P2P Final Project - Report

Student: Andrea Lisi

1 Introduction

The DApp exploits **Truffle suite**[1] for the development, installed via **nodejs**[2], together with other useful packages. The DApp is tested with either a local blockchain exploiting **Ganache**[3] and on Ropsten exploiting an **Infura**[4] node running an Ethereum client. The DApp's frontend is written in html/javascript and it's tested locally with Truffle (i.e. exploiting Truffle's commands).

The DApp project is organized as follows:

- **Contracts** folder, contains the backend:
  - **Migrations.sol**: Truffle's contract for initialization;
  - **Catalog.sol**: the Catalog contract implementation;
  - **BaseContentManagement.sol**: the BaseConentManagement contract implementation;
  - **BaseContentImplement.sol**: 3 examples of contents: PhotoContentManagement contract, SongContentManagement contract and VideoContentManagement contract.
- **Migrations** folder, contains the scripts used by Truffle to deploy contracts to the blockchain:
  - **1_initial_migration.js**: Truffle's file to init migration;
  - **2_deploy_contracts.js**: script which deploys the Catalog, some contents, performs some visual and gives some ratings (see section 4 for more explanation);
- **src/js** source code folder, containing:
  - **catalogApp.js**: the major functionality of the DApp;
  - **deployEditor.js**: the functions managing the content's creation;
  - **helpers.js**: helper functions;
  - **ohter js** such as bootstrap, web3 and truffle-contract;
- **src/index.html**: the Dapp's main page;
- **truffle.js**: Truffle's config file;
- **package.json**: nodejs packages.

2 The Backend: major changes

The major change from the first version of the Catalog includes the insertion of the *categories* to allow users to evaluate a content. This impacts not only on the "first impression", but also influences the reward obtained by a user after a payment is triggered.
The proposed categories are:

- **Quality**: represents how good the content is;

[1] https://truffleframework.com/truffle
[2] https://nodejs.org/en/
[3] https://truffleframework.com/ganache
[4] https://infura.io/

- **Price Fairness**: since the author chooses the price of its content, how much this price suits the content;
- **Rewatchable**: if the content is suited to be watched multiple times;
- **Family Friendly**: if the content is suited for people under the age of 18;

The functions to query the mostRatedBy* are implemented directly as getMostPopularBy*, and thus are supported by additional data structures updated every time a new most rated content by a certain category is triggered. This choice is justified following the same reasoning of the getMostPopularBy* queries, i.e. we expect that the categories updates will be triggered not very often and thus the price overhead will be rarely payed, as well as to avoid loops inside smart contracts.
Another little change to the backend concerns the insertion of the *user preferences*, since a user can ask to the DApp notifications only from interested authors and/or genre. These interests are stored on the Catalog.

Exploit the catagories

Since the amount of auhtor's payment depends on the average rate of a content, and since a customer can rate a content whether is premium or not, then an author could subscribe to the premium service and spam consumptions on its own contents in order to increase the average rating and thus get more reward. In the same way a malicious author could spam in order to decrease other author's rewards. Decreasing an author reward won't change the income of others though, so this behaviour could be induced by personal feelings.
Anyway, an author should think carefully whether or not increase it's own rating in this way. Even if with the premium account a customer does not pay for a content, he/she still pays the transaction. In order to increase its own rating the author should perform 3 transactions for each rating submission: buy, consume and rate. Unless that malicious author isn't a very successful one with the reward triggered frequently, these actions may not worth the investment.

3 The Frontend

The DApp is fully written in html/javascript. The DApp is divided in a Deploy Editor, a simple interface to deploy suitable COBrA contents, and the Catalog, an interface to interact with the COBrA Catalog.

3.1 Deploy Editor

This interface provides to the user the possibility to create a suitable content for the COBrA Catalog and deploy it into the blockchain. A user can select from 3 genres and each genre has attached a few dummy functionalities to personalise it, for example add a cover to a song or change a photo to a black and white one. After a content is deployed, the user gets the content's address, needed to link that content to the Catalog.
The client code is in **deployEditor.js**.

3.2 Catalog

This interface shows a notification sidebar, displaying major events, and the list of contents present in the Catalog. Clicking to a content is possible to see more information, such as author, views, price and ratings, as well as the possibility to buy or gift the content. If the user already has the access to it, the *buy* function is substituted by the *consume* one.

From the interface is possible to buy the premium service (or gift it to someone): buying and consuming a content as premium user is exactly the same as non premium ones, simply the consumption does not increase the views and does not cost content's price.
The DApp provides also a little bar to let the user to query the most popular content or the most rated one. Finally it's present the input field to insert the address of a deployed content (using the Deploy Editor) to link that content to the Catalog.
The button to shut COBrA down is visible only by the owner of the Catalog.
The client code is in **catalogApp.js** and in **helpers.js**.

