SEMESTER 2 2023
COS30049 - Computing Technology Innovation Project

Assignment 1 - Front-end Development (Static Website)
+
Assignment 2 - Back-end Development (Dynamic Website)

By Group 1-64

Due: Sunday, 15/10/2023, 11:59 pm

Members:
Yap Zhe Wei (Leader): 103508895
Vu Gia Thinh Dang: 103177240
Yong Yuan Chong: 101224021

Table of Contents

I. /Project Background and Introduction 3

II. /Team Introduction 3

III. /Project Requirement List and Description 3

/Core Functional Requirements 3

/Description 4

IV. /Project Design 5

/Frontend Prototype (Figma) 5

/Website Design 6

V. /Backend Database Design 27

/FastAPI Design 28

/Functionalities Description 42

/ Smart Contract Implementation 54

/ Project Deployment Instruction 57

VI. /Overall System Architecture Design 59

VII. /Project File Structure 61

VIII. /Conclusion 63

IX. /References 64

I. /Project Background and Introduction

In the digital era, decentralized systems have emerged as solutions to the vulnerabilities of centralized trading platforms, such as limited transparency, hacking risks, high fees, and censorship. Leveraging the blockchain's inherent security and transparency, decentralized trading aims to offer a safer, more open, and fairer trading experience (Maheshwari, 2023).

Through this project, we are expected to develop a decentralized trading platform, capitalizing on the strengths of blockchain technology. The platform is designed to ensure that every transaction made by users is not only transparent but also secured against potential tampering. A crucial part of our project is its web/app platform, it is designed to be responsive with good browsing experience regarding different devices. While the backend development will be done using Node.js, our smart contract's function will be powered by Ether.js.

In this specific assignment of the project, we mainly focus on designing a static website for our trading platform using React.js.

II. /Team Introduction

Vu Gia Thinh Dang: 103177240
Yong Yuan Chong: 101224021
Yap Zhe Wei: 103508895

III. /Project Requirement List and Description

/Core Functional Requirements

1. **Display Digital Assets:** Users can view digital assets available for trading.
2. **Database Integration:** All the listed digital assets information should be stored in the database. – *Assignment 2*
3. **Search and Filter functionality:** The system should provide a search and filter functionality for users to discover specific assets of interest.
4. **Smart Contract Integration:** The website will implement smart contracts to act as escrow during the trading process, and smart contracts should ensure that assets are held securely until the trade is completed or cancelled. – *Assignment 2*
5. **Transaction History:** Users should have access to a transaction history to view their past trades.

/Description

The concept of the Decentralized Trading Platform revolves around providing a solution that meets the evolving needs of purchasing assets online. As blockchain technology gains popularity and digital assets become increasingly in demand there is a pressing requirement, for a system that combines security and user friendliness seamlessly.

At its core this our web application aims to create a platform where individuals can explore and engage in purchasing different types of assets. It goes beyond visualization by incorporating a database that ensures data consistency and allows for quick access. This integration, which will be implemented in Assignment 2 serves as the structure supporting features.

Recognizing the range of assets available to purchase and considering users unique preferences the platform will include an advanced search and filter tool. This feature enables users to easily find assets that align with their preferences streamlining the payment process.

Security holds importance in the realm of exchanges. To address this concern smart contracts will be utilized within the platform. These contracts will be deployed in Assignment 2. Play a role in mediating transactions. By holding assets and ensuring all parties involved fulfill their obligations before releasing them these contracts not enhance asset safety but also instil confidence, in the transactional mechanism.

Moreover, ensuring clarity is of importance. Users will have access, to their records, which will allow them to reflect on exchanges track the statistics of their assets and plan their future purchases.

IV. /Project Design

/Frontend Prototype (Figma)

Draft version

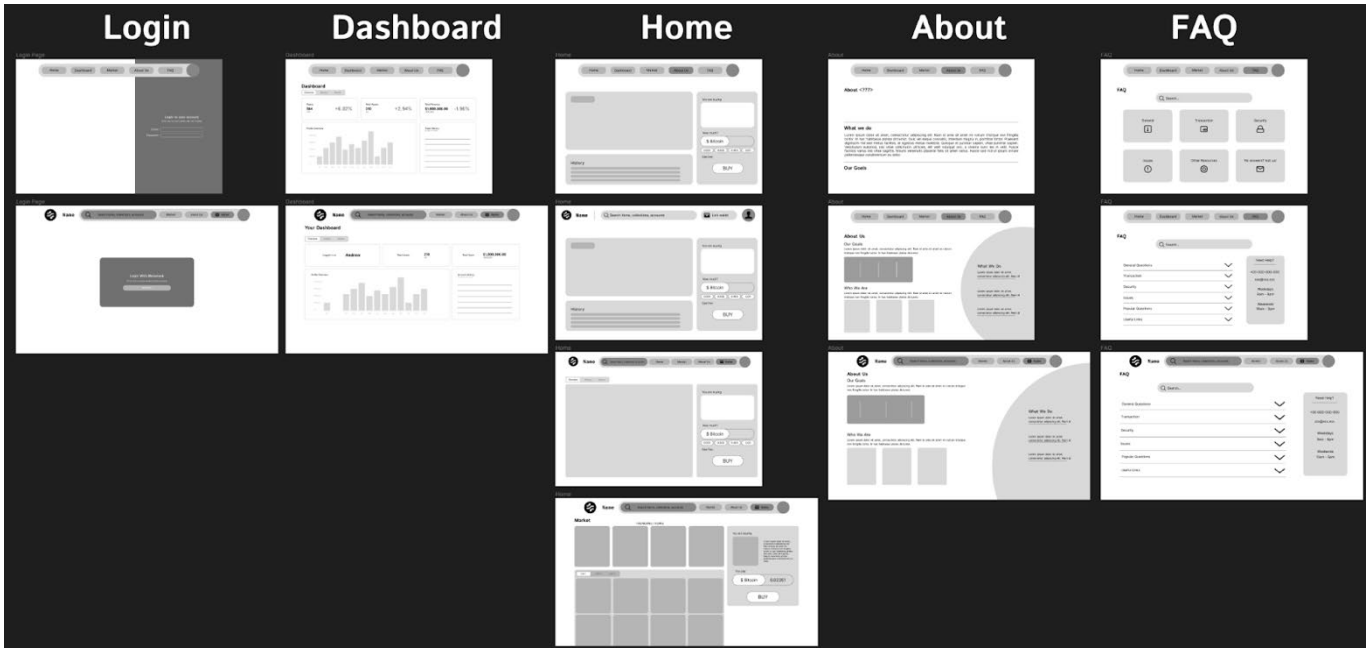

Figure 1.0: Draft Design

Final Version

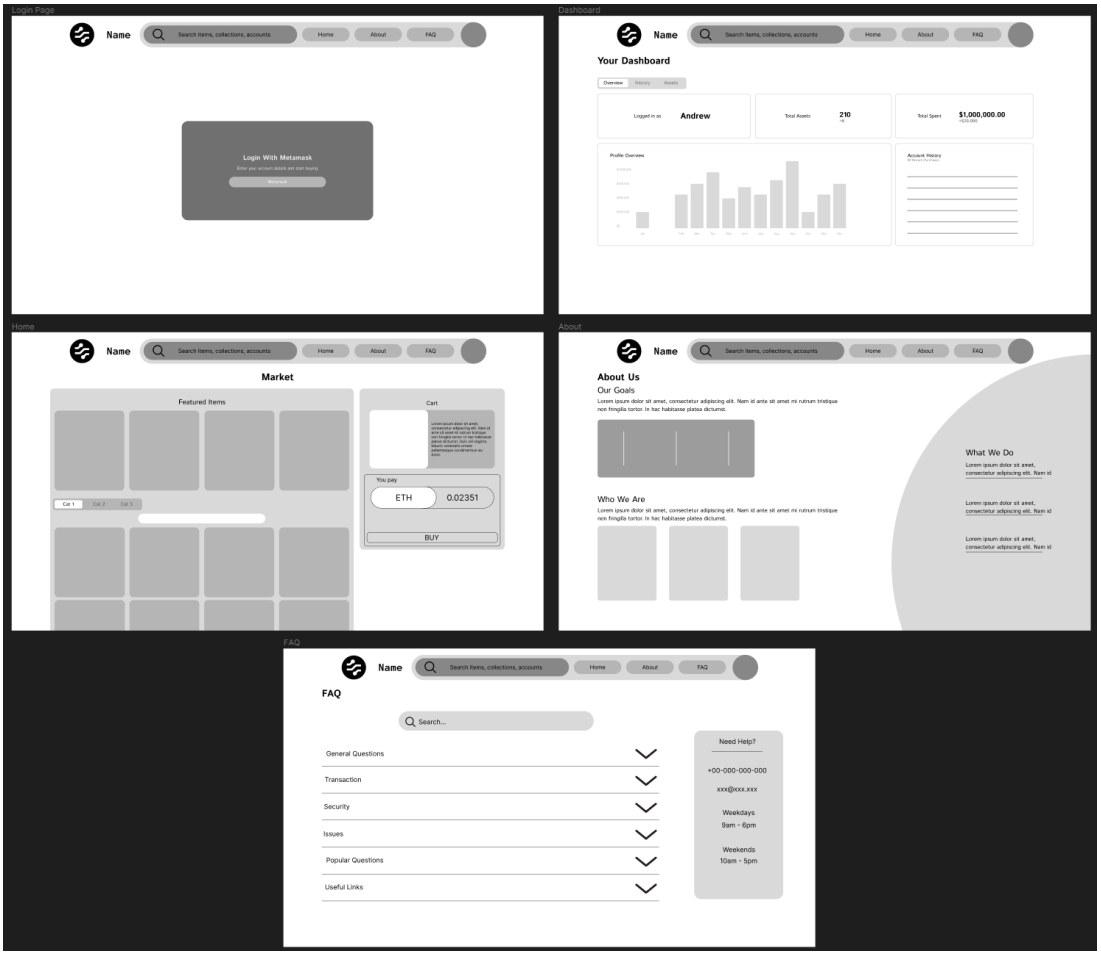

Figure 1.1: Final Design

/Website Design

Navbar

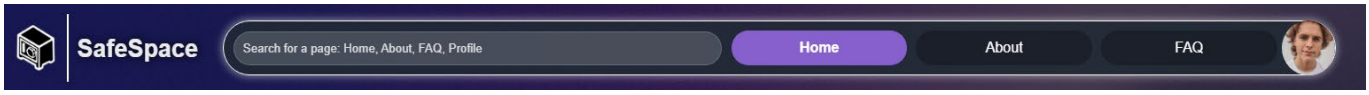

Figure 2.0: Navbar

The navbar is the main navigation in our website's architecture. Serving as the primary channel, it facilitates users in their navigation across different pages of our trading platform. Within the expansive webpage, the navbar helps in steering users across the comprehensive content of our website. In the code, we added multiple features and functionalities across the navbar by utilizing React and CSS. We made sure that the navbar is usable and adaptable across various screen sizes, moreover it is equipped with a search bar so users can easily find the pages they are looking for.

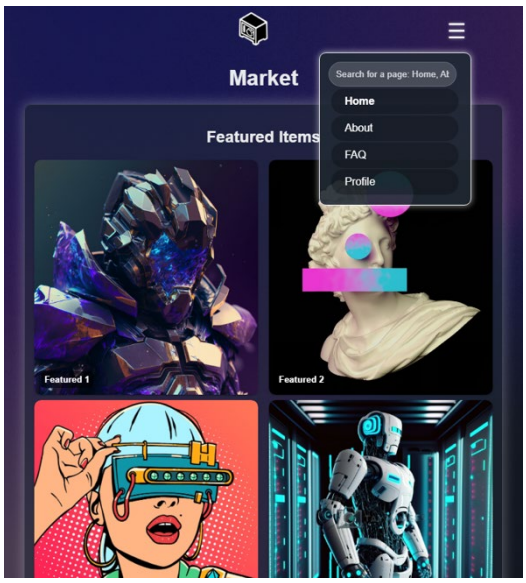

Figure 2.1: Responsive Navbar

Functionalities:

This navbar utilises the react router NavLink, useNavigate, useLocation from the react-router-dom toolkit to redirect the users within the webpage.

1) **Sticky Navbar**

   We made the navbar always stick to the top of the page, allowing easier navigation from any part of the webpage. Users would not need to scroll to the top of each page to navigate the website.

2) **User Friendly Search Bar and Navigation Tab**

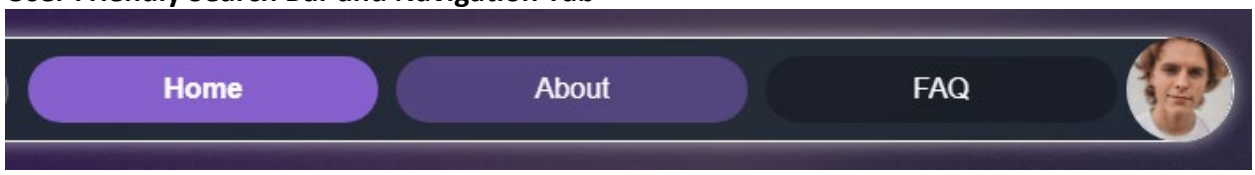

Figure 2.2: Responsive Navbar

The navbar is implemented with user-friendliness in mind, and we added styling for the active tab, hovering tab, and inactive tab. When a user is on a webpage, the active navbar button will appear bright purple (#8660cd) accordingly. Additionally, when a user hovers over a button, it will appear in a more translucent purple (#8660cd7a) to show clearly which tab the user is

hovering over.

Same goes to the search bar, both the placeholder, and input text colour and input background will change colour when user is hovering over it.

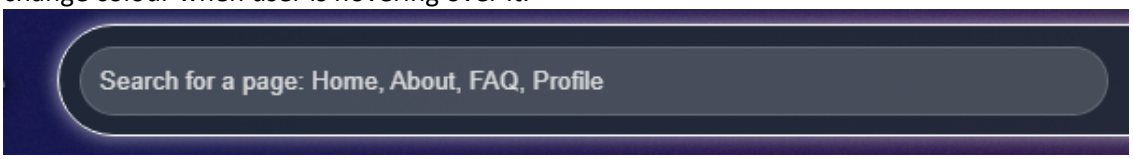

Figure 2.3: Search bar

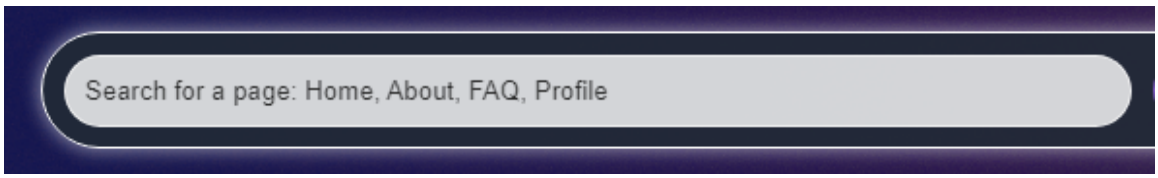

Figure 2.4: Hovered Search bar

3) Search Bar

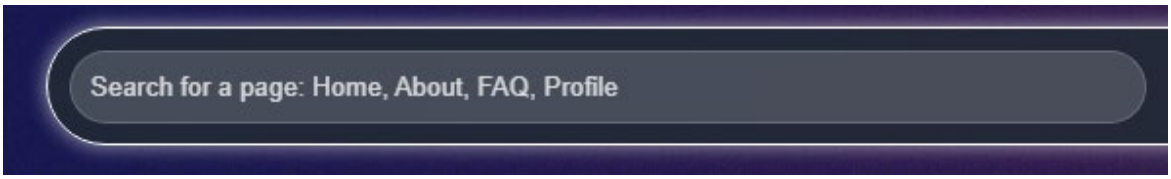

Figure 2.5: Search bar navigation

In the search bar, users have the ability to search for the page(s) they want to navigate to if they feel it's easier to navigate. It helps users seamlessly transition between different pages by inputting terms like "home," "about," etc.

4) Navbar Responsiveness

To make the navbar responsive on smaller screens, we utilized a “hamburger” icon to replace the navbar on smaller devices to clear up the screen, and we compressed all the navbar components (Name, Logo, Search bar, Tabs, and Profile Icon) into a dropdown menu.

a. Dropdown menu

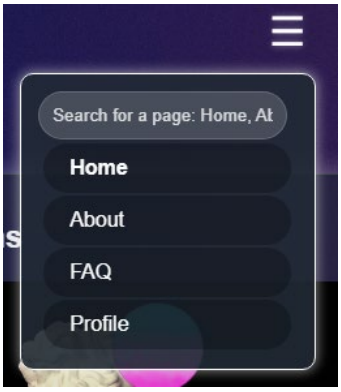

Figure 2.6: Dropdown menu

We removed the website name and only show the logo on smaller screens, then we aligned all the other components in the dropdown menu to be vertically aligned instead of horizontally.

b. Profile button

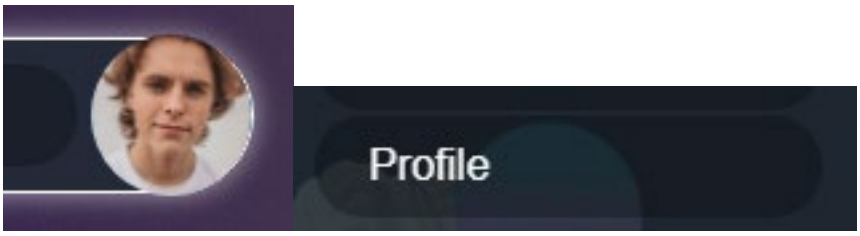

Figure 2.7: Profile icon in dropdown menu

On smaller screens, we replaced the profile icon with a text button like the other navbar buttons to look consistent on smaller devices in the dropdown menu.

c. Logo

Figure 2.8: Logo on smaller devices

To achieve a cleaner view of the overall design of the webpage on smaller devices, we decided to keep only the logo of our web page and keep the name hidden on smaller resolutions.

Login

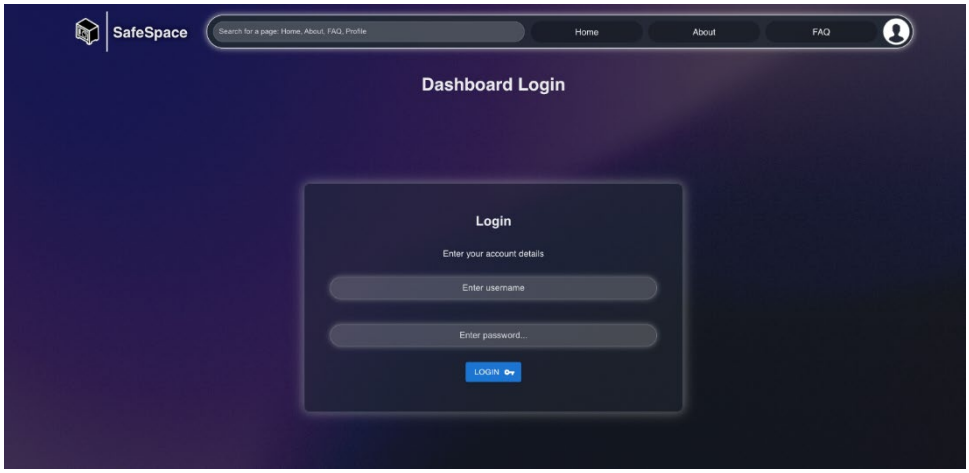

Figure 3.0: Login Page

In the second assignment, we developed a straightforward and unchanging login interface for our trading platform. We utilised traditional usernames and passwords for login. Users can explore the market on *SafeSpace* without the need to log in. Yet, accessing the dashboard or buying item will require user authentication.

Functionalities:

This page utilises the MaterialUI components such as Card, Grid, etc, to create a user-friendly page. It also retrieves user's information from the backend table for user authentication.

1) **Backend user authentication login:**

   We basically have a flag called "isLoggedIn" to keep track of the log in state of the user. When the user clicks on the dashboard for the first time, they are prompted with the sign in screen, and they have to enter their login info to get access to the dashboard. The username and the password are then checked to enable login. When the information is incorrect a warning screen will appear

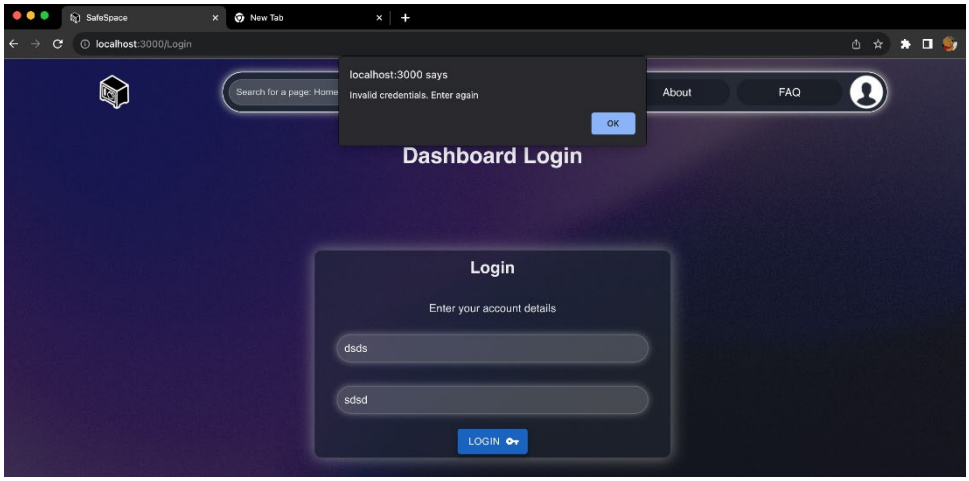

Figure 3.1: Warning Prompt

A warning will pop up if the user clicks the button without providing login information or if they provide wrong login information.

2) Logout

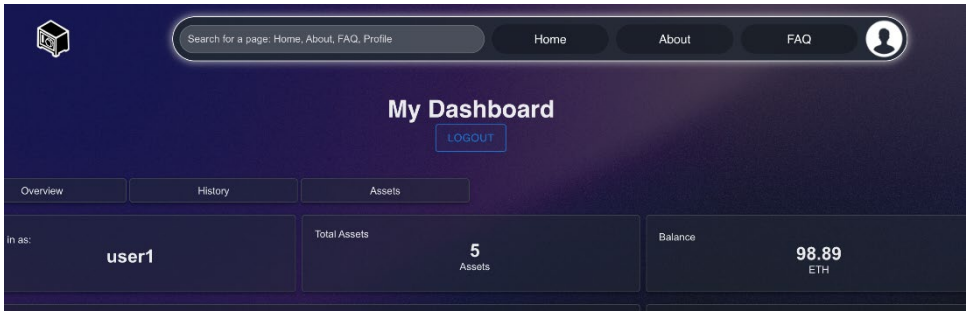

Figure 3.2: Logout Button

We also implemented a logout button for users to go back to the login screen we showcased above. Our logout button simply navigates the current page to the login page (login.js) and requires users to enter user information again to gain access to the main dashboard page and the buy function.

Dashboard

Figure 4.0: Dashboard

We designed a dashboard page that provides an overview of a user's profile once they've logged in. This dashboard features graphs detailing the user's assets based on amount and categories. Additionally, there's a segment showcasing the user's transaction history and essential profile details above. For a more in-depth insight regarding transactions, users can click on the “History” tag. Similarly, the dashboard allows users to see an overview of their current assets and their respective values by clicking on the “Assets” tag. Finally, the “Overview” tag brings them back to Figure 4’s UI.

Functionalities:

The page utilises the grid system provided by MaterialUI to give the user a comprehensive overview of their account’s activity, concise analysis, and current assets.

1) Overview Tab

We decided that the first window when the user accesses the dashboard will be the overview of the account. According to Figure 4 above, users can view their account’s transaction and an account analysis in the form of a bar chart. There is also some very simple logged in information within information cards that are presented before the chart like name, number of assets, and recent spending.

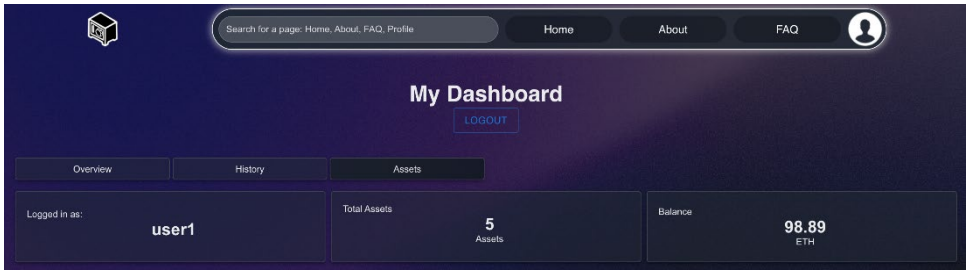

Figure 4.1: Navigation within dashboard

a. Navigation:

If the user wants to get back to the overview window, they can click on the overview tag or the name tag.

b. Donut chart:

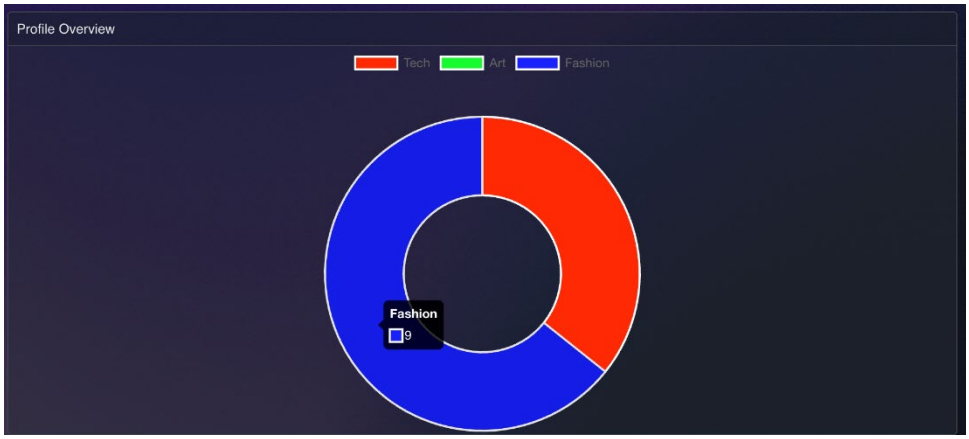

Figure 4.2: Donut Chart

We decided to implement the Donut chart which is provided by chart.js. This chart basically gives an overview of the user's assets detail by splitting them into categories with different colours. This chart is fully backend compatible as it is retrieving the information from the "bought" table for the chart data. When the user buys more items of specific categories, the chart will change accordingly.

