COMP4801

Final Year Project for Computer Science

The University of Hong Kong

Topic:

FYP18003

Blockchain and Smart Contract Application

Final Report

YAM Mei Ki, Vanessa
Abstract

Blockchain has been one of the most trending areas among the field of technology in recent years. In particularly the financial industry has taken the advantage of such distributed ledger technology in creation of cryptocurrency and broadening of a new sector of the digital market. However, blockchain with its unique characteristic of immutability and transparency can provide much more potential usage in other industries. This project makes use of blockchain’s security and traceability to create a platform that can better protect copyright and fair use of media by providing secure storage and usage tracking.

This report is sectioned into project requirement, application design, implementations and result discussion. Chapter 1 demonstrates the background and overview of the project, Chapter 2 then gives an outline of the project organisation with project schedule and division of labour. In chapter 4, methodology is explained regarding the design of website, blockchain architecture, and data storage with detailed justifications and findings. Chapter 5 lays out the design of the system and application interface on which the implementation heavily relies on. The preliminary and final development is presented in Chapter 4 and 5 in details, followed by a performance evaluation in Chapter 7. Lastly, a conclusion is drawn from discussions on the challenges encountered throughout the project and the potential future development on Chapter 8 - 10.
Acknowledgment

I would like to express my greatest gratitude here to anyone who has helped me in completing this report, especially my teammate Alex, who provided a lot of insightful ideas that are essential to the project and in composing the content that establishes this report. I would also like to thank my supervisor Dr. Yuen for his guidance on the blockchain topic and consultation on the project. Furthermore, a thank you to my english lecturer Cezar, who has given me a lot of advice and coaching on professional writing.
# Table of Contents

1. **Introduction**  
   1.1 Background and needs  
   1.2 Project Objectives  
   1.3 Features  
   1.4 Scope  

2. **Project management**  
   2.1 Schedule  
   2.2 Deliverables  
   2.3 Division of work  

3. **Methodology and Findings**  
   3.1 Blockchain design  
   3.2 Website framework  
   3.3 Cloud service  

4. **Design and Analysis**  
   4.1 Interface design  
   4.2 System Architecture Design  
   4.3 System Sequence Diagrams  

5. **Preliminary development**  
   5.1 Demo implementation  
   5.1.1 Settings  
   5.1.2 Features  
   5.2 Evaluation  

6. **Technical specification**  
   6.1 Files organisation  
   6.2 Development environment
6.3 Software implementation 29
  6.3.1 Class-based models 29
  6.3.2 Functional models 32
  6.4 System migration 41

7. Test cases and Performance 43

8. Challenges encountered 50
  8.1 Uncertain design at initial phases 50
  8.2 Finding reliable source 50
  8.3 Trade-off between efficiency and contract gas usage 50

9. Future Scope 52

10. Conclusion 53

List of Figures 54
List of Tables 56
Reference 57
1. Introduction

1.1. Background and needs
Throughout the last decade, social media platforms have rapidly emerged and became a crucial part of our lives. People rely on Facebook in making connections, Instagram in sharing of photos and videos, Twitter in spreading of news... All of these platforms with their popularities share a common trait — centralization. Content that one shared, regardless of the effort one put in, ultimately is being ran and managed by a centralized authority. Thus it is often difficult for people that are used to such scheme of platform to question their ownership rights in what they have contributed, making concept of ownership on the internet unconceivable. Consequently, the nonexistence of concept in property right on the internet also leads to a lack of protection on original work or ideas. Even if one wishes to trace back to the author of a content, it is difficult with the ever-changing nature of the internet.

There is no guarantee in ownership nor originality to the content that one shared in the current media sharing platform, yet blockchain technology can change this. With the development of Bitcoin in 2008, increasing number of blockchains is being built and utilised in different aspects, such as finance and game industry. This project introduces a relatively less familiar concept to the public, a transparent and decentralised way of record storing.

1.2. Project Objectives
This project aims to use blockchain technology to implement a media sharing platform that enables a decentralised storage place for users to share their artworks. The scope of this project is to develop a web-based platform with its database supported by a hybrid approach of centralised cloud storage and existing blockchain architecture. The goal is to facilitate blockchain technology in the creative industry. By taking advantage of blockchain’s immutability, consistency and transparency, this project can drive the art market into a new era of online ownership. Providing reliable trading with cryptocurrency and a sharing platform that is protected by
trustworthiness and security granted from the blockchain technology for artists worldwide.

1.3. Features

This project implements a web application with use of blockchain technology to achieve seven main features:

FE1: For users to create accounts on the website.
FE2: For users to manage their Profiles that are being displayed to others.
FE3: For users to upload artwork on to the website.
FE4: For users to be able to track the derivative and origin of an artwork.
FE6: For users to have the option in pricing their artwork.
FE7: For users to exchange ideas by giving comments.
FE8: For users to get tokens for giving contribution, such as artwork publishing and commenting in the community.

1.4. Scope

The scope is divided in two parts. One as the scope of this project, and the other one as a scope that are being hoped to establish in the future. Table 1 illustrates the two scopes in regards to each feature. FE1, FE2, FE4 and FE3 are treated as key features, therefore will be fully implemented or at least partially implemented in this project. FE6, FE7, and FE8 will be treated as extended features, therefore will only be implemented in the future scope.

<table>
<thead>
<tr>
<th>Features</th>
<th>Scope of this FYP</th>
<th>Scope of subsequent releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE1: Account creation</td>
<td>Fully implemented</td>
<td>-</td>
</tr>
<tr>
<td>FE2: Profile Management</td>
<td>Fully implemented</td>
<td>-</td>
</tr>
<tr>
<td>FE3: Artwork uploading</td>
<td>Only limited to imagery form of work, such as file format of JPEG, PNG, BMP...etc</td>
<td>To be extended to other formats of work, such as text and video</td>
</tr>
<tr>
<td>FE4: Artwork tracking</td>
<td>Fully implemented</td>
<td>-</td>
</tr>
<tr>
<td>Feature ID</td>
<td>Feature Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>FE5: Artwork selling</td>
<td>-</td>
<td>In form of cryptocurrency in exchange for download availability</td>
</tr>
<tr>
<td>FE6: Commenting</td>
<td>-</td>
<td>Available under user profile, with option of private or public</td>
</tr>
<tr>
<td>FE7: Token generating</td>
<td>-</td>
<td>Token can be exchanged into cryptocurrency</td>
</tr>
</tbody>
</table>

Table 1. Scope of features
2. Project management

2.1. Schedule

The timeline of the project is demonstrated in Table 2, and milestones are categorised in stages. The grey boxes represent the initial planned schedule, and tick symbols represent the actual progress. While Requirement and Design and Analysis stages are roughly on schedule, Background research took more time than expected as it has to be parallel conducted with the Design stage. Implementation is also slightly behind schedule as the complexity of the system was higher than anticipated.

