COMP4801

Final Year Project for Computer Science

The University of Hong Kong

Topic:

FYP18003

Blockchain and Smart Contract Application

Individual Report

LAU Siu Ming, Alex
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 iterates the project background and objective, and provides an assessment of the progress to date. There are also results from completed milestones, such as justified engineering decision on the development of web application; conceptual class model that supports the design of blockchain and use cases; user interface design that depicts the anticipated product; and an overview the implementation between preliminary stage and the detailed illustration of Front end development on finalized implementation with the support information of technical details of environment setup.
Acknowledgment

We would like to express our greatest gratitude here to anyone who has helped us in completing this report, especially our supervisor Dr. Yuen for his guidance on the blockchain API and the system architecture of the project, also the patience of solving the issues during the implementation phrase. And Vanessa Yim, the partner of the project who contributed a lot in the backend development of blockchain and database architecture. Furthermore, a thank you to our english lecturer Cezar, who has given me a lot of advice and coaching on professional writing.
# Table of Contents

1. **Overview**  
   1.1. Project Background  
   1.2. Project Objective  
   1.3. Project Features  
   1.4. Report Outline  

2. **Project management**  
   2.1. Progress schedule  
   2.2. Project Deliverables  
   2.3. Work Distribution  

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

4. **Work Accomplished**  
   4.1. Design and Analysis  
   4.2. Interface design  
   4.3. System Architecture Design  
   4.4. Use Cases Design  

5. **Implementation & Testing**  
   5.1. Preliminary implementation  
   5.2. Environment and settings  
   5.3. Features  

6. **Final phase of implementation**  
   6.1. Objective  
   6.2. Page rendering  
   6.3. Upload Artwork on Client side  
   6.4. Retrieving data from server  
   6.4.1. Artworks and Authors rendering  
   6.4.2. Artworks Details rendering  
   6.5. Performance  
   6.6. Quality Testing  

7. **Technical specification**  
   7.1. Environment setup  
   7.2. File organization  
   7.3. React Developer Tools  
   7.4. Cloud service API  

---

**Issue 1.1**

**Page 3**
7.4.1. Cloudinary Compatibility: 46
7.4.2. Cloud storage setting 47

8. **Difficulties encountered** 49
8.1. Navigation from clicking image to render page 49
8.2. Failed to re-render the page for image clicking 49
8.3. Parameter missing while fetching data in sourceList 50

9. **Future work** 50
9.1. Cloudinary add-on implementation 50
9.2. Implementation of subsequent features 51

10. **Conclusion** 52

**List of Figures** 53
**List of Tables** 56
**Reference** 57
1. Overview

1.1. Project Background

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. It introduces a relatively less familiar concept to the public, a transparent and decentralised way of record storing.

With the advantages of this technology, the project proposes a method for implementing blockchain technology into a media sharing platform where the record of uploading and sharing history is transparent to the users and they can trace the original work from a derivative work. It helps to issue the problems of digital sharing while providing a platform to prove the originality of an idea.
1.2. Project Objective

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 immutability, eventual consistency and transparency of blockchain, 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. Project Features

This project implements a web application using blockchain technology to achieve eight 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 downloading previous artwork and publish a derivative.
FE5: 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.

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, FE5 and FE6 will be fully implemented in this project therefore do not have further development in scope of subsequent releases. FE2, FE3, FE4, FE7 and FE8 will be partially implemented in the initial scope and be extended in the future.
### Features | Scope of this FYP | Scope of subsequent releases
--- | --- | ---
FE1: Account creation | Fully implemented | -
FE2: Profile Management | Simple modification of user information should be included | To include more information that is useful for enhancing the transparency of the community
FE3: Artwork uploading | Only limited to imagery form of work, such as file format of JPEG, PNG, BMP...etc | To be extended to other formats of work, such as text and video
FE4: Artwork downloading | Future implementation - Fully implemented for downloading the uploaded image one by one | To be extended to download multiple artworks at once for convenience
FE5: Artwork tracking | Fully implemented | Modify the effectiveness of retrieving and writing in to the storage
FE6: Artwork selling | - | -
FE7: Commenting | - | Available under user profile, with option of private or public
FE8: Token generating | - | Token can be exchanged into cryptocurrency

Table 1. Scope of features

### 1.4. Report Outline

This report is sectioned into progress demonstration, evaluation and conclusion. Chapter 1 demonstrates the background and overview of the project, Chapter 2 then gives an overview of the progress with timeline of each milestone and the work distribution of the project. In chapter 3, methodology is explained regarding the design of website, blockchain architecture, and data storage with detailed justifications and findings. The work accomplished is presented in Chapter 4, including the design of user interface, blockchain architecture, use cases design and the implementation of the demo. Chapter 5 illustrates the difficulties encountered as well as limitation discover while designing the architecture of the application. A
conclusion is given in chapter 6 by summarising the progress of the project and discussing the remaining work.

