





## **FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY**

### **BMIS2003 Blockchain Application Development**

#### **Assignment**

**Academic Session: 202309 (Sem 2, 2023/24)**

#### **Team Members:**

<b>No</b>	<b>Student Name</b>	<b>Student Photo</b>	<b>Student ID</b>	<b>Programme</b>	<b>Tutorial Group</b>
<b>1</b>	<b>Tang Sharren</b>		<b>2101086</b>	<b>RDS3S1</b>	<b>3</b>
<b>2</b>	<b>Harry Chiong Hau Kong</b>		<b>2200656</b>	<b>RSD2S3</b>	<b>1</b>
<b>3</b>					
<b>4</b>					
<b>5</b>					

### **Coursework Declaration**



Academic Session : 202309

Course Code & Title : BMIS2003 Blockchain Application Development

#### **Declaration**

I/We confirm that I/we have read and shall comply with all the terms and condition of TAR UMT's plagiarism policy.

I/We declare that this assignment is free from all forms of plagiarism and for all intents and purposes is my/our own properly derived work.

No	Student Name	Student ID	Signature
1	Tang Sharren	21WMR01086	
2	Harry Chiong Hau Kong	22WMR00656	
3			
4			
5			

Date : 3 January 2023

## Table of Contents

<b>1.0 Introduction.....</b>	<b>4</b>
<b>2.0 Policy / Business Rules.....</b>	<b>5</b>
<b>3.0 Contract Diagram.....</b>	<b>10</b>
<b>4.0 Overall Use Case Diagram.....</b>	<b>12</b>
<b>5.0 Activity Diagram.....</b>	<b>13</b>
<b>6.0 System Architecture Diagram.....</b>	<b>16</b>
<b>7.0 UI Design.....</b>	<b>17</b>
<b>References.....</b>	<b>36</b>

## 1.0 Introduction

In this project, we have created a Decentralized Application (DApp) that implements an Automated Market Maker (AMM) based on the XYK model which is also known the constant product formula.

The smart contract in this project is written in Solidity v8.2 (Solidity, 2016). The contract mainly aligned with Uniswap V1 that is a decentralized exchange protocol that uses an automated market maker (AMM) model (Uniswap, 2018). Uniswap V1 implements the AMM model with the constant product formula:  $x \cdot y = k$  where  $x$  and  $y$  are the amounts of the two tokens in the liquidity pool, and  $k$  is a constant value.

This project serves as a foundational study of AMM principles, drawing inspiration from the XYK model, where the product of reserves ( $k$ ) is consistently maintained, ensuring a dynamic equilibrium in liquidity within the pool. This mathematical model facilitates automatic price adjustments as users trade, providing a decentralized mechanism for setting fair and dynamic token prices (Viera, 2021).

The 2 tokens involved are designed in accordance with the ERC-20 standard, and feature two essential tokens: Flare and Zenix (*ERC20 - OpenZeppelin Docs*, 2017). These tokens play a pivotal role in the liquidity pool, and the contract encompasses three fundamental functions – ‘addLiquidity’, ‘removeLiquidity’, and ‘swap’. Each transaction undergoes rigorous validation, adhering to predefined conditions to uphold the integrity and security of the system. The smart contract and token contracts are deployed on a local blockchain with the help of Ganache and truffle. Ganache is a local blockchain simulator that enables fast and easy DApp development, testing, and deployment on the Ethereum network. In Ganache, transactions are processed instantly. Blocks are added without the need for mining or achieving consensus among a network of nodes. This simplicity allows developers to quickly iterate and test their smart contracts without the delays and complexities associated with a consensus protocol.

*(From Idea to Minimum Viable Dapp - How to Use Ganache to Enhance Your Auction Dapp - Truffle Suite, 2020)*

The user interface is crafted as a React web application, augmented by JavaScript, JSX and CSS (React, 2023). This interface prioritizes user experience, allowing users to seamlessly interact with the DApp. Users can explore the liquidity pool, review their token balances, assess their share in the liquidity pool, and perform key operations such as adding liquidity, removing liquidity, and swapping tokens.

## 2.0 Policy / Business Rules

The Solidity contract used in this project represents a simplified version of implementation of Automated Market Maker(AMM) based on the XYK model. This project specifically used Uniswap V1 as the main reference model for the main functionalities of AMM. In the XYK model ( $x * y = k$ ), the product of the reserves( $k$ ) is kept constant, where  $x$  and  $y$  represent the reserves which is the quantities of the two tokens in the pool. During deployment of the smart contract, owner of the smart contract and address of both ERC-20 tokens are needed.

### ERC-20 Token:

The contract initializes with a pair of ERC-20 tokens (token1 and token2) (*Creating ERC20 Supply - OpenZeppelin Docs*, 2017). Each token represents one side of the trading pair. The ERC-20 token contracts to be deployed are named as Flare and Zenix with the symbol of FLR and ZNX respectively. Generally, both of the token has a fixed total supply of 1000 tokens(initially owned by the owner), and allowing the owner to mint new tokens and burn existing ones. The token contracts support standard ERC-20 operations, including transfers, allowances, and burning, as the contracts is inheriting the contract from the ERC20 and ERC20Burnable extension (*ERC 20 - OpenZeppelin Docs*, 2017). Also, the ownership functionalities are implemented inheriting the contract from the OpenZeppelin Ownable contract, which ensures that certain functions can only be executed by the owner of the contract (*Ownership - OpenZeppelin Docs*, 2017). Therefore, during deployment, the initial owner can be set using the setOwner function.

### Liquidity Pool (Reserves):

Liquidity pool is represented by the reserves of the two tokens where the reserves refer to the quantities of each token in the liquidity pool. In this contract, the variables reserve1 and reserve2 store the quantities of token1 and token2, respectively. These reserves are crucial for determining the pricing and ratios of tokens in the liquidity pool, it represents the constant  $k$  in the XYK model. The product of reserves ( $k$ ) must be remained constant regardless of the tradings, hence preventing drastic shifts in token prices.

### Token Pricing & Ratio:

Pricing is automatic, based on the  $x * y = k$  market making formula which automatically adjusts prices based off the ratio of the reserves and the size of the incoming trade (Uniswap, 2018). As users trade, the reserves change, impacting the prices of tokens. The XYKPool contract doesn't explicitly store token pricing as a state variable because storing pricing as a state variable would require frequent updates and could lead to increased gas costs and storage consumption. Instead, token pricing is calculated dynamically within specific function, getPrice(). Calculating pricing dynamically allows the contract to provide real-time pricing information to users based on the most recent state of the reserves.

Referring to the documentation of uniswapv1 protocol on “Amount Bought” (Uniswap, 2018), the pricing of tokens within the pool can be simplified as this formula:

$\text{token1\_price} = \text{reserve2} / \text{reserve1}$  and  $\text{token2\_price} = \text{reserve1} / \text{reserve2}$  .

The getPrice calculate the expected output amount for a given input amount based on the XYK model.

If the token input is token 1, getPrice() used the formula as below,

$$\text{amountOut} = (\text{reserve2} * \text{amountIn}) / (\text{reserve1} + \text{amountIn})$$

Otherwise,

$$\text{amountOut} = (\text{reserve1} * \text{amountIn}) / (\text{reserve2} + \text{amountIn})$$

### **User Shares:**

The liquidity provided by users(liquidity provider) is represented by shares, and each user's share is tracked using the userShare mapping. This is often represented as LP tokens in other DEX platforms like Uniswap, Sushiswap and PancakeSwap (Lacapra,2023). Initially, shares of users are zero. User shares are maintained through minting and burning mechanisms during liquidity addition and removal operations. The XYK model ensures that the ratio of tokens in the pool is maintained, and user shares represent their portion of the total liquidity.

### **Liquidity Addition :**

Users can add liquidity to the pool by providing both tokens in a specified ratio. The contract owner is not allowed to add liquidity to the pool. Then, users receive a proportionate number of shares representing their ownership in the pool.

Generally, when liquidity is added, both reserves increase, the pricing of the tokens changes based on the new ratio of tokens. The XYK model maintains equilibrium by ensuring that the ratio between the two tokens in the pool stays balanced despite changes in reserves.

If one reserve increases more than the other, it affects the pricing, encouraging traders to perform swaps to restore balance through arbitrage opportunities.

The addLiquidity function allows users to add liquidity to the pool. The function takes amounts of token1 and token2 as parameters. The provided amounts are used to calculate the user's share of liquidity based on the  $x*y=k$  Uniswap v1 formula.

However, there's an additional check in the front end to ensure that the account that initiates liquidity addition is not the owner of the contract.

When a user adds liquidity to the pool, the contract calculates the user's share of liquidity based on the amounts of token1 and token2 provided by the user (Adams, 2020).