<table>
<thead>
<tr>
<th>#</th>
<th>Milestones</th>
<th>Sep 18</th>
<th>Oct 18</th>
<th>Nov 18</th>
<th>Dec 18</th>
<th>Jan 19</th>
<th>Feb 19</th>
<th>Mar 19</th>
<th>Apr 19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Requirement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>Background research</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>Requirement Specification</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design and Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td>Use Case design</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4</td>
<td>System architecture design</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M5</td>
<td>Cloud service analysis</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M6</td>
<td>User Interface design</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M7</td>
<td>Smart Contract building</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M8</td>
<td>Back-end implementation</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M9</td>
<td>Front-end implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>M10</td>
<td>Cloud services intergration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>M11</td>
<td>System intergration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M12</td>
<td>Unit testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>M13</td>
<td>Integration &amp; System testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Table 2. Scheduled milestones and progress

2.2. Deliverables

Different products, including documentation and implementation, are delivered as according to the planned stages. These deliverables represent the completed milestones at each stage. Table 3 layouts the deliverables and its delivery date.

<table>
<thead>
<tr>
<th>Stage(s)</th>
<th>Deliverables</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>Project Plan</td>
<td>30 September, 2018</td>
</tr>
<tr>
<td></td>
<td>Project Website</td>
<td></td>
</tr>
<tr>
<td>Design and Analysis</td>
<td>Intermediate Report</td>
<td>20 January, 2018</td>
</tr>
<tr>
<td></td>
<td>Preliminary implementation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>First Presentation</td>
<td>22 January, 2019</td>
</tr>
<tr>
<td>Implementation &amp; Testing</td>
<td>Final report</td>
<td>14 April, 2019</td>
</tr>
<tr>
<td></td>
<td>Finalised implementation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Final Presentation</td>
<td>17 April, 2019</td>
</tr>
<tr>
<td></td>
<td>Project Exhibition</td>
<td>29 April, 2019</td>
</tr>
</tbody>
</table>

Table 3. Deliverables according to each stage

2.3. Division of work

The size of team is in two people for this project. Hence, the division of work is about half for each person. As the project is a web application, both front-end and back-end developments are involved.

<table>
<thead>
<tr>
<th>#</th>
<th>Tasks</th>
<th>Alex</th>
<th>Vanessa</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Background research</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>T2</td>
<td>Requirement Specification</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>T3</td>
<td>Use Case design</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>T4</td>
<td>Blockchain architecture design</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>T5</td>
<td>Cloud service analysis</td>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>T6</td>
<td>User Interface design</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>T7</td>
<td>Blockchain construction</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>Back-end implementation</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>T9</td>
<td>Front-end implementation</td>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>T10</td>
<td>Cloud services integration</td>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>T11</td>
<td>Risk plan</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>T12</td>
<td>Unit testing</td>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>T13</td>
<td>Integration &amp; System testing</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

E - Equally Responsible  L - Leader  A - Assistant

Table 4. Division of work
3. Methodology and Findings

The following section gives a detailed justification on the decision made regarding each essential aspect that constitutes the project, including choice of Ethereum blockchain, the framework of the Single Page Application and cloud service.

3.1. Blockchain design

In order to process data using blockchain, it is necessary to select an approach that is most suitable to our need among various types of blockchain that are available in the market. Ethereum is used to facilitate the back-end development of this project, for its already well established cryptocurrency — Ether, and its better supported flexibility.

Comparison of Ethereum with other blockchains

Comparing Ethereum to other popular blockchains such as Bitcoin, it offers a better degree of customization to users via Smart Contracts. While Bitcoin was built with predefined operations that support transaction type of activities, Ethereum was built as “a meta-protocol on top of Bitcoin”[1], with its Smart Contract allowing storage of any executable codes. This type of blockchain can secure not just values or records but any type of functionalities, such as storing of image and interaction with users info that are required in this project. While there are also other built blockchains, such as Stratis and EOS, that can support similar flexibility and offer a higher transaction rate, Ethereum has been the most popular platform to be used to build decentralized applications, with the availability of Ethereum Virtual Machine that offers powerful computing power and adaptability to different programming languages that better supports the construction of this project. Furthermore, such popularity also makes its cryptocurrency — Ether widely adopted, hence beneficial to users that wish to use this platform as a market to generate financial value from their artworks.

Comparison of Ethereum with self-build blockchain

While option of building an original blockchain can offer tailor-made functionality for the project without any restriction of existing cryptocurrency as well as a faster
transaction rate, these features are considered less crucial as compared to the advantages from using Ethereum. Even though building our own blockchain would eliminate the restriction of significant low transaction rate from Ethereum, it would require extra effort in implementing the wallet and mining technology that are already extensively developed in Ethereum. Taking into account that this application need not be real-time to be able to operate, the benefits of a well-established platform from Ethereum outweighs the advantage of performance and flexibility from a customized blockchain.

3.2. Website framework

For ease of navigation and to provide a smooth experience to the users, the project will implement a Single Page Application as it web-based application. As compared to Multiple Page Application, Single Page Application can eliminate the hassle in page loading by refreshing only section of the content that the users require, resulting a responsive interaction between users and website.

Areas of concern

Numerous frameworks are available for developing the front-end, where user interacts; the back-end, where the website operates and functions; and the database, where data utilised by the website is stored and retrieved from. Since implementation of blockchain will not restrict the choice of website development, the project will implement with frameworks that gives the most outstanding user experience and that the team is most comfortable working with.

Front-end development

For front-end implementation, AngularJS and ReactJS were both taken into consideration for their popularities and well supported resources in developing Single Page Application. More findings reveal that ReactJS is a better option, for its greater degree of interactiveness that enhances user experience and its easier integration to different platform than AngularJS that make it a more suitable framework for achieving the project’s ultimate goal of implementation on multi-platforms.
Data storage management

For database construction, as opposed to fully operate using blockchain, the project will take a hybrid approach by using both blockchain, database and cloud service. Such approach is to minimize the cost and processing time in storing potentially large image. Storing image is different to typical data storage on blockchain such as plain text and string, as it requires a pixel by pixel data record and thus will be extremely expensive in processing time if done entirely using blockchain. The hybrid approach will make use of the cloud service for storing the actual image and the blockchain for storing a hashed linked which can be used to verify the authenticity of the image on cloud storage server. The user details and metadata will be stored in the database. The approach can share the storage to improve the efficiency on the data processing and ensure a high level of security and traceability from blockchain while solving the issue of a costly storage occupation.

Back-end development

For back-end development, a decision is yet to be finalised for the vast amount of available frameworks that can fit into other determined sectors of the web development including ReactJS and hybrid storage approach. A JavaScript framework such as Node.js will likely be utilised since the front-end development by ReactJS also makes use of JavaScript. The unification of programming languages across different sectors of the development can provide a steady learning curve to the team.