2. Project management

2.1. Progress schedule

The progress of 4 major phases of the project is recorded in Table 2. It demonstrated the expected completion date of milestones, which are categorised in the corresponding phases with the grey box. However, the “✓” represents the actual completion date of the tasks. During the requirement stage, the background research was spent with more time for preparing the proper tools for the project which was in progress with the first half of the Design stage. At the implementation stage, the Front-end implementation was slightly behind the schedule since the complexity of the implementation was challenging than expected.

<table>
<thead>
<tr>
<th>Progress</th>
<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>Requirement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100%</td>
<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>100%</td>
<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></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>85%</td>
<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>100%</td>
<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>100%</td>
<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>100%</td>
<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></td>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100%</td>
<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>95%</td>
<td>M8</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>100%</td>
<td>M9</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>100%</td>
<td>M10</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></td>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95%</td>
<td>M11</td>
<td>Unit testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2. Project Deliverables

There are three phases of deliverables, the corresponding documentations and implementations are listed on Table 3. The deliverables are the completion of the milestones at each phase with the 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 on three phases

2.3. Work Distribution

The works were distributed to two people in this project as the team is in size of two people. The division of labor are divided into half to each of us. Since the project is about web application, the tasks were mainly divided into Front-end and Back-end for the entire progress. The Front-end implementation is mainly involved with cloud service and website construction, where unit testing was also performed. Hence, front-end development was my focus in the project.

<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>Task</td>
<td>Description</td>
<td>Role 1</td>
<td>Role 2</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------</td>
<td>--------</td>
<td>--------</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>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 on each task
3. Project 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 Azure 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. It also enhances the interaction between users and website by being responsive.

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 revealed 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 capacity for storing images that are uploaded to the website, and the other is the functionality of an image recognition service that can facilitate filtering of inappropriate photos or images that might have invaded copyright when uploaded by
users. Table 5 illustrate the details of considered cloud services and gives an comparison in areas of usage, storage service and image recognition service.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Service</td>
<td>CloudSQL</td>
<td>Cloud Storage</td>
<td>Amazon Elastic File System</td>
<td>File Storage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cloud media Storage</td>
</tr>
<tr>
<td>Usage</td>
<td>Provide web framework and content management</td>
<td>Storage for multimedia and blob objects</td>
<td>Content management and web serving, as well as database backup</td>
<td>Enable usage of Windows and Representational State Transfer Application programming interface which is useful for web development</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Media content cloud storage with the support of Client + server side SDKs Add-on integration: Auto-Tagging Analysis Detection Protection Transformations</td>
</tr>
<tr>
<td>Image Recognition</td>
<td>Cloud Vision</td>
<td>Amazon Rekognition</td>
<td>Computer Vision</td>
<td>Customizable add-on from different service providers</td>
</tr>
</tbody>
</table>

Table 5. Cloud service analysis

During the research phase, Azure cloud from Microsoft is selected since it provides a trial version of cloud storage and it is sufficient for storing image files. However, Cloudinary is selected for several advantages. It is convenient and feasible among the choices which provides an integrated environment for testing, developing and deploying application. The file upload and storage of cloudinary is powerful with the client-side SDK(ReactJS) and server-side SDK (NodeJS) (showed in table 5) that allows the establishment of communication between cloud service and the website in front-end and back-end. Also, the Image manipulation from the package of “cloudinary-react” allows developer to utilize the features on image editing with face detection, resizing and cropping, etc. The Add-on integrations on figure 3 also enhance the development of media uploading with auto-tagging and the facial detection feature from google or amazon. It is customizable with high flexibility for
processing media file on our project. Therefore, cloudinary is the most suitable for most of the new project for media storage.
4. Work Accomplished

This chapter demonstrates the work accomplished at the final stage and gives an overview result from each ongoing or completed milestones. The chapter is divided in three sections, the first is the Design and Analysis phase to explain the Interface design and use case design. The second part is the demo implementation during the middle stage of the project, and the final part explains the implementation and testing of the current version of the web application with its architecture and functionality.

4.1. Design and Analysis

4.1.1. 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](image)

Figure 2. Artwork portfolio

![Figure 3. Viewing derivative of an artwork](image)

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.

![Artist Portfolio](image)

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.