We must import a lot of dependencies from chart.js as well as declaring the chart's element that we used for the visualisation. Most of the styling of the donut chart was done within the ChartOptions function provided by chart.js.

c. Transaction

Figure 4.3: Transaction History

We added the transaction history on the right of the screen and these transactions are limited to five most recent transactions using the slice function. Full history transactions can be shown when the user accesses the main window for history. As the user buys more items, they will be shown on the side on the history transaction windows.

2) History Tab

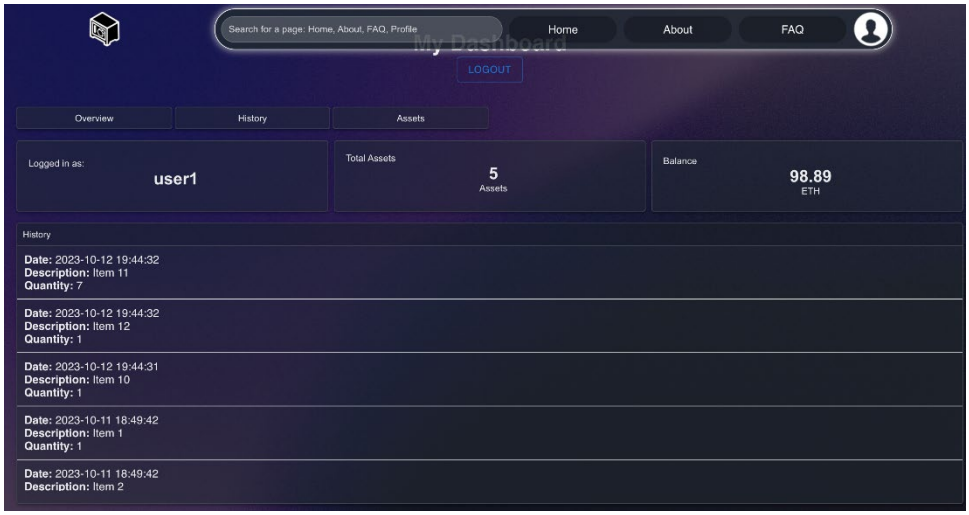

Figure 4.4: History Tab

Our history window showcases all transaction history that has been made by the account. There is scrolling enabled for the history window so users can see older transactions. As we move further with the project, there will be a way to further filter these transactions by time or by monetary value in and out.

a) Navigation

User can click on history tab and the “Total Spent” tab to get to history window.

3) Assets Tab

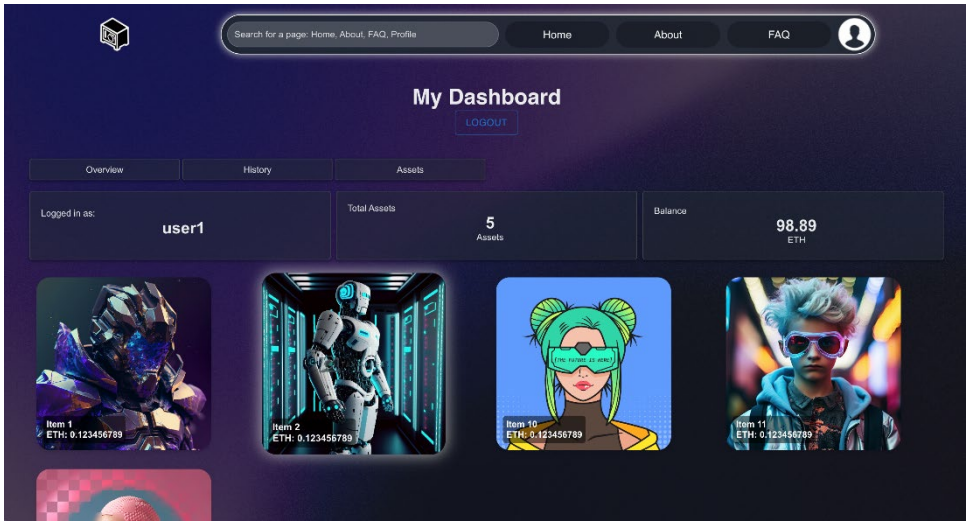

Figure 4.5: total assets window

The assets window gives the user an overview of all their assets and their current values in ETH. The assets have a hover effect as it enlarges and produces a box shadow when we hover mouse over it. This window retrieves all items in the backend and our backend table is updated accordingly when user activates the smart contract when they buy items.

a) Navigation

User can click on assets tag or total assets tag to get to assets window.

4) Responsive

Below are the mobile views of our dashboard/login page.

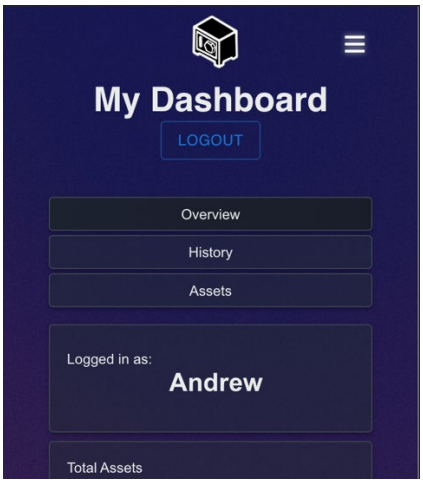

Figure 4.6: Overview

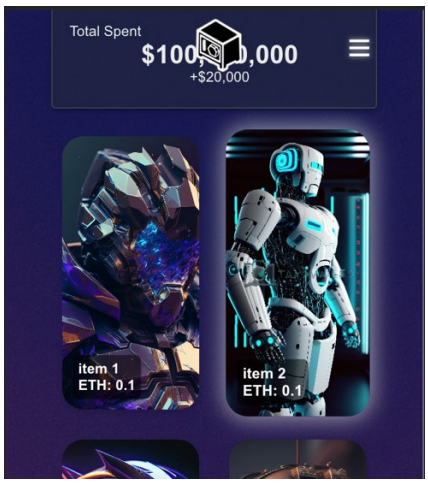

Figure 4.7: Total Assets

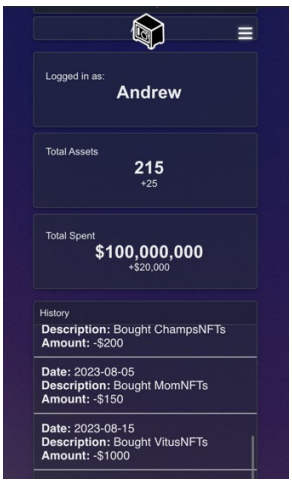

Figure 4.8: History

Using the built-in responsive structure catered for different resolutions provided by MaterialUI, we were able to adjust the responsiveness of our trading platform quite easily and straightforwardly.

We mainly focused on adjusting the grid layout that holds the actual content so that it makes sense in both mobile view and web view. The current screen layout for our elements is consistent across multiple pages.

5) Error Handling

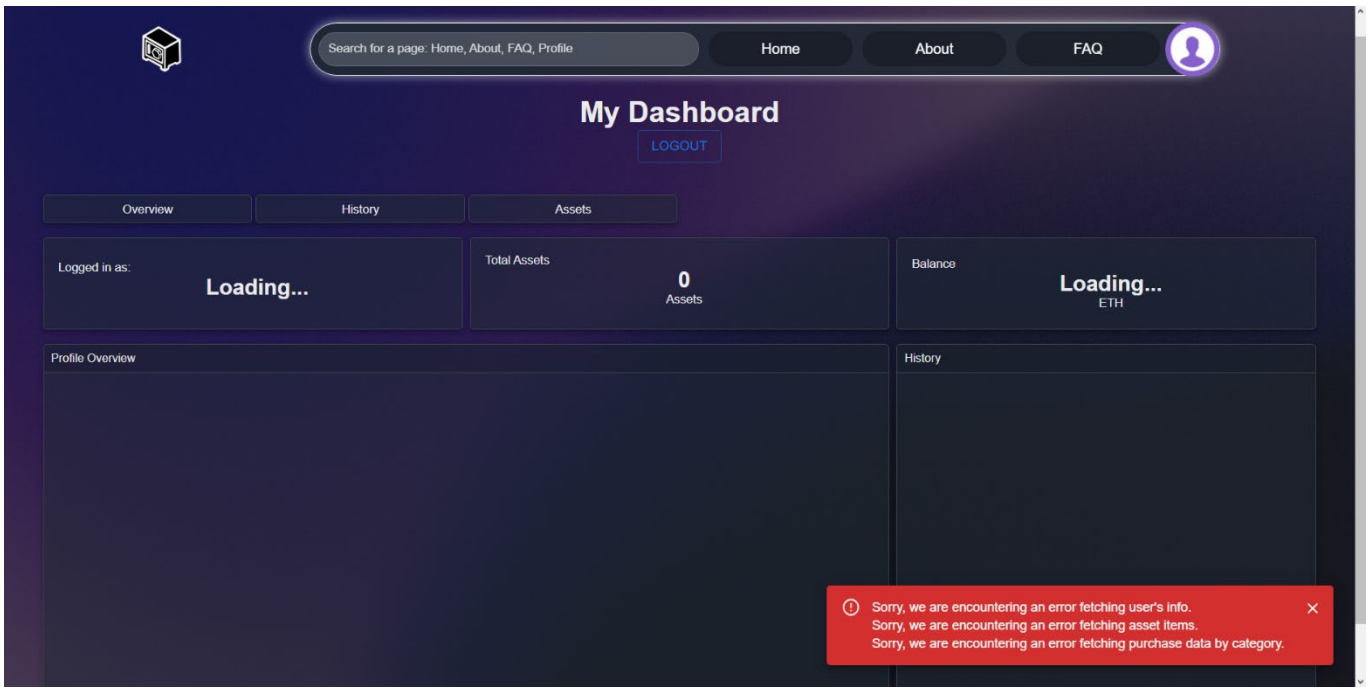

Figure 4.9: Dashboard Error Messages

```
<Snackbar
  open={errorSnackbarOpen}
  autoHideDuration={6000}
  onClose={handleErrorSnackbarClose}
  anchorOrigin={{ vertical: "bottom", horizontal: "right" }}
>
  {/* Display all error messages */}
  <Alert onClose={handleErrorSnackbarClose} severity="error">
    {Array.isArray(errorMessages) && errorMessages.map((message, index) => (
      <div key={index}>{message}</div>
    ))}
  </Alert>
</Snackbar>
```

Figure 4.10: Snackbar usage

Error messages will be displayed using a Snackbar when the page has problem fetching the required data from the backend.

Home/Market

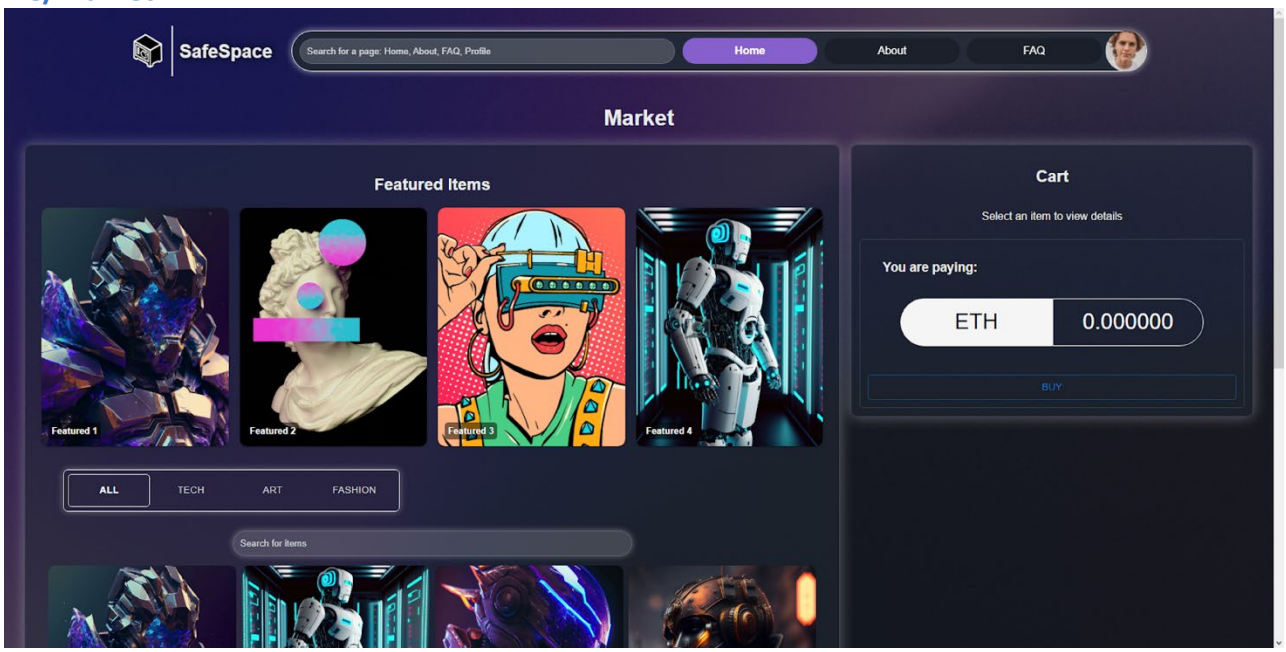

Figure 5.0: Home/Market page

We crafted a market interface, designated as the homepage – the very first page a user encounters. This market spotlights all the highlighted or trending assets available for purchase. Additionally, the platform categorizes assets, enabling users to explore diverse asset types like NFTs, art, music, among others.

To enhance user experience and navigation, we've incorporated a filter function, allowing users to selectively view categories of their preference. Further refining the user-friendly approach, a straightforward purchasing mechanism is integrated right on the market homepage, allowing users to add items to their cart and proceed with purchasing.

Functionalities:
This page utilizes the MaterialUI components, and icons such as Paper, Grid, Typography, Tabs, etc, to create a user-friendly page.

1) Cart

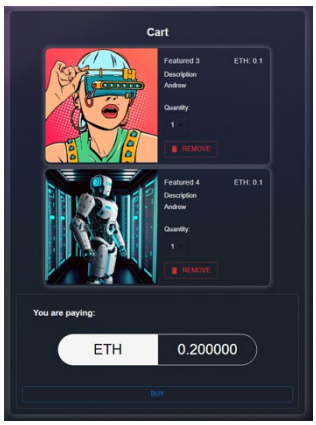

Figure 5.1: Cart

We decided the position the cart section beside the assets at all times, this will help users to keep track of what is in the cart instead of clicking in and out of a cart just to see what is in it. This will allow for easier cart management.

a. Add/Remove from cart

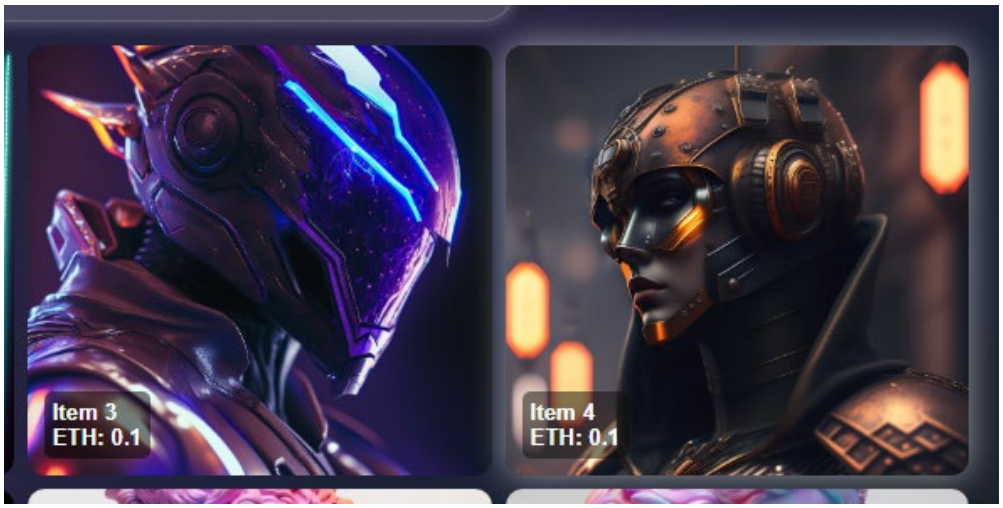

Figure 5.2: Assets Hover Effect

The assets will have a box-shadow behind it if the user hover of it. We implemented the handleItemClick function so user can add/remove the item by clicking on it.

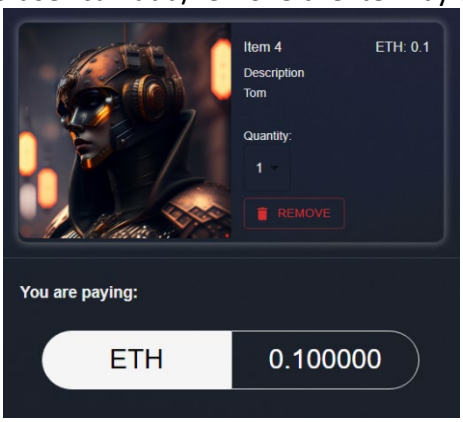

Figure 5.3 Remove Button

Users will also have the option to remove it from their cart by clicking on the remove button.

b. Calculate Total

Figure 5.4: Calculate Total

The total cost of the added item will be updated instantly when the user adds something to their cart. The 'total' constant is calculated using the price of each item and its quantity.

2) Items Filter

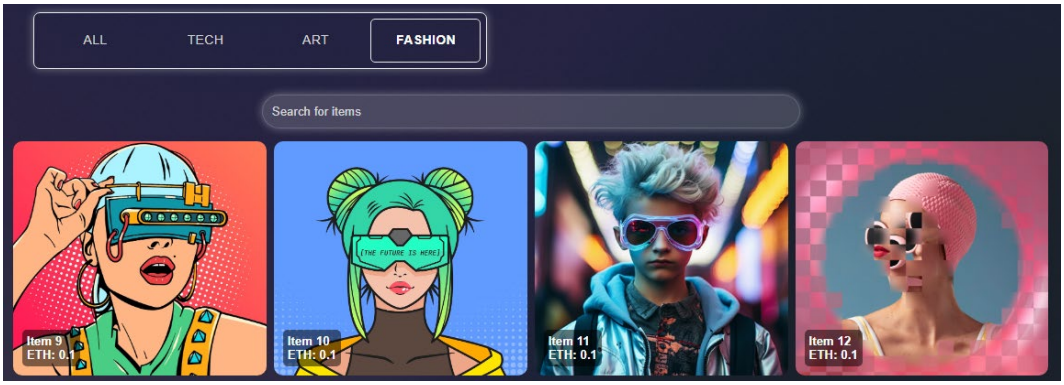

Figure 5.5: Filter Items

Users can use the filter tab above to categorize the items. The filter tabs can display different items depending on the selection. The selected tab will have a white border around it, showing the user which filter is being applied.

3) Items Search

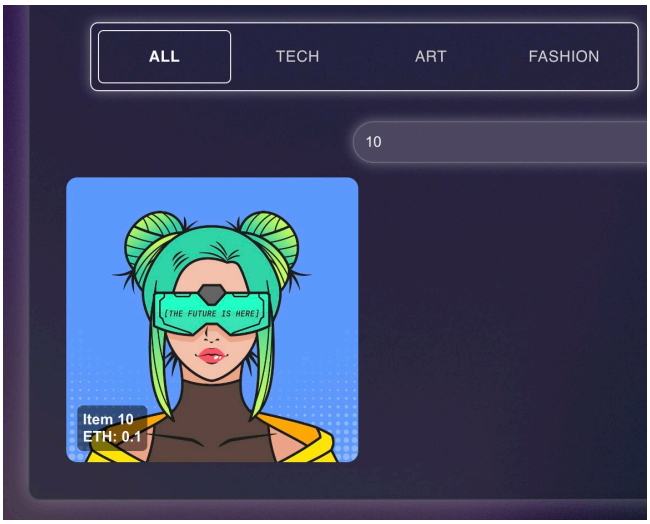

Figure 5.6: Search Item

We also added the search functionality, which allows users to search for any items available for sale below. Users can input into the search bar, and the assets will be filtered to match users' input in real time.

4) Items Buy

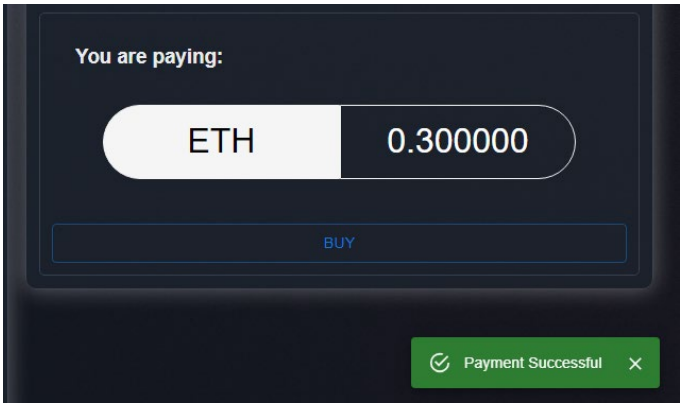

Figure 5.7: Payment

The buy function currently shows a confirmation of the payment. Users can click on the Buy button to finalize their cart. Upon successful purchase, a pop-up message will be displayed at the bottom right of the screen to show that the payment is successful.

5) Responsiveness

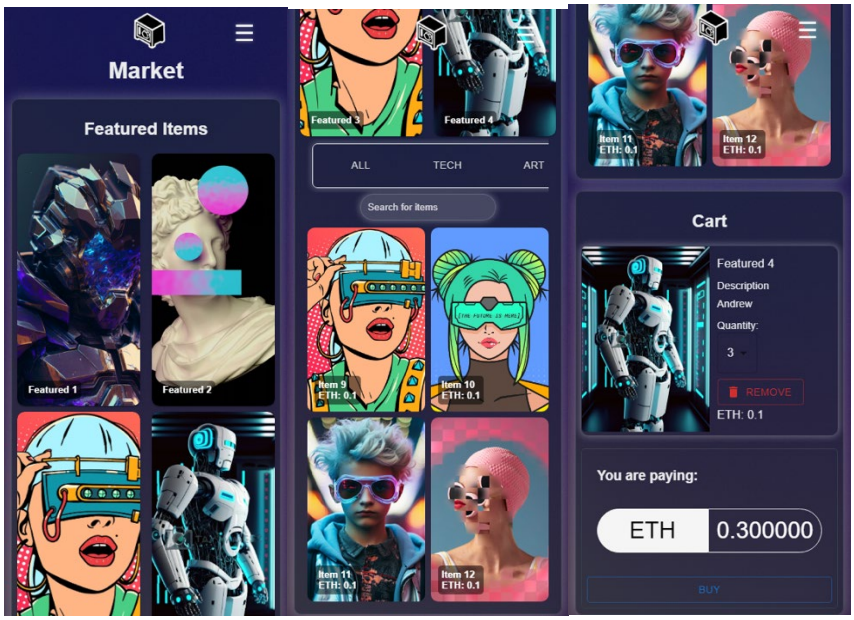

Figure 5.8: Responsive Market page

The Market component is designed to be responsive and adapt to different screen sizes. The responsiveness is achieved using Material-UI's useMediaQuery hook and the theme.breakpoints.down("sm") and external CSS (market.css) methods. Depending on the screen size, certain styles, and layouts are adjusted to provide an optimal viewing experience, which offers an excellent user experience.