3.3. Cloud service

Cloud services are investigated with a few criteria that are crucial to this projects, one is its capability for storing images that are uploaded to the website, and the other is the ease of usage through APIs and other services. Table 3 illustrate the details of considered cloud services and gives an comparison in areas of usage, storage service, availability of APIs and image recognition service.

|------------------|------------|---------------------|---------------|
Table 5. Cloud service analysis

<table>
<thead>
<tr>
<th>Storage Service</th>
<th>CloudSQL</th>
<th>Cloud Storage</th>
<th>File Storage</th>
<th>Image and Video managements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>Provide web framework and content management</td>
<td>Storage for multimedia and blob objects</td>
<td>Enable usage of Windows and Representational State Transfer Application programming interface which is useful for web development</td>
<td>Image uploading with embedded user interface, return Dynamic Responsive Images and optimisation</td>
</tr>
<tr>
<td>APIs</td>
<td>Provided</td>
<td>Self-develop</td>
<td>Provided</td>
<td></td>
</tr>
<tr>
<td>Image Recognition</td>
<td>Cloud Vision</td>
<td>Computer Vision</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

In this project, Cloudinary is selected since it provides a trial version of cloud storage and it is sufficient for storing image files. It also provides smooth and convenient usage with its establish APIs, and embedded user interface that can be employed on the web application. The service to return images dynamically is also very desirable to our front-end with a main page of responsive images grid. Since image filtering function is not the most main objective in the project, a simple cloud storage platform during testing phase is more suitable for simplifying the process without having any extra costs.
4. Design and Analysis

4.1. Interface design

The following are the interface design for the application and it is used to visualize the ideas of the content sharing platform which becomes the reference for building of the website. The website development thus will be heavily relied on the blueprint of the following figure 1 to 6.

![Figure 1. Front page of the website](image)

The user interface make uses of a minimalist design, so that it can provide a clean layout for users to focus on the artworks rather than the website itself. Figure 1 illustrates the interface to the front page of the website, where popular artworks are displayed with an infinite scrolling that allow users to view more artworks by scrolling to the bottom of the page.
Figure 2. Artwork portfolio

Figure 3. Viewing derivative of an artwork
Figure 2 and Figure 3 display the web pages design for viewing an artwork and its derivative works, which is a key feature of the web application. The artwork portfolio contains artwork with its name, description and creator, as well as its popularity presented as the number of “like” given by other users. The page also contains a page-bar in form of dots at the side for navigation between sources and derivatives regarding to the selected artwork. Users can view the entire chain to know the evolution from the root of the artwork to its derivative works. Figure 2 gives an example of portfolio for a selected artwork, and Figure 3 shows an illustration for viewing a derivative portfolio.

![Image of Portfolio](image_url)

**Figure 4. Artist portfolio**
Figure 5. Viewing self portfolio

Figure 6. Profile Editing
Figure 4 and Figure 5 illustrate the webpage layout for viewing artist portfolio. The portfolio contains artworks and biography of artist, as well as different social means for connecting with the artist. The two figures are different in terms of viewer of the page. Figure 4 is a portfolio page in perspective of other users whereas Figure 5 is in view of the artist himself with options to upload artwork or edit profile. Once the artist presses the edit profile link, he will be directed to the edit page as shown in Figure 6, in which he can change or his personal details except his username.

4.2. System Architecture Design

In order to model blockchain architecture and simulate its interactions with the web application and cloud service, a conceptual class model is designed as shown in Figure 7. The diagram illustrates the responsibility of each class and the relationship between different classes across systems. MongoDB holds the classes exist in the web application database, including user details and the access key to artwork information in form of hashing of the user and image ID; Ethereum holds the uRecord class in blockchain that represents the uploaded artwork record and stores artwork metadata including the access link; Cloud contains the image uploaded on cloud service; and MetaMask refers to an external wallet that is connected to the users.

The responsibility of each class is designed in a way so that it aids the implementation of the back-end system of the application as well as efficiently minimises the data processing cost on the blockchain. For example, to avoid data redundancy while verifying the data integrity of each artwork, the Artwork class in MongoDB does not store any artwork metadata but only the key to uRecord class. In such way the integrity of each artwork is automatically protected by Ethereum and does not require cross checking to verify data on either side.
4.3. System Sequence Diagrams

System sequence diagrams simulates the interaction between different system for each use case that involved the users. Figure 8 illustrates the system sequence diagram for a user to upload an image to the web application. It initiates with user interacting with the front-end, which then sends an HTTP request to the back-end and being handled by calling corresponding service providers from cloud service, web application database on MongoDB, and blockchain on Ethereum.
Figure 8. System sequence diagram (SSD) for uploading an image

Figure 9 demonstrates the System sequence diagrams for searching artworks by specific author and Figure 10 illustrates the tracking of a given artwork.

Both use cases initiate with a similar line of interactions that forwards the HTTP request to the back-end and queries the MongoDB for user id (uid) of user or artwork id (aid) of artwork accordingly. The uid and aid together will be used as a key to retrieve artworks from Ethereum. For case of searching artwork by specific user, artworks are queried in a recursive calling to Ethereum; while for case of artwork tracking, a query of related work is first called to Ethereum, and followed by a recursive call to retrieve related artworks while verify the integrity of each related artwork.
Figure 9. System sequence diagram (SSD) for view artworks by specific artist

Figure 10. System sequence diagram (SSD) for tracking an artwork
5. Preliminary development

5.1. Demo implementation
The demo implementation aims to develop a simple web application using smart contracts written in Solidity that serves as a proof of concept in demonstrating the feasibility of the current design and methodology. It shall also act as a bridge between the Design and Analysis phase and the Implementation phase for which the building of the smart contract can be based on the system diagram from Figure 8 to 10.

5.1.1. Settings
Different frameworks and libraries were utilised in the building of the demo application. Smart contracts are written in Solidity in accordance to the decided methodology. The application also makes use of tools provided by the Truffle suite in managing the smart contract, including Truffle framework that compiles and builds contract; and Ganache, a local private Ethereum blockchain for which the contracts can be deployed on. Node JS is also used for the server environment as addressed in chapter 3.2, while web3.js is used to manage the communication between web application and Ethereum.

Since the team has no experience on building decentralised application, online tutorials were used as a reference to initiate the demo implementation, in particularly the tutorial from the Truffle Suite [6] on which the project has its source codes based on.