4.1.2. 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.
4.1.3. Use Cases Design

Use case design simulates the interaction between different system for each use case that involved the users and is represented in a System sequence diagram. 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. **Implementation & Testing**

This section is divided into two stages of the implementation. The first part is the demo application and the second part is the latest version of the final implementation which established successfully between client and server side.

5.1. **Preliminary 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. It explains the basic setup and structure of the smart contract as back end and simple data extraction in front end.

5.1.1. **Environment and 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 accessI;
        //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. Snapshot of the code in smart contract – variables, from Gallery.sol
Web application

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 12. Snapshot of the code in smart contract – functions, from Gallery.sol

```solidity
function addArtwork (string _name, string memory accessi, uint user_id, uint image_id) public {    artwork[sUser] +=;    uint[] memory p;    uint[1] memory q;    bytes32 hash = keccak256(abl.encodePacked(user_id, image_id));    gallery[hash] = artwork[hash, _name, accessi, p, q]; }

function retrieveArtworkInfo (uint user_id, uint image_id) public returns(uint i_id, uint l_id,string memory a, string memory accessi) {    bytes32 hash = keccak256(abl.encodePacked(user_id, image_id));    return (i_id, l_id, gallery[hash].name, gallery[hash].accessi); }

function addSource (uint user_id, uint image_id, uint source_id) public {    bytes32 hash = keccak256(abl.encodePacked(user_id, image_id));    gallery[hash].source.push(source_id); }

function retrieveSource (uint user_id, uint image_id) public returns(uint[] memory source): {    bytes32 hash = keccak256(abl.encodePacked(user_id, image_id));    return gallery[hash].source; }
```

Figure 13. Frontpage of the web application
Figure 14. Code responsible for displaying all artwork from the smart contract, from `app.js`

Figure 14 illustrates the artwork portfolio page after user clicking on an image, in which artwork information is displayed together with its sources and derivatives, and Figure 15 demonstrates section of code that is in charged of the such function.

Figure 15. Artwork portfolio of the web application
App.contracts.Gallery.deployed().then(function(instance) {
    galleryInstance = instance;
    galleryInstance.retrieveSource.call(user_id, image_id).then(function(gallery) {
        return gallery;
    }).then(function(images){
        //Retrieve previous artworks
        one.append("<div id='sources'>&lt;/div&gt;")
        var sources = &lt;div&gt;
        sources.append("<h3> Sources: &lt;/h3&gt;");
        if (images.length == 0) {
            sources.append("&lt;h3&gt;NoSource at the moment&lt;/h3&gt;");
            return
        }
        sources.append("<div id='sourceItems' class='scrollwork'&gt;&lt;/div&gt;");
        var scroll = &lt;div&gt;
        for (var i=0; i&lt;images.length; i++) {
            var l_id=image[i];
            galleryInstance.retrieveArtworkInfo.call(App.Mongo[1_id], L[1], then(function/art) {
                var user_id = art[0];
                var image_id = art[1];
                var name = art[2];
                var link = art[3];
                var displayEach = &lt;div class='scrollcell'&gt;&lt;img id='artwork' src="&lt;/img&gt;&lt;link class='scrolllink'&gt;&lt;/a&gt;&lt;/div&gt;
                scroll.append(displayEach);
            })
        }
    
    
});

Figure 16. A section of code that is responsible for displaying an artwork and its related works, from app.js
6. Final phase of implementation

This part is about the latest version of the web application which is diverged with the preliminary implementation in chapter 5.1. The following is to explain the entire front end development of the application with specific functions and the components rendering, it aims to illustrate the data flow between client and server side communication.

6.1. Objective

The development of the web application is well-supported with the ease of the ReactJS environment, the features of ReactJS environment allows the flexibility of website design and construction.

The main objective of the frontend is to extract the related data from the storage in blockchain, mongoDB, and cloud storage to render a corresponding page by various functions. The logic and the architecture of the Front-end will be explained in the following section.

6.2. Page rendering

![Figure 17. RouterBrowser structure for rendering page in App.js](image)

Issue 1.1
The “Route” component in figure 18 provides the routing function to render corresponding component/layout of a page with defined path. Whenever a NavLink or Link component is called in figure 17, the component will direct it to the defined path in Route. Hence the attribute “to” links in NavLink [7] to the attribute “path” in Route. Moreover, the elements inside the Container will be rendered to the corresponding call from the other component as imported and defined in the component attribute of Route. Hence, it will render the page and content instantly without loading entire page again, but the navigation bar in figure 19 will not be rendered again since it is outside the container <div>. The corresponding option on the navbar will be blue in color as indicating the currently active page. The approach improves the responsiveness of the website as a single page application.
6.3. Upload Artwork on Client side

Upload artwork is one of the main function in the web application which is involved with the communication with client side, cloud service and server. The upload page in figure 20 is used to read the relevant data(artwork_name, artwork_caption and image file) for the client who is logged in to upload their artwork to cloud and store the URL into the blockchain. The following illustrates the data flow inside the function.