6) Login to buy

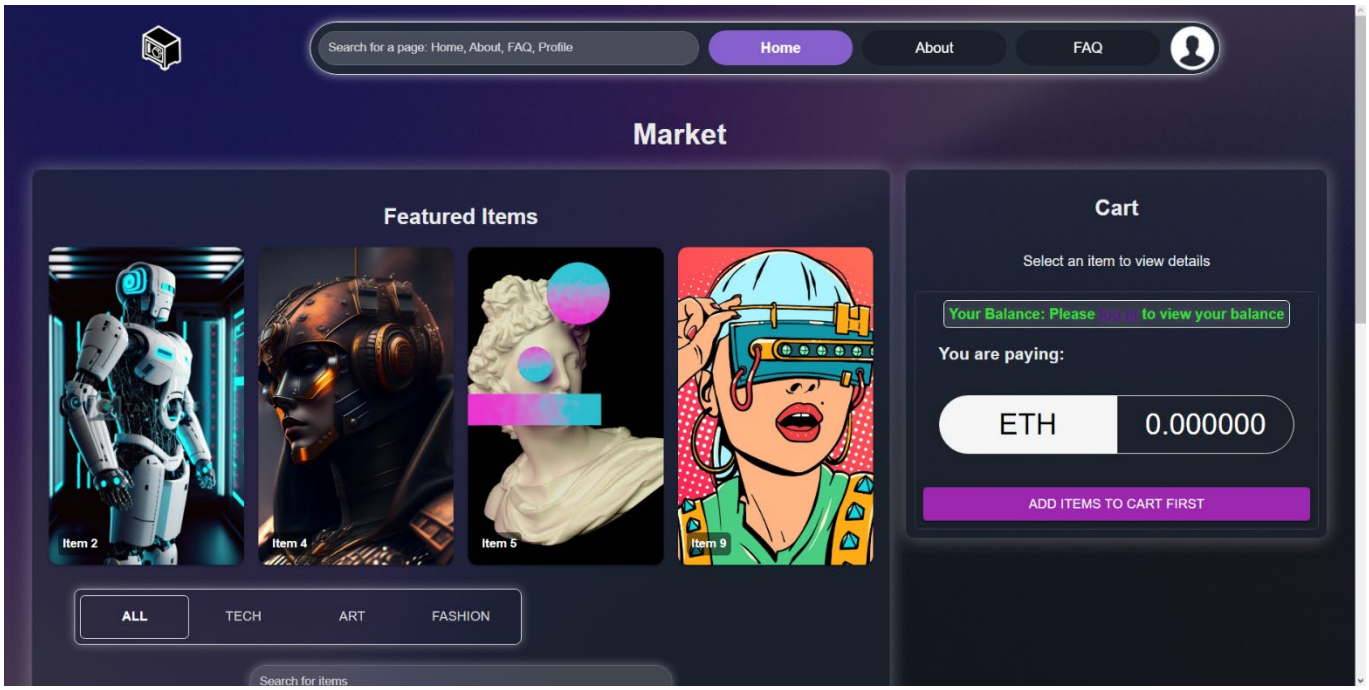

Figure 5.9: Login Check

Figure 5.10: Login Alert

We also implemented a login check in the market page and only allow users to buy when they are logged in. If users are not logged in, it will alert them to log in and redirect them to the login page once they click 'OK' on the alert.

7) Error Handling

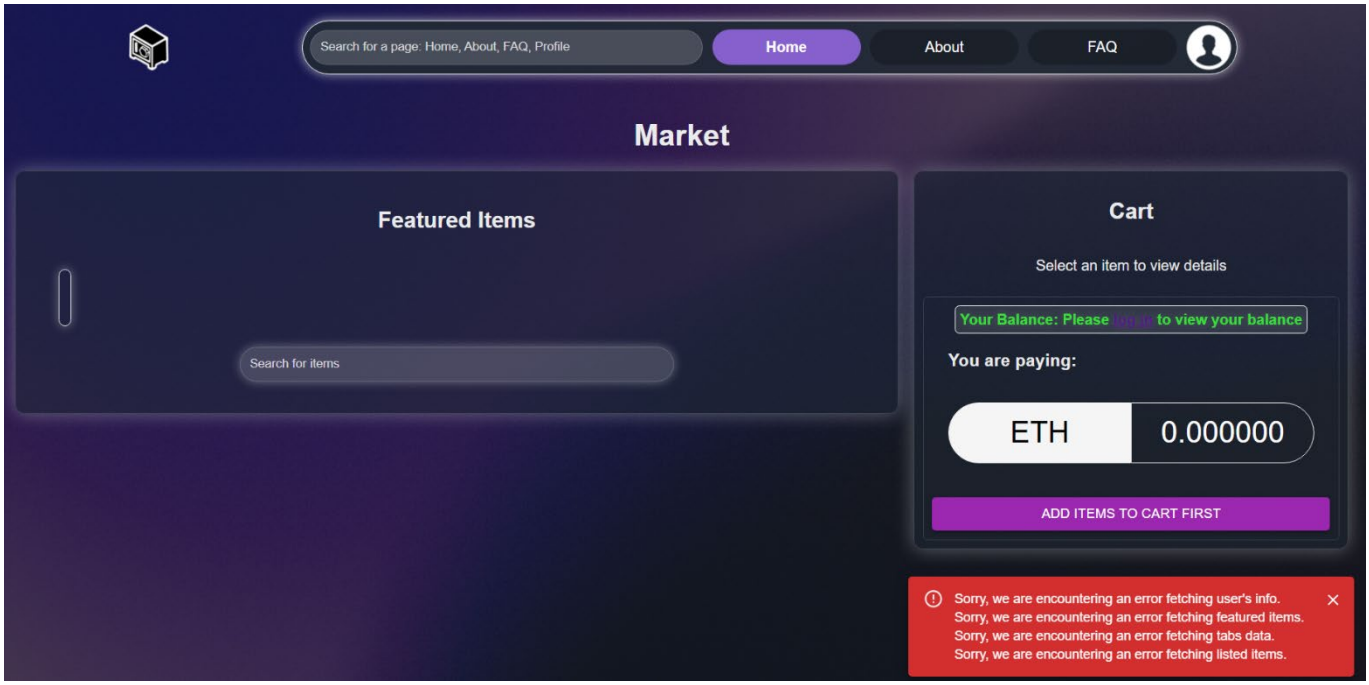

figure 5.11: Market page Error Messages

```
<Snackbar
  open={errorSnackbarOpen}
  autoHideDuration={6000}
  onClose={handleErrorSnackbarClose}
  anchorOrigin={{ vertical: "bottom", horizontal: "right" }}
>
  {/* Display all error messages */}
  <Alert onClose={handleErrorSnackbarClose} severity="error">
    {Array.isArray(errorMessages) && errorMessages.map((message, index) => (
      <div key={index}>{message}</div>
    ))}
  </Alert>
</Snackbar>
```

Figure 5.12: Snackbar usage

This web page uses the Snackbar component from MaterialUI to show all the user-friendly error messages that the page encounters when it cannot fetch the data from the backend to display the data onto the frontend.

About

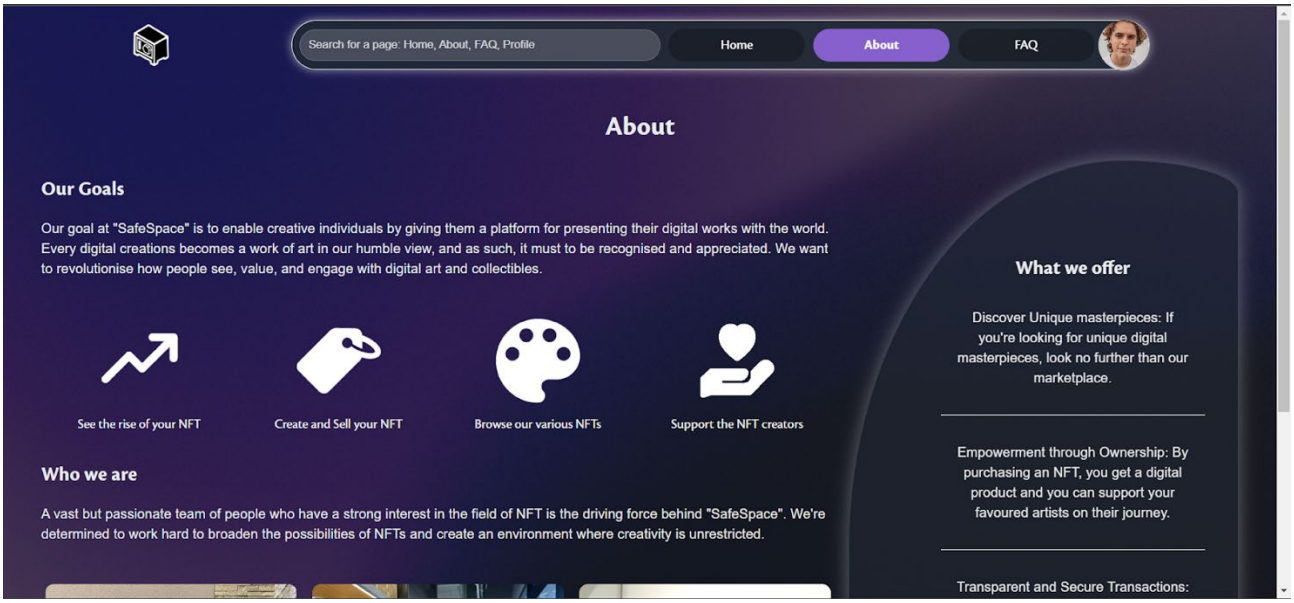

Figure 6.0: About page

For people who are curious about the motivations behind SafeSpace, the About page outlines our goals on the website. Additionally, it introduces the members who made SafeSpace and shows the services offered by the website.

Functionalities:

This page utilizes the MaterialUI components such as Grid, Typography, CardMedia and Divider.

1. **Responsive**

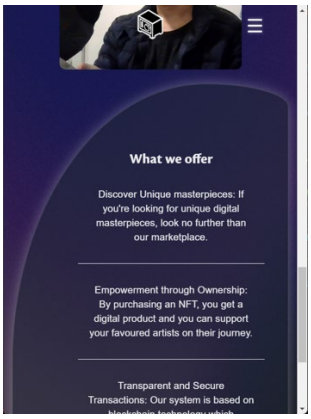

Figure 6.1: Responsive About Page

The screen size of the About page was designed to be responsive by utilising Grid xs and md from Material-UI. If there is enough room, the images would be arranged horizontally, otherwise, they would be arranged vertically. If there is not enough room, the section to the right labelled "What we offer" will be shifted to the bottom of the two sections to the left.

2. Error Handling

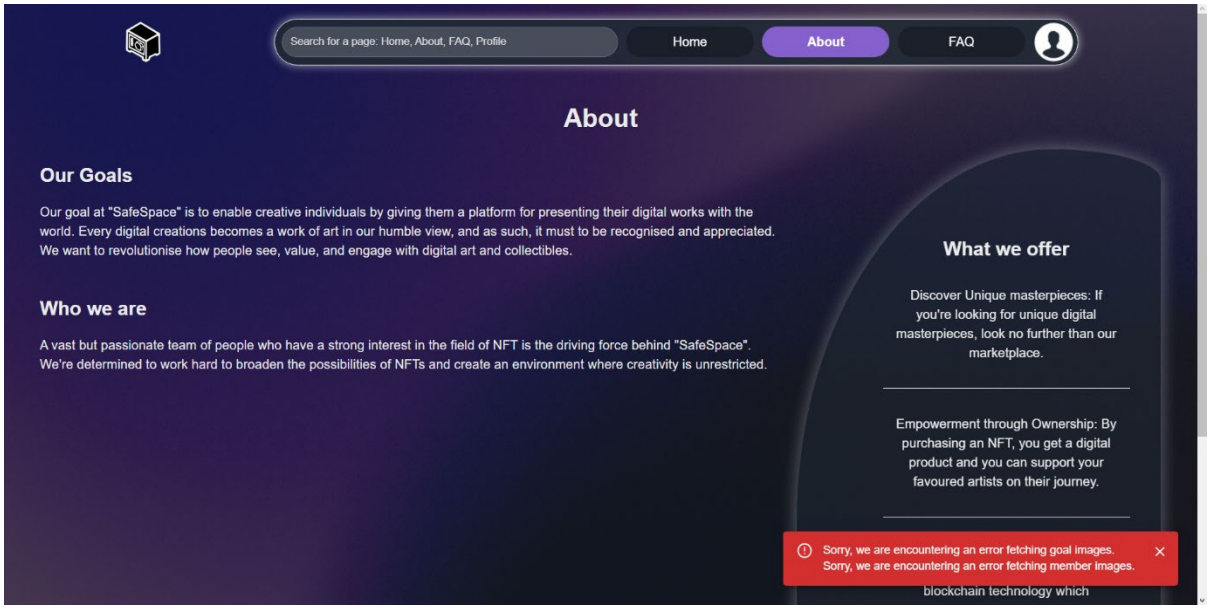

Figure 6.2: About Error Messages

```
<Snackbar
    open={errorSnackbarOpen}
    autoHideDuration={6000}
    onClose={handleErrorSnackbarClose}
    anchorOrigin={{ vertical: "bottom", horizontal: "right" }}
>
    {/* Display all error messages */}
    <Alert onClose={handleErrorSnackbarClose} severity="error">
        {Array.isArray(errorMessages) && errorMessages.map((message, index) => (
            <div key={index}>{message}</div>
        ))}
    </Alert>
</Snackbar>
```

Figure 6.3: Snackbar usage

Error messages will be displayed using a Snackbar when the page has problem fetching the required data from the backend.

FAQ

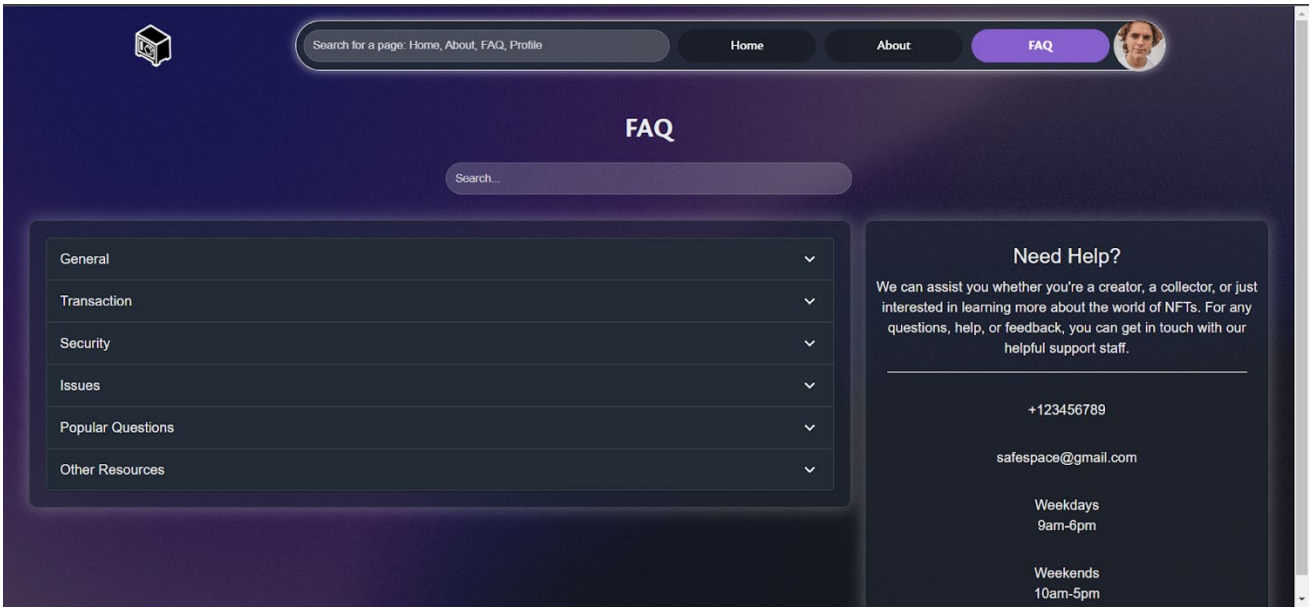

Figure 7.0

Visit the FAQ page to get your questions answered, it provides people easy access to solutions for common questions. Additionally, it provides the phone number and email to contact the support staff and shows the working hours.

Functionalities:

This page utilizes the MaterialUI components, and icons such as Grid, Typography, Accordion, AccordionSummary, AccordionDetails and Divider.

1. **Search**

Figure 7.1: FAQ search bar

The user can use the search bar in FAQ page to find questions. The search bar will be filter out the questions based on the input text.

2. **Dropdown (Accordion)**

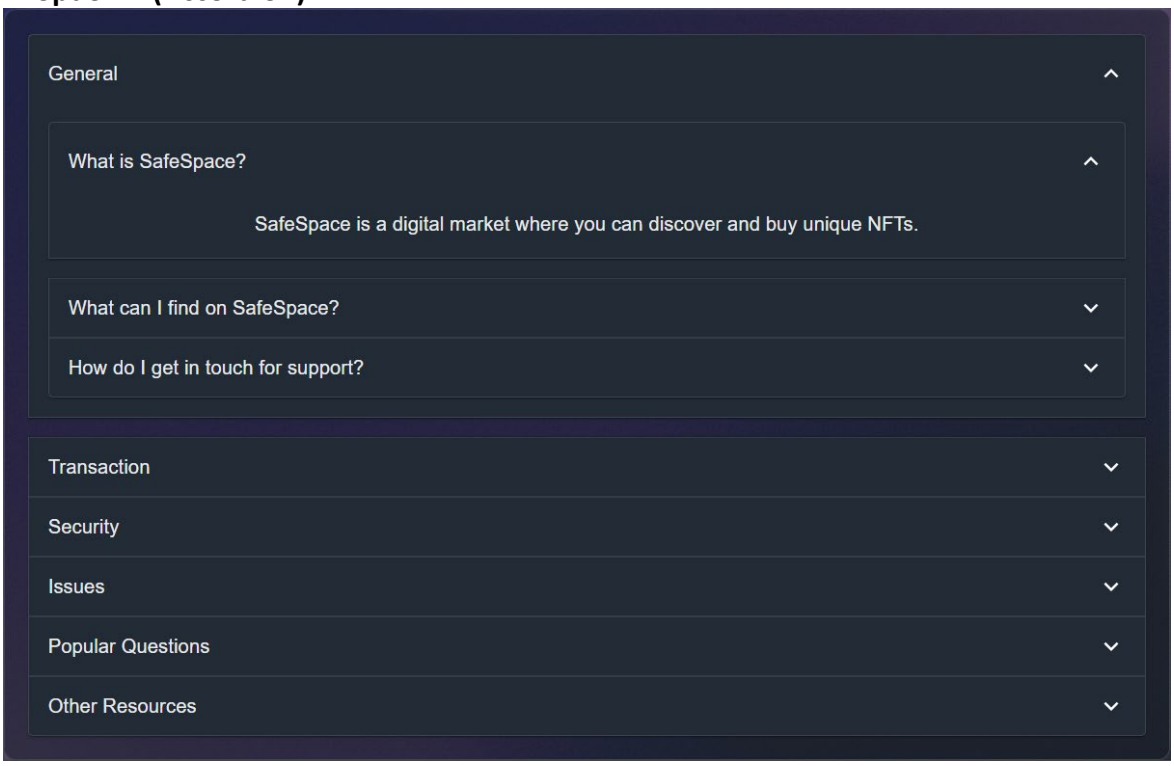

Figure 7.2: Dropdown Menu

The user can click on the dropdown menus to display the questions for the specific section. Then they can click on the question dropdown menus to display the solutions for that question.

3. **Responsiveness**

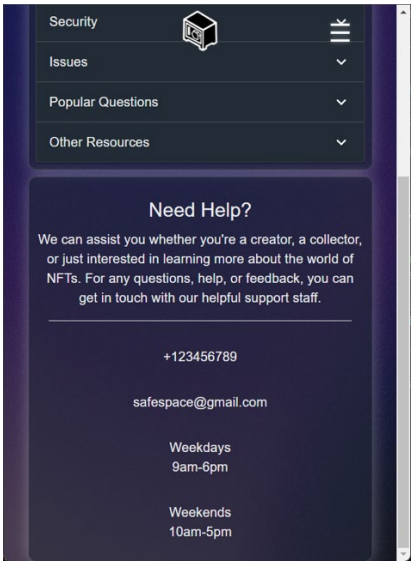

Figure 7.3: Responsive FAQ page

The screen size of the FAQ page was designed to be responsive by utilising Grid xs and md from Material-UI. If there is not enough room, the section to the right labelled "Need Help?" will be shifted to the bottom of the dropdown menu section.

4. Error Handling

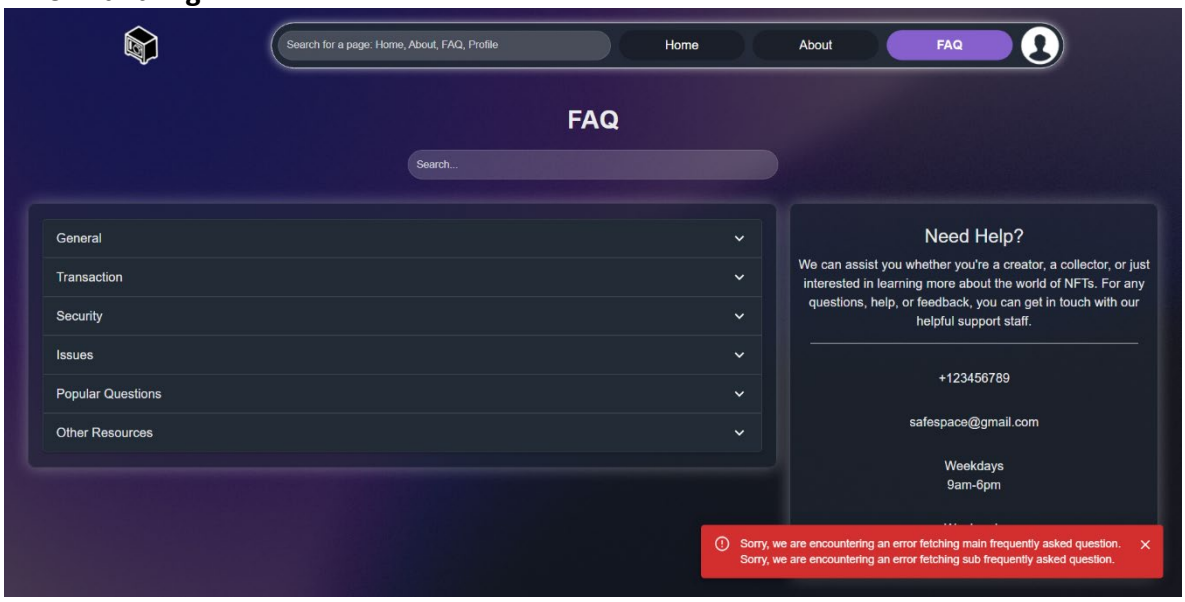

Figure 7.4: FAQ error messages

```
<Snackbar
    open={errorSnackbarOpen}
    autoHideDuration={6000}
    onClose={handleErrorSnackbarClose}
    anchorOrigin={{ vertical: "bottom", horizontal: "right" }}
>
    {/* Display all error messages */}
    <Alert onClose={handleErrorSnackbarClose} severity="error">
        {Array.isArray(errorMessages) && errorMessages.map((message, index) => (
            <div key={index}>{message}</div>
        ))}
    </Alert>
</Snackbar>
```

Figure 7.5: Snackbar usage

Error messages will be displayed using a Snackbar when the page has problem fetching the required data from the backend.

V. /Backend Database Design

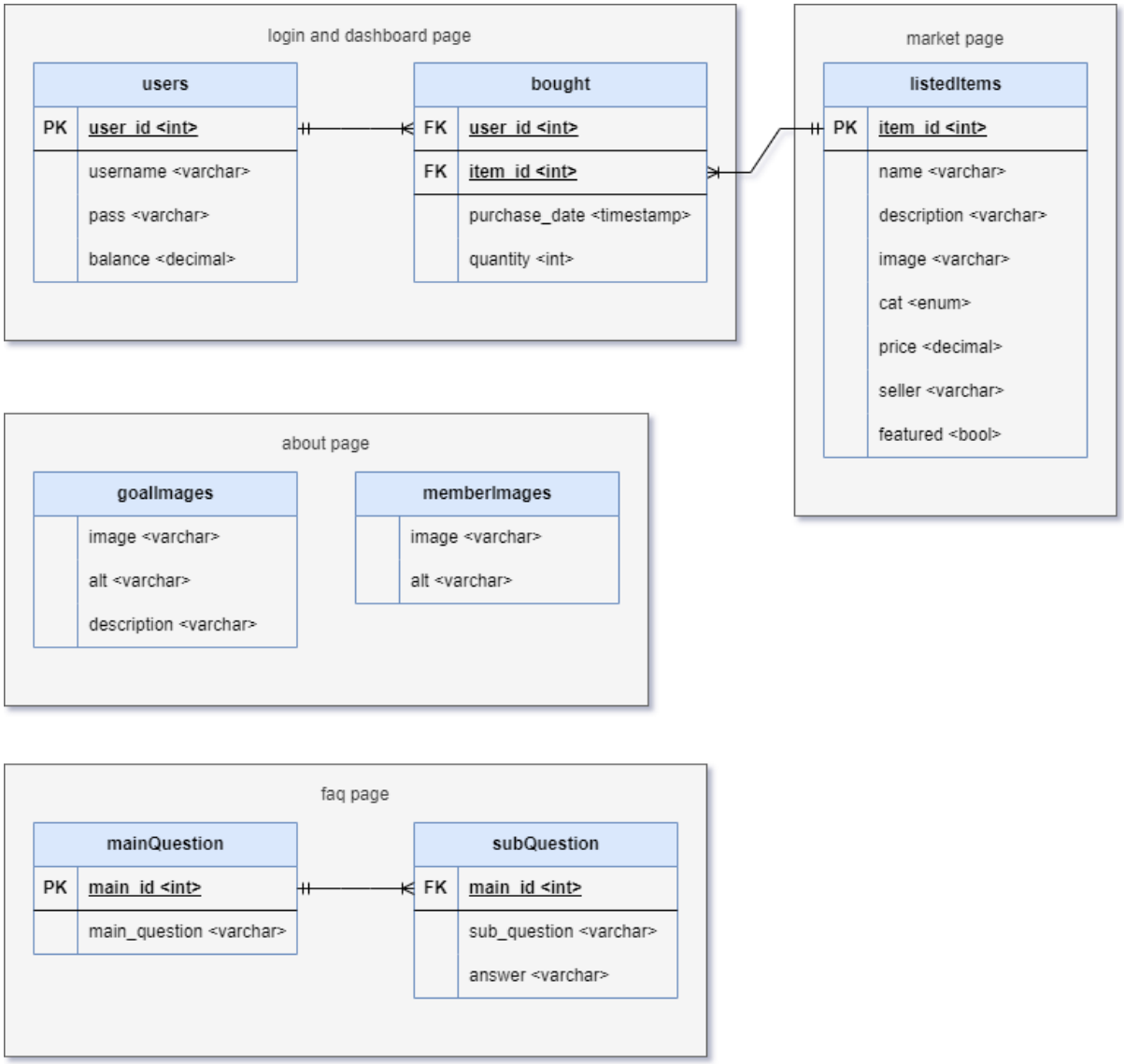

Figure 8.0: Database Design

Description and Justification:
Figure 8.0 shows the database design for our web application, and it is designed to support our digital marketplace for Art. It consists of multiple tables and relationships that serve different purposes. This database is built with robustness, scalability, and data integrity in mind. This database is designed to ensure efficient data retrieval in the backend, and it uses indexing, foreign keys, and unique constraints to maintain data integrity. Normalization has been done on these tables as well to avoid data redundancy. This design can help improve the overall web application's performance to ensure a user-friendly experience.

listItems: Stores data about the Art listings. It is also used in the frontend to display the Art.
goalImages and **memberImages:** Used in frontend About page to display images and descriptions.
mainQuestins and **subQuestion**: Used on the FAQ page to display questions and their corresponding sub-questions.
users: used to store user data such as passwords, usernames, and balance.
bought: record items bought by users using item_id, user_id, purchase_date, and quantity.