5.1.2. Features
Since the demo application serves as a demonstration of feasibility, the focus is placed on the function of smart contract and its communication to the web application. Thus the front-end is written in plain HTML and JavaScript, and serves as a primitive render of data. Cloud service is also not concerned in this stage, such that for this demo application the image addresses are addresses pointing to local storage.
Smart Contract

At the current stage, the team found that a single contract, Gallery, is sufficient in storing all artwork records and handling all CRUD operations. As seen in Figure 10, Artwork records are represented in form of a struct, consisting of its metadata and storing sources and derivatives in form of an array of their image id. The artwork is then mapped to key of user id and image id by a mapping, unique data structure in Solidity which is indeed a hash table. Figure 11 also shows that functions for basic database operation have been written and are available to the web application, for example creating an Artwork instance, retrieving Artwork information from key of user id and image id, and adding a source.

```solidity
contract Gallery {
    // counter
    uint public artworksCount;

    struct Artwork {
        bytes32 hashValue;
        string name;
        string access;
        // store other works in form of image_id that references to Mongo
        uint[] source;
        uint[] derivative;
    }

    // Read/write Artwork
    // a hash table mapping hash(user_id, image_id) to Artwork
    mapping(bytes32 => Artwork) public gallery;
}
```

Figure 11. Pseudocode of smart contract variables from Gallery.sol
By calling functions from the built contracts, the website is able to display existing artworks, retrieve artwork information as well as display sources and derivatives. Figure 12 displays the user interface to the front page, on which list of artworks are displayed and is responsible by the section of code as illustrated in Figure 13.
Figure 14. Pseudocode of displaying contract artworks from app.js

Figure 15 illustrate the artwork portfolio page after user clicking on an image, in which artwork information is displayed together with its sources and derivatives, and Figure 16 demonstrates section of code that is in charged of the such function.

Figure 15. Artwork portfolio of the web application – preliminary development
5.2. Evaluation

The preliminary implementation has successfully demonstrated the feasibility of the current system design. As the demo mainly focused on just contract functionality, future implementation should consider other aspect to the application such as front-end rendering and a solid back-end system. Integration between different services should be set as a priority as it can involve complex solution that the team has not yet forsee. Other aspects such as efficiency of contract methods and compatibility of contracts variable to Mongo database should also be considered.

Figure 16. Pseudocode for displaying an artwork detail from app.js
6. Technical specification

6.1. Files organisation

Table 6 illustrates the key structure of the project, explicitly including important directories. Codes and files organised as according to their responsibility. First-level directory stores file related to contracts, while second-level stores files supporting the web application.

```plaintext
decentral_app
    ├── truffle-config.js
    └── contracts
        ├── Gallery.sol  # Contract written in this project
        ├── Migrations.sol
        └── migrations
            ├── 1_initial_migration.js
            └── 2_deploy_contracts.js  # to deploy contract

    └── test
        └── ...

client
    ├── package.json
    └── ...

    └── model
        ├── Artwork.js  # Model schema of artworks
        └── User.js  # Model schema of users

    └── routes
        ├── Artwork.route.js  # REST API controllers for Artwork
        └── User.route.js  # REST API controllers for User

    └── src
        ├── App.js  # Route to all pages
        └── ...

            └── components  # Reusable components
                └── contracts  # Migrated contract

    └── css
        └── ...

    └── pages
        ├── ArtworkDetails.js  # Artwork portfolio
        ├── Author.js  # Load all Users
        ├── AuthorProfile.js  # Author portfolio
        └── EditInfo.js  # Edit user info
```
6.2. Development environment

The development framework for the final implementation is consistent to the designed methodology as stated in Chapter 3, and integrated with the established environment from the preliminary implementation stated in Chapter 5.1.1. Table 7 gives a detailed list of the environment settings.

<table>
<thead>
<tr>
<th>Smart contract language:</th>
<th>Solidity 0.5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract compilation:</td>
<td>Truffle 5.0.0</td>
</tr>
<tr>
<td>Blockchain enviorment:</td>
<td>Genache 1.2.2 &amp; Ropsten testnet</td>
</tr>
<tr>
<td>Digital wallet:</td>
<td>MetaMask</td>
</tr>
<tr>
<td>Sever environment:</td>
<td>Node JS 8.11.1 + Express 5.6.0</td>
</tr>
<tr>
<td>Front-end rendering:</td>
<td>React</td>
</tr>
</tbody>
</table>

Table 7. Environment of the final implementation

Various node packages are included to aid the implementation, the heavily relied ones include:

**Mongoose**: An Object Data Modeling library for MongoDB and Node.js to communicate exchange data [7]

**Cloudinary**: For image uploading through Node.js to cloud service Cloudinary [5]

**web3**: Allow interact with a Ethereum node, through HTTP, WebSocket or IPC connection [8]

6.3. Software implementation

6.3.1. Class-based models

Class based models include all domain objects that are captured in the conceptual class diagram. As evaluated upon the preliminary implementation, there are a few adjustment made on the conceptual model as seen in Figure 17.
The key update is in User model. Since the method to facilitate authentication of user is re-evaluated, user_id and password is removed as compare to previous conceptual class in Figure. 7, and replaced by account. Account is essential the wallet address from MetaMask, the digital wallet. Such that user authentication can be done by just validating the wallet address, which is also more secure than self-implementing a user system on Mongo with MetaMask secure identity vault.

Figure 17. Updated Conceptual Class Diagram

**Smart Contract**

Smart contract model represents uRecord, the Artwork stored on Ethereum. The artwork struct as seen in Figure 18. has almost identical declaration to the one from preliminary implementation in Figure. 11, except sources and derivatives are declared as array of 32-bytes instead of array of unsigned integer. That is because upon migrating with Mongo Database, sources and derivatives would have to store mongoose.ObjectId in representation of the image_id, which is essentially a 12-bytes value. Yet 32-bytes declaration is used as it is indeed cheaper in gas usage than 12-bytes value. Because of the fact that Ethereum Virtual Machine operate on a 32-bytes words, 12-bytes value will subsequently require more conversion operations hence increase in gas cost [9].
Other than change of variable type, artworkCount is also removed. Since the data of Artworks should be consistent across the two data storages, Mongo Database and Contract, counting of artworks can be recorded on Mongo and do not require redundancy over at the contract. As the redundancy will in reality be traded off by the gas usage on extra variable update during each artwork upload.

```solidity
struct Artwork {
    bytes32 hashValue;
    string name;
    string caption;
    string access;
    bytes32[] sources;
    bytes32[] derivatives;
}

// a hash table mapping hash(author_id, image_id) to Artwork
mapping(bytes32 => Artwork) public gallery;
```

Figure 18. Updated pseudocode of smart contract variables from Gallery.sol

**Mongo collections**

Mongo collections include data structure that are stored over Mongo database, consisting of the entire User structure and partial Artwork structure. User and Artworks are defined as seen in Figure 19. The one-to-many relationship is represented by storing the primary key of User onto Artwork. These data structure will be exported as a schema, that ultimately defines a common structure to the users and artworks collections on Mongodbc Database respectively [7].
Figure 19. Pseudocode for User and Artwork model on Mongo database

6.3.2. Functional models

The section includes operational and functional structures that facilitate system sequences as discussed in Chapter 4.3. The functions can be partitioned in two sections, one includes the function provided by the smart contract; and the other are the controllers constituting to REST API that enable interaction between React front-end and back-end system.

