events that had to be mandatory emitted was the ones concerning access granting on a content and the possibility of rating a content. Considering the events an important form of logging and aiming to make the behavior of the contract as clear as possible, we decided to issue also other events.

We thought important to communicate not only when a user buy a content, but also when a user becomes premium and when a content is consumed.

Another important event is fired when the Catalog contract is closed, to notice all the interest user that they cannot access theirs account anymore. The content deleted event is like this.

Opposite to these, two events are fired when a content is published. The content emits a published event, and the catalog a new content available event.

Finally, it has become necessary to fire an event when an author's payment is available to notice this author that his withdrawals is available.

2.2.14 Content

We have chosen not to store the content in the Content Management contract. This would have implied the input of a large amount of data on the blockchain, which involved various problems, including the cost of gas to deploy the contract.

The task of serving the content to the user is left to the DAPP. The only task we have assigned to the Content Management contract is to manage the rights checks for the content access.

2.3 Implementation choices

2.3.1 Assert, require, throw and revert

Assert and require have the same purpose to verify the conditions and stop the execution of the code in case they are not satisfied. However, in the current version of Solidity, the 0.4.24, assert uses throw and consume all the gas, instead require uses revert and will refund all the unused gas. Therefore, although their behavior is almost equivalent from the point of view of the contract, it is very different from the user's point of view.

Assert vs require is a hotly debated topic about Solidity programming on the web, and we agreed with what Steven McKie wrote in his article (9). The basic idea is that consuming all gas should be a punishment for the user, to be used only when he has done something that he should know he did not have to do. In all the other cases revert should be fine because means that something went wrong or that the user has done some mistakes in passing variables.

After reflecting on the possible cases of use of an assert we considered that among the functions implemented in this project there was no wrong action for which it was revealed a need to punish the user for something. Therefore, we considered it more proper to use require everywhere.

The only case in which the assert could be predicted was in the modifier `onlyOwner`, whose meaning is quite clear, and the user should be aware of the fact that that function is not reserved for him. However, in Remix the modifier is not explicitly visible and therefore the user may not know that this function is reserved. For this reason, we decide to use require also in this case.

2.3.2 Strings

Strings are usually encoded as bytes32. The string type is already available on Solidity but are not fully supported. In particular to be returned by a function call must be enabled the pragma `experimental ABIEncoderV2` that, as the name says, is not yet stable.

Another possibility is to accept the strings as a function parameter and transform the outputs into bytes32, but this requires low-level calls that are to be avoided where possible.

We felt that the best solution was to use bytes32 everywhere, because nowadays is a widespread practice and is currently the standard string type.

The use of bytes32 also entails some advantages: strings are considered bytes and therefore arrays of infinite length. This makes it difficult to estimate the amount of gas needed and exposes you to

possible attacks aimed at consuming an excessive amount of gas to users. `bytes32` instead has a fixed length and therefore guarantees exactly the amount of memory used.

The disadvantage is that `bytes32` is an array of bytes of length 32. Therefore, since each letter occupies exactly 8 bits in UTF-8 encoding, it is possible to handle only 32-letter strings. However, we thought it was enough for a content title, also considering that for example “This is a 32 letters long title.”.

2.3.3 Arrays

We always choose static arrays where possible because they are cheaper than dynamically sized ones.

We also consider different options for storing list of data, but what we notice is that in memory arrays and mappings work behave in similar ways and the gas cost is more or less the same, so we just use the one that is more convenient for the purpose.

For byte arrays we usually use `bytes32` as there is no extra cost for allocating empty position in memory without initializing them.

2.3.4 Reentrancy problem

The reentrancy problem (10) is a common bug that is easy to introduce in the code that leads big problems because involves money, and most likely not only our money but also involves other people’s as well. In this case a reentrancy bug can allow an attacker to withdraw the entire balance of the contract, damaging all the authors.

A reentrancy attack consists of repeatedly and recursively calling the same function until the entire balance has been taken. Usually is done calling a function, i.e. “withdraw”, that start a transaction and in the function that receives this transaction calling again the same “withdraw” function. This will suspend the first call on the transaction and starting another transaction with the second call. In our case this attack could be done on the function `collectPayout`. Consider the following example:

```
function collectPayout(address x) public {
    require (contents[x].author == msg.sender);
    uint amount = payoutAvailable(x);
    require(amount > 0);
    msg.sender.transfer(amount);
}
```

```

        contents[x].uncollectedViews = 0;
        balance -= amount;
    }

```

This function can be exploited with this simple contract:

```

pragma solidity ^0.4.0;
contract CatalogContract {
    function collectPayout() public;
}
contract Exploit {
    function collect(address addr) {
        Catalog memory catalog = Catalog(addr);
        catalog.collectPayout();
    }
    function () payable {
        if (catalog.balance >= msg.value) {
            catalog.collectPayout();
        }
    }
}

```

The collect function calls the collectPayout function. The collectPayout runs until the transfer. The transfer fires the fallback function of the Exploit contract and will return only when the fallback function returns. After the transfer the authors uncollectedViews property is reset. The problem is that the fallback functions does not return but calls again the collectPayout function. The author uncollectedViews property is not yet reset, so the collectPayout function starts another transaction with the same amount. This action is repeated until the Catalog contract has a balance. Then all the recursive fallback functions return and only at this point the collectPayout functions update the uncollectedViews, after having transferred multiples time the amount to the author.

Below you can find the implemented collectPayout function rewritten without the reentrancy bug:

```

function collectPayout(address x) public {
    require (contents[x].author == msg.sender);
    uint amount = payoutAvailable(x);

```

```

    require(amount > 0);
    contents[x].uncollectedViews = 0;
    balance -= amount;
    msg.sender.transfer(amount);
}

```

2.4 Gas estimation

We propose below the list of functions with relative cost in estimated gas.

This list can also be useful to better understand the structure of contracts.

Node that we do not provide the cost of functions that can only be called by other functions, such as internal functions or functions that can be called only by a specific function of a specific contract. The gas cost of this functions is included in the gas cost of the calling function.

2.4.1 CatalogContract

```

// gas cost: 2149126
constructor() public;

// only revert the state
function () public;

// gas cost: at least 17167 (empty contract)
// + 3016 for the first author with his first content
// + 5840 for each additional author with his first content
// + 5014 for each other additional content of any author
// + at list 7927 if there is at least 1 view in all the catalog
// (depends on the number of contents in the catalog)
// + 4028 for each additional author that has at list a visit
function _suicide() public onlyOwner;

// gas cost: 68902 first time,
// 53902 if the user has already purchased and consumed this content previously,
// reverted if the has already purchased but not consumed this content
// (includes the grantAccess function cost)
function getContent(address x) public payable exists(x);

// gas cost: 55482 (includes the grantAccess function cost)