/FastAPI Design

TabsData

API name and description:

tabsData: Return the name of the tabs (item categories)

Endpoints:

Url: http://127.0.0.1:8000/tabsData

Method: Get

Request Parameters:

None

Response Format:


```
{
  "status": "200 OK",
  "content":
            {
              "value": "All",
              "label": "All"
            },
            {
              "value": "Tech",
              "label": "Tech"
            },
            {
              "value": "Art",
              "label": "Art"
            },
            {
              "value": "Fashion",
              "label": "Fashion"
            }
}
```

GetFeaturedItems

API name and description:

getFeaturedItems: Return the items that are featured in market page.

Endpoints:

Url: http://127.0.0.1:8000/getFeaturedItems

Method: Get

Request Parameters:

None

Response Format:


```
{
 "status": "200 OK",
 "content":
        {
          {
          "item_id": 2,
          "name": "Item 2",
          "description": "Item 2 Description",
          "image": "https://i.ibb.co/kJ23Wbb/tech2.jpg",
          "cat": "Tech",
          "price": 0.123456789,
          "seller": "Andrew",
          "featured": 1
         },
         {
          "item_id": 4,
          "name": "Item 4",
          "description": "Item 4 Description",
          "image": "https://i.ibb.co/XyFRTBY/tech4.jpg",
          "cat": "Tech",
          "price": 0.123456789,
          "seller": "Andrew",
          "featured": 1
        }
}
```


Note: This is a shortened version of the response

GetListedItems

API name and description:

getListedItems: Return all the items listed in market page.

Endpoints:

Url: http://127.0.0.1:8000/getListedItems

Method: Get

Request Parameters:

None

Response Format:


```
{
  "status": "200 OK",
  "content":
        {
          "item_id": 1,
          "name": "Item 1",
          "description": "Item 1 Description",
          "image": "https://i.ibb.co/r58jQcq/tech1.jpg",
          "cat": "Tech",
          "price": 0.123456789,
          "seller": "Andrew",
          "featured": 0
        },
        {
          "item_id": 2,
          "name": "Item 2",
          "description": "Item 2 Description",
          "image": "https://i.ibb.co/kJ23Wbb/tech2.jpg",
          "cat": "Tech",
          "price": 0.123456789,
          "seller": "Andrew",
          "featured": 1
        }
}
```


Note: This is a shortened version of the response

GetGoalImages

API name and description:

getGoalImages: Return the image link and description of team goals for the about page.

Endpoints:

Url: http://127.0.0.1:8000/getGoalImages

Method: Get

Request Parameters:

None

Response Format:


```
{
  "status": "200 OK",
  "content":
        {
          "image": "https://i.ibb.co/PQxFkMV/trend.png",
          "alt": "trend",
          "description": "See the rise of your NFT"
        },
        {
          "image": "https://i.ibb.co/5vVP1yn/support.png",
          "alt": "tag",
          "description": "Create and Sell your NFT"
        },
        {
          "image": "https://i.ibb.co/m0qyQr3/tag.png",
          "alt": "palette",
          "description": "Browse our various NFTs"
        },
        {
          "image": "https://i.ibb.co/Xt1dCNZ/palette.png",
          "alt": "support",
          "description": "Support the NFT creators"
        }
}
```

GetMemberImages

API name and description:

getMemberImages: Return the image link of team members for the about page.

Endpoints:

Url: http://127.0.0.1:8000/getMemberImages

Method: Get

Request Parameters:

None

Response Format:


```
{
  "status": "200 OK",
  "content":
        {
          "image": "https://i.ibb.co/KLcTHw2/Tom.jpg",
          "alt": "Tom"
        },
        {
          "image": "https://i.ibb.co/M6NcG4c/Andrew.jpg",
          "alt": "Andrew"
        },
        {
          "image": "https://i.ibb.co/rf6XcSF/Yuan.jpg",
          "alt": "Yuan"
        }
}
```

GetMainQuestion

API name and description:

getMainQuestion: Return the FAQ question categories in the FAQ page.

Endpoints:

Url: http://127.0.0.1:8000/getMainQuestion

Method: Get

Request Parameters:

None

Response Format:


```
{
 "status": "200 OK",
 "content":
        {
         "main_question": "General"
        },
        {
         "main_question": "Transaction"
        },
        {
         "main_question": "Security"
        },
        {
         "main_question": "Issues"
        },
        {
         "main_question": "Popular Questions"
        },
        {
         "main_question": "Other Resources"
        }
}
```

GetSubQuestion

API name and description:

getSubQuestion: Return the FAQ questions and answers for the FAQ page.

Endpoints:

Url: http://127.0.0.1:8000/getSubQuestion

Method: Get

Request Parameters:

None

Response Format:


```
{
 "status": "200 OK",
 "content":
        {
         "main_id": 1,
         "sub_question": "What is SafeSpace?",
         "answer": "SafeSpace is a digital market where you can discover and buy unique NFTs."
        },
        {
         "main_id": 1,
         "sub_question": "What can I find on SafeSpace?",
         "answer": "You can find various types of NFT made by artists and creators from around the world."
        },
        {
         "main_id": 1,
         "sub_question": "How do I get in touch for support?",
         "answer": "If you have any inquiries, you can contact our support staff through phone +123456789 or email safespace@gmail.com."
        },
        {
         "main_id": 2,
         "sub_question": "How do I buy an NFT?",
         "answer": "To buy an NFT, you have to have an account, then browse on the marketplace, click on the NFT and select buy."
        }
```


Note: This is a shortened version of the response

GetUser

API name and description:

getUser: Return the users data.

Endpoints:

Url: http://127.0.0.1:8000/getUser

Method: Get

Request Parameters:

None

Response Format:


```
{
 "status": "200 OK",
 "content":
        {
         "user_id": 1,
         "username": "user1",
         "pass": "pass1",
         "balance": 100
        },
        {
         "user_id": 2,
         "username": "user2",
         "pass": "pass2",
         "balance": 50
        },
        {
         "user_id": 3,
         "username": "user3",
         "pass": "pass3",
         "balance": 40
        }
}
```

Purchase item by category

API name and description:

api/user/{user_id}/puchase_by_category: Return the items bought by user

Endpoints:

Url: http://127.0.0.1:8000/api/user/1/purchases_by_category

Method: Get

Request Parameters:

```
{
  "user_id": 1
}
```

Response Format:

```
{
  "status": "200 OK",
  "content":
            {
            "Tech": 5,
            "Art": 0,
            "Fashion": 0
           }
}
```

DeployContract

API name and description:

deployContract: Deploying solidity smart contract.

Endpoints:

Url: http://127.0.0.1:8000/deployContract

Method: Get

Request Parameters:

None – contract deployed using file `SmartContract.sol`

Response Format:

```
{
  "status": "200 OK",
  "content":
           {
             "Smart Contract deployed":"0xee390B3fA2224fDE5999F0f4e74d7Dd6f66868B9"
           }
}
```

BuyItems

API name and description:

buyItems: buy item/s

Endpoints:

Url: http://127.0.0.1:8000/buyItem

Method: Post

Request Parameters:


```
[
  {
    "item_id": 1,
    "quantity": 2,
    "price": 0.1
  }
]
```


Response Format:


```
{
"status": "200 OK",
  "results": [
    {
      "status": "Item purchased",
      "transaction_receipt": {
        "transactionHash":
"0x437d0d103160b8f1185ef35b0ace32d8c23c40b5de90e35c62ead96bdaa7e10c",
        "transactionIndex": 0,
        "blockNumber": 15,
        "blockHash":
"0xf5fd65af15122686b44c703b71eff77362b7b7e0f997803c92a059200aa276b1",
        "from": "0x192865D77ED4B663a48A066Cae4480DCecFb3696",
        "to": "0xCf052A2cC78854Fbd2D15B00B2d573FFCbD7B4e7",
        "cumulativeGasUsed": 125073,
        "gasUsed": 125073,
        "status": 1
      },
      "transaction_hash":
"0x437d0d103160b8f1185ef35b0ace32d8c23c40b5de90e35c62ead96bdaa7e10c"
    }
  ]
}
```

UpdateUserBalance

API name and description:

updateUserBalance: Update the user's balance in MySQL, takes a user_id and new_balance as input.

Endpoints:

Url: http://127.0.0.1:8000/updateUserBalance

Method: Post

Request Parameters:

```
{
  "user_id": 1,
  "new_balance": 100
}
```

Response Format:

```
{
  "status": "200 OK",
  "message": "User balance updated successfully"
}
```

UpdateUserAssets

API name and description:

updateUserAssets: Update the user's assets in MySQL database.

Endpoints:

Url: http://127.0.0.1:8000/updateUserAssets

Method: Post

Request Parameters:

```
{
  "user_id": 1,
  "item_id": 1,
  "quantity": 1
}
```

Response Format:

```
{
  "status": "200 OK",
  "message": "User assets updated successfully"
}
```

GetTransactionHistory

API name and description:

getTransactionHistory: retrieve the transaction made by a specific user. Using 'bought' and 'listedItems' tables to get data on transaction information.

Endpoints:

Url: http://127.0.0.1:8000/getTransactionHistory/{user_id}
Method: Post

Request Parameters:


```
{
  "user_id": 1
}
```


Response Format:


```
{
  "status": "200 OK",
  "details":
        [
          {
            "date": "2023-09-26T18:59:42",
            "description": "Item 1",
            "quantity": 1
          },
          {
            "date": "2023-09-26T18:59:42",
            "description": "Item 2",
            "quantity": 4
          }
        ]
}
```

Login

API name and description:

login: Login to account

Endpoints:

Url: http://127.0.0.1:8000/login

Method: Post

Request Parameters:

```
{
  "username": "user1",
  "password": "pass1"
}
```

Response Format:

```
{
  "status": "success",
  "message": "User verified",
  "user_id": 1
}
```

getItemsForUser/{user_id}

API name and description:

getItemsForUser/{user_id}: retrieve bought items for a specific user. Uses 'listedItems' and 'bought' table to retrieve data from MySQL about purchases.

Endpoints:

Url: http://127.0.0.1:8000/getItemsForUser/1

Method: Get

Request Parameters:


```
{
  "user_id": 1
}
```


Response Format:


```
{
  "status": "200 OK",
  "content":
        [
            {
              "item_id": 1,
              "name": "Item 1",
              "description": "Item 1 Description",
              "image": "https://i.ibb.co/r58jQcq/tech1.jpg",
              "cat": "Tech",
              "price": 0.123456789,
              "seller": "Andrew",
              "featured": 0
            },
            {
              "item_id": 2,
              "name": "Item 2",
              "description": "Item 2 Description",
              "image": "https://i.ibb.co/kJ23Wbb/tech2.jpg",
              "cat": "Tech",
              "price": 0.123456789,
              "seller": "Andrew",
              "featured": 1
            }
        ]
}
```

/Functionalities Description

This part of the document will showcase the backend functionalities implemented in the frontend for assignment 2 and their use cases.

/User Dashboard

Purpose:

The User Dashboard provides user a central hub to manage and view their assets, track transactions, and gain some insights into their activity in the marketplace. This page is designed to offer users an overview of their interactions within the platform while maintaining a user-friendly and pretty intuitive interface. The dashboard achieves this by having simple but comprehensive donut chart on the assets categories and quantity as well as different windows that users can use to view different aspects of their accounts.

Use Cases:

1.Accessing the User Dashboard:

Step 1: Users need to login using their credentials that are saved in the database table. After successful authentication, they are directed to their personalized dashboard. The login menu could be access by clicking the user icon in the navigation bar or the log in in the market page. After the user logs in, the contract will be deployed automatically, and the user can start buying items. This can be seen in Figure 9.0 and 9.1.

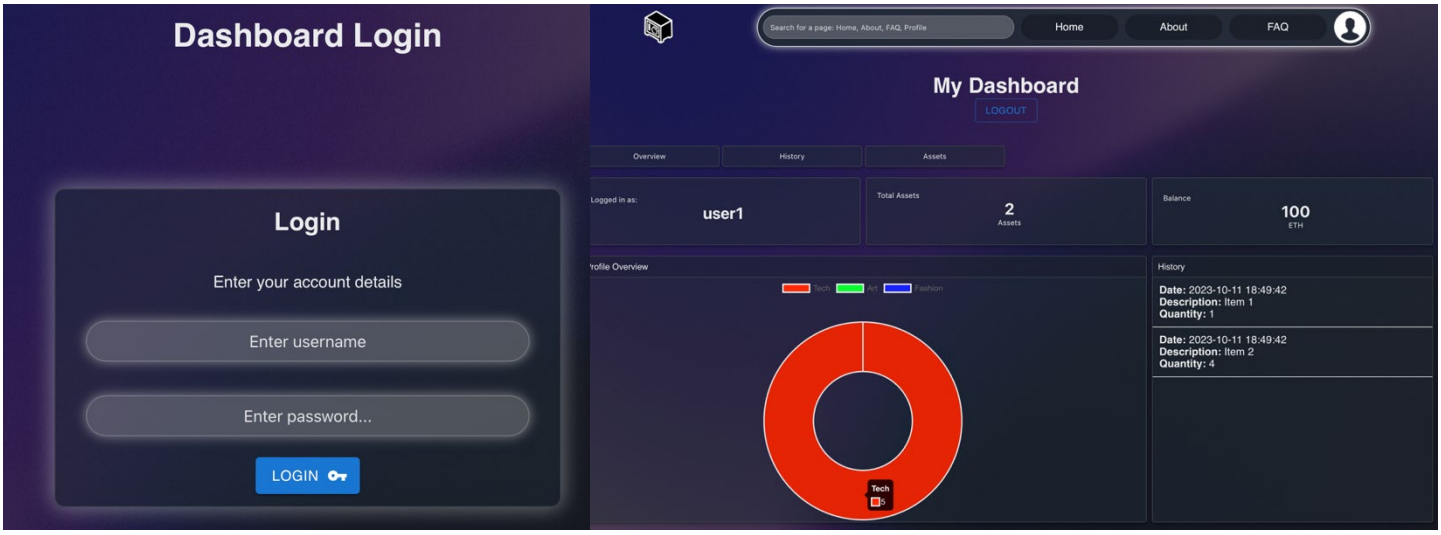

Figure 9.0: Login Page to Dashboard

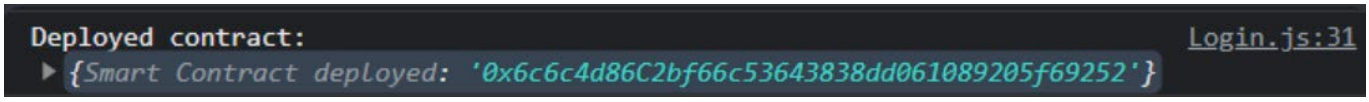

Figure 9.1: Deploy Contract confirmation

2.Viewing Assets:

Step 1: Inside the dashboard page, users can find a section called “Assets”. Clicking on the tab will display a list of assets that users own.

Step 2: Each asset will contain its name and price in Eth

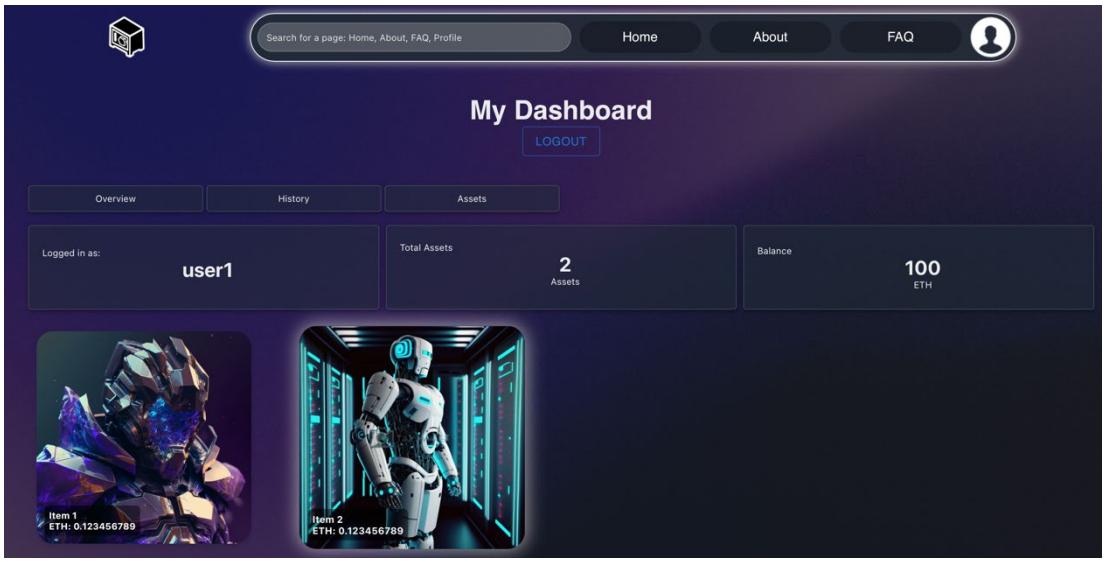

Figure 9.2: Assets tab

3.Monitoring Asset Graph:

Step 1: On the dashboard, there will be a donut chart showcasing the distribution of user’s assets across art, fashion, and tech. This can be viewed in Figure 9.

Step 2: Users can hover over the donut chart to check the quantity of the item assets and they can eliminate the assets that they don’t want to view for a more compact donut chart. This can be seen in Figure 9.2

Step 3: As users buy more items the graph will change accordingly because it retrieves data from the backend. This can be seen in Figure 9.4.

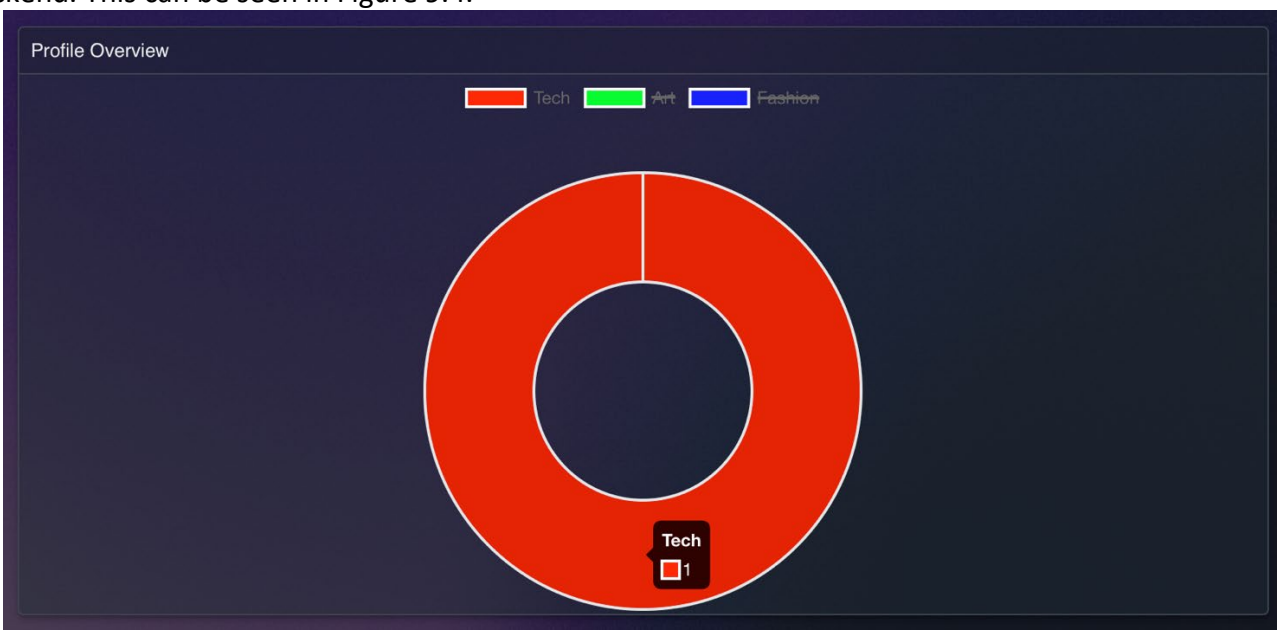

Figure 9.3: Donut Chart

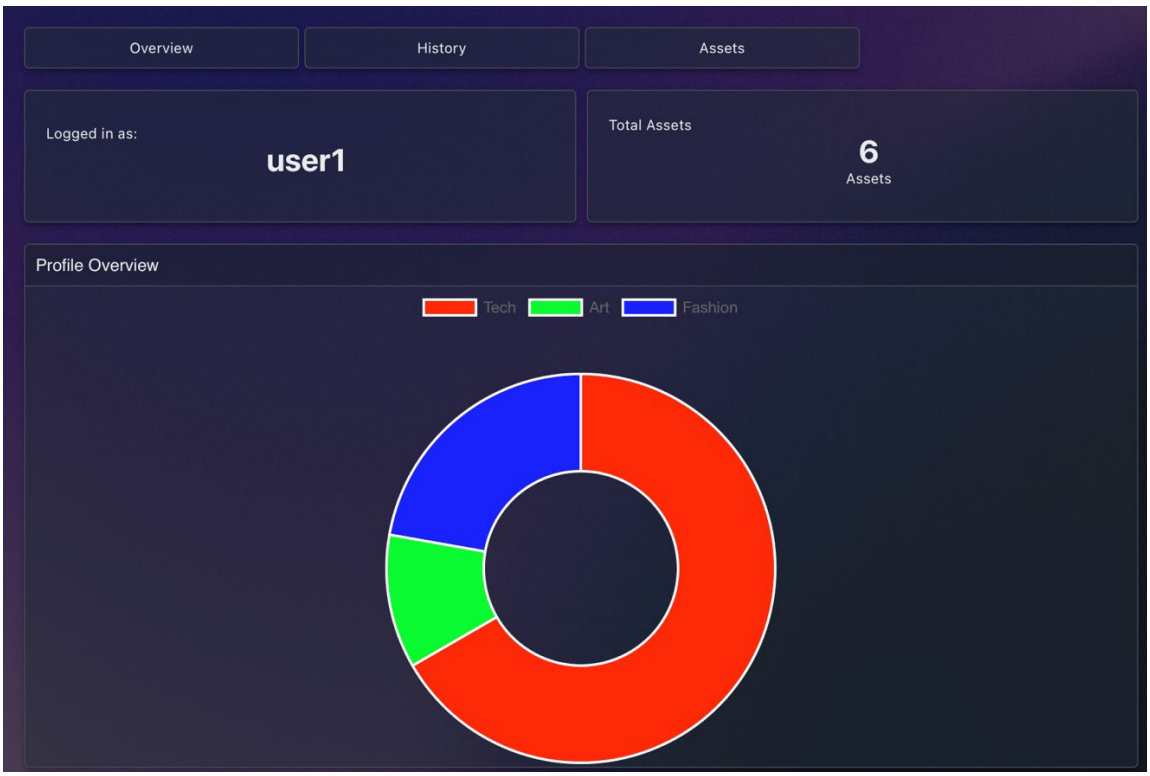

Figure 9.4: Updated Donut chart as buyer buys more items

4.Reviewing Transaction History:

Step 1: Users can view 5 most recent transactions on the right of the dashboard with the full date and time

Step 2: Users can expand this section by clicking on the history tab and view the full transaction history of the account.