3.3 Implementation details

The code in **catalogApp.js** follows this schema (deployEditor.js follows the same idea):

- the script is a big javascript object called App which defines a set of local variables and a set of functions;
- the variable *contracts* will contain the contract abstractions as *TruffleContracts* objects;
- as soon as the window is loaded the script loads web3 with Metamask as provider (if present);
- then the script reads the json files containing the compiled contracts using ajax call (in order to do this, a local server should be running) and stores them into *contracts*;
- after, sets up the listeners for events emitted by the backend. Some listeners start from the current block to listen for events that are supposed to be fired when the user buys/consumes a content; other listeners, instead, start a few blocks earlier in order to fill the notification sidebar up;
- finally, the script loads the content list from the catalog contract and renders the page.

The other functions are just listeners attached to the UI.
Every time the user triggers a transaction (and thus consume gas) the DApp displays a little notification and then opens Metamask to allow the user to confirm or reject the transaction.

3.4 Actions example

- **Perform queries**: it's possible to perform some query such as "Most popular by *". To do this, just combine the search options below "Need some idea?" label, for example "Most popular by author" and type an author/genre name in the input line and press "Watch" button;
- **Personalise notification sidebar**: type an author and refresh page to see more notifications on the sidebar;
- **Buy/Consume content**: select a content from the list, press "buy" and, after the transaction is completed, select the same content again and press "consume". Optionally, rate the content;
- **Gift content**: as before, but click on "make a gift", insert the dest address and press "buy gift". Switch account in order to consume the content;
- **Deploy a new content**: on the nav bar above click on "publish center", fill the inputs and press "Load". This will give the contract's address;
- **Attach deployed content**: in the Catalog, insert the content's address in the lowest input space and press "publish". Refresh the page to see the new content on the list;
- **Destroy COBrA**: with the Catalog's creator account selected on Metamask press the Big Red Button: destructing COBrA will destroy all the linked contents as well.

4 Install and test the project

How to install the project and run it. **Requires: nodejs and Metamask**.
Type `npm install truffle -g` to install Truffle; type `npm install` to install all the project's dependencies and store them into node_modules/;

4.1 Local testing

Install and run Ganache. While Ganache is running, from the script *src/js/2_ deploy_ contracts.js* is possible to index the fake accounts to explicit who should deploy a contract and call a function / start a transaction. To use them from Metamask it's enough to import them copying the private key.

1. Run Ganache (port should be 8545);
2. go to *src/js/2_ deploy_ contracts.js* and un-comment the code starting from the line stating *"UN-COMMENT this for local deployment"*: this will initialize the blockchain with some content and visual;
3. type `truffle migrate --reset`: compiles the contracts (creates *build* folder) and runs the scripts inside *migration* folder in order;
4. open new tab and `npm run dev` to start a local server to enable ajax calls to read contract's abis;
5. open Metamask, connect it to http://127.0.0.1:8545, import one or more accounts from Ganache copying the private key. If error should occur, try to **reset the accounts from settings/reset account** (this is needed for example between different migrations during development). Refresh the page.

4.1.1 Action sequence

Considering the actions listed in section 3.4 and since the Catalog should already have some contents deployed:

- **Perform some query** for example with the combination "most popular of genre", "photo" or "best of category", "quality";
- **Personalise notification sidebar** typing "Paolo Villaggio", refresh the page: some changes should appear on the notification sidebar;
- **Deploy and attach a new content** once or more times;
- **Buy/Consume Content** once or more times;
- **Trigger a payment**: buy and consume the content "Solidity Tutorial" to trigger a payment. The payment notification will be displayed on the notification sidebar;
- **Destroy COBrA**.

4.2 Testing on Ropsten

The testing on Ropsten follows the steps shown in this Truffle page: `https://truffleframework.com/tutorials/using-infura-custom-provider`.

1. go to *src/js/2_ deploy_ contracts.js* and un-comment the code starting from the line stating *"UN-COMMENT this for for deployment on ropsten"*: this will initialize a blockchain with only the Catalog (eventually, comment the block for local deployment);
2. open Metamask and use a Ropsten account (i.e. an account with Ropsten ether);
3. go to *truffle.js*, copy the 12 security words of Metamask in the variable "mnemonic" as string and set the variable "account_index" to the index of the Ropsten Metamask's account to use to deploy the Catalog (the first account should be 0, the second 1 etc);
4. `truffle migrate --network ropsten --reset`: compiles the contracts (create *build* folder) and runs the scripts inside *migration* folder in order. It may take a while in comparison to the local testing. **If it fails** stating *Error: Contract transaction couldn't be found after 50 blocks*[5], then try to increase the gas limit in *truffle.js* (currently set to 30GWei);
5. open new tab and `npm run dev` to start a local server to enable ajax calls to read contract's abis;

4.2.1 Action sequence

Considering the actions listed in section 3.4 and the Catalog being empty:

- **Deploy and attach a new content** once or more times: if contract's address would be lost in case the page refreshes is possible to retrieve it from etherscan, clicking on the transaction from Metamask;
- **Buy/Consume Content** once or more times;
- **Destroy COBrA**.