```

```

function giftContent(address x, address u) public payable exists(x);

// gas cost: 63351 (includes the setPremium function cost)
function giftPremium(address u) public payable;

// gas cost: 62937 (includes the setPremium function cost)
function buyPremium() public payable;

// gas cost: at least 44527 (depends on the length of the category y)
function leaveFeedback(address c, bytes32 y, uint r) public;

// view: no gas consumption
Function payoutAvailable(address x) public view returns(uint)

// gas cost: 25392 (includes the cost of payoutAvailable)
function collectPayout() public;

// included in BaseContentManagementContract.publish()
function addMe() public;

// included in BaseContentManagementContract.consumeContent()
function consumeContent(address u) public exists(msg.sender);

// included in BaseContentManagementContract._suicide()
function removesMe() public exists(msg.sender);

// view: no gas consumption
function getStatistics() public view returns(bytes32[], address[], uint[]);

// view: no gas consumption
function getContentsList() public view returns(bytes32[], address[]);

// view: no gas consumption
function getFullContentsList() public view returns(address[], bytes32[],
address[], bytes32[], uint[], uint[]);

// view: no gas consumption
function getRatingsList() public view returns(address[], uint[], uint[], uint[],
uint[]);

```

```

// view: no gas consumption
function getContentInfo(address x) public view returns(bytes32, address,
bytes32, uint, uint);

// view: no gas consumption
function getContentRatings(address x) public view returns(uint, uint, uint,
uint);

// view: no gas consumption
function getNewContentsList(uint n) public view returns(bytes32[], address[]);

// view: no gas consumption
function getLatestByGenre(bytes32 g) public view returns(bytes32, address);

// view: no gas consumption
function getLatestByAuthor(address a) public view returns(bytes32, address);

// view: no gas consumption
function getMostPopular() public view returns(bytes32, address);

// view: no gas consumption
function getMostPopularByGenre(bytes32 g) public view returns(bytes32, address);

// view: no gas consumption
function getMostPopularByAuthor(address a) public view returns(bytes32,
address);

// view: no gas consumption
function getMostRated(bytes32 y) public view returns(bytes32, address);

// view: no gas consumption
function getMostRatedByGenre(bytes32 g, bytes32 y) public view returns(bytes32,
address);

// view: no gas consumption
function getMostRatedByAuthor(address a, bytes32 y) public view returns(bytes32,
address);

// view: no gas consumption
function hasAccess(address u, address x) public view exists(x) returns(bool);

```

```
// view: no gas consumption
function isPremium(address u) public view returns(bool);

// included in buyPremium and giftPremium
function setPremium(address u) private;

// included in getContent and giftContent
function grantAccess(address u, address x) private;
```

2.4.2 BaseContentManagementContract

```
// gas cost: 800201
constructor() public;

// only revert the state
function () public;

// gas cost: at least 35594
// (it depends on how many contents are published in the catalog)
function _suicide() public onlyOwner;

// gas cost: 14472
function murder() public validAddress(catalog);

// gas cost: usually 41292,
// 86292 if it is the first view for this content,
// 42421 if the paymentAvailable event is emitted
function consumeContent() public returns(bytes);

// gas cost: at least 171503 (it depends on the name and genre length)
// If it is the first content published by the author,
// there is an additional expense of 80739 units of gas.
function publish(address c) public onlyOwner validAddress(c);
```

2.4.3 GenericContentManagementContract (is BaseContentManagementContract)

```
// gas cost: 1016403 (includes BaseContentManagementContract.constructor)
constructor() public;

// gas cost: at least 42510 (it depends on the input string)
function setName(bytes32 n) public onlyOwner;
```



```
// gas cost: at least 42510 (it depends on the input string)
function setGenre(bytes32 g) public onlyOwner;

// gas cost: at least 42510 (it depends on the length of the uint)
function setPrice(bytes32 g) public onlyOwner;
```

2.4.4 DAPPContentManagementContract (is GenericContentManagementContract)

```
// gas cost: 1217381 (includes GenericContentManagementContract.constructor)
constructor() public;

// gas cost: at least 42510 (it depends on the input string)
function setHostname(bytes32 n) public onlyOwner;

// gas cost: at least 42510 (it depends on the length of the uint)
function setPort(uint p) public onlyOwner;

// gas cost: at least 173532 (it depends on the name and genre length)
// (includes BaseContentManagementContract.publish cost)
// If it is the first content published by the author,
// there is an additional expense of 80739 units of gas.
function publish(address c) public onlyOwner;
```

3 DAPP Implementation

3.1 Programming language and tools

We decided, at the suggestion of professor Ricci, to develop the DAPP in Java. We choose Java 10 as programming language to take advantage of all the features of the latest Java versions, including in particular lambdas (11) and Functional Interfaces (12) that we use for the callbacks.

We also use Apache Maven (13) as project management tool.

3.2 Design choices

3.2.1 DAPP

We have chosen to make the modular DAPP, in order to make the code reusable. This allows in the future to develop another interface – CLI, API or web based – without having to rewrite already provided code.

In particular, the contracts module already includes the methods necessary to interact with contracts, and as such represents the heart of the DAPP.

3.2.2 GUI

It was required to develop two separate graphical interfaces, one for authors and one for customers, but since a user can be both author and customer at the same time we found it inconvenient to provide two different jars for each interface. Instead we opted for a more practical solution, which provides a common initial screen and then allows the user to choose which of the two interfaces to show. In this way the two interfaces are two thousand contained within the same archive and the author does not need to have two different programs.

Regarding the design of the style we decided to use the default style, that assumes the graphic style of the operating system to which the user is accustomed. This is to the advantage of usability and allows to reduce the size of the jar archive as we go to use elements already installed on the operating system in use, without having to pack new ones inside it.

3.2.3 Serving the content

Regarding the access content we have evaluated different strategies.

As already mentioned, the content can not be saved on the blockchain due to space problems and therefore to the cost and "pollution" of the blockchain, but it is not possible either to save a link or a reference that allows the user to independently access the content.

Indeed, the consume content function changes the status, so it can not return a value. Consequently, it would have been necessary to call another function before that which gives access to the content. In doing so, however, it would have been necessary to appeal to the good faith of the user who should call the consume content function right after he has accessed the content. However, the user is untrusted by nature, and therefore we can not rely on him. Indeed, he could call the content access function as often as he wants without having to pay it again.

For these reasons access to the content can not be charged to the Content Management Contract but it must instead be provided by the DAPP.

Even this solution is not the best, as it runs counter to the style of the Ethereum contracts that assure to all the parts their rights. Indeed, since the content is provided by DAPP, the user must trust those who developed it – the code may not be open source – and the author or a third part, who must really implement the DAPP and provide the content to the user.

However, since it is assumed that an author or a third part is more trustworthy than a user, we preferred this second solution.

At this point the problem was how the DAPP should provide the content. It was not conceivable to ask the authors to keep the interface always open, but at the same time the content have to be always online.

In this regard we have developed a second, minimal DAPP, to be run server side. It exposes APIs that interface the author's GUI to publish the contents and the user's GUI to request them. It must always be online and save the loaded contents inside it, in order to make them feasible even after the author has closed his GUI.

Therefore, each author must deploy the backend DAPP on his server.

Nowadays, The DAPP we develop only deals with sending the file to the user who requests it. This cannot be the right strategy for the final version, because once the user has obtained the file he can

consume it as many times as he wants, but it is good for a proof of work. A definitive version of the application, specialized therefore in a sector like film or music, will have to worry about managing such file in an proper way, such as a streaming, or protecting it with DRM. In any case, this is beyond the DAPP's tasks as it is a specific software aspect of the final product. The developed DAPP already takes into account the use of this external application – to be developed by the implementer – that should deal with managing the file ensuring the right protection. Therefore, no modifications are need on the software we develop.