If the total liquidity in the pool is zero (indicating an empty pool), the contract initializes the user's share using a square root formula:

$$\text{userShareAmount} = \text{Math.sqrt}(\text{amountToken1} * \text{amountToken2})$$

If the pool is not empty, the contract calculates the minimum share required to maintain the existing ratio of reserves. This ensures that the new liquidity addition does not disrupt the equilibrium using the formula below:

$$\text{userShareAmount} = \text{Math.min}(\frac{(\text{amountToken1} * \text{totalLiquidity})}{\text{reserve1}}, \frac{(\text{amountToken2} * \text{totalLiquidity})}{\text{reserve2}})$$

);

After calculating the user's share, the contract updates the reserves by adding the provided amounts of token1 and token2. The reserves are updated as follows:

```
reserve1 = reserve1 + amountToken1  
reserve2 = reserve2 + amountToken2
```

The contract then mints the user's share by updating both the user's specific share and the total liquidity.

```
userShare[msg.sender] = userShare[msg.sender] + userShareAmount  
totalLiquidity = totalLiquidity + userShareAmount
```

As a result of a successful call of addLiquidity function, the user's share is minted, and the reserves are updated accordingly. Events are emitted to notify changes in user share, total liquidity, and reserves.

To make a successful call of the addLiquidity function, there are many conditions to be met, or else the transaction will be reverted with corresponding error. Firstly, the amount of tokens for adding liquidity(parameters passed to addLiquidity function) must be greater than 0. Then, the user who initiated this function must not be the contract owner. Also, the users must have balance of at least the amount of tokens they inputted to add liquidity. In order to add liquidity, the user must also approved the XYKPool contract to spend at least the amount of tokens for adding liquidity for both tokens. Lastly, the amount of tokens the user provided for adding liquidity must maintain a proper ratio with the existing reserves in the liquidity pool. This is to ensure that the user's addition of liquidity maintains a balanced ratio between the two tokens in the pool. If users were allowed to add liquidity without maintaining the ratio, it could lead to mispricing and inefficiencies in the market. Therefore, in the front, the suggested ratio is given to users after their first trial on liquidity addition is unsuccessful due to an imbalanced ratio of tokens. This way, users can add liquidity easily while maintaining the ratio.

These conditions collectively ensure that users provide valid and approved amounts of tokens, maintain the proper ratio in the liquidity pool, and receive a non-zero share of liquidity in return. If any condition is not met, the transaction is reverted to prevent invalid operations.

### **Liquidity Removal:**

Users can withdraw their share of liquidity from the pool and then receive proportional amounts of both tokens using the removeLiquidity function. The function calculates the amounts of token1 and token2 to be returned to the user based on their share of liquidity. The corresponding reserves and user shares are updated, and the tokens are transferred to the user.

The actual transfer of tokens to the user is done in the \_transferToken function, and the amounts are calculated based on the initial reserves. The amount of tokens withdrawn from the pool (transferred to the user) is calculated as a proportion of the total pool (reserve1) based on the user's share relative to the total liquidity where the user's share is also equivalent to the amount

of liquidity burned upon liquidity removal (Adams, 2020). The amount of tokens to be transferred to the user is calculated using the formula:

$$\begin{aligned}\text{amountToken1} &= (\text{reserve1} * \text{userShareAmount}) / \text{totalLiquidity} \\ \text{amountToken2} &= (\text{reserve2} * \text{userShareAmount}) / \text{totalLiquidity}\end{aligned}$$

Before burning the user share, the amount of tokens is checked to ensure that it is less than a reserve of the tokens. The user shares of both tokens and the total liquidity will also be updated using the ‘\_burnshare’ internal function. This involves subtracting the user's share from both the user's specific share and the total liquidity based on the formula below:

$$\begin{aligned}\text{userShare[msg.sender]} &= \text{userShare[msg.sender]} - \text{share} \\ \text{totalLiquidity} &= \text{totalLiquidity} - \text{share}\end{aligned}$$

The reserves of both token will then be subtracted from the amount of token transferred to the user with the formula below:

$$\begin{aligned}\text{reserve1} &= \text{reserve1} - \text{amountToken1} \\ \text{reserve2} &= \text{reserve2} - \text{amountToken2}\end{aligned}$$

The removeLiquidity function has specific conditions that need to be met for a successful execution. Failure to satisfy these conditions will result in the transaction being reverted with the corresponding error. Firstly, the user attempting to remove liquidity must have a non-zero share in the pool. If the user has no share, the function will revert. Also, the user who initiated this function must not be the contract owner. Then, the calculated amounts of each token that the user is entitled to must be less than or equal to the reserves and the actual balances of the contract in those tokens. Also, before burning the user's share, the function checks again to ensure that the user has a non-zero share. If the user has no share, the function will revert. These conditions are in place to ensure the safety and correctness of the liquidity pool operations.

Upon successful execution of the removeLiquidity function, three events are emitted to notify external entities of the changes:

- ShareChanged: Indicates the change in the user's share.
- LiquidityChanged: Indicates the change in the total liquidity of the pool.
- ReserveChanged: Indicates the change in reserves.

### **Token Swapping:**

When a user swaps one token for another, the XYK model ensures that the product of the reserves before and after the swap remains constant. The swap function is a crucial component in maintaining the balance of the pool and facilitating decentralized trading. Users can swap tokens, and the reserves are adjusted accordingly to reflect the new token ratios in the pool (Adams, 2020).

The \_swap function takes two parameters which are tokenIn represents the input token that the user wants to swap, and amountIn is the amount of tokenIn the user intends to swap. It determines the output token (tokenOut) and the corresponding reserves for both input and output tokens (reserveIn and reserveOut). This is based on whether tokenIn is equal to token1 or token2.



It calculates the expected amount of tokenOut that the user will receive based on the input amount (amountIn) and the current reserves using the XYK model formula:

$$\text{amountOut} = (\text{reserveOut} * \text{amountIn}) / (\text{reserveIn} + \text{amountIn});$$

It checks whether the calculated amountOut is within valid bounds, specifically whether it is less than or equal to the reserve of the output token (reserveOut). This is done by calling the `_checkAmount` function.

In terms of token transferring, it transfers the input amount (amountIn) from the user to the contract using `_transferTokenToContract`. Then, it transfers the calculated output amount (amountOut) from the contract to the user using `_transferTokenToUser`. The formula used to calculate the user balance after swapping token1 is as below:

$$\begin{aligned}\text{token1Balance} &= \text{token1Balance} - \text{amountIn} \\ \text{token2Balance} &= \text{token2Balance} + \text{amountOut}\end{aligned}$$

Based on the swap, the reserves of both input and output tokens also have to be updated. If input token is token1, it increases the reserve of token1 (reserve1) by the input amount (amountIn) and decreases the reserve of token2 (reserve2) by the output amount (amountOut).

Otherwise, it does the opposite. The detailed formulas for calculating the reserve when the tokenIn is token1 are stated as below:

$$\begin{aligned}\text{reserve1} &= \text{reserve1} + \text{amountIn} \\ \text{reserve2} &= \text{reserve2} - \text{amountOut}\end{aligned}$$

The swap function has specific conditions that need to be met for a successful execution. Firstly, the tokenIn address must not be zero. This ensures that the input token is valid and exists. The tokenIn address must be a valid token within the system. This is a check to ensure that the token is recognized by the smart contract. The amountIn provided for swapping should be greater than zero. This ensures that the user is actually swapping some amount of the token. This condition implies that the liquidity pool should not be empty for both token1 and token2. The reserves for both tokens should have some initial value or liquidity. The smart contract should have sufficient balance of tokenIn to facilitate the swap of amountIn. This ensures that the contract has enough tokens to perform the swap. The user must have approved the smart contract to spend at least amountIn of tokenIn on their behalf. Lastly, the user who initiated this function must not be the contract owner. This is essential for the contract to move the tokens during the swap. If any of these conditions are not met when calling the swap function, the transaction will revert, indicating that the swap cannot be executed due to the failure of one or more conditions.

Upon a successful execution of swap, a swapped event is emitted, providing information about the user, input token, output token, input amount, and output amount.