[5] web3 has it's own hardcoded wait limit: 50 blocks: `https://github.com/trufflesuite/truffle/issues/594`

Catalog

```solidity
pragma solidity ^0.4.18;
import "./BaseContentManagement.sol";
import "./BaseContentImplement.sol";

contract Catalog {

    ////////////////////////////////////////////////////////////////
    ///////                  State Variables                  ///////
    ////////////////////////////////////////////////////////////////

    event UserAccess(address _user, bytes32 _content);
    event UserConsume(address _user, bytes32 _content);
    event NewPremiumUser(address _user);
    event NewPopularByAuthor(bytes32 _author, bytes32 _content);
    event NewPopularByGenre(bytes32 _genre, bytes32 _content);
    event NewLatestByAuthor(bytes32 _author, bytes32 _content);
    event NewLatestByGenre(bytes32 _genre, bytes32 _content);
    event ContentRated(bytes32 _content, uint8 _category);
    event AuthorPayed(bytes32 _author, uint _reward);
    event COBrAShutDown();

    // Utilities
    address public COBrA_CEO_Address;

    uint constant public authorRewardPeriod = 5; // author gets payed every 10 views

    uint constant public premiumCost = 0.04 ether; // 18 eur
    uint constant public premiumPeriod = 6170;  // more or less 24h

    uint public totalViews = 0;

    // Popular mappings
    mapping(address => uint) premiumUsers;
    mapping(bytes32 => BaseContentManagement) public contentMap;
    // author => content
    mapping(bytes32 => bytes32) latestAuthorMap;
    mapping(bytes32 => bytes32) latestGenreMap;
    mapping(bytes32 => bytes32) mostPopularAuthorMap;
    mapping(bytes32 => bytes32) mostPopularGenreMap;

    // Content List
    bytes32[] contentList;

    // Categories utilities
        // Note: Average is not a category, is just a placeholder to store the most rated
            content in general
    enum Categories { Quality, PriceFairness, Rewatchable, FamilyFriendly, Average }
    uint constant public numCategories = 4;
    uint constant public minRate = 1;
    uint constant public maxRate = 10;

    // Rate mappings
    mapping(uint8 => bytes32) public mostRatedContent;
    // author => (category => content)
    mapping(bytes32 => mapping(uint8 => bytes32)) public mostRatedByAuthor;
    mapping(bytes32 => mapping(uint8 => bytes32)) public mostRatedByGenre;

    // User preferences
    mapping(address => bytes32[]) public userPreferences;


    ////////////////////////////////////////////////////////////////
```

```
    ///////                     Modifiers                       ///////
    ////////////////////////////////////////////////////////////////////


    /// @notice check whether the sender is a premium account
    modifier isUserPremium(address _user) {
        require(block.number <= premiumUsers[_user], "User isn't active premium");
        _;
    }


    /// @notice check whether the sender is the CEO
    modifier isCEO() {
        require(msg.sender == COBrA_CEO_Address, "Only the CEO can destruct the Catalog ")
            ;
        _;
    }


    ///@notice Check if the payment is correct
    ///@param value payed
    ///@param price to be payed
    modifier priceCorrect(uint value, uint price) {
        require(value == price, "Sent incorrect value");
        _;
    }


    /// @notice check wheter the content is already deployed
    modifier isDeployed(bytes32 _content) {
        require(contentMap[_content] != address(0x0), "Content not deployed");
        _;
    }


    /// @notice check wheter the content is not deployed yet
    modifier isNotDeployed(address _content) {
        require(_content == address(0x0), "Content already deployed");
        _;
    }

    ///@notice Check if a given category is valid
    ///@param _category the category
    modifier validCategory(uint _category) {
        require(_category == uint(Categories.Quality) ||
                _category == uint(Categories.PriceFairness) ||
                _category == uint(Categories.Rewatchable) ||
                _category == uint(Categories.FamilyFriendly) ||
                _category == uint(Categories.Average), "Invalid cateogry");

        _;
    }

    ///@notice Check if an array of ratings is valid
    ///@param _ratings the rating array
    modifier validRating(uint[] _ratings) {

        require(_ratings.length == numCategories, "Rating array not valid");

        for(uint i=0; i<_ratings.length; i++) {
            require(_ratings[i] >= minRate, "Invalid lower bound rating");
            require(_ratings[i] <= maxRate, "Invalid upper bound rating");
        }
        _;
```

```
125     }
126
127     ///@notice Check if the sender is the manager of a given content
128     ///@param _content the content
129     modifier correctManager(bytes32 _content) {
130
131         require(msg.sender == address(contentMap[_content]), "Caller isn't the content's
                manager");
132         _;
133     }
134
135
136     ////////////////////////////////////////////////////////////////
137     ///////              Contract's functions                  ///////
138     ////////////////////////////////////////////////////////////////
139
140     constructor() public {
141
142         COBrA_CEO_Address = msg.sender;
143     }
144
145
146         ///////////////////////////////////
147         ///         Modify the State    ///
148         ///////////////////////////////////
149
150
151
152
153     ///@notice Add a new content to the catalog, if not present
154     ///@param _content The address of the content's contract
155     function addContent(BaseContentManagement _content) public
156                                                 isNotDeployed(contentMap[_content.title()
                                                        ]) {
157
158         // Be sure the content's catalog address is "this" one
159         require(_content.catalog() == this, "Wrong stored Catalog address");
160         require(_content.authorAddress() == msg.sender, "Caller isn't the content's author
                ");
161
162         contentMap[_content.title()] = _content;
163         contentList.push(_content.title());
164
165         // Update latest author's content
166         latestAuthorMap[_content.author()] = _content.title();
167         latestGenreMap[_content.getGenre()] = _content.title();
168
169         emit NewLatestByAuthor(_content.author(), _content.title());
170         emit NewLatestByGenre(_content.getGenre(), _content.title());
171     }
172
173
174
175     ///@notice Gain the access to a content
176     ///@param _content the id of the content