3.3 Project structure

The code from the DAPP is inside the folder of the same name and is divided into 5 modules: contracts, gui, author-server, connections, and test.

All the classes of the project belong to the package `com.aldodaquino.cobra`, except for some classes that I have developed for other projects that belongs to the package `com.aldodaquino.javautils`.

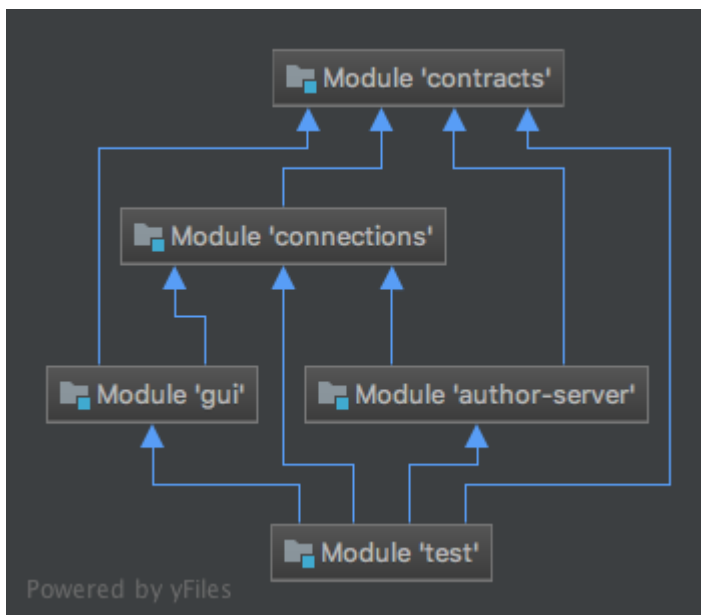


Figure 1 - UML diagram of the DAPP, showing module dependencies.

3.3.1 Contracts

The contracts module is the core of the DAPP. Its classes are divided into two packages: main and contracts. The second one contains the Java classes generated from the compiled solidity contracts. Main instead contains the classes to interact with the contracts.

The Contract Manager class contains the main methods to interact with contracts, like deploy, load, and suicide, but also method to retrieve the contract address and the contract owner's address.

Contract Manager is the superclass of Content Manager and Catalog Manager. This two classes implements method that calls the functions on the real contracts and parse and aggregate the result data. We can consider it a higher-level classes of the compiled ones.

Contents are modelled by the Content class. Content objects contains all the information about a content, but some of them can be null or unspecified. Three constructors allow a fast creation of this object. The purpose is to pass the result data as aggregated object, with an easy and intuitive access, from the contracts to the classes that need to use it.

The Utils class contains utilities to get information about the gas and to convert Strings to bytes32 and vice-versa.

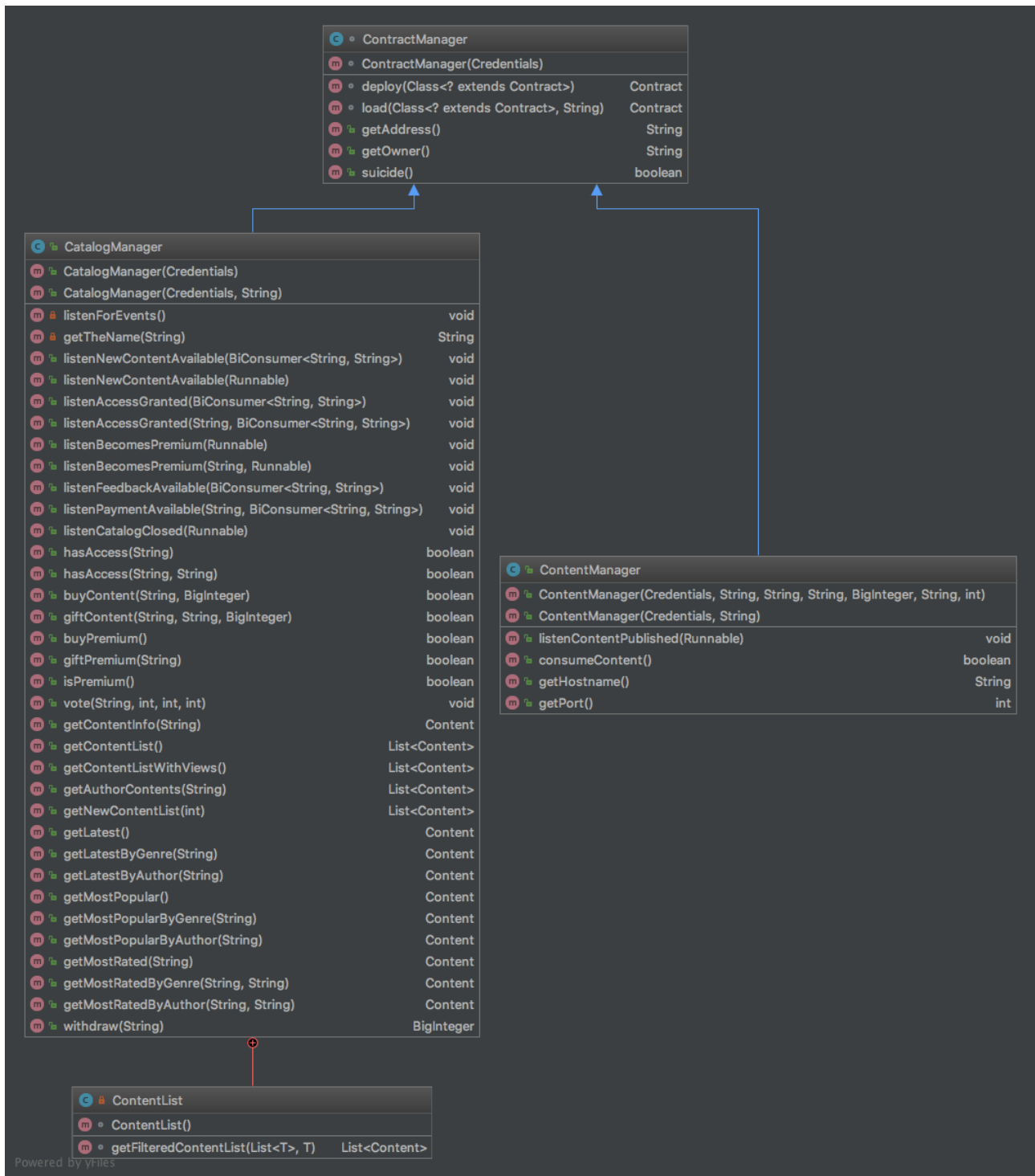


Figure 2 - UML diagram of the contracts module, showing class relationships.

3.3.2 GUI

The gui module contains two folders: res, for the images, and src for the code.

In the src folder we can find 3 main classes: Main, that starts the GUI, Status that includes methods to keep track of the current state and pass information between classes, and Utils with some utilities to create windows.

The GUI itself is divided into 3 packages: constants, components and panels.

Constants contains classes in which are stored colors, strings, images and dimensions used all around the gui. We have created this package to find and modify easy all the constants of the graphic interface, without need to find them around all the code.