Figure 20. upload page in Upload.js

Figure 21. Upload page with the rendering with react add-on
In the render() function, the upload function is executed when the UploadButton is clicked, it will call the widget.open() to open the Cloudinary upload widget. The component is binded with the Upload() function in figure 22 such that the function can be passed to the child component in /components/UploadButton.js. The child component can inherit the function of Upload(). The input elements on figure 20 are updated while the content is changing. With the use of eventHandler for onChange event in <textarea> and <input> in figure 20, the states of artwork name and caption parameters are updated consistently. Hence the rendering content is consistent and clear with the update of its state from the elements on the page. The upload() function will only be executed when the “Upload” button is clicked. It can ensure user would not mess up the cloud storage by inputting text on the page.

```javascript
//cloudinary widget call
let cloudStorage = window.cloudinary.createUploadWidget(
    {
        cloudName: cloudName,
        uploadPreset: uploadPreset,
        folder: folder,
        // use_filename: true,
        resourceType: "image",
        clientAllowedFormats: "jpg, png",
    }, (error, result) => {
        //success --> retrieve artwork info, url
        if (result.event === "success") {
            this.state.accessL = result.info.secure_url;
            this.state.public_id = result.info.public_id;
            signature = result.info.signature;
            // console.log("201 Upload Success to Cloud: access "+
            console.log("201 Upload Success to Cloud: access ");
        }
    }

    Figure 23. CreateUploadWidget call from Cloudinary reactjs SDK

Inside Upload.js, the upload function is mainly divided into three part. The first part (Figure 23) is to create a widget object by constructing the cloud configuration with relevant data, specifying resourceType and format prevents irrelevant document
uploading to cloud. After widget creation, the second part (figure 24) is about retrieving the data from the callback function of the upload widget API provided. Since cloudinary upload provides callback function with the data in JSON format, developer can retrieve the URL, public_id and other information needed.

```javascript
// fetching artwork data to backend
console.log("Fetching :POST --> uploadArtwork route");
let uploadArtwork = "http://localhost:4000/artworks/uploadArtwork";
fetch(uploadArtwork,
    { method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(
        { author_id: this.state.user_id, 
          title: this.state.title,
          description: this.state.description,
          access: this.state.access,
          captions: this.state.captions,
    })
).then(res => {console.log("POST: uploadArtwork to Ethereum : Error when fetching: $error");});
// fetch source artwork
```

Figure 25. Artwork data fetching by POST method

The final part is establishing the communication with backend with fetch() in POST method as shown in figure 25. By specifying ‘Content-type’ to declare the data format in JSON and constructing the body of JSON response with JSON.stringify on all parameters of the current state, the client side can fetch the data to the /artwork/uploadArtwork route in the server for storing data in mongoDB and blockchain correspondingly.

### 6.4. Retrieving data from server

This part illustrate the related functions which is getting the data from the server side. The following figures are key functions in the corresponding pages which determines the parameters of current state.

**Artworks and Authors rendering**

The method of retrieving list of artworks and authors is the same by fetching to the “/artworks” and “/users” routes which return the data in JSON format. The only difference is the artworks is returned through accessing the smart contract for validating the confidentiality of artwork data, the author list is retrieved by accessing the database. The get() function is called inside async
didComponentMount(), which `get()` will be called once the components of the page is loaded. Then the request will be fetched to the server side.

```javascript
getAllArtworkFromServer() {
  let urlArtworkList = "http://localhost:4000/artworks";
  console.log("Fetching /artwork route");
  fetch(urlArtworkList).then(results => 
    results.json().then(results => 
      this.setState({ artworkList: results.Artworks }));
    catch(error => {
      console.log('400 Retrieving Artwork List Error when fetching: ' + error);
      this.setState({ isFetched: false });
    });
  console.log("Fetching /artwork route");
  this.setState({ isFetched: true });
}
```

Figure 26. Retrieve All artwork from server

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>artworkDetails: {}</td>
</tr>
<tr>
<td>artwork_id: &quot;Scb20936f39f6a2b77219a3a&quot;</td>
</tr>
<tr>
<td>author: &quot;Alan&quot;</td>
</tr>
<tr>
<td>author_id: &quot;Sca5ec2fa5ef65243d100a72&quot;</td>
</tr>
<tr>
<td>derivatives: Array[1]</td>
</tr>
<tr>
<td>isFetched: true</td>
</tr>
<tr>
<td>isLoading: true</td>
</tr>
<tr>
<td>sources: Array[0]</td>
</tr>
</tbody>
</table>

Figure 27 the states in react add-on and constructor in Gallery.js
Figure 28. Rendered page from Gallery.js

Figure 28 is the rendered page of Gallery.js which displays all the artworks in the cloud storage with the proof of work in smart contract. The loading method is to fetch the data from the server and set isLoaded and isFetched to ensure the data is successfully loaded into the current state.