177     function getContent(bytes32 _content) external payable
178                                 isDeployed(_content)
179                                 priceCorrect(msg.value, contentMap[_content].price()) {
180
181         contentMap[_content].grantAccess(msg.sender);
182         emit UserAccess(msg.sender, _content);
183     }
184
185
```

```
186
187     ///@notice Gain the premium access to a content
188     ///@param _content the title of the content
189     function getContentPremium(bytes32 _content) external
190                                             isDeployed(_content)
191                                             isUserPremium(msg.sender) {
192
193         contentMap[_content].grantAccess(msg.sender);
194         emit UserAccess(msg.sender, _content);
195     }
196
197
198
199     ///@notice Let a user buy a premium account
200     function buyPremium() external payable
201                                 priceCorrect(msg.value, premiumCost) {
202
203         premiumUsers[msg.sender] = block.number + premiumPeriod;
204         emit NewPremiumUser(msg.sender);
205     }
206
207
208
209     ///@notice Give to a user the access to a content
210     ///@param _content the id of the content
211     ///@param _dest the address of the receiver
212     function giftContent(bytes32 _content, address _dest) external payable
213                                 priceCorrect(msg.value, contentMap[_content].price()) {
214
215         contentMap[_content].grantAccess(_dest);
216         emit UserAccess(_dest, _content);
217     }
218
219
220
221     ///@notice Give to a user the premium account
222     ///@param _dest the address of the receiver
223     function giftPremium(address _dest) external payable
224                                 priceCorrect(msg.value, premiumCost) {
225
226         premiumUsers[_dest] = block.number + premiumPeriod;
227         emit NewPremiumUser(_dest);
228     }
229
230
231     /// @notice Get a notification from a content manager that it was consumed
232     /// @param _content the calling contract
233     /// @param _user the sender
234     /// @param _premiumView if the content was consumed by a premium user
235     /// @dev this function should be called only by a ContentManagement contract
236     function notifyConsumption(bytes32 _content, address _user, bool _premiumView)
            external  correctManager(_content){
237
238         emit UserConsume(_user, _content);
239
240         if(!_premiumView) totalViews++;
241
242         // Update current most popular content of the author
243         updateMostPopularByAuthor(_content);
244
245         // Update current most popular content of a genre
246         updateMostPopularByGenre(_content);
247
248         // Check for payment
```

```
        if(contentMap[_content].views() % authorRewardPeriod == 0){

            uint reward = computeReward(_content);
            contentMap[_content].authorAddress().transfer(reward);

            emit AuthorPayed(contentMap[_content].author(), reward);
        }
    }

    /// @notice Rate a content
    /// @param _content The content to rate
    /// @param ratings The rating array
    function rateContent(bytes32 _content, uint[] ratings) external validRating(ratings) {

        contentMap[_content].rateContent(ratings);
    }


    /// @notice Get a rating notification by the manager of a content
    /// @param _content The rated content
    function notifyRating(bytes32 _content) external correctManager(_content) {

        updateCategory(_content, uint8(Categories.Quality));
        updateCategory(_content, uint8(Categories.PriceFairness));
        updateCategory(_content, uint8(Categories.Rewatchable));
        updateCategory(_content, uint8(Categories.FamilyFriendly));

        uint sum = contentMap[_content].getRateSum();
        bytes32 _bestRatedContent = mostRatedContent[uint8(Categories.Average)];
        // Update best rated content
        if(_bestRatedContent == 0x0) {
            mostRatedContent[uint8(Categories.Average)] = _content;
        }
        else {

            if(sum > contentMap[_bestRatedContent].getRateSum())
                mostRatedContent[uint8(Categories.Average)] = _content;
        }

        // Update best rated by author
        updateBestRatedByAuthor(_content, sum);

        // Update best rated by genre
        updateBestRatedByGenre(_content, sum);
    }


    /// @notice Add a user preference (author or genre)
    /// @param _label The name of the author or genre
    function addPreference(bytes32 _label) external {

        userPreferences[msg.sender].push(_label);
    }

    /// @notice Get the number of preferences of that user
    /// @return The number of preferences of that user
    function getPreferenceCount() external view returns(uint) {

        return userPreferences[msg.sender].length;
    }



        ////////////////////////////////////
```

```
313         ///             Views           ///
314         ////////////////////////////////////
315
316
317     /// @notice Check if a user is premium
318     /// @param _user the requested user
319     /// @return true if the user is premium, false otherwise
320     function isPremium(address _user) public view returns(bool) {
321
322         return block.number <= premiumUsers[_user];
323     }
324
325
326     /// @notice Get the list of contents and their number of views
327     /// @return the list of the titles of the contents
328     /// @return the list of the views of each content
329     function getStatistics() external view returns(bytes32[], uint[]) {
330
331         uint[] memory _viewList = new uint[](contentList.length);
332
333         for(uint i = 0; i < contentList.length; i++)
334             _viewList[i] = contentMap[contentList[i]].views();
335
336         return (contentList, _viewList);
337     }
338
339
340
341     /// @notice Get list of the contents in the catalog
342     /// @return the list of the titles of the contents
343     function getContentList() external view returns(bytes32[]) {
344
345         return contentList;
346     }
347
348
349
350     /// @notice Get list of the newest contents in the catalog
351     /// @param _number the number of newest contents requested
352     /// @return the list of the _number titles of the newest contents
353     /// @dev the casting is needed since using uint instead of int while iterating
            backward faces the underflow problem when i = 0 and the loop performs i--
354     function getNewContentList(uint _number) external view returns(bytes32[]) {
355
356         // Keep the minimum between _number and the length of the catalog