### 3.0 Contract Diagram

Clearer diagrams can be accessed through

<https://drive.google.com/file/d/1UTIA9rj4BaKZ3IWn73Vcc377OcjaHcy4/view?usp=sharing>

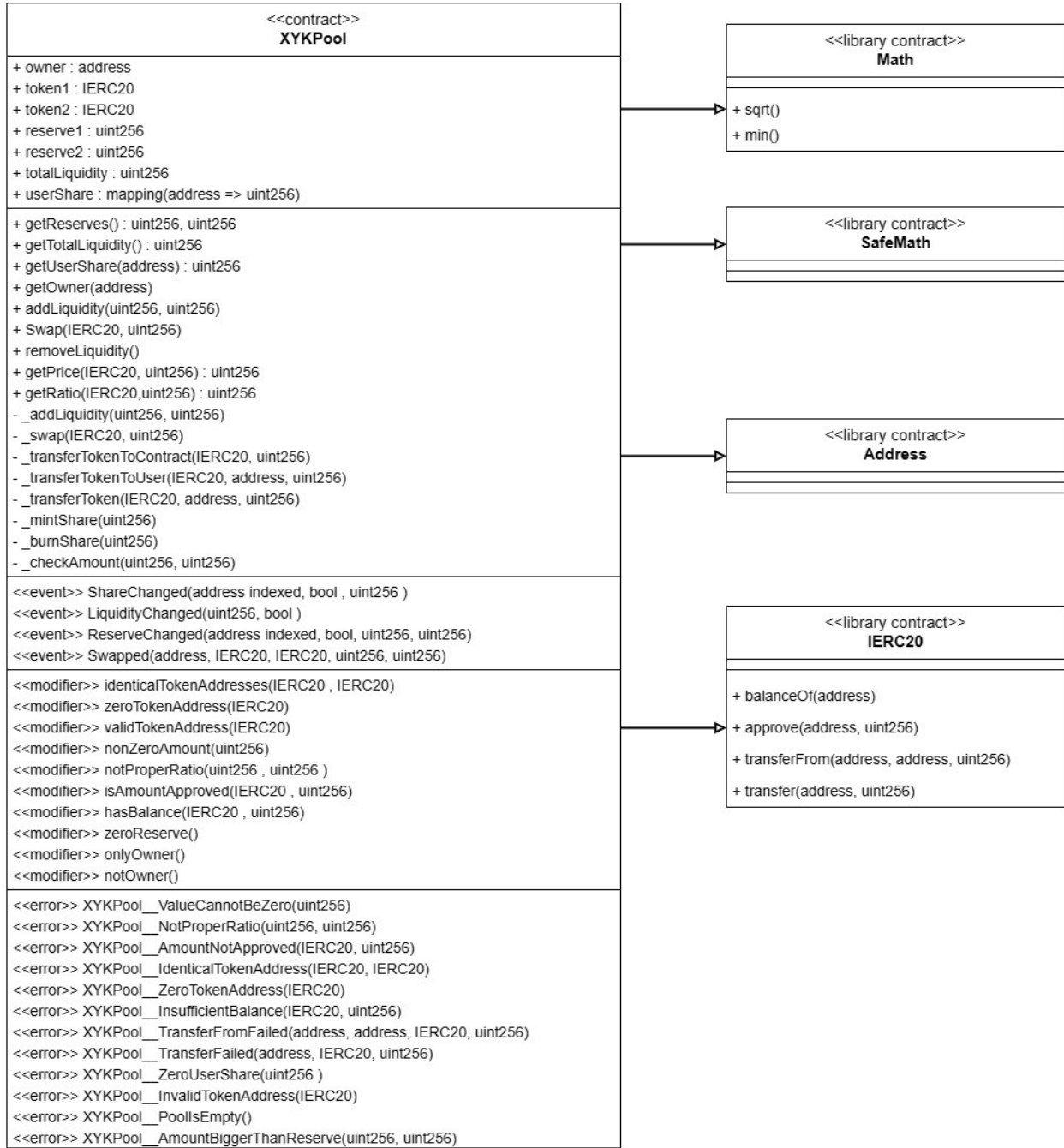


Diagram 3.1: XYKPool contract diagram

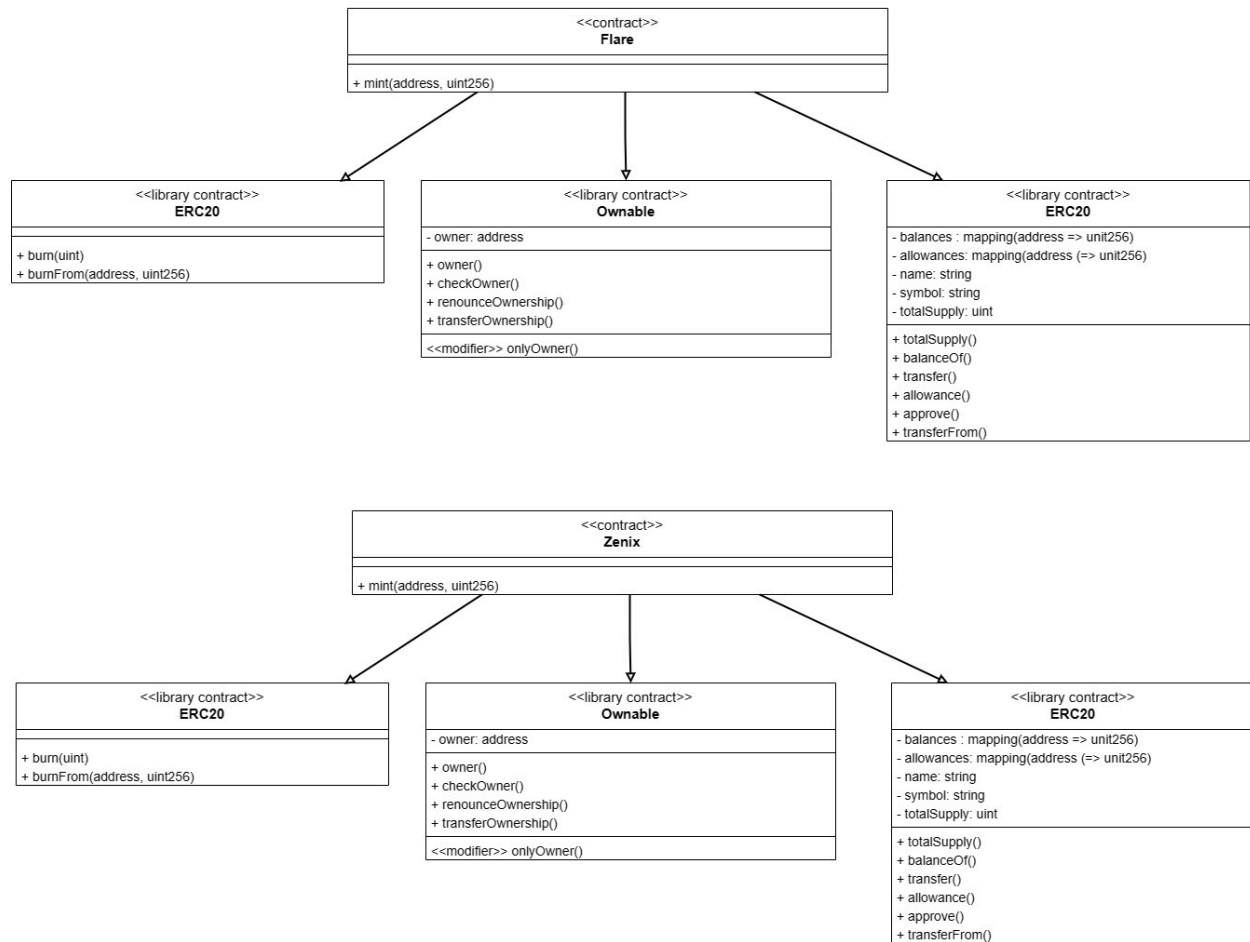


Diagram 3.2: Flare & Zenix token contract diagram

## 4.0 Overall Use Case Diagram

Clearer diagrams can be accessed through

<https://drive.google.com/file/d/1UTIA9rj4BaKZ3IWn73Vcc377OcjaHcy4/view?usp=sharing>