```javascript
// Check the rendering : is artwork loaded
isArtworkLoaded(){
  // console.log( "isArtworkloaded function " + this.state.artworkList.length);
  // checking the loading state
  if (this.state.artworkList.length > 0 && this.state.isLoaded==false){
    console.log("artworklist is loaded " +this.state.artworkList.length);
    this.setState({isLoaded : true});
  }
  if (this.state.artworkList.length == 0)
    console.log("authorlist is null " +this.state.artworkList.length);
}

//whenever a component is updated
async componentDidMount(){
  this.isArtworkloaded();
}
```

Figure 29. isArtworkloaded inside componentDidUpdate in Gallery.js

From figure 29, by calling isArtworkloaded() in componentDidUpdate() to check the artworkList’s length for handling error while data is not loaded or fetched successfully. The componentDidUpdate() is able to check if there is any update of parameter in the current state of the page. Hence, when the getAllArtworkFromServer() in figure 26 is called, the componentDidUpdate() will be called since the previous function changed the state of the page.
With the use of map() function, the nested data in the array is assigned into an array object to store each artwork info for rendering. Then the artwork rendered in figure 30 can link to the detail of the artwork by passing url /artwork/:name/details which is defined inside component Route in the App.js. The data will be store in localStorage as same as the figure 31. It will be retrieved when the Detail page is directed and rendered, which will be discussed in Artworks Details rendering.
Artworks Details rendering

State

- artworkDetails: {}  
  artwork_id: "5cb20936f39f6a2b77219a3a"
  author: "Alan"
  author_id: "5cb20936f39f6a2b77219a3a"
  derivatives: Array[1]
  fetched: true
  loaded: true
  sources: Array[0]

Figure 32. Artwork Details page in ArtworkDetails.js and state of the application

```javascript
async shouldComponentUpdate(nextProps, nextState) {
  if (this.state.artwork_id !== nextState.artwork_id) {
    console.log("This state artwork_id: " + this.state.artwork_id + " next state artwork_id: " + nextState.artwork_id);
    console.log("Should update");
    this.getArtworkDetailFromServer();
    this.render();
    return true;
  }
  console.log("not update");
  return false;
}
```

Figure 33. shouldComponentUpdate function call
After the artwork details are rendered on the page, the current state will be storing the relevant parameters in figure 32, this.state.artwork_id is used to check whether user clicks the artwork in derivative or source list at the bottom of the page. Once the artwork is clicked, the page will be re-rendered with conditional state check function shouldComponentMount() in figure 33. The nextState.artwork_id will be storing the artwork_id of the clicked image which can identify the change of parameter in current page.

Figure 34. getArtworkDetailsFromServer() in ArtworkDetails.js

After the if statement, the data will be updated with the same NavLink method in the artwork isClicked() triggered. The data in localStorage will also be updated with method in figure 31. Finally, the page will be re-rendered with the new updates on the state parameters. The clicked artwork will be rendered with this approach. Hence content of the artwork details will be updated while the client will fetch the request to the server with the click of an artwork.

Validating user identity

Figure 35. isSignedIn function in App.js
The `isSignedIn()` function in App.js is to retrieve the response message from server since the server has read the metamask account of the client and validate whether the metamask wallet account match with the existing user in database.

```javascript
checkUserlogin(){
    //
    console.log("checkUserlogin");
    //
    if (this.state.login_msg == "rr"){
        console.log("Have meta mask account but no user account");
        this.directToSignInform();
    }