Functions from Small contract

As the contract class has not evolved a lot from the class built in preliminary implementation, there are also minimal changes applied to the functions. Figure 20 shows the contract functions that will induce a transaction when called upon, including addArtwork and addSource. Compared to Figure 12, the function has modified to correspond to the changed variable type of sources and derivatives. addSource is also modified to achieve add source for a given artwork while also adding the given artwork as a derivative to its source.
Figure 20. Updated pseudocode for smart contract functions from Gallery.sol

Figure 21 displays the pseudocode of contract functions that will not induce transaction as it only involves data reading operations. Such functions included `retrieveArtwork()`, `retrieveArtworkInfo()` and `retrieveArtworkAccess()`. The three functions differ in terms of the number of returning arguments, such that minimum data is passed only as accordance to the situation and enable a more efficient data communication.

As compared to Figure 12, all the updated contracts functions also have names removed from its returning argument. Such approach is employed so that methods to access value is distinguishable on objects returned from Mongo and contract. For example accessing returned value from contract will be based on index, response[3], rather than name, response.access. That is to allow further clarification on controller side by avoiding multiple variables using “access” as its name.

Figure 21. Updated pseudocode for smart contract functions from Gallery.sol

```solidity
function addArtwork (bytes32 author_id, bytes32 image_id, string memory _name, string memory caption, string memory access) public {  
    bytes32 hashV = keccak256(abi.encodePacked(author_id, image_id));  
gallery[hashV] = Artwork(hashV, _name, caption, access, 1, 0);  
}

function addSource (bytes32 author_id, bytes32 image_id, bytes32 source_aid, bytes32 source_id) public {  
    bytes32 hashV = keccak256(abi.encodePacked(author_id, image_id));  
gallery[hashV].sources.push(source_id);  
    bytes32 source_hashV = keccak256(abi.encodePacked(source_aid, source_id));  
gallery[source_hashV].derivatives.push(image_id);  
}
```

```solidity
function retrieveArtwork (bytes32 author_id, bytes32 image_id) public returns (bytes32, bytes32, string memory, string memory) {  
    bytes32 hashV = keccak256(abi.encodePacked(author_id, image_id));  
    return (author_id, image_id, gallery[hashV].name, gallery[hashV].access);  
}

function retrieveArtworkInfo (bytes32 author_id, bytes32 image_id) public returns (bytes32, bytes32, string memory, string memory, string memory, bytes32[] memory, bytes32[] memory) {  
    bytes32 hashV = keccak256(abi.encodePacked(author_id, image_id));  
    return (author_id, image_id, gallery[hashV].name, gallery[hashV].caption,  
gallery[hashV].access, gallery[hashV].sources, gallery[hashV].derivatives);  
}

function retrieveArtworkAccess (bytes32 author_id, bytes32 image_id) public returns (string memory) {  
    bytes32 hashV = keccak256(abi.encodePacked(author_id, image_id));  
    return (gallery[hashV].access);  
}
```
REST APIs as Artwork controllers

Artwork controllers are relatively complex as they involve retrieving data from both Mongo database and contract instance. Before any APIs can be used, the function `componentDidMount()` is first called to establish connection to deployed blockchain and get related contract and account info. As shown in Figure 22, a web3 instance is first created by connecting it with the MetaMask wallet connect to local blockchain Ganache. Various methods are then called on the instance to get metadata of accounts, blockchain network, and contract instance whom the contract methods will be called on.

```javascript
// Get web3Provider
const web3 = new Web3(Web3.givenProvider || "ws://localhost:7545");

// Get contract instance
let accounts, networkId, deployedNetwork, instance, balance;
const componentDidMount = async function() {
    try {
        // Use web3 to get the user's accounts.
        accounts = await web3.eth.getAccounts();

        // Let accounts[0] be default account and get balance
        await web3.eth.getBalance(accounts[0], (err, b) => {
            balance = b/(10**10);
        });

        // Get the contract instance.
        networkId = await web3.eth.net.getId();
        deployedNetwork = GalleryContract.networks[networkId];
        instance = new web3.eth.Contract(
            GalleryContract.abi,
            deployedNetwork.address,
        );
    } catch (error) {
        // Catch any errors for any of the above operations.
        console.error(error);
    }

    componentDidMount();
}
```

Figure 22. Pseudocode for getting deployed contract from Artwork.route.js
To retrieve all artworks to display on the front page, the controller first queries the Mongo database for image_id and author_id of all artworks, the two is then used as a key to call retrieveArtwork( ) of contract instance in loop and push the found Artworks as an array back to the client.

```javascript
// To Get List Of Artworks from Ethereum
artworkRoute.route('/').get(function(req, res) {
    // Find all artworks in Mongo
    ArtworkModel.find(function(err, Artworks) {
        let json = {
            Artworks: []
        };

        // Retrieve artworks from Ethereum
        const callInstance = async function() {
            try {
                for (var i=0; i<Artwork.length; i++) {
                    // parse string into bytes32 for contract
                    let author_id = web3.utils.asciiToHex(ARTWORK[Artwork[i].author_id.replace(/\['"]+/g,''), 32);
                    let image_id = web3.utils.asciiToHex(ARTWORK[Artwork[i].image_id.replace(/\['"]+/g,''), 32]);

                    const response = await instance.methods.retrieveArtwork(author_id, image_id).call();
                    // if this is a valid artwork uploaded to Ethereum (with a name)
                    if (response[2] !== '') {
                        let result = {'author_id': web3.utils.hexToUtf8(response[0]), 'image_id': web3.utils.hexToUtf8(response[1]),
                            'name': response[2], 'author': Artwork[i].author, 'access': response[3]};
                        json.Artworks.push(result);
                    }
                }
                res.json(json);
                callInstance();
            }
        };
        callInstance();
    });
});
```

Figure 23. Pseudocode for retrieve artworks from Artwork.route.js

All artwork controllers wrap the calling of instance method in an asynchronous function, such that it enables the contract method calling be expressed as `await`, to block-wait for contract returns result before executing the rest of the code.