Step 3: This transaction will be updated as the user buys more items. This can be seen in Figure 9.6 below

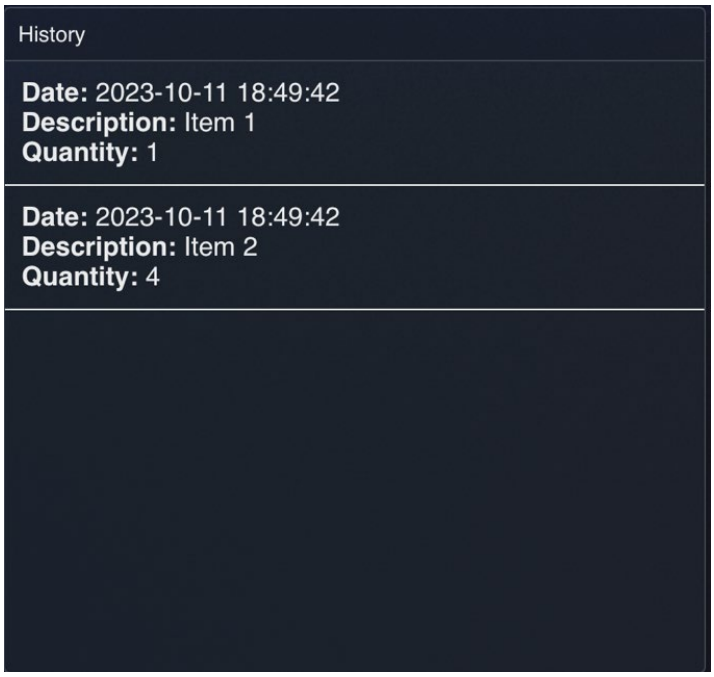

Figure 9.5: Compact history transaction

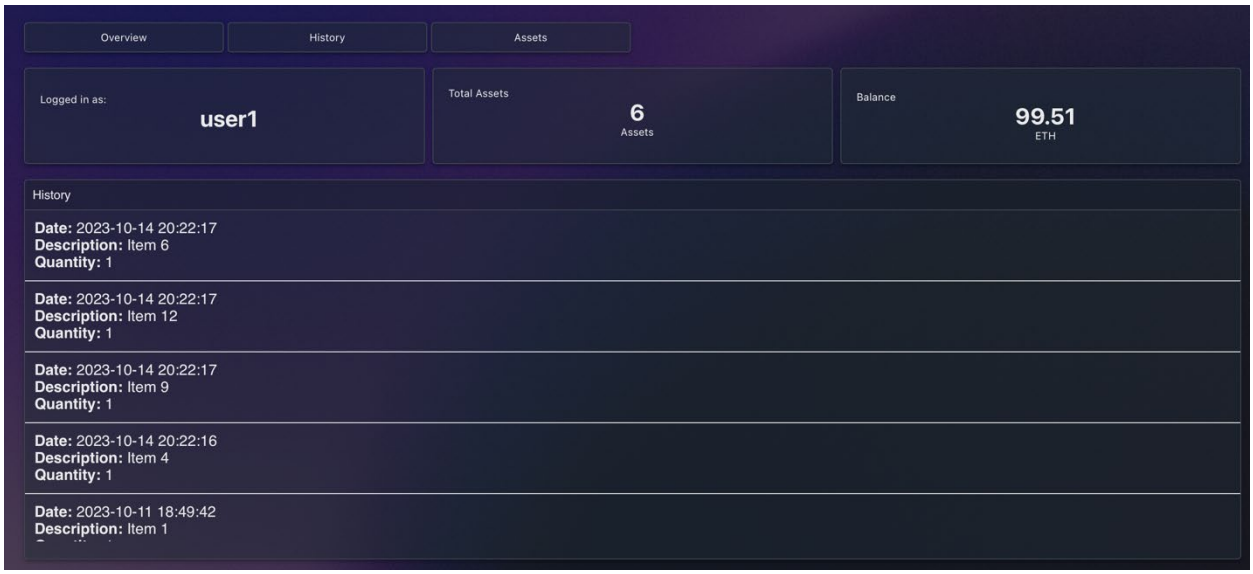

Figure 9.6: Full scrollable history transaction

/Market

Purpose:

The main functionality of the market page is to display the items that are available for the users to buy and to interact with the smart contract when buying the selected item(s). In section IV. /Project Design under /Website Design for Home/Market page, we discussed the item category filter, calculate total, cart functionalities that we implemented in the frontend for assignment 1.

This page servers as the platform where users can browse, select, filter, and purchase items using Smart Contract which aims to provide a secure and seamless transaction for the user.

Use Cases:

1. **Fetch data from backend**

   We created a fetchApiData function to fetch and set the data to the corresponding variables in our React page which we can utilize to perform other functions. This helps to eliminate redundancy of repeating code. Not only will this function get the response and set the data, but it also checks for errors and logs it.

```
//Function to Fetch data from backend and handle errors
const fetchApiData = async (url, setData, errorMessage) => {
  let tempErrorMessage = '';
  let errorLogged = false;

  try {
    const response = await axios.get(url); //fetch data from url
    if ('error' in response.data) {
      if (!errorLogged) {
        tempErrorMessage += errorMessage; //append error message - RESPOND error
        errorLogged = true; //only log error once
      }
      console.log(response.data.error);
    } else if (Array.isArray(response.data) && response.data.length === 0) { //add error message if no data
      tempErrorMessage += 'No items available. ';
    } else {
      setData(response.data); //set data if no error
    }
  } catch (error) {
    if (!errorLogged) {
      tempErrorMessage += errorMessage; //append error message - FECTHING error
      errorLogged = true;
    }
  }

  return tempErrorMessage;
};
```

Figure 10.0: fetchApiData function

```
//Call to backend to fetch data
useEffect(() => {
  const fetchData = async () => {
    let allErrorMessages = [];

    const userError = await fetchApiData(
      'http://127.0.0.1:8000/getUser/', //api url
      setUser, //setData
      'Sorry, we are encountering an error fetching user\'s info. ' //errorMessage
    );
    if (userError) allErrorMessages.push(userError)

    const featuredItemsError = await fetchApiData(
      'http://127.0.0.1:8000/getFeaturedItems/',
      setFeaturedItems,
      'Sorry, we are encountering an error fetching featured items. '
    );
    if (featuredItemsError) allErrorMessages.push(featuredItemsError);

    const tabsDataError = await fetchApiData(
      'http://127.0.0.1:8000/tabsData',
      setTabs,
      'Sorry, we are encountering an error fetching tabs data. '
    );
    if (tabsDataError) allErrorMessages.push(tabsDataError);

    const listedItemsError = await fetchApiData(
      'http://127.0.0.1:8000/getListedItems/',
      setListedItems,
      'Sorry, we are encountering an error fetching listed items. '
    );
```

Figure 10.1: fetch data

Steps on how the function is used:

1) first, we passed in the API endpoint as the first parameter.
2) Add the setter for the variable.
3) Then the error message we want to display if there were errors.

The error messages will then be displayed using the Snackbar.

2. Buy item

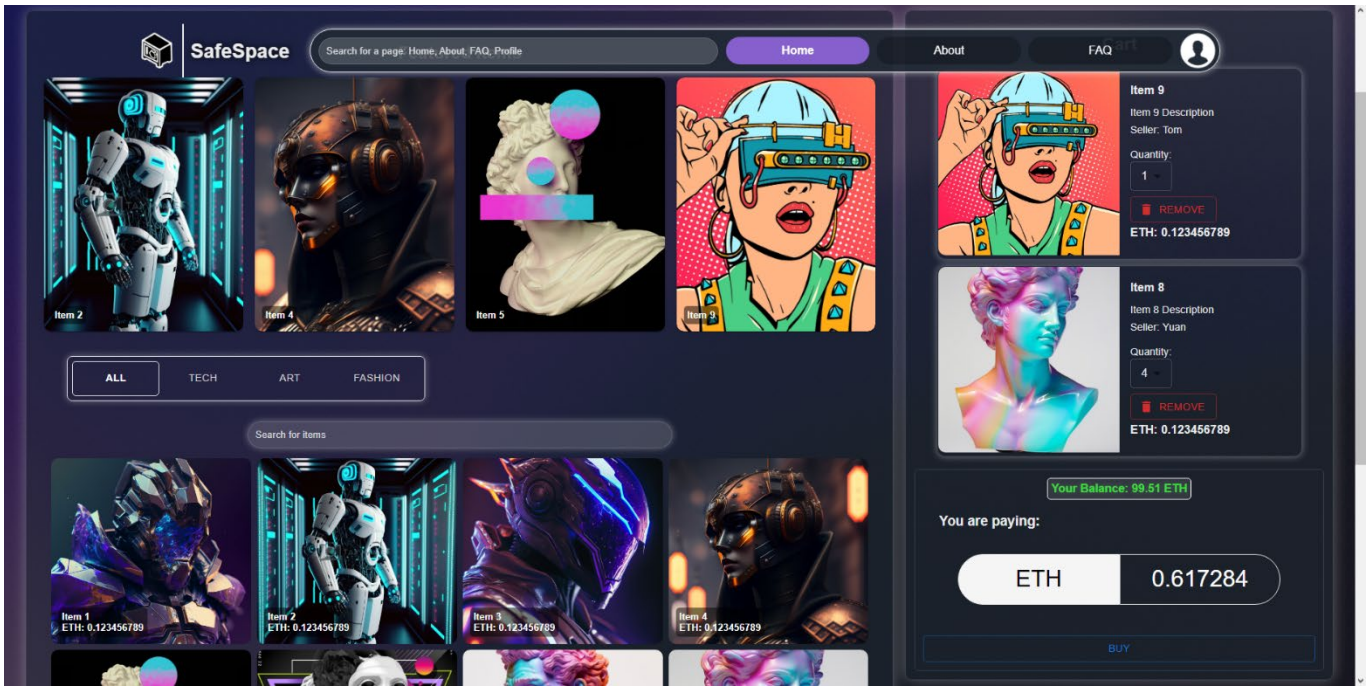

Figure 10.2: Items displayed.

All the items displayed in the marketplace utilizes the data fetched in Figure: 10.2 from our MySQL database through FastAPI to dynamically display them in the market page.

Steps:

1) User Login

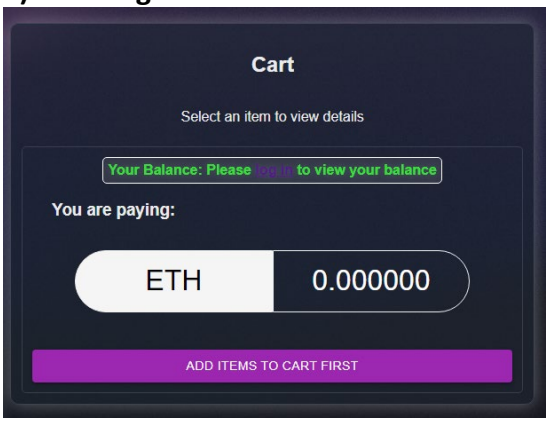

Figure 10.3: Check balance

First user would have to login to buy and display balance.

The system will check whether user is logged in before letting them buy.

It will alert and redirect them to the login page if they are not logged in.

Figure 10.4: Alert user to login

2) Fetch user's data

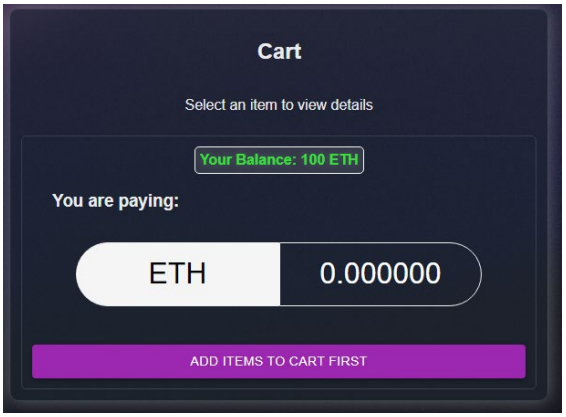

Figure 10.5: Get balance.

After successful login, the market page will fetch the data from MySQL database and display user's balance in the cart.

This makes the marketplace user-friendly as user can easily check their balance when buying/selecting items.

3) Choose item and quantity

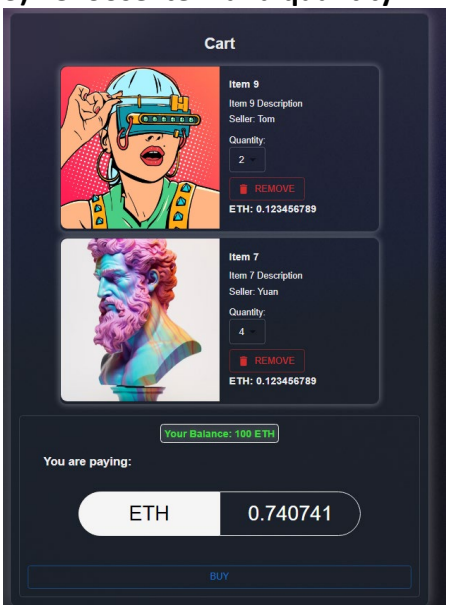

Figure 10.6: Item selection

User can then choose their items and add it to the cart.

User can also specify the quantity of each item, and the web page will automatically calculate the total cost and displays it in real-time as shown in Figure 10.6.

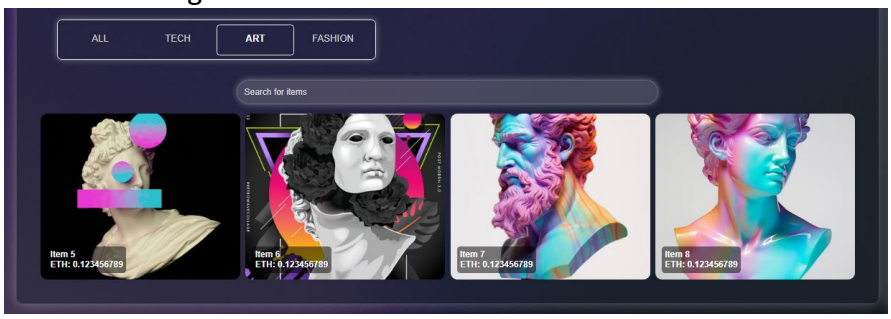

Figure 10.7: Filter by category

Additionally, user can also filter the items based on their category or search for it in the search bar as shown in Figure 10.8.

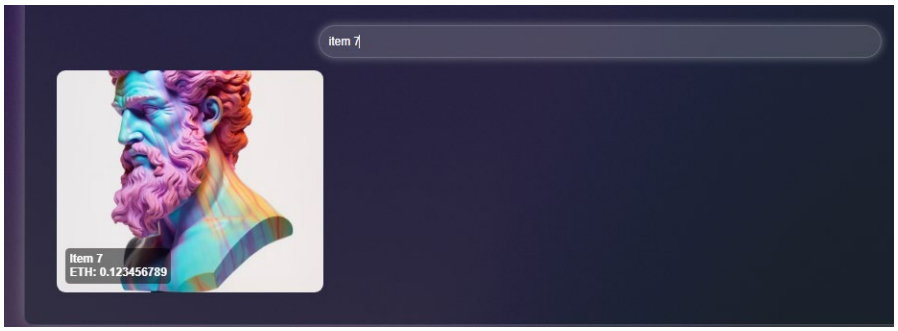

Figure 10.8: Search Item

4) Buy item (handleClick function)

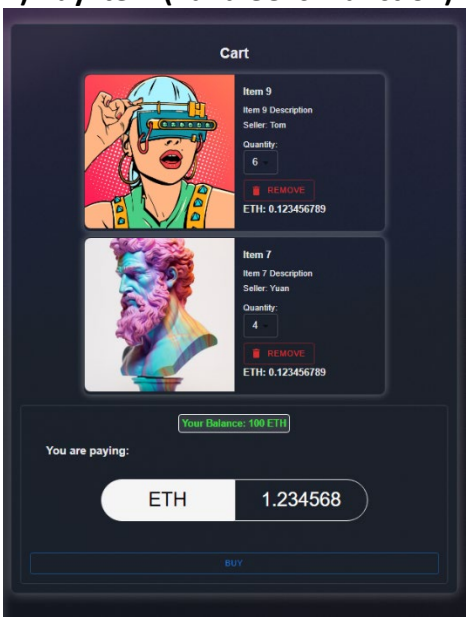

Figure 10.9: Buy Item

```
//On click of buy button
const handleClick = async () => {
  // Check if the user is logged in
  if (!localStorage.getItem("isLoggedIn")) {
    window.alert("Please log in to buy.");
    navigate("/Login");
    return;
  }
  // Check if the user and balance are defined
  if (currentUser && typeof currentUser.balance !== 'undefined') {
    // Check if the user has sufficient balance
    if (parseFloat(currentUser.balance) >= parseFloat(total)) {
      const newBalance = parseFloat(currentUser.balance) - parseFloat(total);
```

Figure 10.10: Check logged in and balance.

After clicking the buy button, the **handleClick** function bind to the 'BUY' button will be called.

The function will first check if user is logged in, then it will check the user's balance to make sure user has sufficient balance to make the purchase.

```
// Prepare the payload for all selected items
const payload = selectedItems.map(item => ({
  item_id: item.item_id,
  price: item.price,
  name: item.name,
  quantity: itemQuantities[item.name] || 1
}));
```

Figure 10.11: Prepare payload to send.

```
try {
  //API call to purchase all selected items
  const response = await axios.post('http://localhost:8000/buyItems', payload);
  console.log("Purchasing item:", payload);  'Purchasing item:' | [ { item_id: 9,
  console.log("Purchase response:", response.data.results[0].status);  'Purchase
  console.log("Receipts:", response.data.results);  'Receipts:' | [ { status: 'It
```

Figure 10.12: API call to buy items.

After all the checks, a payload is prepared for each of the selected items, so the smart contract knows what is being bought.

Using the payload, it will make the API call to the buyItems function in our FastAPI backend to purchase all selected items using the buyItems function in the smart contract. The code in Figure 10.12 handles the API response to check if the purchase is successful.

```
Receipts:                                                    Market.js:204
 ▼ Array(1)
   ▼ 0:
       status: "Item purchased"
       transaction_hash: "0x2caf748f5034331f7377fdce8855b5531ad4
     ▼ transaction_receipt:
         blockHash: "0x4ae23d44dc9e8e5d70a236f43b23827bb0b1db8f1
         blockNumber: 7
         cumulativeGasUsed: 962277
         from: "0xE9B6f078C52E74bFc31417157DE2d8EEa20BdC16"
         gasUsed: 962277
         status: 1
         to: "0xaDAAFC4a3A34E42acdE1a7f0651d7916d9AAd67A"
         transactionHash: "0x2caf748f5034331f7377fdce8855b5531ad
         transactionIndex: 0
       ▶ [[Prototype]]: Object
     ▶ [[Prototype]]: Object
     length: 1
   ▶ [[Prototype]]: Array(0)
```

Figure 10.13: Receipt of transaction

A transaction receipt will be returned from the /buyItems function in FastAPI to confirm that the items are purchased shown in figure 10.13.

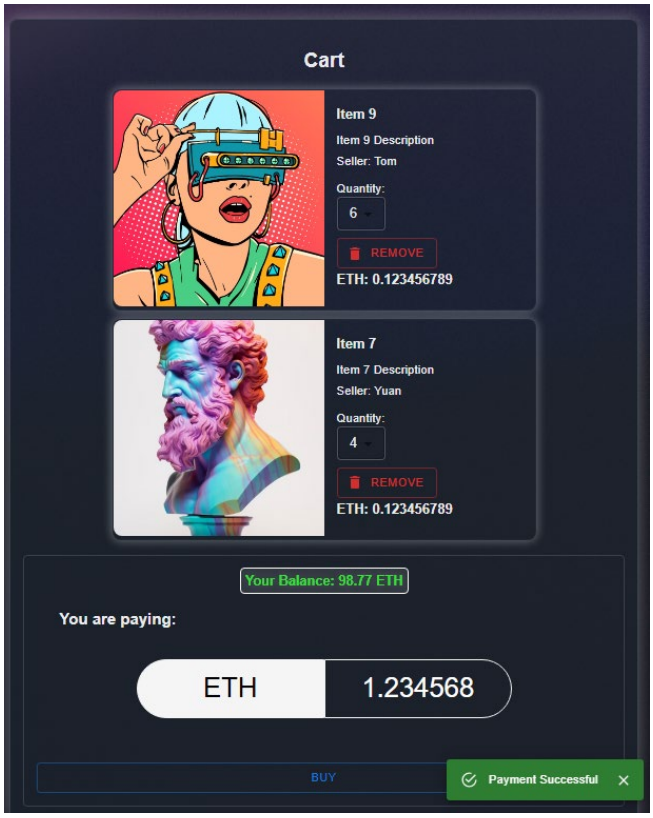

Figure 10.14: Payment successful

The Snackbar as shown in Figure10.14 will then display 'Payment Successful' to indicate that the transaction went through.

The Snackbar will also show an error message if that is a problem with the transaction.

```
mysql> select * from users  where user_id = "1";
+---------+----------+-------+---------+
| user_id | username | pass  | balance |
+---------+----------+-------+---------+
|       1 | user1    | pass1 |   98.77 |
+---------+----------+-------+---------+
```

Figure 10.15: updated user's balance

```
mysql> select * from bought where user_id = "1";
+---------+---------+---------------------+----------+
| user_id | item_id | purchase_date       | quantity |
+---------+---------+---------------------+----------+
|       1 |       1 | 2023-10-11 18:49:42 |        1 |
|       1 |       2 | 2023-10-11 18:49:42 |        4 |
+---------+---------+---------------------+----------+
2 rows in set (0.00 sec)
```

Figure 10.16: bought table before.

```
mysql> select * from bought where user_id = "1";
+---------+---------+---------------------+----------+
| user_id | item_id | purchase_date       | quantity |
+---------+---------+---------------------+----------+
|       1 |       1 | 2023-10-11 18:49:42 |        1 |
|       1 |       2 | 2023-10-11 18:49:42 |        4 |
|       1 |       9 | 2023-10-14 21:41:12 |        6 |
|       1 |       7 | 2023-10-14 21:41:13 |        4 |
+---------+---------+---------------------+----------+
```

Figure 10.17: bought table after.

The user's new balance will also be calculated and updated in the frontend and backend database as shown in Figure 10.16 and Figure 10.17.

Each item in the cart will then be added to the 'bought' table in the backend along with their item id, quantity and the user id that bought the items as shown in Figure 10.17.

Summary of how buying works :

1. The FastAPI /buyItems function is called with a list of items and their quantities.
2. FastAPI calculates the total cost and prepares a transaction to call the buyItems function in the smart contract.
3. The transaction is signed and sent to the Ethereum network.
4. The smart contract's buyItems function is executed, performing validations, and storing the transaction details.
5. A transaction receipt is returned to the FastAPI function, which can then be sent back to the client.
6. Database will be updated with new user's balance and user's items in the 'bought' table.

/About

Purpose:

Users can get an in-depth understanding of the team, its goals, values and members through the About page. This page aims to forge a stronger bond of trust and credibility with the users.

Use Case:

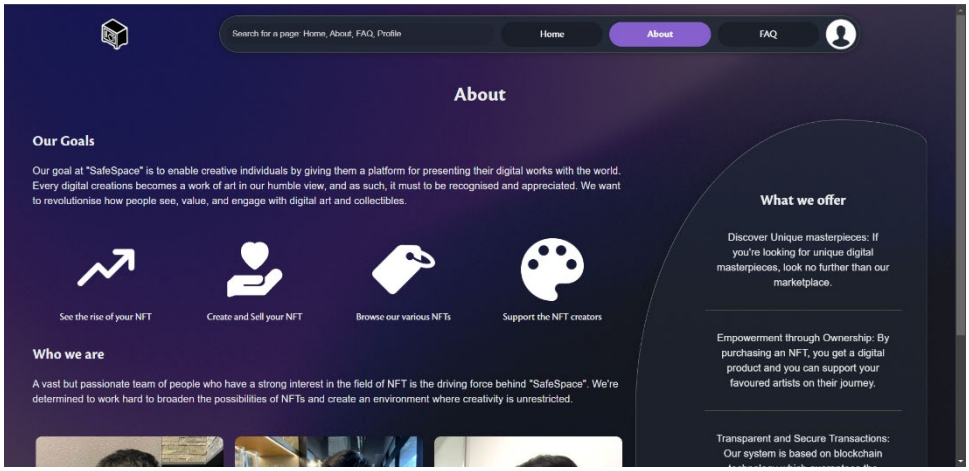

Figure 11.0: About page

Step 1: To visit the about page, users can simply click on the About button in the navigation.
Step 2: Users can view the goals and values of the team under the section “Our Goals”.
Step 3: Users can view the team members in the “Who we are” section.
Step 4: Users can view the website's offerings under the “What we offer” section.

/FAQ

Purpose:

Users can quickly and easily find answers for common questions, concerns, or other inquiries they may have on the FAQ page. This page aims to offer a comprehensive resource for common questions and reduce support requests.