357         int len = int(contentList.length);
358         int min = int(_number);
359
360         if(min > len)
361             min = len;
362
363         bytes32[] memory _list = new bytes32[](uint(min));
364
365         for(int i = len - 1; i >= len - min; i--)
366             _list[uint(len - 1 - i)] = contentList[uint(i)];
367
368         return _list;
369     }
370
371
372     /// @notice Get the newest content of a given genre
373     /// @param _genre the genre of the requested content
374     /// @return the title of the newest content
375     function getLatestByGenre(bytes32 _genre) external view returns(bytes32) {
```

```
376
377         return latestGenreMap[_genre];
378     }
379
380
381
382     /// @notice Get the newest content of a given author
383     /// @param _author the author of the requested content
384     /// @return the title of the newest content
385     function getLatestByAuthor(bytes32 _author) external view returns(bytes32) {
386
387         return latestAuthorMap[_author];
388     }
389
390
391     /// @notice Get the most popular content of a given genre
392     /// @param _genre the genre of the requested content
393     /// @return the title of the most popular content
394     function getMostPopularByGenre(bytes32 _genre) external view returns(bytes32){
395
396         return mostPopularGenreMap[_genre];
397     }
398
399
400
401     /// @notice Get the most popular content of a given author
402     /// @param _author the author of the requested content
403     /// @return the title of the most popular content
404     function getMostPopularByAuthor(bytes32 _author) external view returns(bytes32){
405
406         return mostPopularAuthorMap[_author];
407     }
408
409
410     /// @notice Get the most rated content of a given category
411     /// @param _category the category of the requested content
412     /// @return the title of the most popular content
413     function getMostRated(uint8 _category) external view validCategory(_category)
414                                                          returns(bytes32) {
415
416         return mostRatedContent[_category];
417     }
418
419
420     /// @notice Get the most rated content of a given category of a given author
421     /// @param _category the category of the requested content
422     /// @param _author the author of the requested content
423     /// @return the title of the most popular content
424     function getMostRatedByAuthor(bytes32 _author, uint8 _category) external view
425                                                                     validCategory(
                                                                         _category)
426                                                                     returns(bytes32) {
427
428         return mostRatedByAuthor[_author][_category];
429     }
430
431
432     /// @notice Get the most rated content of a given category
433     /// @param _category the category of the requested content
434     /// @param _genre the genre of the requested content
435     /// @return the title of the most popular content
436     function getMostRatedByGenre(bytes32 _genre, uint8 _category) external view
437                                                                   validCategory(_category)
438                                                                   returns(bytes32) {
```

```
439
440         return mostRatedByGenre[_genre][_category];
441     }
442
443
444         ////////////////////////////////
445         ///         Self Destruct      ///
446         ////////////////////////////////
447
448
449     /// @notice Destruct the catalog, pay the authors proportionally to the views: if any
            content was consumed, pay every content equally
450     function destructCOBrA() external isCEO {
451
452
453         uint i = 0;
454         uint _factor = totalViews;
455         uint _totalBalance = address(this).balance;
456         bytes32 _content = 0x0;
457
458         if(_factor == 0) // No content was viewed
459             _factor = contentList.length;
460
461         if(_factor > 0) {
462             // At least  a view or a content
463
464             if(totalViews > 0){
465                 // Divide the balance proportionally to the views
466                 for(i=0; i<contentList.length; i++) {
467
468
469                     _content = contentList[i];
470                     payAndDestroy(_content, contentMap[_content].views(), _factor,
                            _totalBalance);
471                 }
472             }
473
474             else {
475                 // Divide the balance equally for each content
476                 for(i=0; i<contentList.length; i++){
477
478                     _content = contentList[i];
479                     payAndDestroy(_content, 1, _factor, _totalBalance);
480                 }
481             }
482         }
483
484         emit COBrAShutDown();
485         selfdestruct(COBrA_CEO_Address);
486     }
487
488     function payAndDestroy(bytes32 _content, uint _multiplier, uint _factor, uint
            _totalBalance) private {
489
490         finalPayment(_content, _multiplier, _factor, _totalBalance);
491         contentMap[_content].destruct();
492     }
493
494
495
496         ////////////////////////////////
497         ///            Helpers         ///
498         ////////////////////////////////
499
```

```
500
501     /// @notice Send the payment to authors during the last payment cycle
502     /// @param _content the current content
503     /// @param _multiplier the multiplier for the amount computation
504     /// @param _factor the factor for the amount computation
505     function finalPayment(bytes32 _content, uint _multiplier, uint _factor, uint _balance)
            private {
506
507         uint _amount = (_multiplier * _balance) / _factor;
508         address _author = contentMap[_content].authorAddress();
509         _author.transfer(_amount);
510         emit AuthorPayed(contentMap[_content].author(), _amount);
511     }
512
513
514     /// @notice Compute the raward for an author
515     /// @param _content The content viewed
516     /// @return The amount of wei for the author
517     function computeReward(bytes32 _content) private view returns(uint) {
518
519         uint rate = contentMap[_content].getRateSum();
520         uint max = numCategories * maxRate;
521         return (contentMap[_content].price() * rate) / max;
522     }
523
524
525     /// @notice Check whether to update the best content of a given category
526     /// @param _content The viewed content