Artwork uploading reflects the system sequence illustrated in Figure 8 and takes similar procedure as retrieving artwork. As shown in Figure 24, it first queries the Mongo database to get author name, which is then used to create a new Artwork instance to stored to Mongo. The same Artwork object with details of caption and access link is then passed to contract to store in the gallery mapping.
Figure 24. Pseudocode for uploading artworks from Artwork.route.js

```javascript
// To upload artwork
artworkRoute.route('/uploadArtwork').post(function (req, res) {
  let author_id = req.body.author_id;
  let name = req.body.name;
  let caption = req.body.caption;
  let access = req.body.access;

  // Find author name
  UserModel.findById(author_id, function (err, author) {
    let artwork = new ArtworkModel({author_id: author_id, name: name, author: author.name});

    Artwork.save(function (err, artwork) {
      try {
        // Parse id from string to bytes32 for contract
        let author_id32 = web3.utils.asciiToHex(JSON.stringify(artwork.author_id).replace(/[^a-z]*/g, '').slice());
        let image_id32 = web3.utils.asciiToHex(JSON.stringify(artwork.image_id).replace(/[^a-z]*/g, '').slice());

        let instance = await ArtworkModel.addArtwork(author_id, image_id, name, caption, access);
        res.send({ from: accounts[0], gas: 3000000 });
      } catch (error) {
        callInstance();
      }
    });
  });
});
```

Figure 25 illustrates the APIs for retrieve artworks corresponding to a given User, which also reflects to designed system sequence in Figure 9. Conversion between ASCII and Hex of author_id and image_id is prevaded across APIs for Artwork. That is because the contract instance reads in autho_id and image_id as bytes32 value for the benefit mentioned in Chapter 6.3.2, and require conversion from Hex to ASCII when the id values are retrieved from contract.
Figure 25. Pseudocode for finding artworks of given author from Artwork.route.js

Getting artwork details means not only retrieving Artworks info but also its source and derivatives. The pseudocode for this operation as shown in Figure 26 is slightly clumsy. The main reason is that for each source and derivative, author_id is required from Mongo database, and is used as key to retrieve all related works from contract. Hence multiple await methods are needed to be executed in a loop, as well as frequent conversion between ASCII and Hex on id values for retrieval and reading of artworks.
Figure 26. Pseudocode for finding artworks details, including sources and derivatives from Artwork.route.js

Adding source for an artwork is comparably more straight-forward as illustrated in Figure 27. Upon evaluation of the preliminary implementation, an improvement is made on the previous system sequence designed in Figure 10. The operations is simplified by combining add source and add derivative into a single contract method, thus only a single call to contract instance is required.
Figure 27. Pseudocode for add source for an artwork from Artwork.route.js

Note that all snapshots of artwork controller pseudocode has simplified to focus on the success operation flows. Error handling and console logging are dismissed in snapshots but are present in the source code. In particular the logic for error handling is explained in Figure 29.

REST APIs as User controllers

The controllers for User objects are less complicated than Artwork ones because User data is only stored in Mongo database. The CRUD operations for user are done through four different APIs as shown in Figure 28. /registrate and /update are routers for HTTP POST request, and return message of success back to user. /get/:id and /delete/:id are for HTTP GET request, where /get/:id returns the required user object and /delete/:id returns success messages.
Figure 28. Pseudocode for CRUD operation on user from User.route.js

Error handling

The pseudocode of User and Artwork controllers as illustrated are simplified to show only successful operation flows. Error handling on both controller are done by the same logic as displayed in Figure 29, where error is caught and logged onto console. The error message will be sent to the user, with a HTTP response status codes of 500 to indicate internal server error. Note that on success operations a default HTTP response status codes of 200 ‘OK’ is sent along side with the data require or message of success [10].
6.3.3. System migration

While the local private blockchain network can simulate the blockchain environment and provide convenient access, it is the test network that can truly simulate contract behaviours on the main network, ‘Mainnet’. The project make use of Ropsten as test network to migrate, since it is one of the three supported by MetaMask, which is the digital wallet this application heavily relies on. It is also the only Proof-of-Work network out of the three, hence the best simulation to the Ethereum environment on the Mainnet. [11]

Migrating smart contracts to testnet involves different means:

**Metamask**

New account is created on Metamask to deploy contracts on to the Ropsten test network. Test ether is also needed to run transactions, and was requested on various faucets such as Ropsten Ethereum Faucet[12] and MetaMask Ether Faucet [13] that act at API endpoints to provide free Ether on test network.

**Infura for accessing Ropsten test network**

Infura is an API endpoint that acts as an Ethereum node supporting deployed contracts to interact with Ropsten network. Registration with Infura is required to obtain API endpoint that is specific to each project on site.

**HDWalletProvider**

HDWalletProvider is a npm package provided by Truffle Suite that support connection to Ropsten test network by utilising the API endpoint from Infura and mnemonic from MetaMask account.
The migrations was successfully done through terminal with the following transaction receipt:

```
Starting migrations...

> Network name: 'ropsten'
> Network id: 3
> Block gas limit: 0

1_initial_migration.js

Replacing 'Migrations'

> transaction hash: 0x336f2ba380ef9842955b4a7e81badd88600ac5337f0005586dddb1504d710887
> blocks: 1
> contract address: 0xc74222f909f338753507894bc983d327c383a
> account: 0x48e221664502e2705b84a686d17a5891fa691b
> balance: 3.99763
> gas used: 264508
> gas price: 28 gwei
> value sent: 0 ETH
> total cost: 0.08559816 ETH

> Saving migration to chain.
> Saving artifacts

---

> Total cost: 0.08559816 ETH

2_deploy_contracts.js

Deploying 'Gallery'

> transaction hash: 0xc96a719bce669b7d0023471bc5f4c3981b3e6a8723f3a115c8c827e0e5b06675
> blocks: 1
> contract address: 0x2e3658307d64f7aa630d349f7499a80d7397f77e
> account: 0x484d222f909f338753507894bc983d327c383a
> balance: 3.5585362
> gas used: 1401935
> gas price: 28 gwei
> value sent: 0 ETH
> total cost: 0.0828327 ETH

> Saving migration to chain.
> Saving artifacts

---

> Total cost: 0.0828327 ETH

Summary

---

> Total deployments: 2
> Final cost: 0.83480088 ETH
```

Figure 30. A snapshot of the contract deployment receipt
7. Test cases and Performance

Unit testing were conducted in particular to test the functionality of the built REST APIs. As the APIs are fundamentally interacting with the back-end system on Mongo Database and on contract, the following test cases can also provide an overview on the feasibility of each individual method.

Test cases on User controllers

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that end-user is able to registrate via creating an account</td>
</tr>
<tr>
<td>Related features</td>
<td>FE2: Account creation</td>
</tr>
<tr>
<td>Pre-requisites</td>
<td>End-user has logged in MetaMask wallet with a valid account</td>
</tr>
<tr>
<td>Procedure</td>
<td>Sending HTTP POST request to /users/registrate</td>
</tr>
<tr>
<td>Test data</td>
<td>Request body:</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;name&quot;: &quot;Tessa&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;bio&quot;: &quot;hi I am Tessa&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;avatar&quot;: &quot;<a href="https://res.cloudinary.com/fyp18003/image/upload/v155451663/artworks/kitten2.jpg">https://res.cloudinary.com/fyp18003/image/upload/v155451663/artworks/kitten2.jpg</a>&quot;</td>
</tr>
<tr>
<td>Expected result</td>
<td>HTML Status: 200</td>
</tr>
<tr>
<td></td>
<td>Respond body: { &quot;message&quot;: &quot;Success&quot; }</td>
</tr>
<tr>
<td>Actual result</td>
<td>As expected</td>
</tr>
<tr>
<td>Follow up</td>
<td>TC02 to check if user has really been created</td>
</tr>
</tbody>
</table>