    if (this.state.login_msg == "rm")
        console.log("meta mask is not logged in");
    if (this.state.login_msg == "success"){
        this.setState(isLoggedin:true);
    }
}
```

Figure 36. checkUserlogin() function in App.js

The return message will be returned as three scenerios in Figure 36. When the user successfully login to the application, the client side will fetch the GET request to “users/get/:id” and return the info of current user. Also, upload function will be enabled for user to update his/her account info and upload new artwork. The “rr” message will direct user to the sign in form page to fill in the details for registration. Then the data will be fetch to “/users/registrate” by POST method to server for updating the info in database.

### 6.5. Performance

After the illustrations in different part of the functions and rendering, this project is well developed as a Single Page Application which can provide a responsive and functionable Artwork sharing platform. Since the front end development consists of three essential tasks which should be implemented in order to provide the service in our objective. The responsiveness of the data display from server side to client side can be improved since it always takes a short delay during the loading of the data.
Comparing with the demo in the preliminary stage in chapter 5, the latest version of the application establish the communication between client-side and server-side which is functioning in a practical situation and the demo was just testing on the data access to the storage of smart contract.

## 6.6. Quality Testing

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>TC01</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>To test the upload function</td>
</tr>
<tr>
<td><strong>Related features</strong></td>
<td>FE3: Artwork Upload</td>
</tr>
</tbody>
</table>
| **Pre-requisites** | - TC01 and TC02 is executed and passed  
- author_id is known |
| **Procedure** | Sending HTTP POST request to /artworks/uploadArtwork and request to cloudinary storage |
| **Test data** | request from upload to server  
Request body:  
```  
{  
    author_id: "5ca5ec14a5ef65243d180a71",  
    name: "Dou Dou",
    caption: "cute cat",
    access:  
"https://res.cloudinary.com/fyp18003/image/upload/v1554541663/artworks/kitten2.jpg"  
}  
```
request from upload to cloud handled by widget:  
sample img file:
### 7. Technical specification

In this section, the technical details of the environment setup and the architecture of the application are specified. The structure of the system is also illustrated with the supported directory tree and diagram. The cloud service setup is included for explaining the structure of Cloudinary.

#### 7.1. Environment setup

From Table 6, the setup of server and smart contract is consistent to the approach of the methodology which is in Chapter 3 and the development environment of the back end is same as the preliminary implementation. However, the front end environment is different with the demo stage, which is developed in reactJS such that provides a responsive rendering page for the application. The cloud service is also implemented with the Cloudinary SDK with the support in NodeJS and ReactJS.

<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 environment:</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>

Cloud service: Cloudinary NodeJS and ReactJS SDK provided for client and server side development
Mongoose: Object Data modeling library for MongoDB and Node.js
web3: Enable interaction between Ethereum node through HTTP, WebSocket or IPC connection

Table 6. Environment and node packages of the final implementation

Figure 37. Updated Conceptual Class Diagram

From the figure 37, the conceptual class was updated with the simplicity of the structure, most of the artwork related info like access link is storing in the smart contract. The basic relationship between mongoDB, Ethereum and Cloud storage follows figure 7 which is in the design phase of the project. The structure is successfully implemented on both client and server side.

7.2. File organization

Table_ shows the structure of the project, including the relationship and functionality of each important files and directories. The front end page rendering is well-organized with the illustration in Table _. It indicates parent-child relationship between components and pages. The first layer of the application stored with the smart contract related files, the second layer mainly focuses the development on the web application.