527     /// @param _category A content's category
528     function updateCategory(bytes32 _content, uint8 _category) private {
529
530         // Most rated in general
531         bytes32 _bestRated = mostRatedContent[_category];
532         uint _popularRate = 0;
533
534         if(_bestRated == 0x0) {
535             // First rating
536             mostRatedContent[_category] = _content;
537         }
538         else {
539
540             _popularRate = contentMap[_bestRated].getRate(_category);
541
542             if(contentMap[_content].getRate(_category) > _popularRate)
543                 mostRatedContent[_category] = _content;
544         }
545
546
547         // Update most rated by author
548         updateMostRatedByAuthor(_content, _category);
549
550         // Update most rated by genre
551         updateMostRatedByGenre(_content, _category);
552     }
553
554
555     ////
556     // MOST  == The most rated for a category
557     ////
558     /// @notice Check whether to update the best content of an author of a given category
559     /// @param _content The viewed content
560     /// @param _category A content's category
561     function updateMostRatedByAuthor(bytes32 _content, uint8 _category) private  {
562
```

```
563         bytes32 _author = contentMap[_content].author();
564         bytes32 _bestRatedByAuthor = mostRatedByAuthor[_author][_category];
565         uint _popularRate = 0;
566 
567         if(_bestRatedByAuthor == 0x0) {
568             // First rating to a content to that genre
569             mostRatedByAuthor[_author][_category] = _content;
570         }
571         else {
572 
573             _popularRate = contentMap[_bestRatedByAuthor].getRate(_category);
574 
575             if(contentMap[_content].getRate(_category) > _popularRate) {
576 
577                 mostRatedByAuthor[_author][_category] = _content;
578             }
579         }
580     }
581 
582 
583     /// @notice Check whether to update the best content of a genre of a given category
584     /// @param _content The viewed content
585     /// @param _category A content's category
586     function updateMostRatedByGenre(bytes32 _content, uint8 _category) private  {
587 
588         bytes32 _genre = contentMap[_content].getGenre();
589         bytes32 _bestRatedByGenre = mostRatedByGenre[_genre][_category];
590         uint _popularRate = 0;
591 
592         if(_bestRatedByGenre == 0x0) {
593             // First rating to a content to that genre
594             mostRatedByGenre[_genre][_category] = _content;
595         }
596         else {
597 
598             _popularRate = contentMap[_bestRatedByGenre].getRate(_category);
599 
600             if(contentMap[_content].getRate(_category) > _popularRate)
601                 mostRatedByGenre[_genre][_category] = _content;
602         }
603     }
604 
605 
606     ////
607     // BEST == The most rated in avg
608     ////
609     /// @notice Check whether to update the best content of an author (i.e. the the
        highest category's avg rating)
610     /// @param _content The viewed content
611     /// @param _contentSum The sum of the content's ratings
612     function updateBestRatedByAuthor(bytes32 _content, uint _contentSum) private {
613 
614         bytes32 _author = contentMap[_content].author();
615         uint8 avgPos = uint8(Categories.Average);
616 
617         if(mostRatedByAuthor[_author][avgPos] == 0x0)
618             mostRatedByAuthor[_author][avgPos] = _content;
619         else {
620 
621             bytes32 _bestByAuthor = mostRatedByAuthor[_author][avgPos];
622 
623             if(_contentSum > contentMap[_bestByAuthor].getRateSum())
624                 mostRatedByAuthor[_author][avgPos] = _content;
625         }
```

```
626     }
627
628
629     /// @notice Check whether to update the best content of a genre (i.e. the the highest
            category's avg rating)
630     /// @param _content The viewed content
631     /// @param _contentSum The sum of the content's ratings
632     function updateBestRatedByGenre(bytes32 _content, uint _contentSum) private {
633
634         bytes32 _genre = contentMap[_content].getGenre();
635         uint8 avgPos = uint8(Categories.Average);
636
637         if(mostRatedByGenre[_genre][avgPos] == 0x0)
638             mostRatedByGenre[_genre][avgPos] = _content;
639         else {
640
641             bytes32 _bestByGenre = mostRatedByGenre[_genre][avgPos];
642
643             if(_contentSum > contentMap[_bestByGenre].getRateSum())
644                 mostRatedByGenre[_genre][avgPos] = _content;
645         }
646     }
647
648
649     /// @notice Check whether to update the most popular content of an author
650     /// @param _content The viewed content
651     function updateMostPopularByAuthor(bytes32 _content) private {
652
653         bytes32 _author = contentMap[_content].author();                    // author name
654         bytes32 _currentPopularByAuthor = mostPopularAuthorMap[_author];    // pop content
655         uint _popularViews = 0;
656
657         if(_currentPopularByAuthor == 0x0){
658             // First access to a content to that author
659             mostPopularAuthorMap[_author] = _content;
660             emit NewPopularByAuthor(_author, _content);
661         }
662         else {
663             _popularViews = contentMap[_currentPopularByAuthor].views();    // pop views
664
665             if(contentMap[_content].views() > _popularViews) {
666
667                 mostPopularAuthorMap[_author] = _content;
668                 emit NewPopularByAuthor(_author, _content);
669             }
670         }
671
672     }
673
674
675     /// @notice Check whether to update the most popular content of a genre
676     /// @param _content The viewed content
677     function updateMostPopularByGenre(bytes32 _content) private {
678
679         bytes32 _genre = contentMap[_content].getGenre();               // genre name
680         bytes32 _currentPopularByGenre = mostPopularGenreMap[_genre];   // pop content
681         uint _popularViews = 0;
682
683         if(_currentPopularByGenre == 0x0){
684             // First access to a content of that genre
685             mostPopularGenreMap[_genre] = _content;
686             emit NewPopularByGenre(_genre, _content);
687         }
688         else {
```

```
            _popularViews = contentMap[_currentPopularByGenre].views(); // pop views

            if(contentMap[_content].views() > _popularViews) {

                mostPopularGenreMap[_genre] = _content;
                emit NewPopularByGenre(_genre, _content);
            }
        }
    }
}
```