Panels package contains all the final panels that are showed in the windows. The GUI opens with the Starter Panel, then will show the Customer Panel or the Author Panel, depending on the role that the user chooses. Genre Info Panel and Author Info Panel shows the charts of the most viewed and rated content of the chosen genre or author. Content Info Panel shows all the information about a selected content and allow the customer to buy it, gift it to another user or access it. After the customer has accessed it, he can rate the content through the Voting Panel. Pick User Panel is used to select the user to which gift a content or a premium subscription. Deploy Content Panel is used by authors to deploy a new content.

Components includes all the panels and components used by the classes in panels.

Component Factory makes the components creation faster.

Async Panel is a special panel that listens for ancestor changes and saves the window pointer in a protected variable. It also contains a protected method doAsync that execute a runnable in another thread and put the window in a state of loading until it finishes.

Upgradable panel is a child of Async Panel that allows also to replace components on-the-fly.

Logo, Login Form, Catalog Form and Role Form are used in the Starter Panel.

User Info shows information about the user, its premium status and the catalog to which the user is connected.

Info Panel is the super class of Author Info Panel, Genre Info Panel and Chart Widget, that is shown in the Customer Panel. It uses inside it the Label Panel.

New Content Widget is shown in the Customer Panel and allow the customer to choose how many contents to show in the new content list. This list is consultable in the Content List, that is present also in the Customer Panel to show the full list of contents in the catalog. This list is replaceable with the Views Content Table that shows also the number of view for each content.

In the Author Panel instead authors can find a more advanced list, that we display in the Author Content Table.

Finally, the Star Panel show the rating for a content or ask the user to set the rate for it.

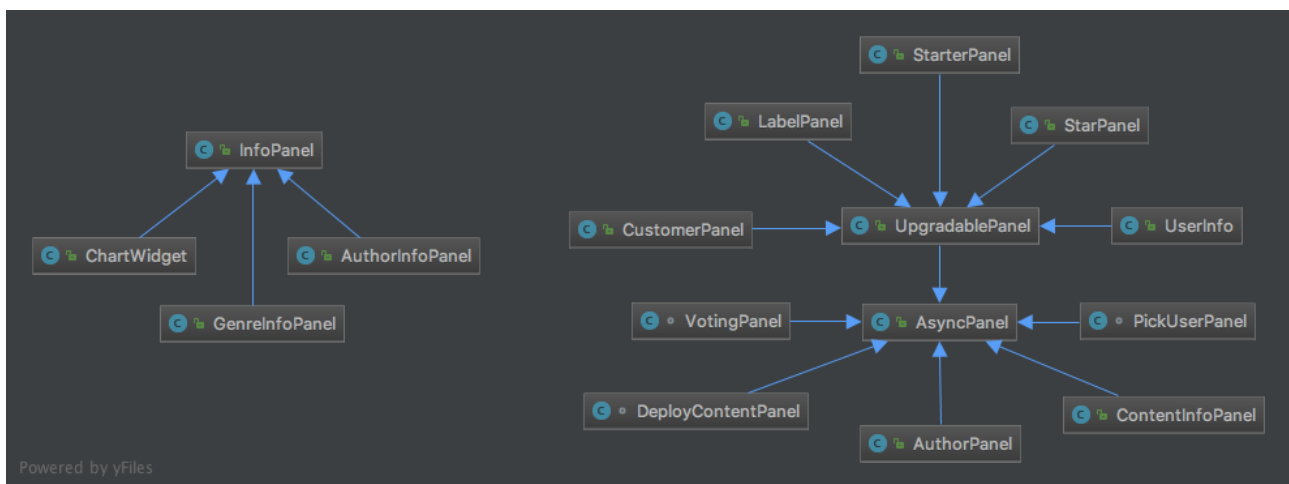


Figure 3 - UML diagram of the GUI module, showing class relationships.

3.3.3 Author server

The author-server module is a simple single-file DAPP. The Main class contains the main method that parses the command line options and prepares the server, and two listeners for the only two URL paths available: /deploy to deploy a new content and /access to consume a content.

There is also the Cli Helper class of our javautils package that helps to parse the CLI options passed as args to the main method.

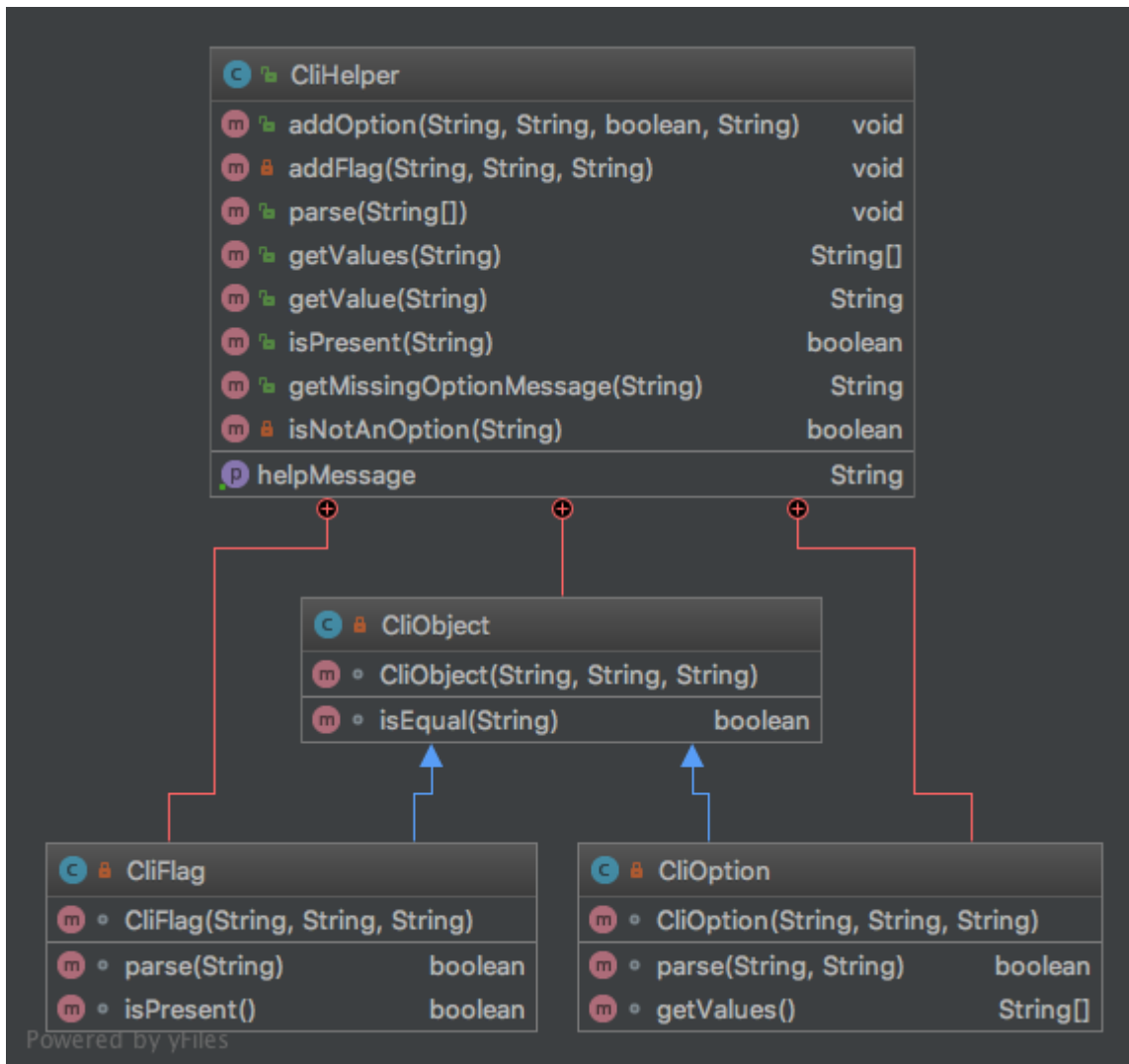


Figure 4 - UML diagram of the author server module, showing class relationships.