Table 8. TC01 test case for user registration

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that specific user data can be returned</td>
</tr>
<tr>
<td>Related features</td>
<td>FE2: Profile Management</td>
</tr>
</tbody>
</table>
| Pre-requisites | - TC01 is executed and passed  
|               | - author_id is known for retrieve specific user |
| Procedure     | Sending HTTP GET request to  
|               | /users/get/ca5ec14a5ef65243d180a71 |
| Test data     | author_id: ca5ec14a5ef65243d180a71 |
| Expected result | HTML Status: 200  
|                | Respond body:  
|                | {  
|                |   "_id": "5ca5ec14a5ef65243d180a71",  
|                |   "name": "Tessa",  
|                |   "account": "0x404E221E6845D2eA7D5804A6BEd17a5891Fa691b",  
|                |   "bio": "hi I am Tessa",  
|                |   "avatar": "https://res.cloudinary.com/fyp18003/image/upload/v1554541663/artworks/kitten2.jpg",  
|                |   "__v": 0  
|                | } |
| Actual result | As expected |
| Follow up     | None |

Table 9. TC02 test case for retrieving user

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that user details can be updated</td>
</tr>
<tr>
<td>Related features</td>
<td>FE2: Profile Management</td>
</tr>
</tbody>
</table>
| Pre-requisites | - TC01 and TC02 is executed and passed  
|               | - author_id is known |
| Procedure     | Sending HTTP POST request to  
|               | /users/update/ca5ec14a5ef65243d180a71 |
| Test data     | Altered response from TC02  
| Request body: |  
|               | {  
|               |   "name": "Tessa",  
|               |   "bio": "I love to draw",  
|               |   "avatar": "https://res.cloudinary.com/fyp18003/image/upload/v1554541663/artworks/kitten2.jpg"  
|               | } |
### Expected result

HTML Status: 200  
Respond body: `{ "message": "Success" }`

### Actual result

As expected

### Follow up

Execute TC02 again to check if the altered caption has been updated

Table 10. TC03 test case for updating user

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that user can be deleted</td>
</tr>
<tr>
<td>Related features</td>
<td>FE2: Profile Management</td>
</tr>
</tbody>
</table>
| Pre-requisites | - TC01 is executed and passed  
- author_id is known |
| Procedure | Sending HTTP GET request to /users/delete/ca5ec14a5ef65243d180a71 |
| Test data | author_id: ca5ec14a5ef65243d180a71 |
| Expected result | HTML Status: 200  
Respond body: `{ "message": "Success" }` |
| Actual result | As expected |
| Follow up | Execute TC02 again to check if the user has been deleted |

Table 11. TC04 test case for deleting user

### Test cases on Artwork controllers

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that an artwork can be uploaded</td>
</tr>
<tr>
<td>Related features</td>
<td>FE3: Artwork uploading</td>
</tr>
<tr>
<td>Pre-requisites</td>
<td>TC01 has been executed to create a usable user</td>
</tr>
<tr>
<td>Procedure</td>
<td>Sending HTTP POST request to /artworks/uploadArtwork</td>
</tr>
</tbody>
</table>
| Test data | Request body:  
\`
{  
   "author_id": "5ca5ec14a5ef65243d180a71",  
   "name": "artwork by Tessa",  
   "caption": "Cute dog",  
\`

Table 12. TC05 test case for uploading artwork

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that all artworks can be retrieved</td>
</tr>
<tr>
<td>Related features</td>
<td>FE4: Artwork tracking</td>
</tr>
<tr>
<td>Pre-requisites</td>
<td>TC05 has been executed and passed</td>
</tr>
<tr>
<td>Procedure</td>
<td>Sending HTTP GET request to /artworks/</td>
</tr>
<tr>
<td>Test data</td>
<td>None</td>
</tr>
</tbody>
</table>

| Expected result | HTML Status: 200  
Respond body: { "message": "Success" } |
| Actual result | As expected |
| Follow up | execute TC05 and TC06 few more times to ensure the test case is indeed success in retrieving all existing artworks |

Table 13. TC06 test case for retrieving all artwork

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test that all artworks can be retrieved</td>
</tr>
<tr>
<td>Related features</td>
<td>FE4: Artwork tracking</td>
</tr>
<tr>
<td>Pre-requisites</td>
<td>TC05 has been executed and passed</td>
</tr>
<tr>
<td>Procedure</td>
<td>Sending HTTP GET request to /artworks/</td>
</tr>
<tr>
<td>Test data</td>
<td>None</td>
</tr>
</tbody>
</table>

| Expected result | HTML Status: 200  
Respond body: { "Artworks": [ {  
"author_id": "5ca5ec14a5ef65243d180a71",  
"image_id": "5c76586ec8dd530a9f247b4e",  
"name": "artwork by Tessa",  
"author": "Tessa",  
"access": "https://res.cloudinary.com/fyp18003/image/upload/v1554541663/artworks/boxer.jpg" } ] } |
| Actual result | As expected |
| Follow up | execute TC05 and TC06 few more times to ensure the test case is indeed success in retrieving all existing artworks |
Summary
To test that artworks of given artist can be retrieved

Related features
FE3: Artwork tracking

Pre-requisites
TC05 has been executed and passed

Procedure
Sending HTTP GET request to
/artworks/byAuthor/ca5ec14a5ef65243d180a71

Test data
author_id: ca5ec14a5ef65243d180a71

Expected result
HTML Status: 200
Respond body: {
   "Artworks": [
      {
         "author_id": "5ca5ec14a5ef65243d180a71",
         "image_id": "5c76586ec8dd530a9f247b4e",
         "name": "artwork by Tessa",
         "author": "Tessa",
         "access": "https://res.cloudinary.com/fyp18003/image/upload/v1554541663/artworks/boxer.jpg"
      }
   ]
}

Actual result
As expected

Follow up
To execute TC05 and TC07 few more times to ensure the test case is indeed success in retrieving all existing artworks of the given author

Table 14. TC07 test case for retrieving all artwork by given author

Test Case ID | TC08
---|---
Summary | To test operation of adding source for an artworks
Related features | FE3: Artwork tracking
Pre-requisites | TC01 is executed twice to create two users
| TC05 is executed wice by different author_id
Procedure | Sending HTTP POST request to /artworks/addSource
Test data | Request body:
| {
|   "author_id": "5ca5ec14a5ef65243d180a71",
|   "image_id": "5c76586ec8dd530a9f247b4e",
|}
Table 15. TC08 test case for adding source for an artwork

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>To test retrieve artwork details for artwork portfolio</td>
</tr>
<tr>
<td>Related features</td>
<td>FE3: Artwork tracking</td>
</tr>
<tr>
<td>Pre-requisites</td>
<td>TC08 is executed and passed</td>
</tr>
<tr>
<td>Procedure</td>
<td>Sending HTTP GET request to /artworks/details/5ca5ec14a5ef65243d180a71/5c76586ec8dd530a9f247b4e</td>
</tr>
</tbody>
</table>
| Test data | author_id: 5ca5ec14a5ef65243d180a71  
image_id: 5c76586ec8dd530a9f247b4e |
| Expected result | HTML Status: 200  
Respond body: |
<table>
<thead>
<tr>
<th>Actual result</th>
<th>As expected</th>
</tr>
</thead>
</table>
| Follow up     | execute TC09 again on test data of: 
author_id: 5ca5ec2fa5ef65243d180a72 
image_id: 5cb2097df39f6a2b77219a3b 
to check whether this artwork is updated as derivative of its source |
8. Challenges encountered

Going through each milestone has given the team several difficulties to be resolved. The phase of design and analysis arose the most difficulties as the team has not had sufficient in depth knowledge to produce an accurate and precise system design to confidently rely on. Going into the implementation phase has in fact resolved the difficulties faced in previous phase, as it provided hands-on opportunity for the team to explore different options and select the most feasible approach. Nonetheless, the challenges faced in throughout the project cycle can be summarised by the followings sections.