```
decentral_app
    │   truffle-config.js
    │
    │   ...
```
<table>
<thead>
<tr>
<th>contracts</th>
<th>Gallery.sol</th>
<th>Contract written in this project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Migrations.sol</td>
<td></td>
</tr>
<tr>
<td>migrations</td>
<td>1_initial_migration.js</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2_deploy_contracts.js</td>
<td>to deploy contract</td>
</tr>
<tr>
<td>test</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>client</td>
<td>package.json</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>Artwork.js</td>
<td>Model schema of artworks</td>
</tr>
<tr>
<td></td>
<td>User.js</td>
<td>Model schema of users</td>
</tr>
<tr>
<td>routes</td>
<td>Artwork.route.js</td>
<td>REST API controllers for Artwork</td>
</tr>
<tr>
<td></td>
<td>User.route.js</td>
<td>REST API controllers for User</td>
</tr>
<tr>
<td>src</td>
<td>App.js</td>
<td>Route to all pages</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>components</td>
<td>Reusable components</td>
<td></td>
</tr>
<tr>
<td>contracts</td>
<td>Migrated contract</td>
<td></td>
</tr>
<tr>
<td>css</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pages</td>
<td>ArtworkDetails.js</td>
<td>Artwork portfolio</td>
</tr>
<tr>
<td></td>
<td>Author.js</td>
<td>Load all Users</td>
</tr>
<tr>
<td></td>
<td>AuthorProfile.js</td>
<td>Author portfolio</td>
</tr>
<tr>
<td></td>
<td>EditInfo.js</td>
<td>Edit user info</td>
</tr>
<tr>
<td></td>
<td>Gallery.js</td>
<td>Load all artworks</td>
</tr>
<tr>
<td></td>
<td>Home.js</td>
<td>Home page</td>
</tr>
<tr>
<td></td>
<td>SignInForm.js</td>
<td>Sign in page</td>
</tr>
<tr>
<td></td>
<td>Upload.js</td>
<td>Upload page</td>
</tr>
<tr>
<td></td>
<td>utils</td>
<td>fetch blockchain provider</td>
</tr>
</tbody>
</table>

Table 7. Project files organisation
index.js  
  - App.js  -> style.css App.css

  - /components/NavBar.js

  ../pages/
  - Home.js  //default routing
  - /components/Header.js
  - Banner.js

  - About.js

  - Featured.js

  - Gallery.js
  - ArtworkDetails.js
  - itself  // re rendering

  - Author.js
  - AuthorProfile.js

  - Upload.js
  - /components/UploadButton.js

  - SignIn.js

Figure 38. Components file organization for rendering
7.3. React Developer Tools

React developer tools to chrome was installed to analyse the data flow between client and server side. It is useful for testing and understanding the responses and requests between client and server. It can indicate the change of parameters in the current state.

The Element window can browser the routed components and show the selected component with its value and state. The React developer tools is useful in testing the functionalities in each .js file.
7.4. Cloud service API

7.4.1. Cloudinary Compatibility:

For the selection of cloud service of storing image files, cloudinary is selected as the host for handling data storage for images uploaded from the client side. Cloudinary supports reactJS for web development with various add-ons to enhance the image processing and establish the communication between cloud, client side and server side. Since the front-end development is in reactJS with provided ReactJS Software Development Kit, it enhances the functionality of the application by supporting with image manipulation and other customization of host setting. And the back-end development is mainly in NodeJS, it is also compatible to it with supported SDK, which allows the future enhancement on establishing different server communication with the cloud service to provide a comprehensive platform with various functions supported.
7.4.2. Cloud storage setting

The environment setup of the cloud storage in figure 40 allows the use to modify the type of uploading as the signed and unsigned uploading. Both are easy to implement as signed upload is required with the configuration of cloud name, API key and secret key to establish the connect with cloudinary storage. The unsigned upload is the setting of this project since it only requires the unique upload preset in figure 41 and

Figure 40. Cloud storage properties

Figure 41. Cloud storage upload presets settings
the cloud name to establish connection. The upload presets can define the behavior for the uploads, which is more convenient for handling some specific image for different purposes. The details of setting of the cloud in this project is shown in figure_. Since the application is to store the image file and extract the URL of it, the current setting of unsigned upload is sufficient for it. Also, 25 credits are provided as the maximum storage of 25GB and it is large enough for storing the data of avatar of uses and artwork uploaded.

Figure 42. Pre-load sample image for testing

In figure 40, there are some sample images uploaded to the cloud for function testing for cloud environment setup. Also in figure 41, there are two folders for image storage which is for artwork and user_avatar. The user’s avatar will also be stored on the cloud and retrieved by its URL which is stored in the user’s info in mongoDB. This approach is the simple to get the avatar from database. However, the access link of the artwork is stored in the block of the blockchain during upload process. They are accessed in different ways such that the retrieve of the data of artwork has to gone through the verification of smart contract in the backend and return it back to the front-end which is explained in Chapter 5.2.2 and 5.2.3.
8. Difficulties encountered

Going through each milestone has given the team several difficulties to be resolved, especially in the part of design and analysis. The team has been facing most of the challenges on the adjustment and error debugging for system enhancement during testing phase.

8.1. Navigation from clicking image to render page

During the implementation in gallery.js and ArtworkDetails.js, it was required to provide the link to each image for directing the path from the defined route component in app.js. At the beginning, it was confused to figure out how to make use route on the dynamic rendered components like the images on the gallery.

The solution is to make use of NavLink and withRoute with provided path in <Route> components in App.js, NavLink enables the dynamic rendering components with linking properties to the correct url. Hence, the rendered images are able to direct to their corresponding pages with image details rendered by passing correct data via localStorage() function.

8.2. Failed to re-render the page for image clicking