3.3.4 Connections

The connection module hosts all the utilities for the HTTP server shared between the GUI and the author-server.

You can find the API interface that specifies the URLs to be called on the author-server to deploy new contents or access the existent ones, the Status class that contains all the status information of the server, and the Cobra Http Helper that extends the functionalities of our Http Helper.

The Http Helper class and the File Exchange class in the javautils package helps to create a server, parse parameters and make HTTP requests and exchange files via web socket.

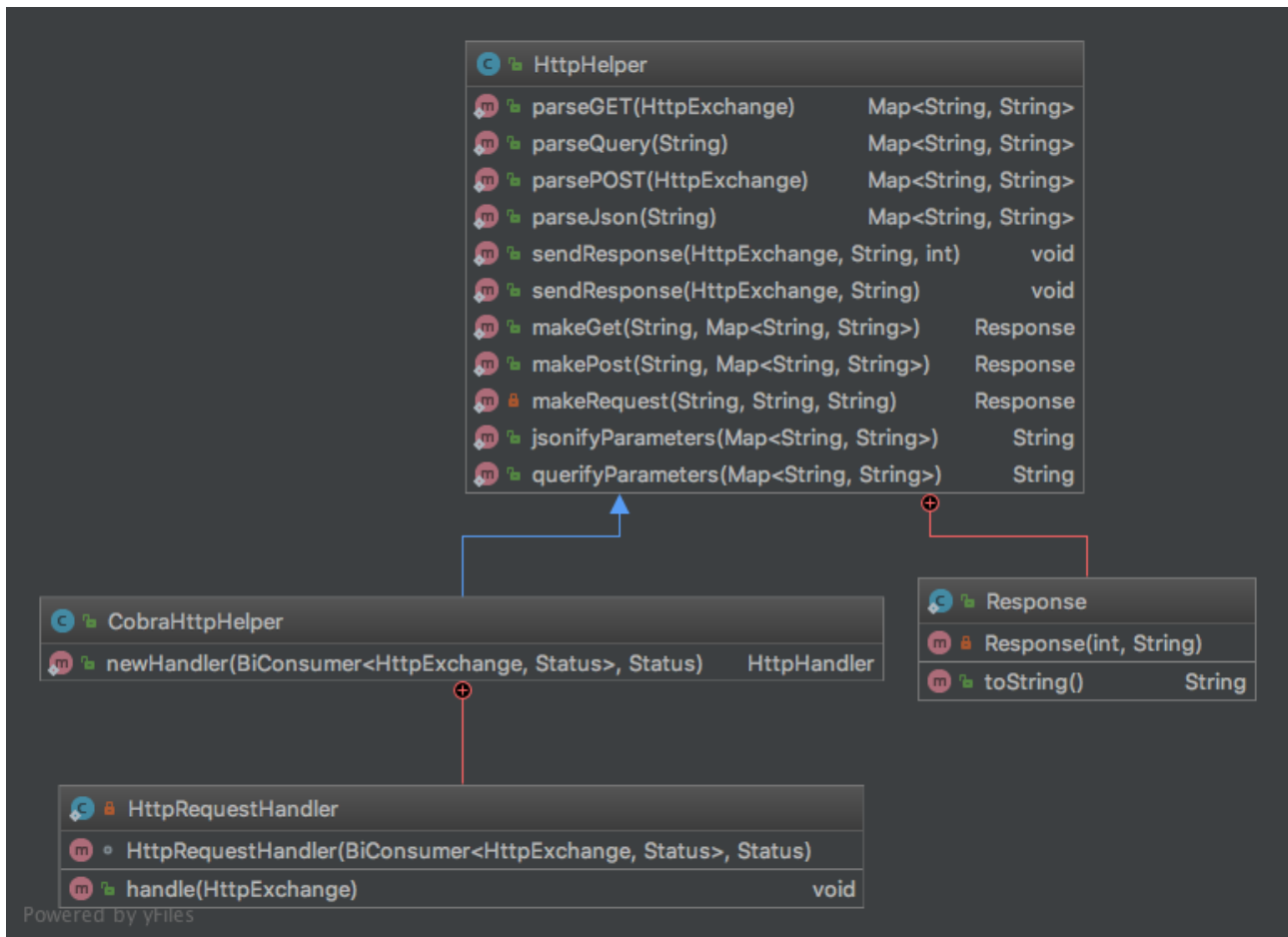


Figure 5 - UML diagram of the connections module, showing class relationships.

3.3.5 Test

The test package contains only a Main class and is an option package. It is not required but helps to create an environment in which test the DAPP. It deploys a Catalog, starts two author servers and publish some contents. Then starts 2 GUI, one for the Customer and one for the Author.

3.4 Execute the DAPP

3.4.1 Requirements

Ethereum (geth) (14)

- Ubuntu:
 - `sudo add-apt-repository -y ppa:ethereum/ethereum`
 - `sudo apt-get update`
 - `sudo apt-get install ethereum`

- **MacOS:**
 - `brew tap ethereum/ethereum`
 - `brew install ethereum`
- **Compile sources (requires Go (15)):**
 - `go install github.com/ethereum/go-ethereum/cmd/geth`

Solidity compiler (16)

- **Ubuntu:**
 - `sudo add-apt-repository ppa:ethereum/ethereum`
 - `sudo apt-get update`
 - `sudo apt-get install solc`
- **MacOS:**
 - `brew tap ethereum/ethereum`
 - `brew install solidity`
- **Build from sources (17)**

Java 10

- **JRE (18)**
- **JDK (19)**

Apache Maven (20)

- **Ubuntu:**
 - `sudo apt-get install maven`
- **MacOS:**
 - `sudo brew install maven`
- **Binaries (21)**

3.4.2 Compile sources

To compile solidity contracts, generate Java contracts with web3j and build JARs run the following command.

```
bash install.sh
```

3.4.3 Start Ethereum client

Start an Ethereum node on the testnet Ropsten using geth running the following command or start an emulated node with ganache-cli.

```
command geth --rpcapi personal,db,eth,net,web3 --rpc --testnet
```

3.4.4 Run the app

Run the GUI with the following command.

```
java -jar DAPP/jar/gui-1.0-jar-with-dependencies.jar
```

It starts a wizard that allows you to create credentials from your private key, deploy a new catalog or connect to an existent one and choose if you want the author's or the customer's view.