Use Case:

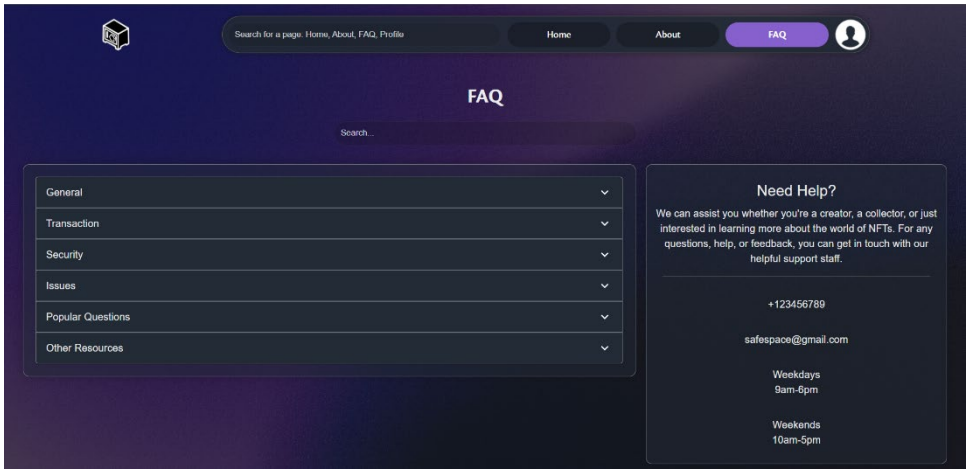

Figure 12.0: FAQ page

Step 1: To visit the faq page, users can simply click on the FAQ button in the navigation.
Step 2: Users can use the search bar to quickly find the question they are searching for.
Step 3: Users can find the answers in the dropdown menus.
Step 4: If any additional help is needed, the users can find the contact number for our help desk, email and working hours in the “Heed Help?” section.

/ Smart Contract Implementation

/ Introduction

In this assignment, we implemented smart contracts into our web application to offer an increase in security for the transactions when buying items through our web application. Our smart contract is written in Solidity and developed on the Ethereum private blockchain using Ganache, our smart contract serves as a decentralized marketplace.

In this section, we will discuss the architecture, features, and implementation of the smart contract into our marketplace.

/ Architecture

Using the pragma version 0.6.0, we developed the smart contract using Solidity and tested the functionalities using Remix.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

contract SmartContract {
    struct Item {
        string name;
        string description;
        string image;
        string category;
        uint256 price;
        string seller;
    }

    struct Transaction {
        address toAddress;
        address fromAddress;
        uint256 price;
        uint256 item;
    }
```

Figure 13.0: Struct

struct Item - represents the items that we have listed in the market page, it contains all the necessary details of the items, we made it so that the structure matches our database 'listedItem' table's structure to keep consistency.

struct Transaction - represents the transaction between the smart contract and the buyer, it records the transaction between 2 addresses and captures the price and item involved I the transaction.

/ Features

```
event ItemPurchased(
    uint256 indexed itemId,
    address buyer,
    uint256 price,
    uint256 time
);
event TransactionAdded(
    uint256 indexed transactionId,
    address toAddress,
    address fromAddress,
    uint256 price,
    uint256 item
);
```

Figure 13.1: Struct

Event logging:
We have added 2 events into our smart contract, 'ItemPurchased' and 'TransactionAdded', which are emitted during the transaction process to help track and audit transactions on the blockchain.

```
constructor() public {
    owner = msg.sender;
    //123456789000000000 Wei equivalent to 0.123456789 Ether.
    items[1] = Item(
        "Item 1",
        "Item 1 Description",
        "https://i.ibb.co/r58jQcq/tech1.jpg",
        "Tech",
        123456789000000000,
        "Andrew"
    );
    items[2] = Item(
        "Item 2",
        "Item 2 Description",
        "https://i.ibb.co/kJ23Wbb/tech2.jpg",
        "Tech",
        123456789000000000,
        "Andrew"
    );
```

Figure 13.2: Item list

Item Management:
The smart contract populates a list of the listed items in different categories. Each of the itme has a unique identifier and is associated with their metadata such as, name, description, image, price, etc.

The reason we prepopulate the items in the smart contract is because we only have 12 items, which doesn't take much gas fee ,and since our website only allow users to buy items and not list items, the listed items won't be changed.

```
function buyItems(
    uint256[] memory _itemIds,
    uint256[] memory _quantities
) public payable {
    // Check that the lengths of the _itemIds and _quantities arrays match
    require(
        _itemIds.length == _quantities.length,
        "Mismatched array lengths"
    );

    uint256 totalCost = 0;

    // Calculate the total cost of all items
    for (uint i = 0; i < _itemIds.length; i++) {
        uint256 itemId = _itemIds[i];
        uint256 quantity = _quantities[i];
        require(items[itemId].price > 0, "Item does not exist");
        totalCost += items[itemId].price * quantity;
    }

    // Check that the total cost matches the amount sent
    require(msg.value == totalCost, "Incorrect ETH value sent");

    // Process each item
    for (uint i = 0; i < _itemIds.length; i++) {
        uint256 itemId = _itemIds[i];
        uint256 quantity = _quantities[i];

        // Add transactions and emit events
        for (uint j = 0; j < quantity; j++) {
            transactions[transactionCount] = Transaction(
                owner,
                msg.sender,
                items[itemId].price,
                itemId
            );
            emit TransactionAdded(
                transactionCount,
                owner,
                msg.sender,
                items[itemId].price,
                itemId
            );
            transactionCount++;
        }

        // Emit ItemPurchased event
        emit ItemPurchased(
            itemId,
            msg.sender,
            items[itemId].price * quantity,
            block.timestamp
        );
    }
```

Figure 13.3: buyItem function

Transaction Management:
The 'buyItems' function allows users to purchase single/multiple items in a single transaction. It uses an array of the predefined item IDs and their price and quantity to calculate the total cost and processes the transaction and emit the event 'TransactionAdded', and 'ItemPurchased' if the sent ETH matches the total cost, otherwise it will revert the transaction.

function buyItems:
1.Accepts _itemIds and _quantities as parameters.
2.Checks that the lengths of _itemIds and _quantities are the same.
3.Checks that each item exists and calculates the total cost in Ether.
4.Verifies that the total cost matches the amount of Ether sent in the transaction (msg.value).
5.For each item, it creates a new Transaction struct and stores it in the transactions mapping.
6.Emits TransactionAdded and ItemPurchased events.

Python FastAPI backend Integration:
Our Python FastAPI backend interacts with the smart contract by calling the functions such as 'deployContract' and 'getItemDetails' and 'buyItems' when they are called in the frontend using the API endpoints.

/ Functions

- **deployContract:** Deploys the smart contract onto the private Ethereum blockchain on Ganache.
- **getItemDetails:** Fetches details of an item from the smart contract.
- **buyItems:** Sends a transaction to purchase items, updating both the blockchain and the MySQL database.

/ Benefits

Trust less Transactions: Smart contracts eliminate the need for middle, enabling fast trust less transactions between parties.
Transparency and Security: All transactions are publicly recorded on the blockchain, ensuring transparency and security between all parties.
Automated Compliance: It can automatically enforce the rules of the marketplace, reducing the risk of fraud or non-compliance transactions

/ Project Deployment Instruction

A/ Environment setup:

Note: *all commands for both MAC and Windows can be found in the ReadMe file included in the project folder. This example of deployment is done on MAC.*

Python Version used: 3.9.17

Step 1: Connect to tester's MySQL terminal or command line

Step 2: Changing MySQL's password to 'password' for the purpose of testing

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'password';
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Figure 14.0: MySQL Alter native password

Step 3: In backend folder directory and run sql script: `mysql -u root -p < safespace_script.sql` ***OR*** `/usr/local/mysql/bin/mysql -u root -p < safespace_script.sql` (for Mac)

Step 4: as well as fast api: ` uvicorn main:app --reload`

```
(base) thinhdang@Thinhs-MacBook-Pro COS300049 % conda activate testenvi
(testenvi) thinhdang@Thinhs-MacBook-Pro COS300049 %
(testenvi) thinhdang@Thinhs-MacBook-Pro COS300049 % cd backend
(testenvi) thinhdang@Thinhs-MacBook-Pro backend % /usr/local/mysql/bin/mysql -u root -p < safespace_script.sql
Enter password:
(testenvi) thinhdang@Thinhs-MacBook-Pro backend %
(testenvi) thinhdang@Thinhs-MacBook-Pro backend % uvicorn main:app --reload
INFO:     Will watch for changes in these directories: ['/Users/thinhdang/Documents/GitHub/COS300049/backend']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [3044] using WatchFiles
INFO:     Started server process [3046]
```

Figure 14.1: FastAPI

Step 5: run `npm install`

Step 6: run `npm start`

B/ Ganache deployment guide:

Tester must enter their ganache's wallet address and private key in the deployContract section in main.py that is in our backend folder. After that, the contract should be auto deployed when tester signs in with the sign in credentials provided below. This enables buying more items in the marketplace for users.

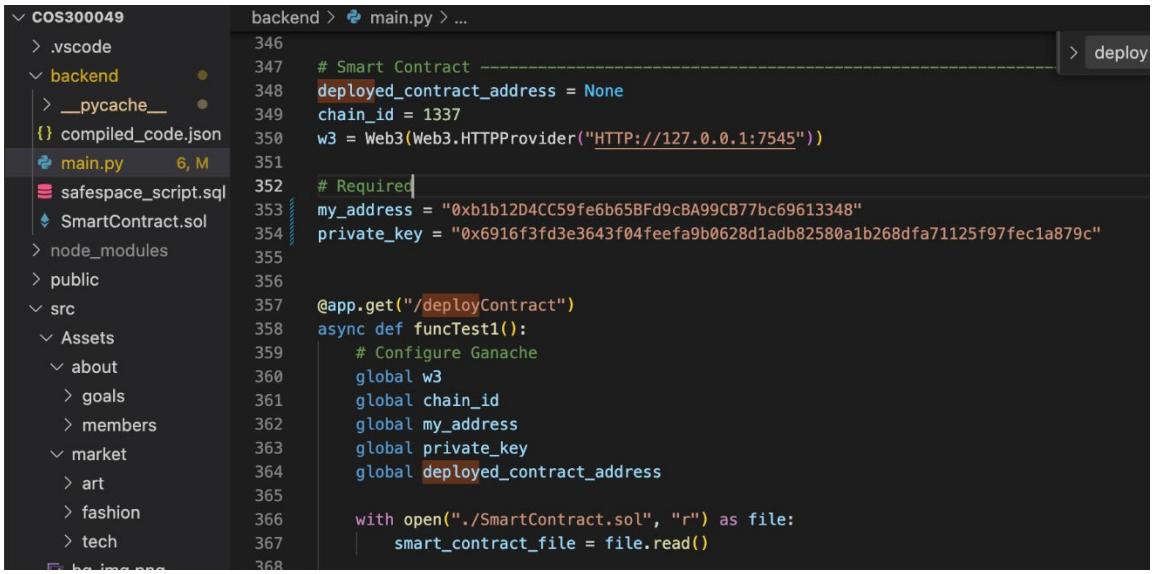

Figure 14.2: Ganache Config

C/ User sign-in guide:

We have created 3 users for testing from our backend, these users have existing assets and balance that can be used to test the backend compatibility of the website as well as the smart contract functionality. The login information can be found below:

For User1:
Username: user1
Password: pass1

For User2:
Username: user2
Password: pass2

For User3:
Username: user3
Password: pass3

Note: The overall system architecture design has been updated from assignment 1

VI. /Overall System Architecture Design

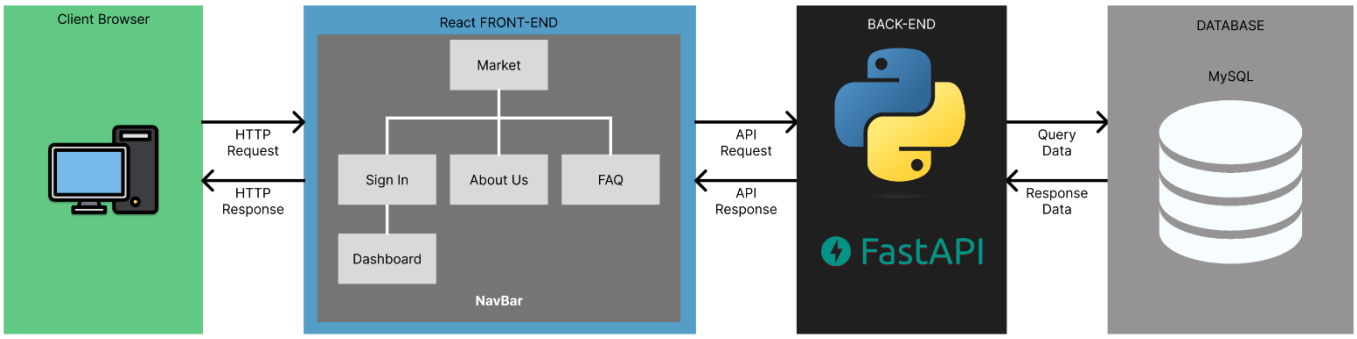

Figure 15.0: System Architecture Design

Client Browser:
The client browser is the part of the application that users interact with. Its primary purpose is to display the React webpages and provide a representation of the website.

The client browser sends HTTP requests to request webpages to communicate with the React frontend. The frontend layer then responds by serving those requested webpages through HTTP responses. This ensures a better user experience regardless of the device or browser.

FRONTEND (React):
The frontend layer interacts with the backend through API requests and responses. This separation allows for flexibility in design and scalability as it decouples the user interface from the data processing logic.

Component:

I. **Navbar:** This component offers a gateway to different pages of the website; it consists of different navigation buttons and a search function which searches for the available pages offered by the website.

Pages:

I. **Market (Home Page):** This serves as the website's landing page/homepage. It provides an overview of the website's offerings and serves as a gateway to other sections of the site.

II. **Sign In:** A dedicated page for users to authenticate themselves. Once authenticated, users are directed to their personal dashboard.

III. **Dashboard:** A personalized space for authenticated users to access and manage/view their data and assets.

IV. **About:** Provides information about the website, its mission, team, and other relevant details.

V. **FAQ:** Addresses common questions and concerns users might have, enhancing user trust and reducing the need for customer support.

BACKEND (FastAPI + Python): *modified from assignment 1.*
The backend will be built using FastAPI and Python which handles user requests performs business logic operations and manages data interactions. It acts as a bridge between the frontend layer and the database.

In assignment 1 we opted for Node.js and ExpressJS for our backend due to their nature, efficiency, and scalability. However, we have decided to change to FastAPI and Python, FastAPi offers easy type validation, serialization, and it uses Python, which makes it easy to debug and implement.

The backend communicates with the database using data queries and responses. This ensures that data retrieval, storage, and manipulation are done securely and efficiently.

DATABASE (MySQL):
For assignment 2's data storage and management, we opted for MySQL which is a relational database. This is used in the frontend display the necessary information like listed items, user data, graph, etc. We utilize it in the FastAPI backend to query the MySQL to perform complex functions for the website.

MySQL offers fast data retrieval, reliability, and its user friendly since all the team members is experienced with it. This makes it ideal for this project.

VII. /Project File Structure

```
public
    index.html
    logo.png
src
    Assets
        about
            goals
                palette.png
                support.png
                tag.png
                trend.png
            members
                Andrew.jpg
                Tom.jpg
                Yuan.jpg
        market
            art
                art1.png
                art2.png
                art3.png
                art4.png
            fashion
                fashion1.png
                fashion2.png
                fashion3.png
                fashion4.png
            tech
                tech1.png
                tech2.png
                tech3.png
                tech4.png
        bg-img.png
        database.js
        logo.png
    Components
        Navbar
            navbar.css
            Navbar.js
    Pages
        About
            about.css
            About.js
        Dashboard
            dashboard.css
            Dashboard.js
        FAQ
            faq.css
            FAQ.js
        Login
            login.css
            Login.js
        Market
            market.css
            Market.js
        NoMatch
            NoMatch.js
    App.css
    App.js
    App.test.js
    index.css
    index.js
    reportWebVitals.js
    setupTest.js
```

Figure 16.0: Project File Structure

Our project file structure consists of 3 main folders in “src”, which are the “Assets”, “Component”, and “Pages”.

“Assets” consists of “about”, and “market”. The “about” folder contains “goals”, and “members” folder which consist of each of the members’ pictures which are displayed on the About page.

The “goals” folder consists of the icons that are used on the About page.

The “market” folder holds all the necessary images of the assets that are displayed on the Market page.

The “Assets” folder also consists of three individual files which are the “bg-img.png”, “logo.png”, and “database.js”.

“bg-img.png” consists of the website’s background image which is used as the default background of the website in “index.css”.

“logo.png” consists of the website’s logo which is used in several areas of the website.

“database.js” acts as the database for our website, it includes all the necessary data like the items’ descriptions and prices.

“Components” consists of the "Navbar” folder, inside the “Navbar” folder, we have “navbar.css” which consist of the styling on the navbar, “Navbar.js” consist of the logic and structure of the navbar component.

“Pages” folder stores the files needed to construct each page, for example the “About” folder consists of two files, which are the “about.css” for styling, and “About.js” for the page’s logic and structure. The only exception is the NoMatch component as there is nothing to style on that page.

We decided to have a separate CSS file for each component as it would be easier to manage and work with, in comparison to one CSS file which might look messy if the styling for all components is mixed.