BaseContentManagement

```
pragma solidity ^0.4.18;

import "./Catalog.sol";

contract BaseContentManagement {

    /// Content information
    address public authorAddress;
    Catalog public catalog;
    bytes32 public author;
    bytes32 public title;
    uint public views = 0;
    uint public price = 0;

    mapping(address => bool) public accessRightMap;

    // Rating information
    uint public times;
    uint[] public ratingMap;

    /// @notice Check if the caller is the Catalog
    modifier isCatalog() {
        require(msg.sender == address(catalog), "The caller isn't the Catalog");
        _;
    }


    /// @notice Check if the user has access to the content
    /// @param _user the address of the user
    modifier hasAccess(address _user) {
        require(accessRightMap[_user] == true, "Access denied");
        _;
    }

    /// @notice Check if the user doens't already have the access to the content
    /// @param _user the address of the user
    modifier hasNoAccess(address _user) {
        require(accessRightMap[_user] == false, "Access already granted");
        _;
    }

    /// @notice Check if a given category is valid
    /// @param _category the category
    modifier validCategory(uint _category) {

        require(_category == uint(Catalog.Categories.Quality) ||
                _category == uint(Catalog.Categories.PriceFairness) ||
                _category == uint(Catalog.Categories.Rewatchable) ||
                _category == uint(Catalog.Categories.FamilyFriendly), "Invalid cateogry");

        _;
    }


    // @notice returns the type of the content
    // @returns the genre of the content
    function getGenre() public pure returns(bytes32);



    constructor(bytes32 _author,
                bytes32 _title,
```

```
63                uint _price,
64                Catalog _catalogAddress) public {
65
66        authorAddress = msg.sender;
67        catalog = _catalogAddress;
68        author = _author;
69        title = _title;
70        price = _price;
71
72        ratingMap = new uint[](catalog.numCategories());
73        times = 0;
74        views = 0;
75    }
76
77
78    /// @notice grant the access at the content to the user
79    /// @param _user the address of the user
80    function grantAccess(address _user) external isCatalog hasNoAccess(_user) {
81        accessRightMap[_user] = true;
82    }
83
84
85    /// @notice remove the access at the content to the user
86    function consumeContent() external hasAccess(msg.sender) {
87
88        accessRightMap[msg.sender] = false;
89
90        if(!catalog.isPremium(msg.sender)){
91
92            views++;
93            catalog.notifyConsumption(title, msg.sender, false);
94        }
95        else {
96            catalog.notifyConsumption(title, msg.sender, true);
97        }
98    }
99
100
101   /// @notice Rate this content
102   /// @param ratings The rating array
103   function rateContent(uint[] ratings) external isCatalog {
104
105       ratingMap[uint(Catalog.Categories.Quality)] += ratings[uint(Catalog.Categories.
              Quality)];
106       ratingMap[uint(Catalog.Categories.PriceFairness)] += ratings[uint(Catalog.
              Categories.PriceFairness)];
107       ratingMap[uint(Catalog.Categories.Rewatchable)] += ratings[uint(Catalog.Categories
              .Rewatchable)];
108       ratingMap[uint(Catalog.Categories.FamilyFriendly)] += ratings[uint(Catalog.
              Categories.FamilyFriendly)];
109
110       times++;
111
112       catalog.notifyRating(title);
113   }
114
115
116   /// @notice Get a rate of a category of this content
117   /// @param _category The requested category
118   /// @return The rate
119   function getRate(uint _category) external view validCategory(_category) returns(uint)
           {
120
121       if(times == 0)
```

```
            return 0;
        else
            return uint(ratingMap[_category] / times);
    }

    /// @notice Get the sum of all ratings
    /// @return The sum of all ratings
    function getRateSum() external view returns(uint) {

        return (ratingMap[uint(Catalog.Categories.Quality)] / times) +
                (ratingMap[uint(Catalog.Categories.PriceFairness)] / times)+
                (ratingMap[uint(Catalog.Categories.Rewatchable)]  / times) +
                (ratingMap[uint(Catalog.Categories.FamilyFriendly)] / times);
    }

    /// @notice Destruct method
    function destruct() external isCatalog {

        selfdestruct(this);
    }
}
```