Authors also need a running author-server in which store their content in order to deploy a ContentManagementContract. You can run an author-server with the following command.

```
java -jar DAPP/jar/author-server-1.0-jar-with-dependencies.jar -k <your-private-key> -c <existent-catalog-contract-address>
```

The contract deploy can be done from the author's GUI, then the GUI can be stopped and the content will remain available as far as the author-server remain online.

3.5 GUI snapshots

3.5.1 Starter panel

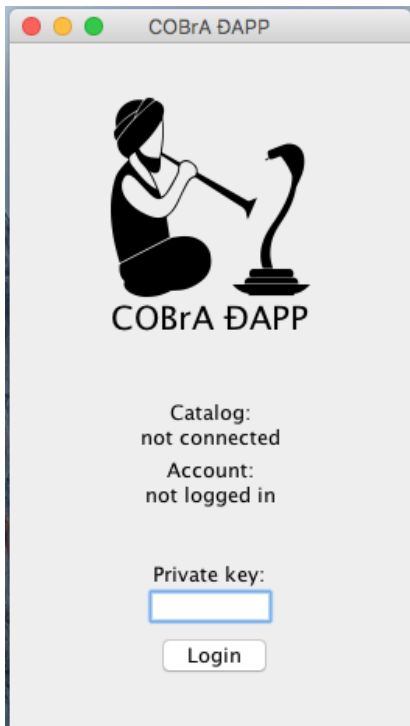


Figure 6 - The starter panel asking for private key.



Figure 7 - The starter panel asking for catalog address. The user can also deploy a new one.

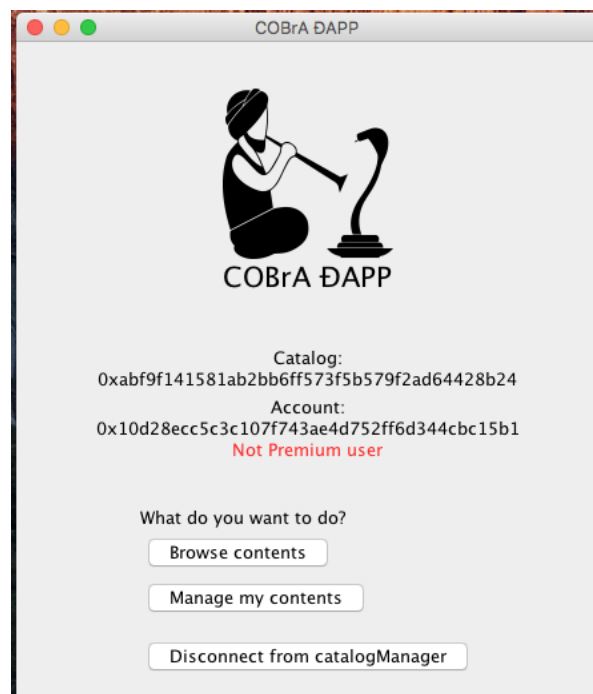


Figure 8 - The starter panel asking for the user role. The "Browse contents" buttons will show the customer panel, the "Manage my contents" one the author panel.

3.5.2 Customer panel



Figure 9 - The customer panel.

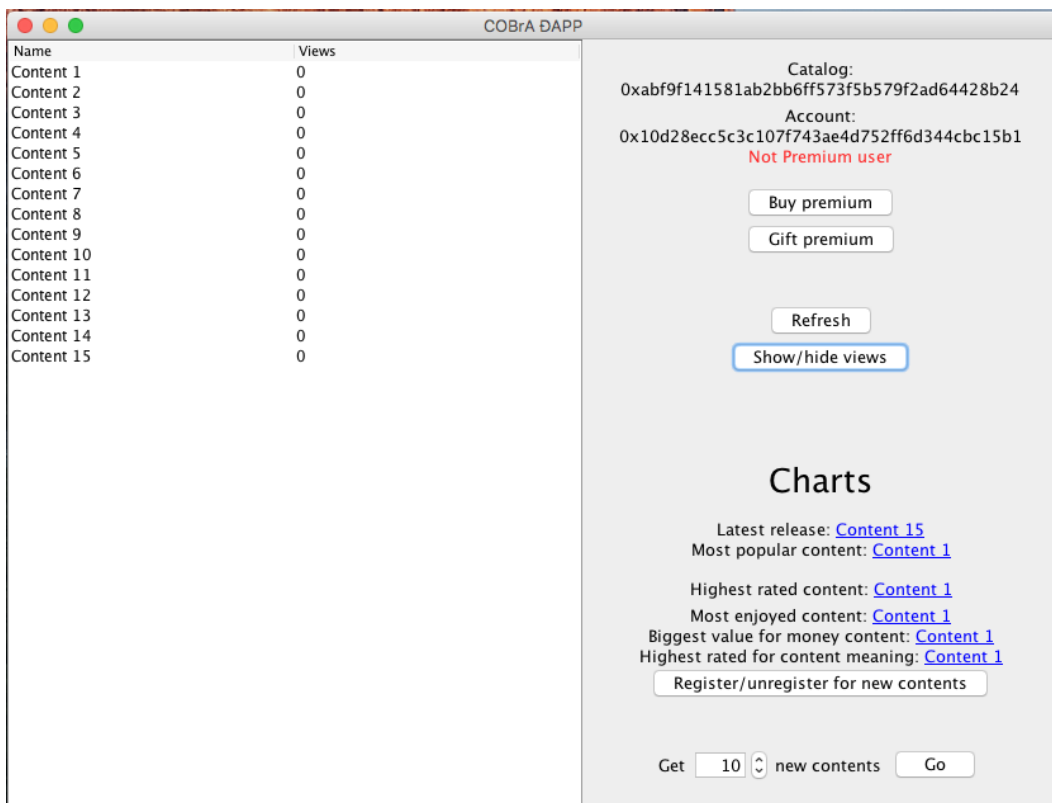


Figure 10 - The customer panel showing the number of views.

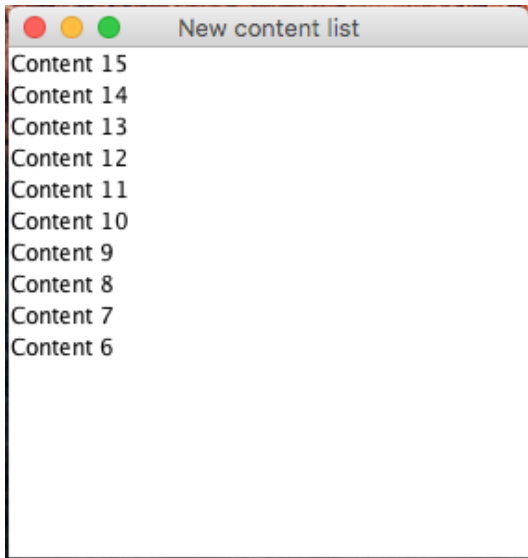


Figure 11 - The new content list.