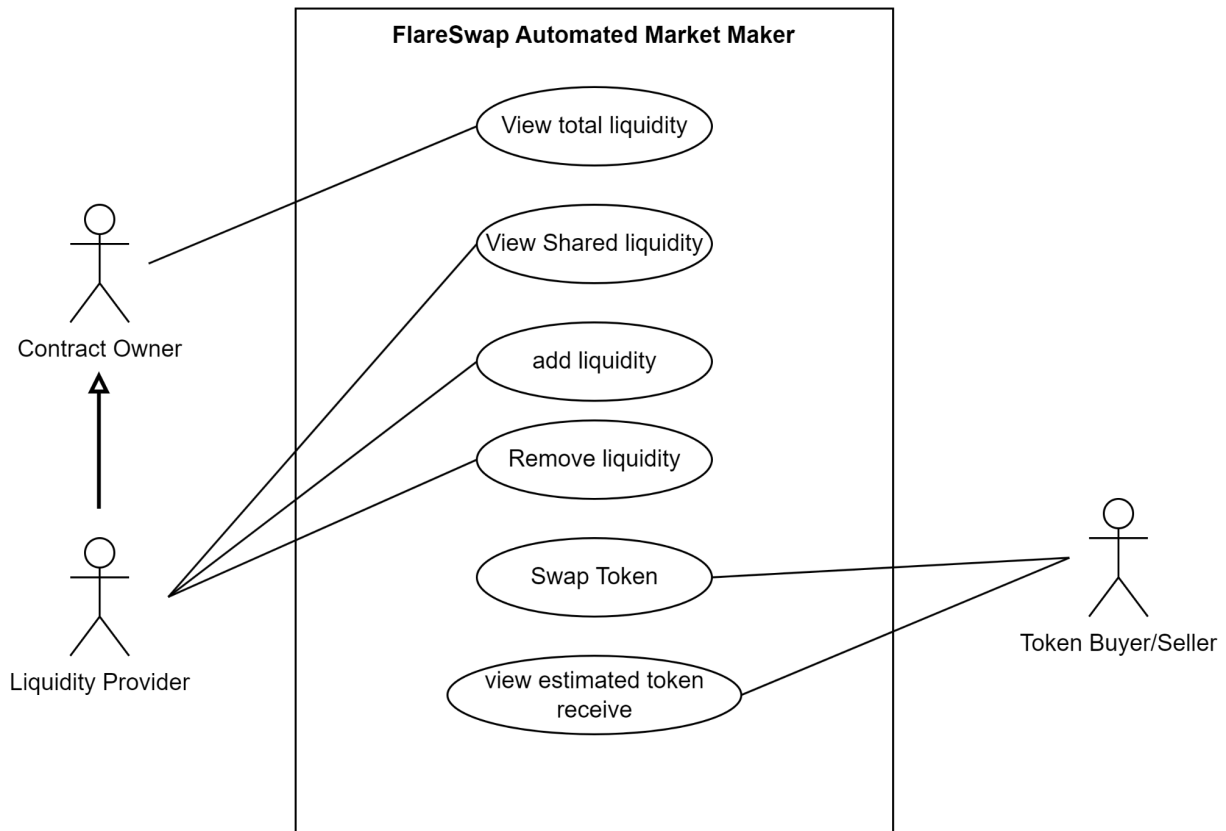


Diagram 4.1: FlareSwap AMM use case diagram

## 5.0 Activity Diagram

Clearer diagram can be viewed through

<https://drive.google.com/file/d/1UTIA9rj4BaKZ3IWn73Vcc377OcjaHcy4/view?usp=sharing>