8.1. Uncertain design at initial phases

Use case design was one of the milestones that the team has most struggled on, and is also a key factor that delayed the overall implementation schedule. The slow progress is due to the nature that each use case takes an enormous amount of time to construct. In particularly the process of use case designing for a web application is complicated by the different elements involved from blockchain and cloud service. Fortunately, consultations with supervisor has provided a clearer picture, followed by implementation phase that enable practical experimenting on the most feasible design.

8.2. Finding reliable source

Since Ethereum and Smart contracts are both relatively new technologies, in particularly that Ethereum was only released a few years ago in 2015[14], resulting in a lack of reputable source. There are inadequate authoritative reference books thus the project mainly relied on online tutorials. Subsequently it also costed the team a great deal of time in exploring feasible and trustworthy tutorials, as well as time to experiment with different implementation approaches.

8.3. Trade-off between efficiency and contract gas usage

A difficulty unique to a project on smart contract is the consideration of gas usage. While considering the data type most suitable to a given data structure, it is also crucial to recognise the potential gas usage for the data structure and operations. For
example, as mentioned in Chapter 6.3.1, while storing the object ids as 12-bytes value is the most intuitive approach, the most gas-usage efficient approach is actually storing the 12-bytes value as 32-bytes.
9. Future Scope

This chapter discusses aspects that was not included in the project, but should be considered in future development to fulfil the maximum potential of this project in a production environment.

9.1. Subsequent features
Features scope is defined in Chapter 1.4, and subsequent features include diversifying artwork format, enabling more user interaction and cryptocurrency usage. Including cryptocurrency usage will be the most imperative function, as it aligns with the project objective of a decentralised media sharing platform as well as enhances the overall functionality on top of the existing architecture. By further incorporating the use of MetaMask on the web application, it can allow artwork selling to be achieved via accounts that are already connected to web3. The extensive function can also facilitate token for potential Ether exchange between users, enhancing the overall user experience and providing an incentive for users to participate on the platform.

9.2. Security enhancement
As the project acts as a proof of concept and demo product, most of the security concerns are not considered. Future development should focus on implementing measurements that improve the overall security of the web application, for example to include input validation or input encoding to avoid code injection attack, and setting the HTTPOnly flag to avoid cross-site scripting[15]. While smart contract apparently implies high level of security with its blockchain properties, it can still be vulnerable to attacks. For example, including logics in contracts that prevent an overflow or underflow attack similar to the case of 92 billion Bitcoin hack in 2010[16].
10. Conclusion

This report summarises the development of this Blockchain and Smart Contract Application, with justified engineering choices and methodologies; design of system interaction and architecture; implementation of the product, and performance testing. Background research and findings support the implementation decision for developing a responsive Single Page Application with ReactJS as it front-end and hybrid approach of cloud storage, database and blockchain as its data storage. The system architecture design demonstrates the feasibility and approach in developing web application as well as simulates users’ interactions with different functionalities across the front-end and the back-end. Technical specification lays out the development environment used, and explains the implementation and rationale of each functionality. Comparisons are made in sections where the final implementation is different to the designed models. Various test cases are conducted to demonstrate the performance of the developed application.

The team was able to resolve various challenges faced during developing the projects, and that includes overcoming uncertainty, determining feasible approach on an unfamiliar system and considering aspects that are unique to a decentralised application such efficiency of contract and gas fee usage. This project has succeeded in demonstrating a feasible solution for implementing a decentralised sharing platform. With future implementation of cryptocurrency usage and security enhancement, it can introduce a renovated concept of blockchain application.
# List of Figures

1. Front page of the website 15
2. Artwork portfolio 16
3. Viewing derivative of an artwork 16
4. Artist portfolio 17
5. Viewing self portfolio 18
6. Profile Editing 18
7. Conceptual Class Diagram of the web application with blockchain and cloud service 20
8. System sequence diagram (SSD) for uploading an image 21
9. System sequence diagram (SSD) for view artworks by specific artist 22
10. System sequence diagram (SSD) for tracking an artwork 22
11. Pseudocode of smart contract variables from Gallery.sol 24
12. Pseudocode of smart contract functions from Gallery.sol 25
13. Frontpage of the web application – preliminary development 25
14. Pseudocode of displaying contract artworks from app.js 26
15. Artwork portfolio – preliminary development 26
16. Pseudocode for displaying an artwork detail from app.js 27
18. Updated pseudocode of smart contract variables from Gallery.sol 31
19. Psuedocode for User and Artwork model on Mongo database 32
20. Updated pseudocode for smart contract functions from Gallery.sol (1) 33
21. Updated pseudocode for smart contract functions from Gallery.sol (2) 33
22. Pseudocode for getting deployed contract from Artwork.route.js  

23. Pseudocode for retrieve artworks from Artwork.route.js  

24. Pseudocode for uploading artworks from Artwork.route.js  

25. Pseudocode for finding artworks of given author from Artwork.route.js  

26. Pseudocode for finding artworks details, including sources and derivatives from Artwork.route.js  

27. Pseudocode for add source for an artwork from Artwork.route.js  

28. Pseudocode for CRUD operation on user from User.route.js  

29. Pseudocode for error handling for controllers  

30. A snapshot of the contract deployment receipt
List of Tables

1. Scope of features 6 - 7
2. Scheduled milestones and progress 8
3. Deliverables according to each stage 9
4. Division of work 9 - 10
5. Cloud service analysis 13 - 14
6. Project files organisation 28 - 29
7. Environment of the final implementation 29
8. TC01 test case for user registration 43
9. TC02 test case for retrieving user 43 - 44
10. TC03 test case for updating user 44 - 45
11. TC04 test case for deleting user 45
12. TC05 test case for uploading artwork 45 - 46
13. TC06 test case for retrieving all artwork 46
14. TC07 test case for retrieving all artwork by given author 47
15. TC08 test case for adding source for an artwork 48
Reference