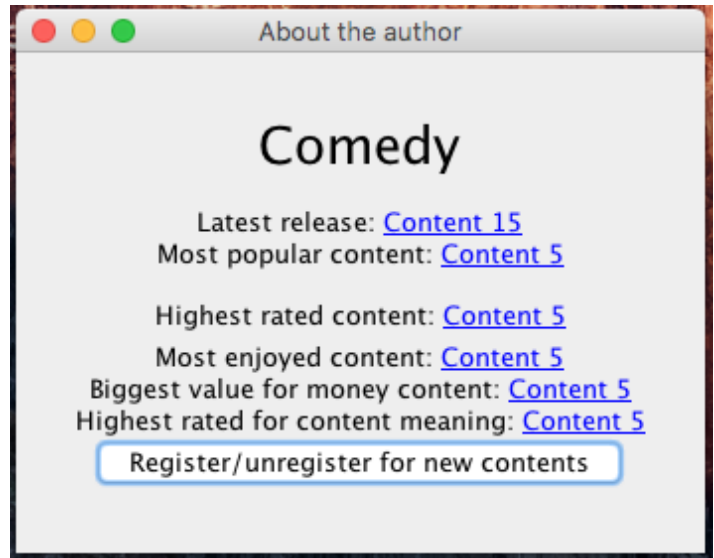


Figure 12 - Statistics about a genre.

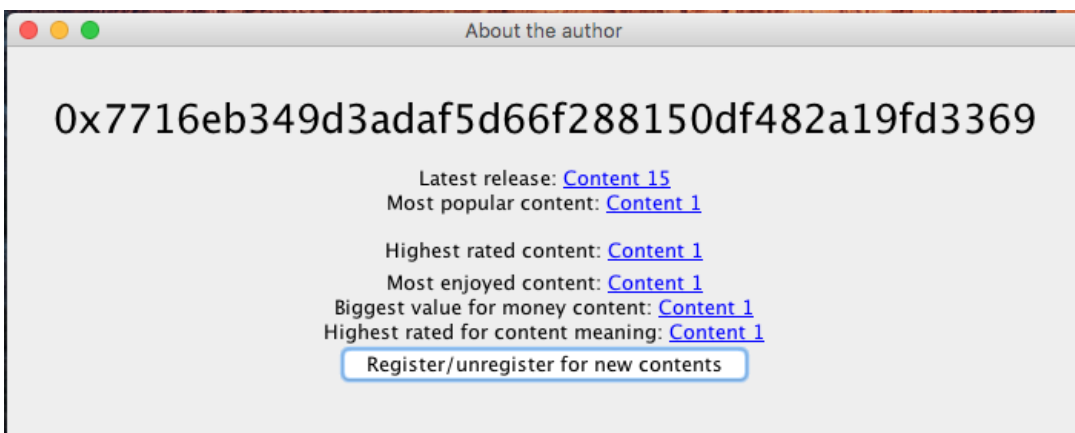


Figure 13 - Statistics about an author.

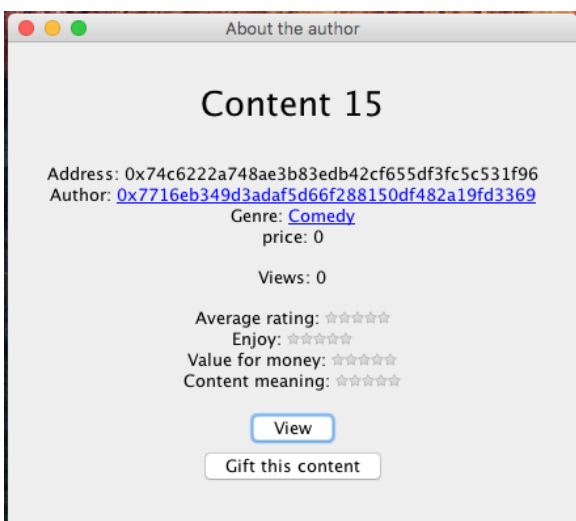
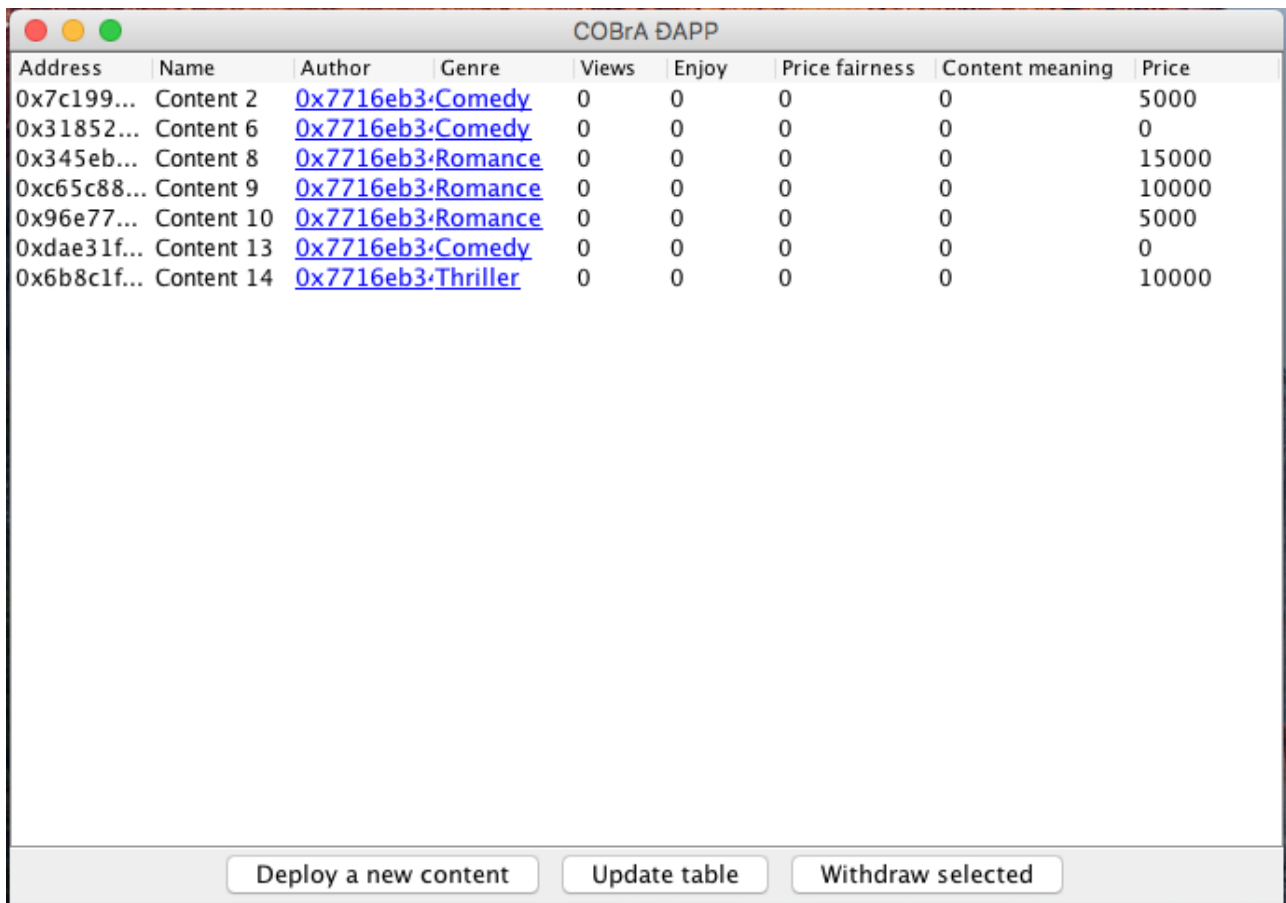


Figure 14 - Information about a content.