During the implementation in artwork details rendering which is in Chapter 6.4, it was unable to render the same page again with the specified path of “/artworks/:name/details”. The state was updated with the correct function goToImageDetails() in figure 31. However, the fetching and render() were not called by anyone which led to the static page which was just rendered by once even the state was updated.

The solution is to make use of shouldComponentUpdate() [8] function call which is provided in ReactJS. It provides the customized conditional check for
any states change in the current page. It allow users to re-render the page if the condition match.

### 8.3. Parameter missing while fetching data in sourceList

The sourceList and derivativeList are not loaded after the successful fetching to the route “/artworks/:authorid/:artworkid”. The response return from the server with correct data set except the sourceList and derivativeList is empty. There could be some issues from the backend data retrieving which has to be studied and discovered the order of the data parsing from the server while retrieving info from smart contract and database. There could be unknown conflict during request fetching.

### 9. Future work

This section is about the potential implementations which can enhance the completeness of the web application. There are some features not implemented on Table 2 and Cloudinary add-ons, which can be accomplished in the future which can improve our quality of service and provide interactive experience on our artwork sharing platform.

#### 9.1. Cloudinary add-on implementation

From the features described in table 5, our application is built with the uploading feature to the cloud storage only. Hence the image recognition feature from Azure or google which are provided on cloudinary can be implemented for securing the originality of each artwork. This feature can be utilized for the duplication upload of an artwork by detecting the properties of an artwork.

The image recognition feature can also identify the density of human faces which may be useful for building different categories for all artworks. The add-on can get rid of the section of inputting artwork category for an image during uploading.
9.2. Implementation of subsequent features

Since our platform aims to establish a healthy community of artwork sharing, comment and rating system could be implemented for the enhancement on user experience. Also, the token reward system should also be implemented with the use of cryptocurrency, if the user contributes more by having a great reputation (not copying others' work), they should be rewarded with token for trading with certain amount of cryptocurrency.
10. Conclusion

This report summarises the progress of this Blockchain and Smart Contract Application to date, with justified design and engineering choices; design of conceptual model; Use cases design; and illustration of user interface. 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 user interface demonstrates the developing web application and simulate users’ interactions with the website on different functionalities. The conceptual class model also helps to partially understand the blockchain architecture and institutes the process of use case design. The completion of use cases aids the understanding of the flow of system operations.

As reflected by the progress appraisal, an application demo is for testing the environment proposed during the design phase. After all the preparation phase, the finalized version of the web application developed with major functionalities. The establishment of channeling data between cloud, smart contract and database is successful which allows client side to upload and browse artworks. This project allows our team to discover the potential of having an artwork sharing platform with the use of blockchain and smart contract for preserving the originality of artwork digitally. The next major step shall be on enhancing functionalities and interface with the aid of tools from cloudinary service, and make our data channeling more secure.
List of Figures

1. Front page of the website 12
2. Artwork portfolio 13
3. Viewing derivative of an artwork 13
4. Artist portfolio 14
5. Viewing self portfolio 15
6. Profile Editing 15
7. Conceptual Class Diagram of the web application with blockchain and cloud service 16
8. System sequence diagram (SSD) for uploading an image 17
9. System sequence diagram (SSD) for view artworks by specific artist 18
10. System sequence diagram (SSD) for tracking an artwork 18
11. Snapshot of the smart contract – variables 20
12. Snapshot of the smart contract – functions 20
13. Frontpage of the web application 21
14. Code responsible for displaying all artwork from the smart contract, from app.js 21
15. Artwork portfolio of the web application 22
16. A section of code that is responsible for displaying an artwork and its related works, from app.js 22
17. RouterBrowser structure for rendering page in App.js 29
18. <NavLink> components in NavBar.js 30
19. Navigation Bar rendered in NavBar.js 30
20. upload page in Upload.js 31
21. Upload page with the rendering with react add-on

22. UploadButton Component

23. CreateUploadWidget call from Cloudinary reactjs SDK

24. Retrieve data from the upload callback

25. Artwork data fetching by POST method

26. Retrieve All artwork from server

27. the states in react add-on and constructor in Gallery.js

28. Rendered page from Gallery.js

29. isArtworkloaded inside componentDidUpdate in Gallery.js

30. conditional rendering and NavLink for artwork routing and mapping

31. goToImageDetail() with the use of localStorage

32. Artwork Details page in ArtworkDetails.js and state of the application

33. shouldComponentUpdate function call

34. getArtworkDetailsFromServer() in ArtworkDetails.js

35. isSignedIn function in App.js

36. checkUserlogin() function in App.js

37. Updated Conceptual Class Diagram

38. Components file organization for rendering

39. React developer tools

40. Cloud storage properties

41. Cloud storage upload presets settings

42. Pre-load sample image for testing
43. Cloud Storage environment
## List of Tables

1. Scope of features 5 - 6  
2. Scheduled milestones and progress 7  
3. Cloud service analysis 11  
4. Division of work on each task 10  
5. Cloud service analysis 14  
6. Environment and node packages of the final implementation 42  
7. Project files organisation 43
Reference


Available from: https://cloudinary.com/documentation/cloudinary_add_ons


Available from: https://reacttraining.com/react-router/web/guides/basic-components

Available from: https://reactjs.org/docs/react-component.html