BaseContentImplement

```
pragma solidity ^0.4.18;

import "./BaseContentManagement.sol";
//import "./Catalog.sol";

contract PhotoContentManagement is BaseContentManagement {

    constructor(bytes32 _author,
                bytes32 _title,
                uint _price,
                Catalog _catalogAddress) BaseContentManagement(_author,
                                        _title,
                                        _price,
                                        _catalogAddress) public {

    }

    function getGenre() public pure returns(bytes32) {
        // "photo"
        return 0x70686f746f0000000000000000000000000000000000000000000000000000;
    }
}



contract SongContentManagement is BaseContentManagement {

    constructor(bytes32 _author,
                bytes32 _title,
                uint _price,
                Catalog _catalogAddress) BaseContentManagement(_author,
                                        _title,
                                        _price,
                                        _catalogAddress) public {

    }

    function getGenre() public pure returns(bytes32) {
        // "song"
        return 0x736f6e6700000000000000000000000000000000000000000000000000000000;
    }
}



contract VideoContentManagement is BaseContentManagement {

    constructor(bytes32 _author,
                bytes32 _title,
                uint _price,
                Catalog _catalogAddress) BaseContentManagement(_author,
                                        _title,
                                        _price,
                                        _catalogAddress) public {

    }

    function getGenre() public pure returns(bytes32) {
        // "video"
        return 0x766964656f0000000000000000000000000000000000000000000000000000;
    }
}
```