```
└── backend
    ├── compiled_code.json
    ├── main.py
    ├── safespace_script.sql
    └── SmartContract.sol
```

Figure 16.1:backend Project File Structure

Our backend file structure consists of 4 files: compiled_code.json, main,py, safespace_script.sql, and SmartContract.sol.

“compiled_code.json”
consists of the bytecode and ABI for the smart contract “SmartContract.sol” solidity file. This is a compiled version of our smart contract, which is deployed to the Ethereum blockchain. It is basically a representation of our smart contract structure in binary form.

“main.py”
consists of the FastAPI backend, which has all the API endpoints that we used in the frontend for various functionalities of the website. It handles HTTP requests and responses as well as other functionalities such as deploying smart contract, retrieving from MySQL, etc.

“safespace_script.sql”
This is our MySQL database schema. It is a script file to set up the database. The queries to create and insert tables defining relationships between tables are done inside here. This script should be run before starting the application to ensure the website can retrieve the necessary data from the database.

“SmartContract.sol”
This is our smart contract, which is written in solidity. It has the business rules and methods that can be used on the Ethereum blockchain. This is responsible for buying items. This is the file that generates “compiled_code.json” when it is deployed to the Ethereum network.

VIII. /Conclusion

While developing this marketplace, our team has built a robust and dynamic backend infrastructure that not only supports traditional marketplace functionalities but also seamlessly integrates with blockchain technology through the deployment of smart contracts. We have enabled important functionalities such as, user login, detailed personal asset management, buying items from the market with smart contract deployment. Additionally, the inclusion of FAQ and 'About Us' pages also ensures our users are well-informed with additional details about the website.

This project was a good learning experience for three of us, we gained insights into the challenges and the requirements from this assignment and were able to learn a lot along the way. Looking forward, we are enthusiastic about refining, expanding and building upon this foundation. We are confident that with continuous improvements, our marketplace can be a more innovative group project in the future.

IX. /References

Behance (n.d.). *UI/UX*. [online] www.behance.net. Available at: https://www.behance.net/galleries/ui-ux [Accessed 30 Aug. 2023].

dribbble.com. (n.d.). *Browse thousands of Webdesign images for design inspiration | Dribbble*. [online] Available at: https://dribbble.com/search/webdesign [Accessed 30 Aug. 2023].

HQSoftware. (n.d.). *Web Applications Architectures: Types and Main Components*. [online] Available at: https://hqsoftwarelab.com/blog/web-application-architecture/.

Maheshwari, R. (2012). *Why Is the Crypto Market Rising Today?* [online] Forbes. Available at: https://www.forbes.com/advisor/in/investing/cryptocurrency/why-is-crypto-going-up/.

mui.com. (n.d.). *React Accordion Components - Material UI*. [online] Available at: https://mui.com/material-ui/react-accordion/.

mui.com. (n.d.). *React Components - Material UI*. [online] Available at: https://mui.com/material-ui/.

mui.com. (n.d.). *React Divider Components - Material UI*. [online] Available at: https://mui.com/material-ui/react-divider/.

mui.com. (n.d.). *React Grid component - Material UI*. [online] Available at: https://mui.com/material-ui/react-grid/.

mui.com. (n.d.). *React Typography Components - Material UI*. [online] Available at: https://mui.com/material-ui/react-typography/.

Nguyen, C. (2020). *A Complete Guide to Build a Well Structured 3-tier Architecture (MERN Stack — ES6)*. [online] Medium. Available at: https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805.

SEMESTER 2 2023

COS30049 - Computing Technology Innovation Project

Assignment 3 – Smart Contracts, Blockchain, and Cybersecurity

By Group 1-64

Due: Sunday, 5/11/2023, 11:59 pm

Members:
Yap Zhe Wei (Leader): 103508895
Vu Gia Thinh Dang: 103177240
Yong Yuan Chong: 101224021

Project Information:

Project Name:

Uniswap V3 Smart Contracts, Blockchain, and Cybersecurity

Project Description:

Purpose:

The goal of this project is to give a full examination of the UniswapV3 smart contracts architecture and its role in blockchain technology. It intends to investigate the nature of smart contracts.

Objectives:

Decentralisation: Smart contracts cant by edited/modified once they are deployed ,and there is no involvement of a third party (Uniswap 101: What is Uniswap? 2023).

Liquidity: Can access liquidity from retail users by enabling anyone to establish and supply liquidity to a pool (Uniswap 101: What is Uniswap? 2023).

Accessibility: More widely accessible to users, even those who might not have access to conventional banking services (Uniswap 101: What is Uniswap? 2023).

Functionalities:

Concentrated Liquidity: It allows liquidity providers to concentrate their assets at smaller price intervals (Kant 2022).

Oracle: It offers access to historical price and liquidity data (Danill 2022).

Range Orders: it can approximate a limit order by giving a single asset as liquidity inside a particular range (Danill 2022).

Swaps: It is a permissionless process of exchanging one cryptocurrency to another with a small fee (Danill 2022).

Team Information:

Zhe Wei Yap (103508895) - Wrote ABI illustration, Wrote Inheritance Graph Discussion

Vu Gia Thinh Dang (103177240) - Wrote ABI illustration, Wrote Call Graph Discussion.

Yong Yuan Chong (101224021) - Wrote Project Information, Wrote Executive Summary

Version Information:

Smart Contract Version:

0.8.22

Dependencies Version:

Solidity: The primary programming language used to develop the smart contracts in Uniswap V3.

Ethereum Virtual Machine: Uniswap V3 uses the Ethereum Virtual Machine to run smart contracts.

ERC-20 and ERC-721: Cryptocurrencies standards to enable compatibility.

Other sources that may interact with Uniswap V3:

- Web3.js
- Chainlink
- InterPlanetary File System

Executive Summary:

This report is an analysis of contracts. It dives into the structures, functions, methodologies, and cybersecurity of them. Specifically, it focuses on UniswapV3 contracts which are a decentralized exchange (DEX) protocol built on the Ethereum blockchain and represents the third version of Uniswap.

The report explains how by using contracts and liquidity providers users can do many things such as: exchanging cryptocurrencies with low fees, accessing historical prices and liquidity data. Users can even leverage these providers to concentrate their assets within a specific price range. To all be done with one DEX.

These functionalities rely on Ethereum tools such as Web3.js for interacting with Ethereum, Ethereum Virtual Machine for execution purposes Solidity for smart contract development and ERC 20 and ERC 721 standards for cryptocurrency compatibility. Uniswap V3 aims to improve user experience by reducing gas fees and transaction times through integration with Layer 2 scaling solutions like Arbitrum and Optimism.

The report also explains the understanding of relationship between contracts while highlighting the significance of contract structures that involve external libraries usage and inheritance. To gather data on each contracts Application Binary Interface (ABI) and construct them accordingly Remix was utilized. By obtaining the ABI information various aspects such, as call graphs, inheritance graphs ABI illustrations are discussed to provide a representation of the structure of these smart contracts.

A big part of this report delves into the ABI illustrations, which provide explanations and descriptions for each function of the Smart Contracts used in Uniswap. The report sheds light on aspects such, as readability, structure, and more offering valuable insights into how these functions operate within the Uniswap smart contract ecosystem. This report also talks about the inheritance graph in Solidity, it discussed the main concepts of inheritance hierarchy, which are important for developer's readability, security, extensibility, and maintainability.

ABI Illustration:

Application Binary Interface for smart contracts in UniswapV3 is a convenient way to interact and understand contracts on the blockchain as it defines how functions and data structures are being accessed. Through studying the ABI, we will have a clear expectation of what kind of inputs, outputs, and state mutability a function can have and a decent understanding of their purposes. The ABI is presented in JSON format and can be obtained after compiling the sol file on Remix.

a) NoDelegateCall.sol:

ABI Generation on Remix

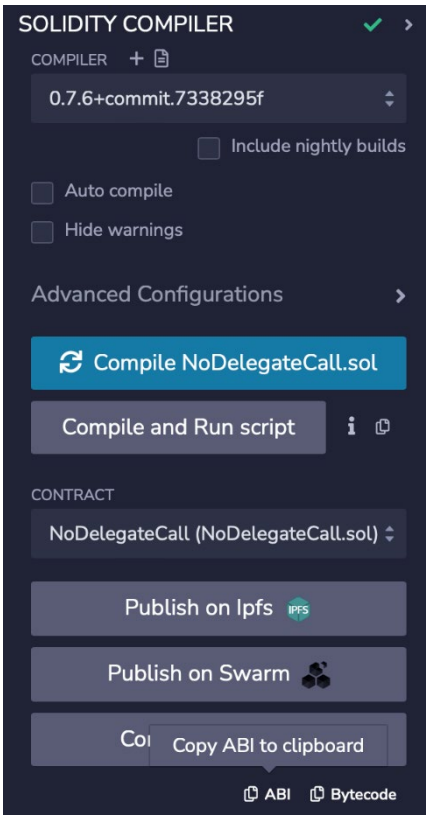

Figure 1.0: Compiled NoDelegateCall

We successfully compiled the NoDelegateCall.sol. This is one of the delegates calls in Solidity that allow contracts to execute code in the context of another contract's state. Since the contract is designed to be inherited by other contracts, it does not contain public functions which resulting in our ABI being empty.

Function explained:

1) **Constructor():**
   a. **Inputs:** None
   b. **Outputs:** None
   c. **State Mutability:** Nonpayable
   d. **Description:** Initialises the contract. The variable cannot be altered after initial assignment
2) **checkNotDelegateCall():**

a. **Inputs:** None
b. **Outputs:** None
c. **State Mutability:** View
d. **Description:** This function compares the current contract's address with stored original address to make sure the function in which this is called is not through a delegate call.

3) **noDelegateCall():**
   a. **Inputs:** None
   b. **Outputs:** None
   c. **Description:** this is a modifier that precents other functions from being executed through delegate call.

b) UniSwapV3Factory.sol:

ABI Generation on Remix:

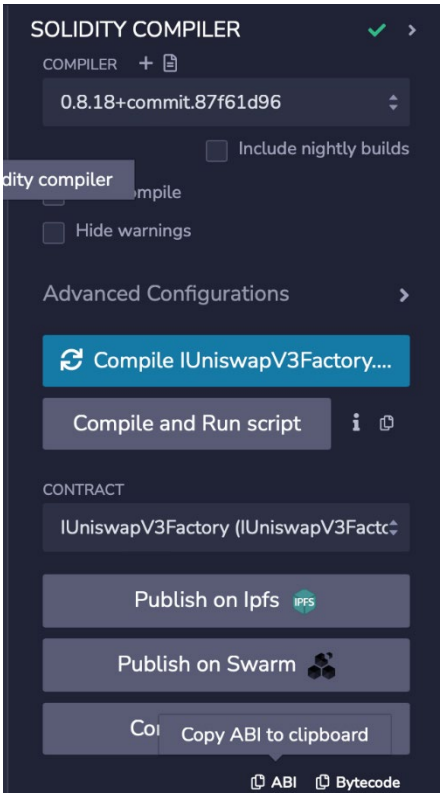

Figure 1.1: Compiled Factory sol

We have successfully compiled the UniswapV3Factory.sol with Remix. We copied the ABI file and analysed it below:

Function explained:

1) **createPool():**
   a. **Inputs:**
      i. tokenA(address): address of first token
      ii. tokenB(address): address of second token
      iii. fee(uint24): fee amount for the pool
   b. **Outputs:**

i. pool(address): address of constructed pool

c. **State Mutability:** nonpayable

d. **Description:** This function creates new liquidity pool for a pair of tokens with associated fee rate.

2) **enableFeeAmount():**

a. **Inputs:**

i. fee (uint24): the enabled fee amount

ii. tickSpacing (int24): tick spacing value for fee

b. **Outputs:** None

c. **State Mutability:** nonpayable

d. **Description:** This function is called to create new fee amount and associated tick spacing within the factory contract

3) **setOwner():**

a. **Inputs:**

i. _owner (address): The address of the previous owner

b. **Outputs:** None

c. **State mutability:** nonpayable

d. **Description:** This function is called for changing contract owner

4) **Owner():**

a. **Inputs:** None

b. **Outputs:**

i. Owner(address): The address of the current owner

c. **State mutability:** view

d. **Description:** returns the address of current owner

5) **feeAmountTickSpacing():**

a. **Inputs:**

i. fee(uint24): the fee amount

b. **Outputs:**

i. tickSpacing(int24): the tick spacing of the given fee amount

c. **State Mutability:** view – read only

d. **Description:** This function returns the tick spacing of the fee amount

6) **getPool():**

a. **Inputs:**

i. tokenA(address): address of first token

ii. tokenB(address): address of second token

iii. fee(uint24): fee amount

b. **Outputs:**

i. pool(address): address of the associated pool

c. **State Mutability:** view

d. **Description:** Provides the address of the specific pool for a pair of tokens and fee

c) UniSwapV3Pool.sol:

ABI Generation on Remix:

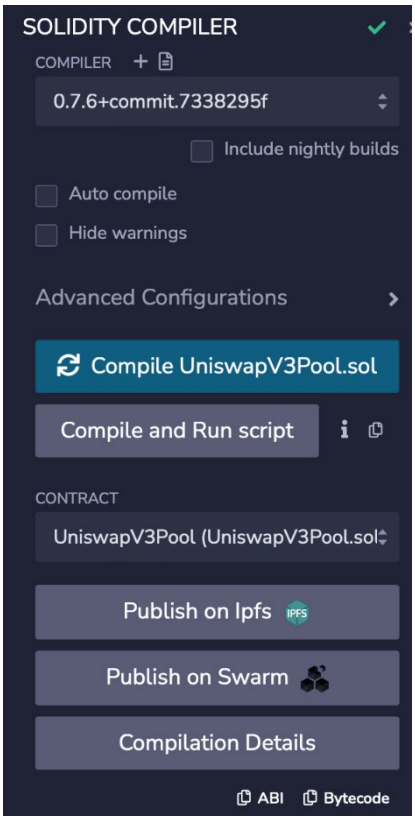

Figure 2.2: Compiled Pool sol

We have successfully compiled the UniswapV3Factory.sol with Remix. We copied the ABI file and analysed it below:

Function explained:

1) **checkTicks(int24 tickLower, int24 tickUpper):**
   a. **Inputs:**
      i. tickLower (int24): Lower end of the tick range.
      ii. tickUpper(int24): : Upper end of the tick range.
   b. **Outputs:**
      i. None (may revert if conditions are not met ()
   c. **State Mutability:** view
   d. **Description:** Checks the validity of tick range provided and help ensure **tickLower** is lesser than **tickUpper**, also checks if both ticks are initialized. Transaction will be reverted if these checks failed.

2) **_blockTimestamp():**
   a. **Inputs:** None
   b. **Outputs:**
      i. time(uint32): The current block timestamp.
   c. **State Mutability:** view
   d. **Description:** This function retrieves the current block timestamp. This function is private therefore it can't be called externally.

3) **balance0():**
   a. **Inputs:** None
   b. **Outputs:**
      i. balance(uint256): The amount of token0.
   c. **State Mutability:** view
   d. **Description:** Returns the current balance of token0 in the smart contract.

4) **balance1():**
   a. **Inputs:** None
   b. **Outputs:**
      i. balance(uint256): The amount of token1.
   c. **State Mutability:** view
   d. **Description:** Returns the current balance of token1 in the smart contract.

5) **snapshotCumulativesInside(int24 tickLower, int24 tickUpper):**
   a. **Inputs:**
      i. tickLower (int24): Lower end of the tick range.
      ii. tickUpper(int24): : Upper end of the tick range.
   b. **Outputs:**
      i. tickCumulativeInside(int56): Total summed tick (portion of time pool was in the tick range)
      ii. secondsPerLiquidityInsideX128(uint128): Seconds per liquidity within specified range.
      iii. secondsInside(uint32): The total number of seconds pool spent in the specified tick range.
   c. **State Mutability:** view
   d. **Description:** provides snapshot of the cumulative tick and liquidity values within a tick range, and total time spent in the tick range. Snapshots must only be compared to other snapshots taken during the time that a position existed. Thus, snapshots cannot be compared if the position is not retained for the full time between the first and second snapshots.

6) **observe(uint32[] calldata secondsAgos):**
   a. **Inputs:**
      i. secondsAgos (uint32[]): seconds passed since each liquidity and cumulative tick returned a value.
   b. **Outputs:**
      i. tickCumulatives (int56[]): Cumulative tick values from each secondsAgos since the current block timestamp.
      ii. secondsPerLiquidityCumulativeX128s (uint160[]): Cumulative seconds per liquidity at each secondsAgos.
   c. **State Mutability:** view

d. **Description:** This function returns the cumulative tick and liquidity values at specified points in the past, allowing for the calculation of time-weighted averages. This function must be called with 2 values to provide time weighted average tick, one value should represent start period and another should represent end period.

7) **increaseObservationCardinalityNext(uint16 observationCardinalityNext):**
   a. **Inputs:**
      i. observationCardinalityNext (uint16): desired future cardinality for the observation buffer.
   b. **Outputs:** None
   c. **State Mutability:** nonpayable
   d. **Description:** used to increase max size of observations for price and liquidity, which is used for storing historical data for calculating time weight average prices.

8) **initialize(uint160 sqrtPriceX96):**
   a. **Inputs:**
      i. sqrtPriceX96 (uint160): initial square root price of the pool.
   b. **Outputs:** None
   c. **State Mutability:** nonpayable
   d. **Description:** initializes a pool with a specified square root price. Can only be called once by the pool's factory contract during creation of the pool.

9) **_modifyPosition(ModifyPositionParams memory params):**
   a. **Inputs:**
      i. params (ModifyPositionParams): : Struct containing details about position to be modified (owner, tickLower, tickUpper, and liquidityDelta).
   b. **Outputs:** Includes updated liquidity, fee growth, and tokens owed.
   c. **State Mutability:** nonpayable
   d. **Description:** A private function to changes position's liquidity which may result in minting/burning of pool tokens. Updates position's data and accumulate fees owed.

10) **_updatePosition(address owner, int24 tickLower, int24 tickUpper, int128 liquidityDelta, int24 tick):**
   a. **Inputs:**
      i. owner (address): address of position owner
      ii. tickLower (int24): lower end of the tick range for position.
      iii. tickUpper (int24): upper end of the tick range for position.
      iv. liquidityDelta (int128): change in liquidity for the position.
      v. tick (int24): current tick of pool.
   b. **Outputs:** Includes updated liquidity, fee growth, and tokens owed.
   c. **State Mutability:** nonpayable
   d. **Description:** Private function that updates position's liquidity, fee growth, and tokens owed based on provided parameters. Important for position management within the pool.

11) **mint(address recipient, int24 tickLower, int24 tickUpper, uint128 amount, bytes calldata data):**
   a. **Inputs:**

i. recipient (address): Recipient address for receiving the liquidity tokens
ii. tickLower (int24): lower end of the tick range for position.
iii. tickUpper (int24): upper end of the tick range for position.
iv. amount (uint128): amount of liquidity to mint.
v. data (bytes): Arbitrary data passed to call back.

b. **Outputs:**
   i. amount0(uint256): Amount of token0 paid to mint specified liquidity.
   ii. amount1(uint256): Amount of token1 paid to mint specified liquidity.
c. **State Mutability:** nonpayable
d. **Description:** this function adds liquidity to the specified recipient/tickLower/tickUpper position. The amount of token0 and token1 paid will be returned, and a callback will be executed to transfer the amount to the pool.

12) **collect(address recipient, int24 tickLower, int24 tickUpper, uint128 amount0Requested, uint128 amount1Requested):**
   a. **Inputs:**
      i. recipient(address): Recipient address to receive the collected tokens.
      ii. tickLower (int24): lower end of the tick range for position.
      iii. tickUpper (int24): upper end of the tick range for position.
      iv. amount0Requested(uint128): Amount of token0 to collect from fess owed.
      v. amount1Requested(uint128): Amount of token1 to collect from fess owed.
   b. **Outputs:**
      i. amount0 (uint128): Actual amount of fess collected from token0.
      ii. amount1 (uint128): Actual amount of fess collected from token1.
   c. **State Mutability:** nonpayable
   d. **Description:** Allows a position owner to collect total fees in token0 and token1 from a range within the pool.

13) **burn(int24 tickLower, int24 tickUpper, uint128 amount):**
   a. **Inputs:**
      i. tickLower (int24): lower end of the tick range for position.
      ii. tickUpper (int24): upper end of the tick range for position.
      iii. amount (uint128): Amount of liquidity to burn.
   b. **Outputs:**
      i. amount0 (uint128): Actual amount of fess sent from token0 to recipient.
      ii. amount1 (uint128): Actual amount of fess sent from token1 to recipient.
   c. **State Mutability:** nonpayable
   d. **Description:** Used to remove liquidity from a pool. It burns the sender's liquidity and the account tokens owing for the position's liquidity.

14) **swap(address recipient, bool zeroForOne, int256 amountSpecified, uint160 sqrtPriceLimitX96, bytes calldata data):**
    a. **Inputs:**
        i. recipient (address): Recipient address to receive the swap's output.
        ii. zeroForOne (bool): Order of swapping (true for token0 to token1, false for token1 to token0)
        iii. amountSpecified (int256): Amount of the input token to swap. (specifies whether the swap should be exact input (positive) or exact output (negative))
        iv. sqrtPriceLimitX96 (uint160): Price limit of swap. If the price is zero for one, it cannot be less after the swap. If one for zero, the price following the swap cannot be more than this value.
        v. data (bytes): Arbitrary data passed to the callback.
    b. **Outputs:**
        i. amount0 (uint128): The difference in value of the pool's token0 balance, specific when negative and minimum when positive.
        ii. amount1 (uint128): Difference in value of the pool's token1 balance, specific when negative and minimum when positive.
    c. **State Mutability:** nonpayable
    d. **Description:** executes a swap in the pool, which allows a user to swap between token0 and token1. The direction and limit of the swap are specified, and a callback can be executed to ensure the necessary funds are provided for the swap.

15) **flash(address recipient, uint256 amount0, uint256 amount1, bytes calldata data):**
    a. **Inputs:**
        i. recipient (address): Recipient address to receive the token0 and token1 amounts.
        ii. amount0 (uint256): Token 0's amount to send.
        iii. amount1 (uint256): Token 1's amount to send.
        iv. data (bytes): Arbitrary data passed to the callback.
    b. **Outputs:** None
    c. **State Mutability:** nonpayable
    d. **Description:** Allow borrowing of token0 and token1 with a promise for paying back the borrowed amount plus a fee in a transaction block.

16) **setFeeProtocol(uint8 feeProtocol0, uint8 feeProtocol1):**
    a. **Inputs:**
        i. feeProtocol0 (uint8): new protocol fee percentage for token0.
        ii. feeProtocol1 (uint8): new protocol fee percentage for token1.
    b. **Outputs:** None
    c. **State Mutability:** nonpayable
    d. **Description:** sets the protocol fee percentages for token0 and token1. It can only be executed by the pool's administration.

17) **collectProtocol(address recipient, uint128 amount0Requested, uint128 amount1Requested):**
    a. **Inputs:**
        i. recipient (address): Recipient address to receive the protocols fees.
        ii. amount0Requested (uint128): max amount of token0 to send (will be 0 if only collecting fees in token1)
        iii. amount1Requested (uint128): max amount of token1 to send (will be 0 if only collecting fees in token0)
    b. **Outputs:**
        i. amount0 (uint128): actual amount of token0 protocol fees collected.
        ii. amount1 (uint128): actual amount of token1 protocol fees collected.
    c. **State Mutability:** nonpayable
    d. **Description:** Allows the pool's governance to collect the accrued protocol fees in token0 and token1.

d) UniSwapV3PoolDeployer.sol:

ABI Generation on Remix:

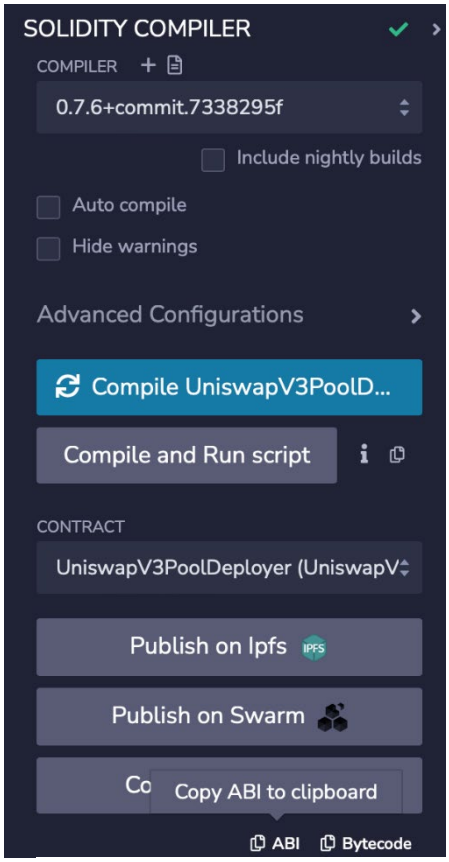

Figure 1.3: Compiled PoolDeployer sol

We have successfully compiled the UniswapV3PoolDeployer.sol with Remix. We copied the ABI file and analysed it below:

Function explained:

1) **parameters():**
    a. **Inputs:** None

b. **Outputs:**
   i. factory(address): address of factory contract that deployed the pool
   ii. token0(address): address of the first token in the pool
   iii. token1(address): address of the second token in the pool
   iv. fee(uint24) the fee from each trade within the pool
   v. tickSpacing(int24): tickSpacing of the pool
c. **State Mutability:** view
d. **Description:** This function returns important parameters regarding a deployed pool.

Elaboration of functionality of a single function (collect function in UniswapV3Pool):

```
/// @inheritdoc IUniswapV3PoolActions
function collect(    infinite gas
    address recipient,
    int24 tickLower,
    int24 tickUpper,
    uint128 amount0Requested,
    uint128 amount1Requested
) external override lock returns (uint128 amount0, uint128 amount1) {
    // we don't need to checkTicks here, because invalid positions will never have non-zero tokensOwed{0,1}
    Position.Info storage position = positions.get(msg.sender, tickLower, tickUpper);

    amount0 = amount0Requested > position.tokensOwed0 ? position.tokensOwed0 : amount0Requested;
    amount1 = amount1Requested > position.tokensOwed1 ? position.tokensOwed1 : amount1Requested;

    if (amount0 > 0) {
        position.tokensOwed0 -= amount0;
        TransferHelper.safeTransfer(token0, recipient, amount0);
    }
    if (amount1 > 0) {
        position.tokensOwed1 -= amount1;
        TransferHelper.safeTransfer(token1, recipient, amount1);
    }

    emit Collect(msg.sender, recipient, tickLower, tickUpper, amount0, amount1);
}
```

Figure 2.0: collect function in UniswapV3Pool

The 'collect' function in the UniswapV3Pool.sol is used to allow liquidity providers to collect fees owed to them from their liquidity positions. Specifically, to collect fees in token0 and token1 from a range within the pool. This function must only be called by the position owner to ensure only the authentic owner receives the fees.

This function can be called by owner of a liquidity position to collect the total fees accumulated from trading activities within their specified price range. These fees are specified in the form of token0 and token1. However, the fees owed will not be calculated automatically, this can be solved by using the 'mint' or 'burn' function of any liquidity amount to recalculate the fees owed.

Liquidity providers can specify how much fees to collect from token0, and token 1. They can choose to collect all the accumulated fees by passing a value more than actual token owed, or they can specify an actual amount to be collected from each token.

Call Graph Discussion:

A call graph is a visual diagram representation of the relationships between functions within a program or between different programs within an environment. These sections will showcase how each important function relates to each other and how their individual functions call upon other functions. We put our focus on the UniSwapV3Factory.sol and UniSwapV3Pool.sol because the other 2 functions are only referring to script or account that is used to deploy contracts to the blockchain, or a low-level function for delegate call.

A) High-level call graph of UniswapV3:

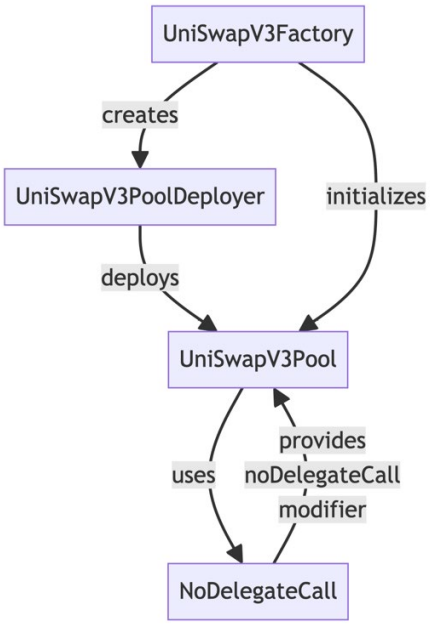

Figure 3.0: Call graph for UniswapV3

Main functionality: UniSwap is a decentralized finance protocol on the Ethereum blockchain that allows users to swap between multiple cryptocurrencies.

Explain call graph's relationship:

- **UniSwapV3Factory.sol:** This contract creates new pool by calling the deployer and then initializes it
- **UniSwapV3Deployer.sol:** This contract deploys new UniSwapV3Pool contracts
- **UniSwapV3Pool.sol:** This contract represents a pool in the whole decentralized protocol. It uses the noDelegateCall modifier to protect methods from being called from delegateCall
- **NoDelegateCall.sol:** modifier for add an extra layer of security

B) UniswapV3Factory.sol:

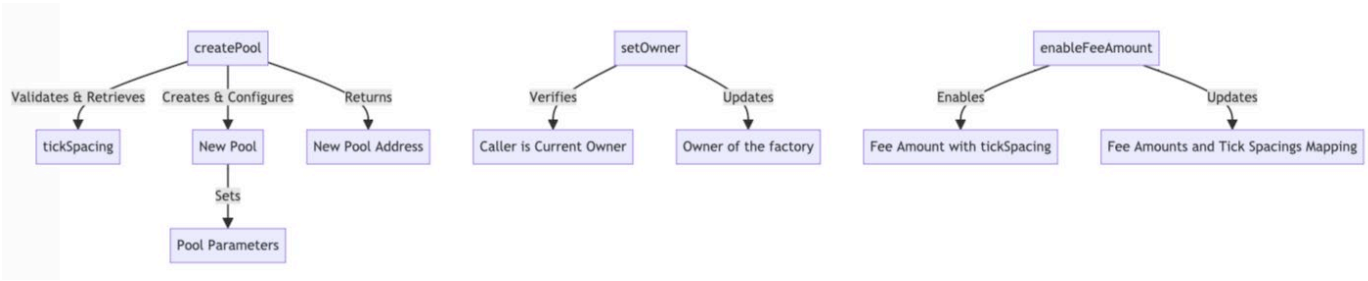

Figure 3.1: Call graph for UniswapV3Factory

Main functionality: This solidity file creates new trading area known as pools for different types of currencies. It also keeps track of the owner of the setup for each pool. Lastly, it oversees the collected fees for each trade.

Explain Individual function:

- **createPool():** This function is in charge of create a new liquidity pool for the exchanged tokens with a specialized fee rate.
- **setOwner():** This function is used to update the owner of the contract and it can only be called by the current owner of such contract.
- **enableFeeAmount():** This function allows new fee for the Uniswap pool setup with associated tick spacing. This function also supports future pool creations.

Explain call graph's relationship:

- **createPool():** This function validates to make sure input tokens and fee rate existed and are valid. It also retrieves the appropriate tick spacing associated with the provided fee. Tick spacing is the fixed distance between ticks in price range. With successful validation, the function deploys new pool contract with specified parameters and returns address of the newly setup pool.
- **setOwner():** This function is used to update owner for the factory contract. This function makes sure the caller is the current owner before the updating process.
- **enableFeeAmount():** This function enables fee amount for specified Uniswap pools along with custom tick spacing. Once the amount is enabled, it cannot be removed. The function also updates the mappings that happens inside factory contract to record the relationship between fee and its tick spacing which allows for future pool creation.

C) UniswapV3Pool.sol:

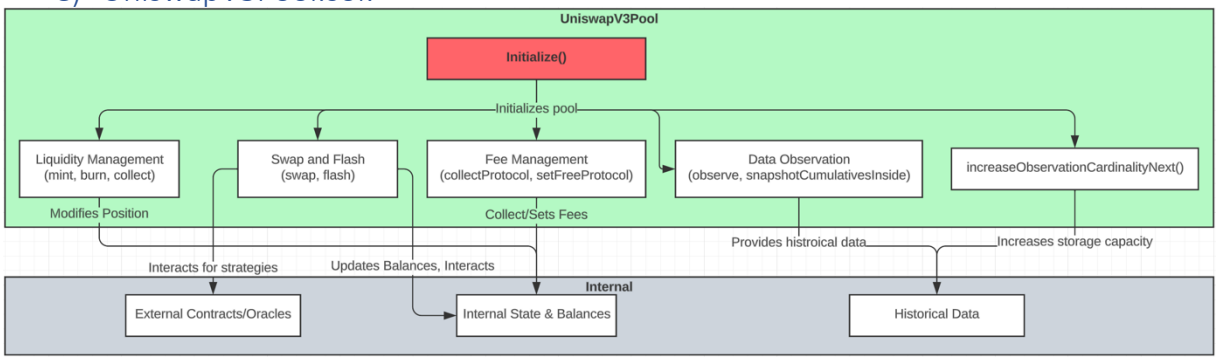

Figure 3.2: Call graph for UniSwapV3Pool

Main functionality: This solidity file initializes the smart trading platform in UniSwapV3 where users can swap and trade cryptocurrencies, add cryptocurrencies to earn fees, change settings from specific price ranges.

Explain Individual function: The graph is a high-level call graph where functions performing an aspect of the core functionality are grouped together. In a Uniswap pool, each trading pair will consist of 2 different tokens which are referred to as token0 and token1.

- **Initialize():** This function sets the initial price for the pool. After the initialization, other functions can be called and processed.
- **Liquidity management:**
  - **Mint():** this function adds liquidity for a position in the specified/initialized pool
  - **Burn():** this function removes liquidity from a position and takes care of tokens owed
  - **Collect():** this function collects tokens owed to a position depending on fees created from swaps and liquidity provided
- **Swap and Flash:**
  - **Swap():** this function allows swapping between token0 and token1 in the pool
  - **Flash():** this function allows borrowing of tokens within the pool as long as they are paid back within the same transaction with a fee
- **Fee Management:**
  - **collectProtocol():** this function collects the protocol fees assigned to the pool
  - **setFeeProtocol():** this function sets the protocol fee rates for token0 and token1
- **Data Observation:**
  - **Observe():** this function shows cumulative tick and liquidity data at specific timestamp
  - **snapshotCumulativeInside():** this function returns a snapshot of tick and liquidity data at a specific price range.
- **increaseObservationCardinalityNext():** this function alters the capacity of the pool to store extra price and liquidity observations

Explain call graph's relationship:

- **Initialize -> Liquidity Management, Swap and Flash, Fee Management, Data Observation, increaseObservationCardinalityNext:** the initialize() function basically sets the configuration for the pool which allows the other lower functions to function correctly. It makes sure that the pool configuration is valid and ready for liquidity and trading action to be added and performed.
- **Liquidity Management -> Internal State and Balances:** Internal state is referring to variable and data stored on the blockchain that represents the contract whereas balances are referring to the amounts of each token held in the specified pool. All functions in this group modify the liquidity positions and account for tokens inside the pool.
- **Swap and Flash -> Internal State and Balances, External Contracts:** this group interacts with the internal state to update token balances. It can interact with external contract for strategies and price info.
- **Fee Management -> Internal State and Balances:** this group handles the collection and sets the fee for protocol by observing and using the internal state and balances of such pool
- **Data Observation -> Historical Data:** this group of function can view the past prices and liquidity data. External contracts can query this function to view past state information.

- **increaseObservationCardinalityNext -> Historical Data:** this function increases the pool's capability to store more past price and liquidity data, ensuring the pool can maintain a sufficient history of its state for potential queries.

Inheritance Graph Discussion:

In Solidity, the inheritance diagram (or inheritance hierarchy) can reflect the following information:

In this part of the report, we will discuss about the concept of inheritance in Solidity which is involved in creating new contracts that are based on an existing contract and its properties. With inheritance, we are able to reuse code, enhance functionality while simplifying complex contract structures. We will be discussing the topics below which are related to inheritance graph in Solidity.

a) **Inheritance Relationships:** The diagram clearly shows the inheritance relationships between contracts, indicating which contract inherits from another.
b) **Code Reusability:** Through inheritance, Solidity allows developers to reuse existing code. The inheritance diagram can help developers and auditors understand the origins and reuse of code.
c) **Function and State Variable Overriding: In** Solidity, a derived contract can override functions and state variables of its parent contract. The inheritance diagram can help identify these overriding scenarios.
d) **Function Visibility and Access Control:** The diagram can assist developers and auditors in understanding which functions remain visible in derived contracts and whether their access permissions have been altered.

a) Inheritance Relationships

Inheritance relationships in Solidity are one of the most important aspects of contract interactions within a blockchain ecosystem. It shows the relationship between one contract and another, either as a parent or a child of a derived contract. In terms of a parent contract(base contract), it has a common properties and functionality that can be inherited by a child contract(derived contract).