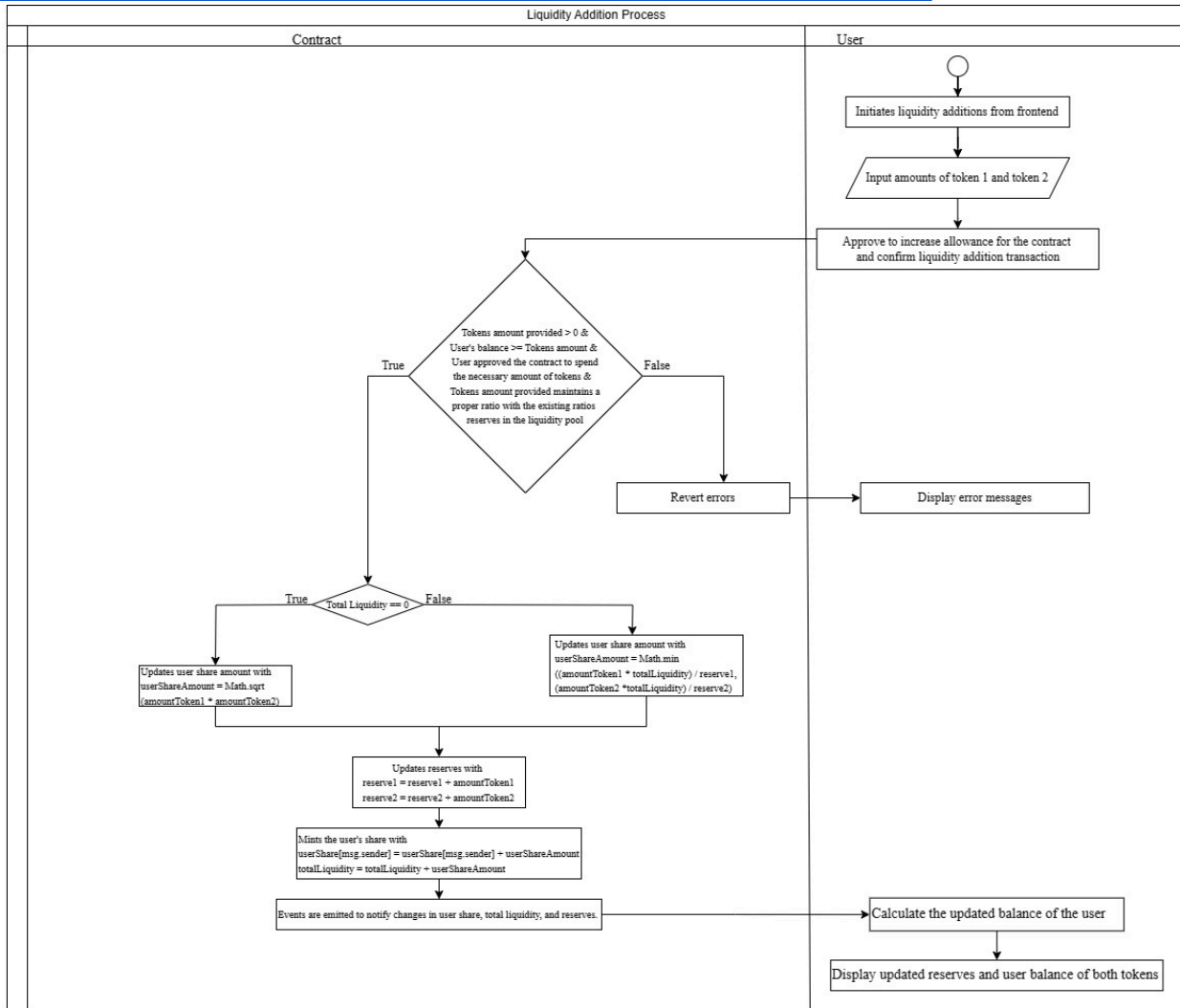


Diagram 5.1: Liquidity Addition Process Activity Diagram

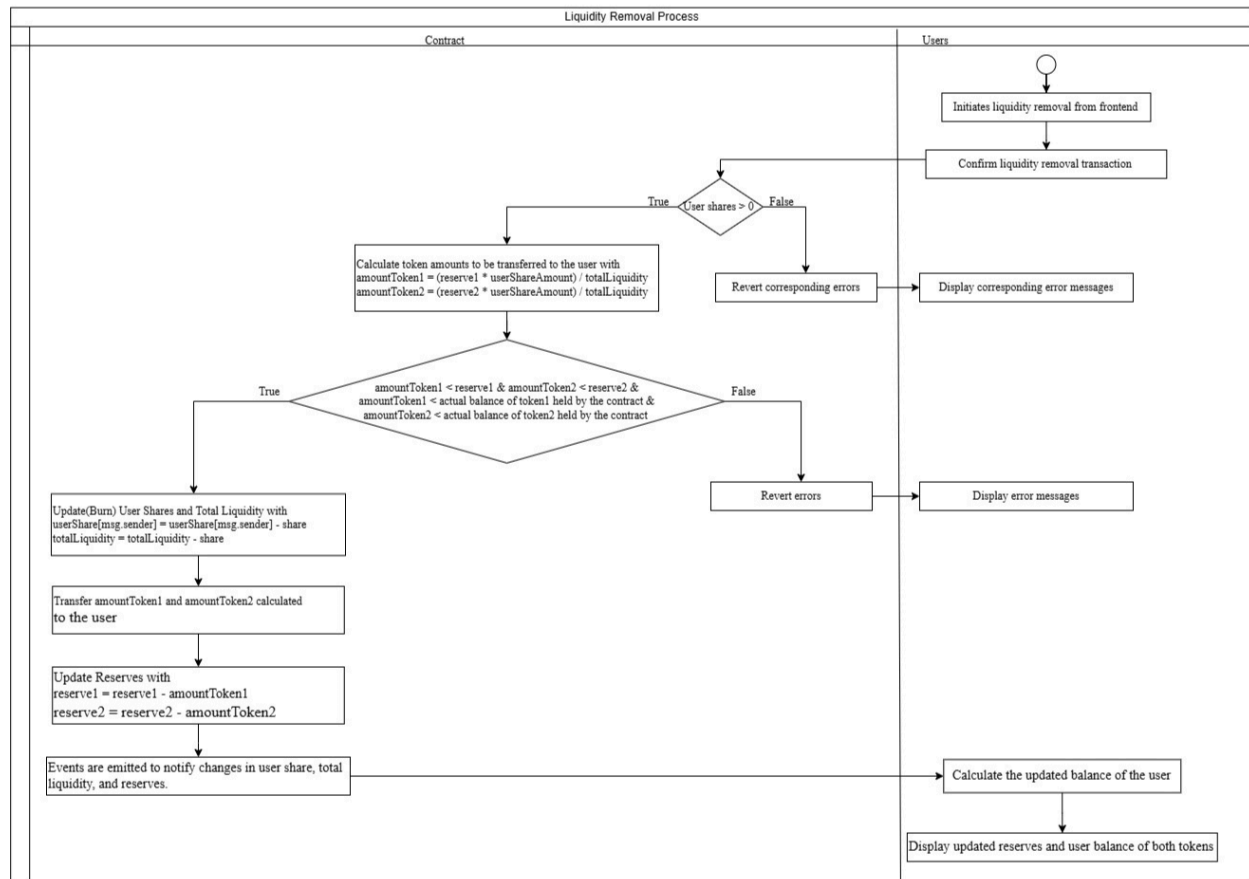


Diagram 5.2: Liquidity Removal Process Activity Diagram

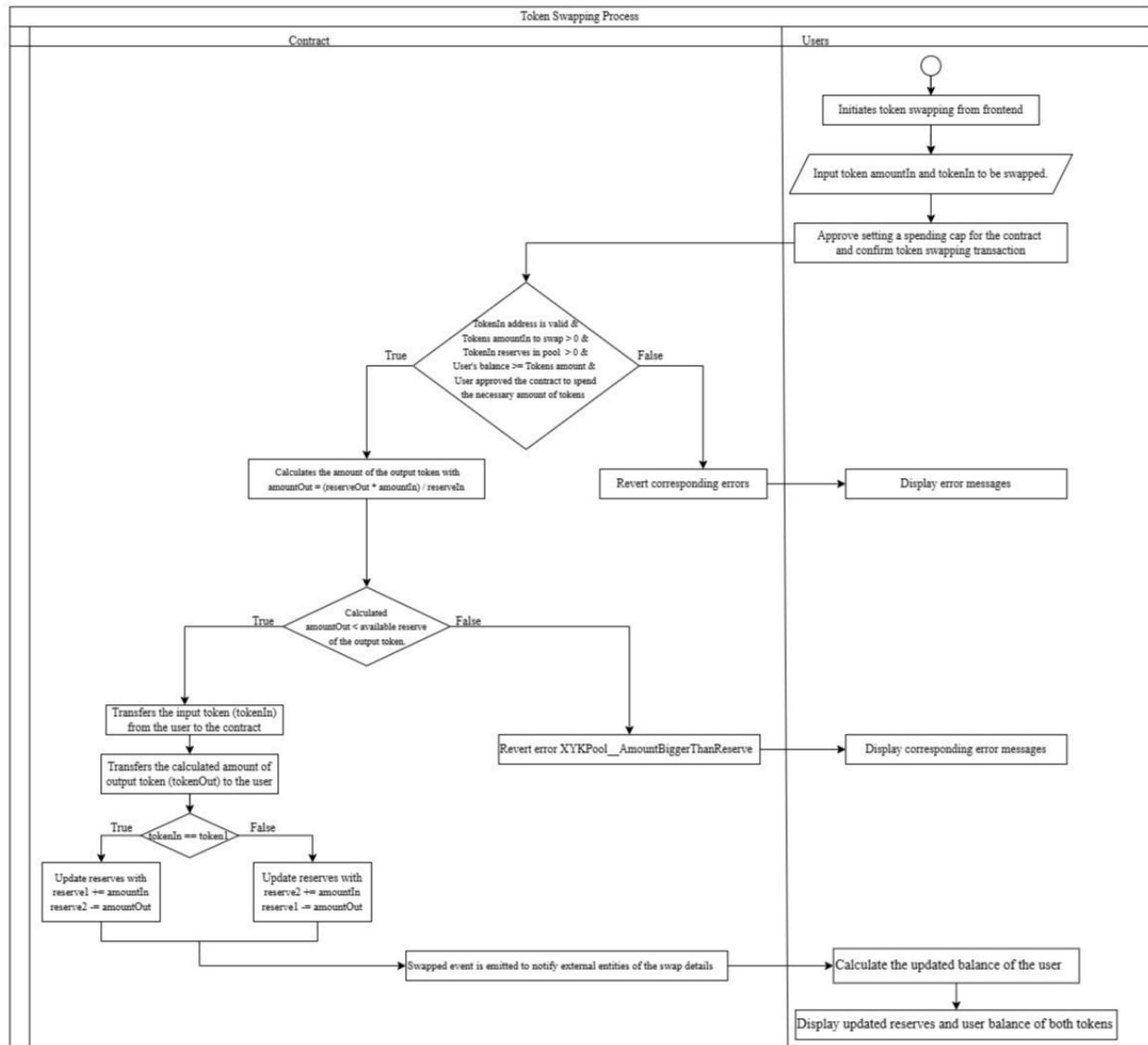


Diagram 5.3: Token Swapping Process Activity Diagram

## 6.0 System Architecture Diagram

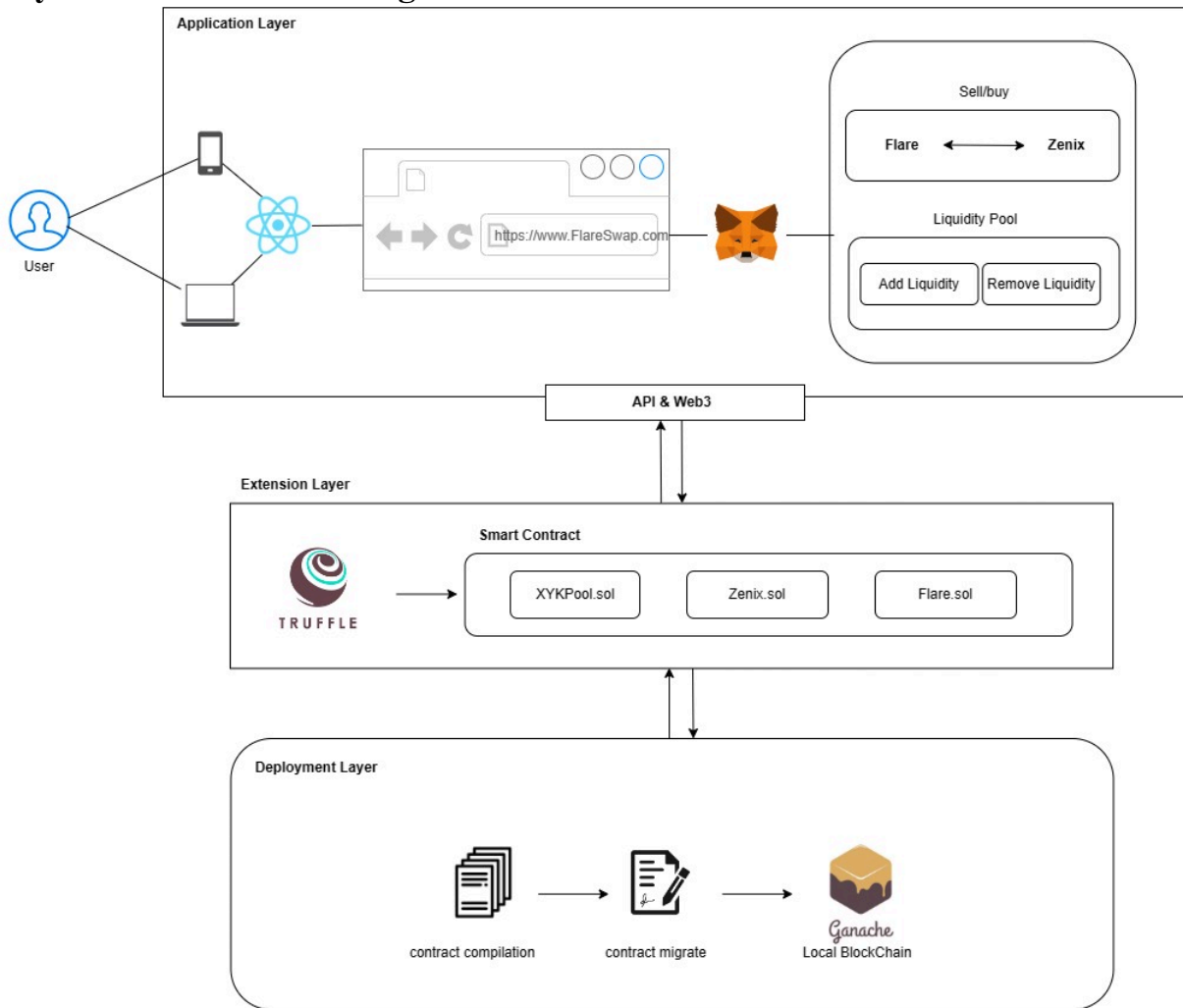


Diagram 6.1: FlareSwap System Architecture Diagram



## 7.0 UI Design

The React web application serves as the frontend of this DApp for users to interact with the system. The markup and styling components supporting the React application are JSX,JS and CSS.

Initially, the web application is not connected to any account and the Add Liquidity container is displayed. Users can click on the “Connect to MetaMask” button to connect to their account.