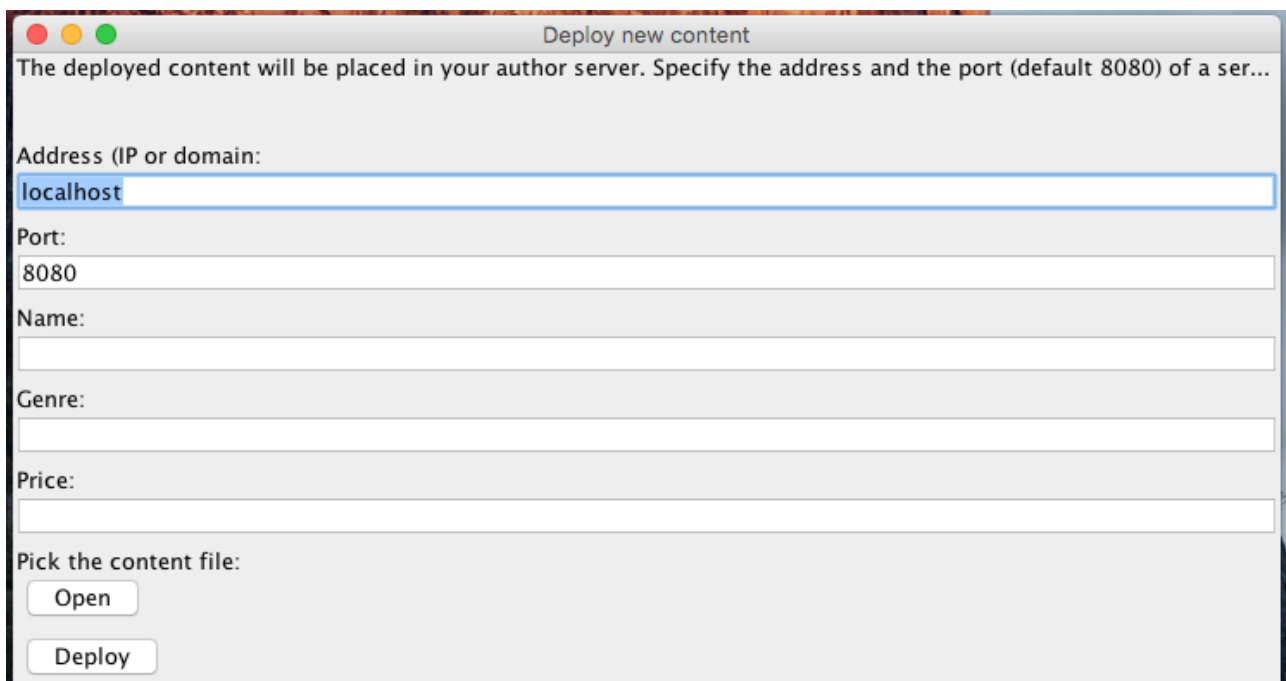
3.5.3 Author panel



Address	Name	Author	Genre	Views	Enjoy	Price fairness	Content meaning	Price
0x7c199...	Content 2	0x7716eb3•Comedy		0	0	0	0	5000
0x31852...	Content 6	0x7716eb3•Comedy		0	0	0	0	0
0x345eb...	Content 8	0x7716eb3•Romance		0	0	0	0	15000
0xc65c88...	Content 9	0x7716eb3•Romance		0	0	0	0	10000
0x96e77...	Content 10	0x7716eb3•Romance		0	0	0	0	5000
0xdae31f...	Content 13	0x7716eb3•Comedy		0	0	0	0	0
0x6b8c1f...	Content 14	0x7716eb3•Thriller		0	0	0	0	10000

Deploy a new content Update table Withdraw selected

Figure 15 - The author panel.



Deploy new content

The deployed content will be placed in your author server. Specify the address and the port (default 8080) of a ser...

Address (IP or domain):

Port:

Name:

Genre:

Price:

Pick the content file:

Figure 16 - The deploy content window.

4 Works Cited

1. Ethereum organization. Solidity Style Guide. *Read The Docs*. [Online] May 09, 2018. <http://solidity.readthedocs.io/en/v0.4.24/style-guide.html>.
2. —. Solidity version 0.4.24. *GitHub*. [Online] May 16, 2018. <https://github.com/ethereum/solidity/tree/v0.4.24>.
3. —. Solidity Style Guide. *Git Hub*. [Online] <https://github.com/ethereum/solidity/blob/develop/docs/style-guide.rst>.
4. —. Solidity. *GitHub*. [Online] <https://github.com/ethereum/solidity>.
5. Heesch, Dimitri van. Documenting the code. *Doxygen Manual*. [Online] May 23, 2018. <http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html>.
6. Wood, Gavin. Yellow Paper. *Ethereum repository on GitHub*. [Online] May 05, 2018. <https://ethereum.github.io/yellowpaper/paper.pdf>.
7. *ethereumprice*. [Online] [Cited: June 13, 2018.] <https://ethereumprice.org/>.
8. Ethereum organization. *Etherscan*. [Online] [Cited: June 9, 2018.] <https://etherscan.io/>.
9. McKie, Steven. Solidity Learning: Revert(), Assert(), and Require() in Solidity, and the New REVERT Opcode in the EVM. *Medium*. [Online] September 27, 2017. <https://medium.com/blockchannel/the-use-of-revert-assert-and-require-in-solidity-and-the-new-revert-opcode-in-the-evm-1a3a7990e06e>.
10. Ethereum organization. Solidity re-entrancy. *Read the Docs*. [Online] May 9, 2018. <http://solidity.readthedocs.io/en/develop/security-considerations.html#re-entrancy>.
11. Oracle. Lambda Expressions. *Java Website*. [Online] <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>.
12. Horstmann, Cay S. Choosing a Functional Interface. *Java SE8 for the Really Impatient: A Short Course on the Basics*. s.l. : Addison-Wesley Professional, 2014.
13. Apache. Home page. *Apache Maven*. [Online] <https://maven.apache.org/>.
14. Ethereum. Downloads. *Ethereum Geth*. [Online] <https://geth.ethereum.org/downloads/>.
15. Google. Download. *Golang*. [Online] <https://golang.org/dl/>.
16. Ethereum. Solidity. *Ethereum GitHub Repository*. [Online] <https://github.com/ethereum/solidity>.
17. —. Installing Solidity - Build from source. *Solidity Read the Docs*. [Online] <http://solidity.readthedocs.io/en/v0.4.24/installing-solidity.html#building-from-source>.

18. Oracle. Java Runtime Environment 10 Downloads. *Oracle*. [Online] <http://www.oracle.com/technetwork/java/javase/downloads/jre10-downloads-4417026.html>.
19. —. Java JDK 10 Downloads. *Oracle*. [Online] <http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html>.
20. Apache. Maven Install. *Apache Maven*. [Online] <https://maven.apache.org/install.html/>.
21. —. Binaries Download. *Apache Maven*. [Online] <https://maven.apache.org/download.cgi>.