```
contract UniswapV3Factory is IUniswapV3Factory, UniswapV3PoolDeployer, NoDelegateCall {
    /// @inheritdoc IUniswapV3Factory
    address public override owner;

    /// @inheritdoc IUniswapV3Factory
    mapping(uint24 => int24) public override feeAmountTickSpacing;
    /// @inheritdoc IUniswapV3Factory
    mapping(address => mapping(address => mapping(uint24 => address))) public override getPool;
```

Figure 4.0: contracts/UniswapV3Factory.sol inheritance

In the figure above, UniswapV3Factory.sol is a child contract which inherits from 3 different contract, IUniswapV3Factory, UniswapV3PoolDeployer, and NoDelegateCall.

UniswapV3Factory.sol contract now has access to the public and internal properties, state variables and functions of the parent contracts, this creates a modular design which allows for extensive functionality without modifying the base contract.

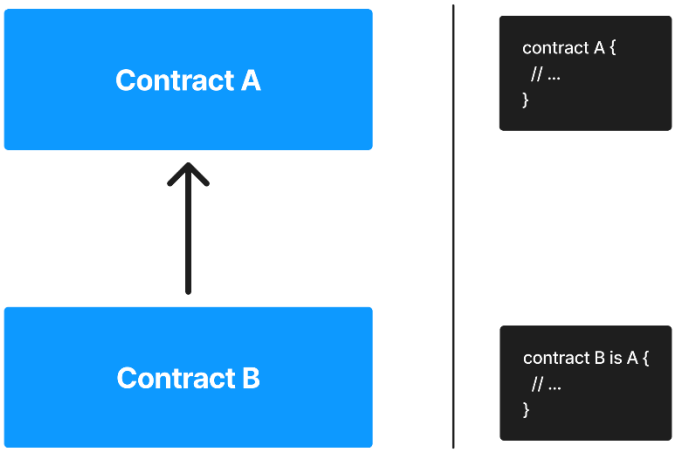

Figure 4.1: Inheritance Example

The 'is' keyword in contract B tells us that it is derived from the contract A. (Solidity - Inheritance 2020). Additionally, functions that can be overridden must be declared as 'virtual' and when a child contract is overriding the function, it must use 'override' to override the parent's 'virtual' function. (Solidity by Example n.d.)

Inheritance order is also important when inheriting from other contracts. They are ordered from "most base-like" to "most derived" if there are multiple inheritance in a contract. (Solidity by Example n.d.)

1. Single Inheritance

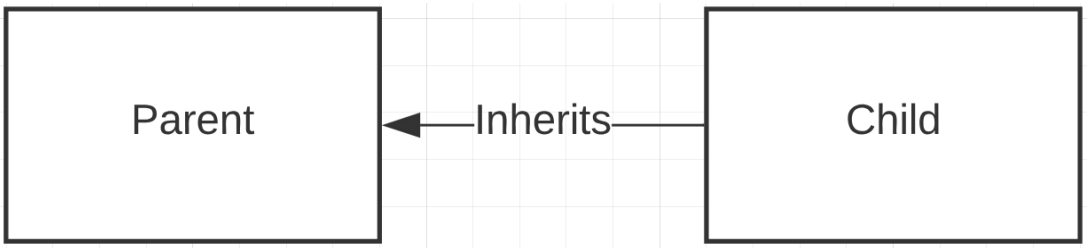

Figure 4.2: Single Inheritance Example

In Single level inheritance, the contract's properties, functions, and variables are only inherited by child one contract. As you can see from the above example, this is the simplest form of inheritance where the child contract(derived) inherits all the public and internal properties and functions of the base contract(parent). This results in a one-to-one relationship with the parent and child contract.

2. Multilevel Inheritance

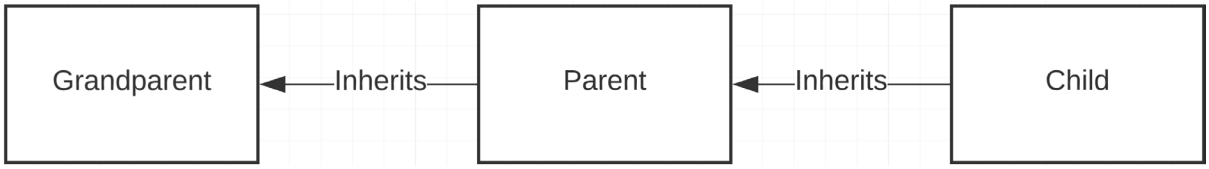

Figure 4.3: Multilevel Inheritance Example

For multilevel inheritance, it is similar to single inheritance. A contract can inherit from a parent, who inherits from another grandparent contract. Essentially it forms a chain relationship from the grandparent to parent to child contract. In the above example, the Child contract has access to both the 'Grandparent' and 'Parent' contracts' functions as it inherits the Parent contract which inherits the Grandparent contract.

3. Hierarchical Inheritance

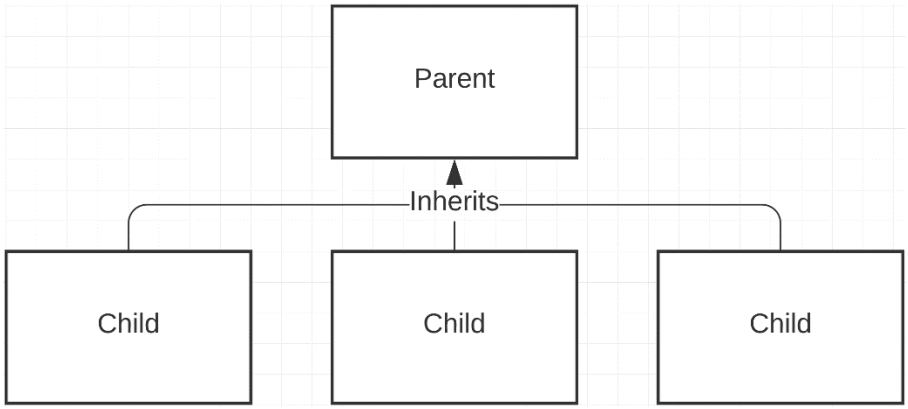

Figure 4.4: Hierarchical Inheritance Example

Hierarchical inheritance is also similar to single inheritance except one parent contract can be inherited by many child contracts, creating a one-to-many relationships. In the above example, all the child contracts are derived from the parent contract and have access to its properties, but none of the child classes have any relations to one another.

4. Multiple Inheritance

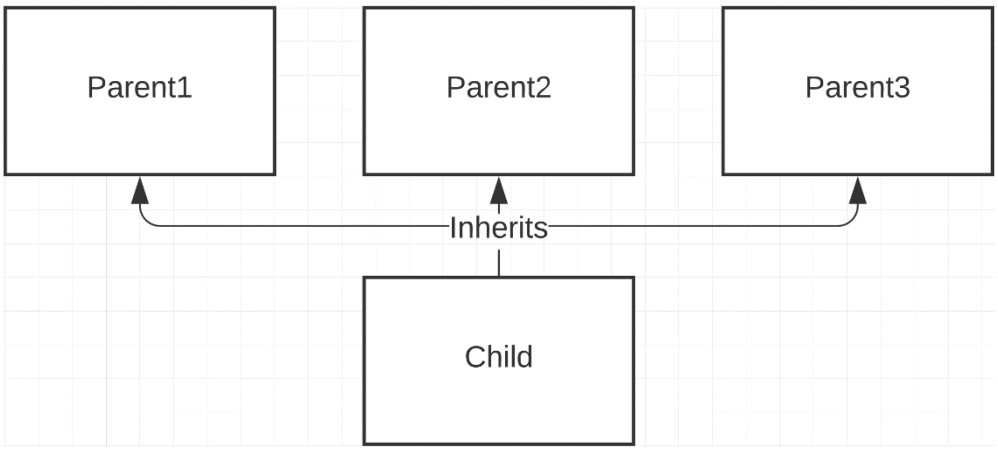

Figure 4.5: Multiple Inheritance Example

In multiple inheritance, the child contract can inherit properties from many parents contract creating a many-to-one relationship between the parent contracts and the child contract. When the child

contract inherits from the parent contracts, it inherits all the functions and state variables from all the inherited parent contracts. From the above example, the Child contract inherits from Parent1, Parent2, and Parent3, gaining access to all their functions.

Summary of Inheritance Types:

1. **Single Inheritance:** One parent, one child
2. **Multilevel Inheritance:** Chain of inheritance through multiple levels
3. **Hierarchical Inheritance:** One parent, many children
4. **Multiple Inheritance:** Many parents, one child

b) Code Reusability:

In smart contract development, code reusability plays one of the most important advantages when using inheritance in Solidity. Blockchain developers can utilize code reusability to build smart contracts by just building on top of existing smart contracts, this helps to reduce redundancy and human errors when it comes to developing smart contracts. One of the main ways to utilize code reusability is through inheritance.

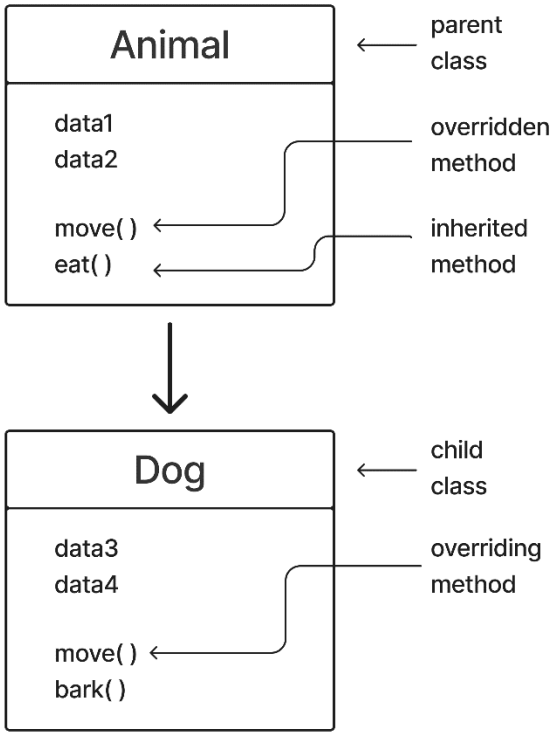

Figure 4.6: Code Reusability

Code Reusability Using Inheritance

As shown in figure 5.6, code reusability can be done using inheritance, a contract can access the state variables and functions of another contract and reuse the existing functions and variables. (What is Smart Contract inheritance? n.d.)
This gives a lot of advantages such as:

1. **Avoids Duplications and Errors:** Instead of rewriting the same functions/variables when creating new smart contracts, developers can just inherit these properties from a preexisting parent contract and pass it onto multiple child contracts. Not only would this accelerate the process of creating new smart contracts, but it also helps to avoid errors in code, as the pre-existing code is already proven to be reliable and most likely free of errors/bugs.

2. **Improve Maintainability:** It also helps with maintainability of the code, as changes will only have to be made to the parent contract instead of all the created contracts. If there are updates in the future, it will be easier for developers to update/maintain the code.

3. **Improve Extensibility:** Child contracts that inherits the functions from a parent contract can extend the inherited functions to allow for more complex functionality and features, while reusing the base code from the parent contract.

4. **Improve Consistency:** Code reusability can improve consistency as all the child contracts have consistent implementation of the inherited properties which is important when it comes to ERC20/ERC721 token standards. *(ERC-721 Non-Fungible Token Standard n.d.)*

Inheritance Graph in Understanding Code Reusability

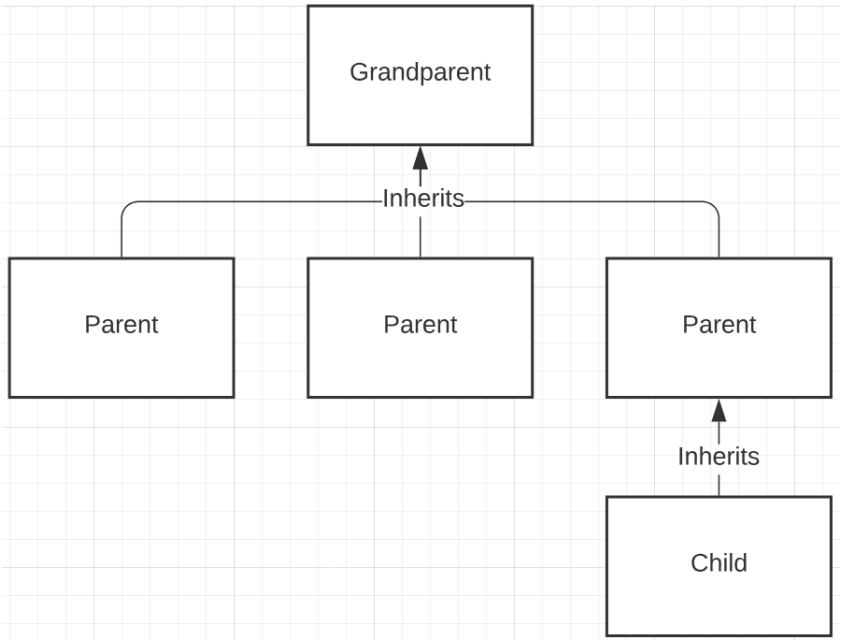

Figure 4.7: Inheritance Diagram

Inheritance graph helps to quickly shown where functions and state variables are inherited from, this can help developers to understand, maintain, and extend the code functionality by looking at the parent contract's functions. Additionally, developers can also identify the commonly reused functions and identify ways to improve and optimize the code.

For example, in figure 5.7, inheritance graph can also show the dependencies of all the contracts, developers can carefully modify smart contracts by identifying the relationships and dependencies of the contracts. This will ensure that changes in one contract would not affect another contract. In figure 5.7, developer will know that modifying Child contract would not affect the Parent or Grandparent contracts, as these 2 contracts do not depend on the Child contract.

Conclusion:

Code reusability is a power method when developing smart contracts in Solidity as it significantly improves the maintainability and consistency of the contracts. Together with the inheritance graph, developers can effectively develop and manage the smart contracts.

c) Function and State Variable Overriding:

In Solidity, we can use overriding with inheritance. Derived contracts can override functions and state variables of the parent contracts. This allows the child contracts to be flexible with their implementation of the inherited functions and state variables. The child contracts can use overriding to extend these functionalities to have different behaviour from the parent contracts without modifying the original code.

In terms of inheritance graph in Solidity, inheritance graph can help in representing the relationships between the parent and child contracts which can be important to developers when understanding how the functions and state variables are overridden from the parent contract to the child contract. Developers can also backtrack the origins of the source code and identify what is being overridden and understand the data flow and control of the contracts ensuring integrity and security of the smart contracts.

Type of Overriding

Function Overriding

A function that can be overridden will have the keyword “virtual” to allow the derived contracts to override its behaviour. The keyword “override” will be used by the derived contracts to override the base function. (Lifanova 2020). In Solidity, a child contract can override and have a new implementation of a function in the parent contract.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.22;

contract A {
    function foo() public virtual {}
}

contract B {
    function foo() public virtual {}
}

contract Child is A, B {
    function foo() public override(A, B) {}
}
```

Figure 4.8: Function Overriding

In the example in Figure 5.8, the Child contract can be seen overriding the 'virtual' foo functions from contract A and contract. B. The child contract can modify and have its own implementation of the foo function from contract A and B.

State Variable Overriding

Similarly, to function overriding, state variable overriding can be done in child contract. To achieve this, the variable must have the same name. However, the approach will be different to Function Overriding. State variables in Solidity can't be directly overridden by redeclaring the variables, instead it is done through shadowing by declaring the same variable name as the parent contract with a different value.

```
contract A {
    int public x = 1;

    function getValue() public view returns (int value) {
        return x;
    }
}

contract B is A {
    constructor() {
        x = 2;
    }
}
```

Figure 4.9: State Variable Overriding(shadowing)

In the example in Figure 5.9, contract A has a x value of 1 and a getValue function that returns the value of x. In contract B, which is derived from contract A, it also has a x variable in its constructor which is inherited from contract A. The overridden is done in the constructor in contract B, when the function getValue is called from contract B like this B.getValue, it will return the value 2. 'override' keyword is not used here as we are only modifying the value. (Academy 2023)

Example in Uniswap

```
contract UniswapV3PoolDeployer is IUniswapV3PoolDeployer {
    struct Parameters {
        address factory;
        address token0;
        address token1;
        uint24 fee;
        int24 tickSpacing;
    }

    /// @inheritdoc IUniswapV3PoolDeployer
    Parameters public override parameters;
```

Figure 4.10: UniswapV3PoolDeployer.sol overriding.

Using the UniswapV3PoolDeployer as an example, the contract UniswapV3PoolDeployer is derived from IUniswapV3PoolDeployer. Overriding is used here to modify and extend its functionality. The 'parameters' state variable is declared as 'public override' which is overriding the state variable from the parent interface IUniswapV3PoolDeployer. The reason it uses the 'override' keyword here compared to figure 5.9 where it doesn't use 'override', is because in figure 5.9, the implementation of the state variable 'x' is provided. However, in figure 5.10, 'parameters' from the interface IUniswapV3PoolDeployer doesn't have an implementation so 'override' keyword will need to be used to provide an implementation for the state variable 'parameters.

Conclusion:

Function and state variable overriding can be done in Solidity to achieve code reusability and modular design. They allow developers to reuse the code and modify the behaviours of the parent contract to extend its functionality in the child contract, which is important and effective when creating smart contracts.

d) Function Visibility and Access Control:

Function visibility and access control is important in Solidity to control how and which functions in the contracts can be accessed or called. It is important for developers and auditors to understand the functions' visibility and access to carefully identify any functionality implications and security risks.

Types on Function Visibility

Public

Public visibility allows the function to be accessed by everyone outside the contract. The default visibility of a function will be set to public if not defined. This might incur security risks as it can expose vulnerabilities and allow for unintended or unauthorized changes.

Private

Privat functions visibility can only be accessed inside the contract itself and cannot be inherited by other contracts. This can help with lowering security risks as the access is limited to within the contract itself.

Internal

Internal visibility is similar to private visibility, the function can only be accessed within the contract itself, however 'internal' functions can also be accessed by derived contracts.

External

External functions can only be called by contracts outside of the containing contract. This can be used when developer expect the functions to only be called outside the containing contract. External function uses less gas compared to public as they can directly read from calldata whereas 'public' copies the entire arguments to memory.

Access Control

Function visibility and access control are used to manage security and functionality. The visibility of the functions derived from the parent contract can affect how these inherited functions are accessed/overridden by the derived contracts.

```
    function deploy(
        address factory,
        address token0,
        address token1,
        uint24 fee,
        int24 tickSpacing
    ) internal returns (address pool) {
        parameters = Parameters({factory: factory, token0: token0, token1
        pool = address(new UniswapV3Pool{salt: keccak256(abi.encode(token
        delete parameters;
    }
}
```

Figure 4.11: UniswapV3PoolDeployer.sol internal deploy function.

For example, in the UniswapV3PoolDeployer.sol, the deploy function can only be accessed by derived contracts and within the UniswapV3PoolDeployer contract.

Important of Function Visibility and Access Control

Understanding the importance of function visibility and access control is needed for auditors to identify any potential and unwanted security risks such as unwanted external calls or state changes/modifications. Auditors need to ensure that the function visibility and access control aligns with the intended purpose of the contracts' functionality and security control.

By looking at inheritance graph, auditors can effective and efficiently identify each functions visibility from the derived contracts and their access permissions. This can help with streamlining the auditing process.

Conclusion:

In smart contracts, function visibility and access control are important for securing the functional design of the smart contracts in Solidity. Setting an appropriate access and visibility can help developers to create a secure system and prevent unwanted and unauthorized access.

References:

Academy, S 2023, *#100DaysOfSolidity Mastering Shadowing Inherited State Variables in Solidity #24*, Medium, viewed 31 October 2023, <https://medium.com/@solidity101/100daysofsolidity-mastering-shadowing-inherited-state-variables-in-solidity-24-e4673ed6cd4b#:~:text=In%20Solidity%2C%20when%20a%20contract>.

Danill, D 2022, *Oracle*, Uniswap Docs, viewed 3 November 2023, <https://docs.uniswap.org/concepts/protocol/oracle>.

Danill, D 2022, *Range Orders*, Uniswap Docs, viewed 3 November 2023, <https://docs.uniswap.org/concepts/protocol/range-orders>.

Danill, D 2022, *Swaps*, Uniswap Docs, viewed 3 November 2023, <https://docs.uniswap.org/concepts/protocol/swaps>.

ERC-721 Non-Fungible Token Standard n.d., ethereum.org, <https://eips.ethereum.org/EIPS/eip-721>.

Garnett, A 2023, *Uniswap (UNI): What it is, Pros and Cons, FAQs*, Investopedia, viewed 2 November 2023, <https://www.investopedia.com/uniswap-uni-definition-5217463>.

Kant, A 2022, *Concentrated Liquidity*, Uniswap Docs, viewed 3 November 2023, <https://docs.uniswap.org/concepts/protocol/concentrated-liquidity>.

Lifanova, K 2020, Solidity override vs. virtual functions, Upstate Interactive, viewed 31 October 2023, <https://medium.com/upstate-interactive/solidity-override-vs-virtual-functions-c0a5dfb83aaf>.

Solidity - Inheritance 2020, GeeksforGeeks, viewed 27 October 2023, <https://www.geeksforgeeks.org/solidity-inheritance/>.

Solidity by Example n.d., solidity-by-example.org, viewed 27 October 2023, <https://solidity-by-example.org/inheritance/>.

Uniswap 101: What is Uniswap? 2023, Uniswap Labs Blog, viewed 2 November 2023, <https://blog.uniswap.org/what-is-uniswap>.

What is Smart Contract inheritance? n.d., Alchemy Docs, viewed 30 October 2023, <https://docs.alchemy.com/docs/smart-contract-inheritance>.