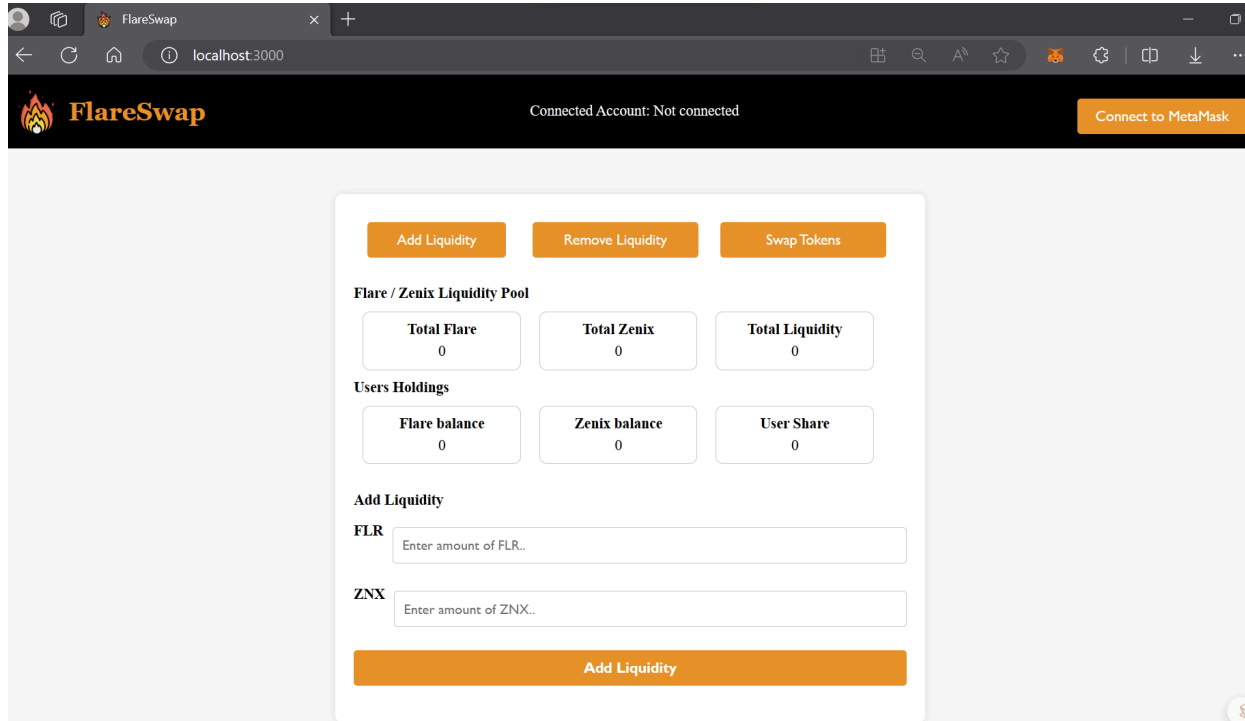


Diagram 7.1: Initial state of the web application

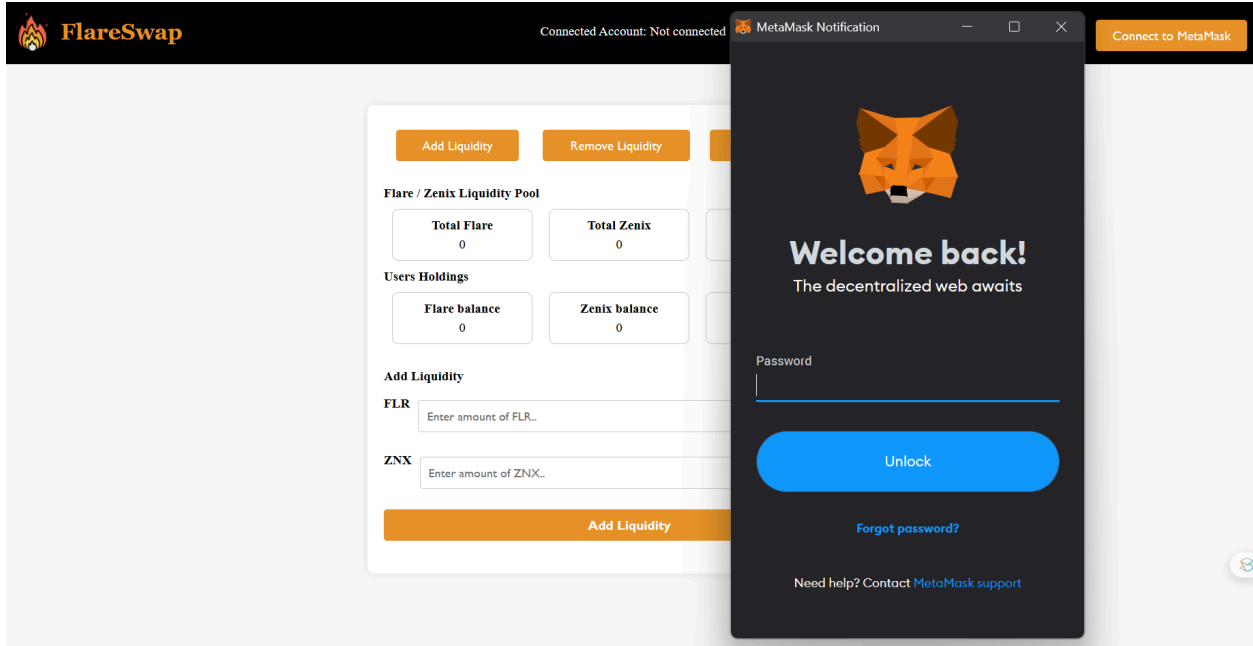


Diagram 7.2: After clicking connect to metamask button, Metamask window is popped up for user to unlock with password.

Once it is successfully connected to an account, the header of the page will display the account address of the current connected account. Also, reserves of both tokens in the liquidity pool and user holdings are shown accordingly.

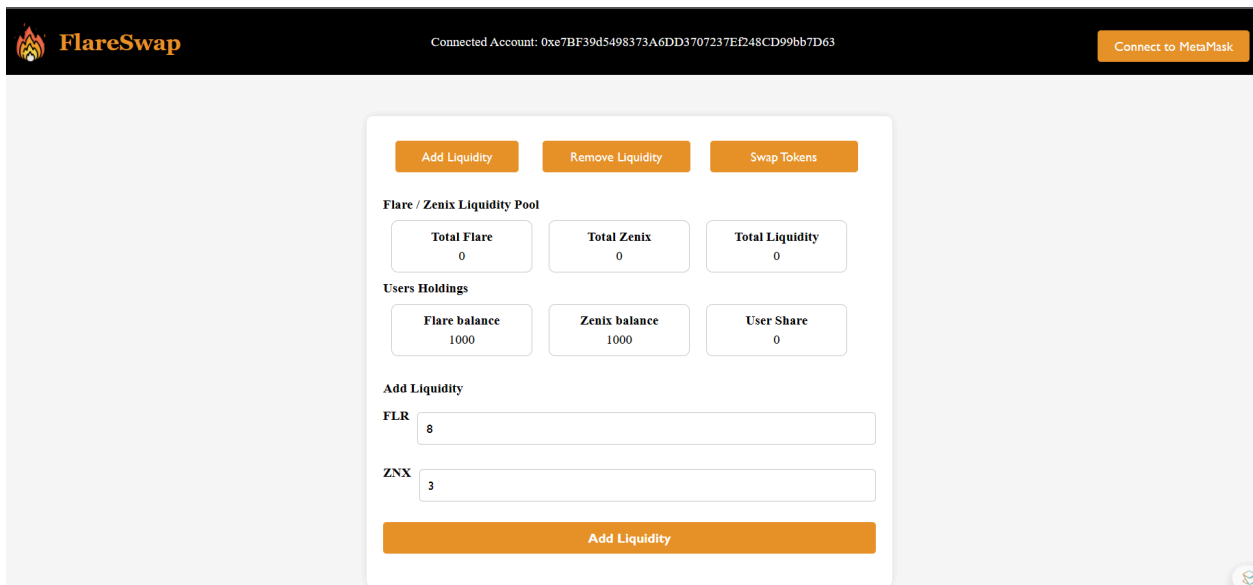


Diagram 7.3: Interface updated after successful connection to an account.

User is able to switch to another account by selecting the desired account on Metamask plugin, and click the "Connect to MetaMask" button again. The interface will be updated to show the corresponding details for the current connected user.

Connected Account: 0x2b78F1C5AE78C9962dCd022cE50aE590Cb81BEcE

Add Liquidity

Remove Liquidity

Swap Tokens

**Flare / Zenix Liquidity Pool**

<b>Total Flare</b> 0	<b>Total Zenix</b> 0	<b>Total Liquidity</b> 0
-------------------------	-------------------------	-----------------------------

**Users Holdings**

<b>Flare balance</b> 100	<b>Zenix balance</b> 100	<b>User Share</b> 0
-----------------------------	-----------------------------	------------------------

**Add Liquidity**

**FLR**

**ZNX**

Add Liquidity

Diagram 7.4: Interface updated after switching to another account.

User can add liquidity by entering the token amounts. Token amount entered must not exceed the user's balance for the token, else it will display an error message.

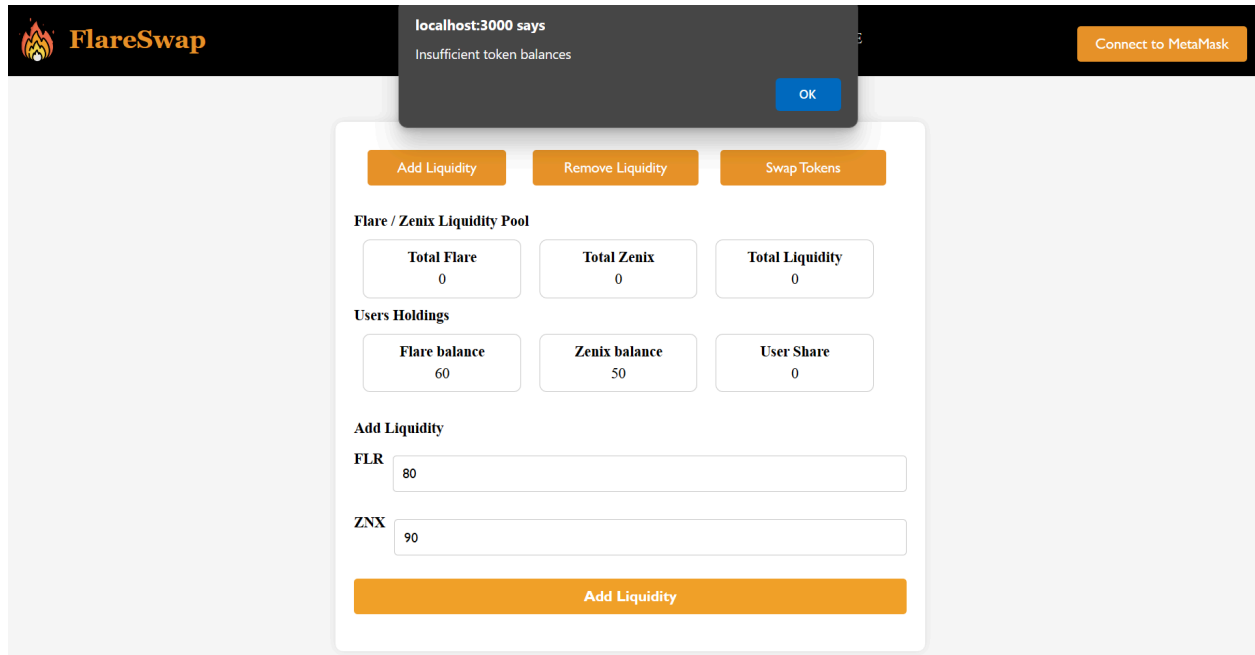


Diagram 7.5: Alert message is popped up when the token amount exceeded the token balance owned by the user.

When the liquidity pool is empty, user can add any amount of token. However, when the liquidity pool is not empty, user must add liquidity based on a proper ratio. After user clicked on the “Add Liquidity” button in the bottom of the page. Metamask window is popped up for user to approve an increased amount of allowance of both tokens for the XYKPool contract to use.

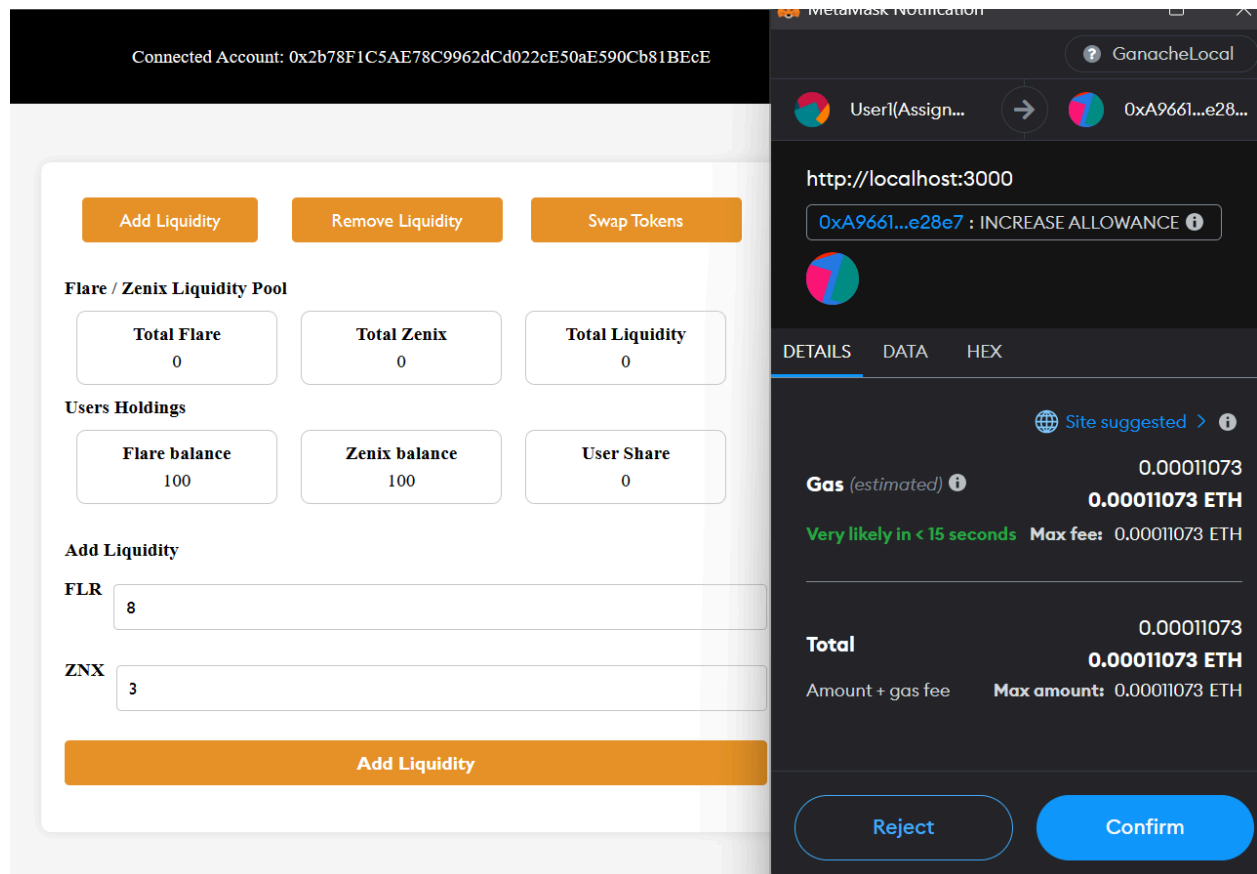


Diagram 7.6: Allowance approval before adding liquidity.

After clicking the “Confirm” button on the MetaMask window, another window is popped up for user to confirm the execution of liquidity addition.

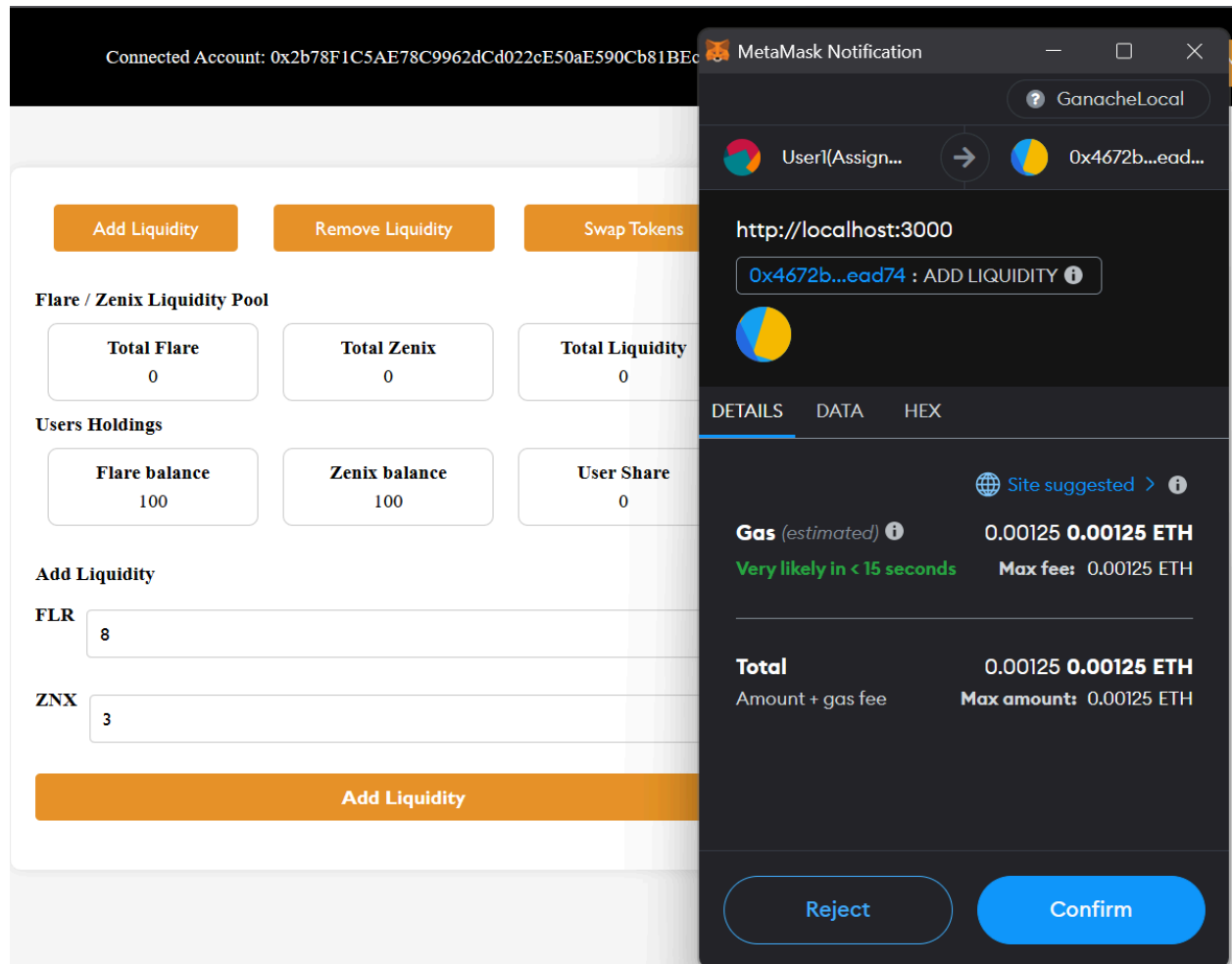


Diagram 7.7: Transaction confirmation for adding liquidity.

**Add Liquidity** **Remove Liquidity** **Swap Tokens**

**Flare / Zenix Liquidity Pool**

<b>Total Flare</b> 8	<b>Total Zenix</b> 3	<b>Total Liquidity</b> 4
-------------------------	-------------------------	-----------------------------

**Users Holdings**

<b>Flare balance</b> 92	<b>Zenix balance</b> 97	<b>User Share</b> 4
----------------------------	----------------------------	------------------------

**Add Liquidity**

**FLR**

**ZNX**

**Add Liquidity**

Diagram 7.8: Liquidity pool and user holdings details is updated after liquidity addition.

If the token amounts to add liquidity is not in proper ratio between tokens, the transaction is failed and an alert message is displayed. As the transaction is failed, no tokens are withdrawn from the user account. Then, user can add liquidity by using the suggested ratio, which is the proper ratio, it generally use the user input ZNX to calculate the corresponding token amount for FLR. If users were allowed to add liquidity without considering the proper ratio, it could lead to significant price imbalances and fluctuations. A proper ratio helps stabilize the token prices within the pool which is crucial in implementation of XYK model.

The screenshot shows a web interface for a 'Flare / Zenix Liquidity Pool'. At the top, a dark grey alert box from 'localhost:3000' displays the message: 'Transaction failed: Not proper ratio between tokens. Suggested ratio(FLR:ZNX) is 16:6'. Below the alert, there are three orange buttons: 'Add Liquidity', 'Remove Liquidity', and 'Swap Tokens'. The 'Flare / Zenix Liquidity Pool' section contains three white boxes with black text: 'Total Flare' with value 8, 'Total Zenix' with value 3, and 'Total Liquidity' with value 4. Below this, the 'Users Holdings' section has three white boxes: 'Flare balance' with value 92, 'Zenix balance' with value 97, and 'User Share' with value 4. The 'Add Liquidity' section features two input fields: 'FLR' with the value 8 and 'ZNX' with the value 6. At the bottom of this section is a large orange button labeled 'Add Liquidity'.

localhost:3000 says

Transaction failed: Not proper ratio between tokens. Suggested ratio(FLR:ZNX) is 16:6

OK

Add Liquidity Remove Liquidity Swap Tokens

**Flare / Zenix Liquidity Pool**

<b>Total Flare</b> 8	<b>Total Zenix</b> 3	<b>Total Liquidity</b> 4
-------------------------	-------------------------	-----------------------------

**Users Holdings**

<b>Flare balance</b> 92	<b>Zenix balance</b> 97	<b>User Share</b> 4
----------------------------	----------------------------	------------------------

**Add Liquidity**

**FLR**

**ZNX**

**Add Liquidity**

Diagram 7.9: Alert message when no proper ratio between tokens is used to add liquidity. The proper ratio between tokens is 16:6 in this case.

When the user add liquidity with the suggested ratio, the same procedures involved in approving allowance and confirming the transaction mentioned above are done. Then, the liquidity pool and user holdings will be updated accordingly. (eg. Flare balance  $100 - 8 = 92$ )



Connected Account: 0x2b78F1C5AE78C9962dCd022cE50aE590Cb81BEcE

Add LiquidityRemove LiquiditySwap Tokens

Flare / Zenix Liquidity Pool

Total Flare24

Total Zenix9

Total Liquidity4

Users Holdings

Flare balance76

Zenix balance91

User Share4

Add Liquidity

FLR16

ZNX6

Add Liquidity

Diagram 7.10: Adding liquidity using the suggested ratio is successful and details are updated accordingly.

Additionally, the web application will pop up an alert if the current account that initiated liquidity addition is the contract owner, as an owner is not allowed to add liquidity. This is to maintain fairness and prevent potential manipulation done by the owner.

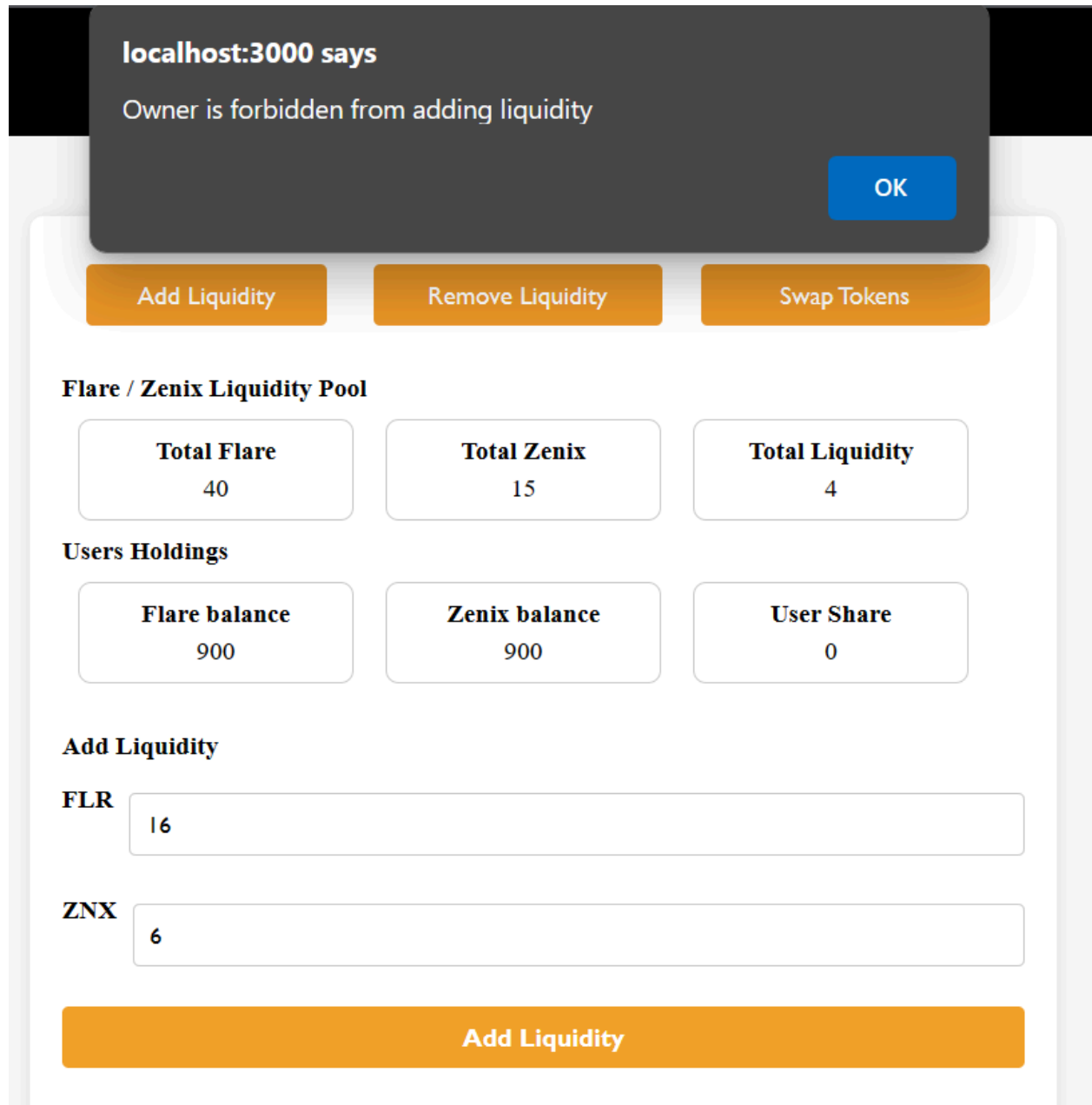


Diagram 7.11: Alert message when an owner attempts to add liquidity.

Users can switch to the “Remove Liquidity” container by clicking the “Remove Liquidity” button at the top of the page. Liquidity pool and user holdings are also displayed here. The “Remove Liquidity” button at the bottom is used to initiate the liquidity removal process. After clicking, users have to confirm the liquidity removal transaction.

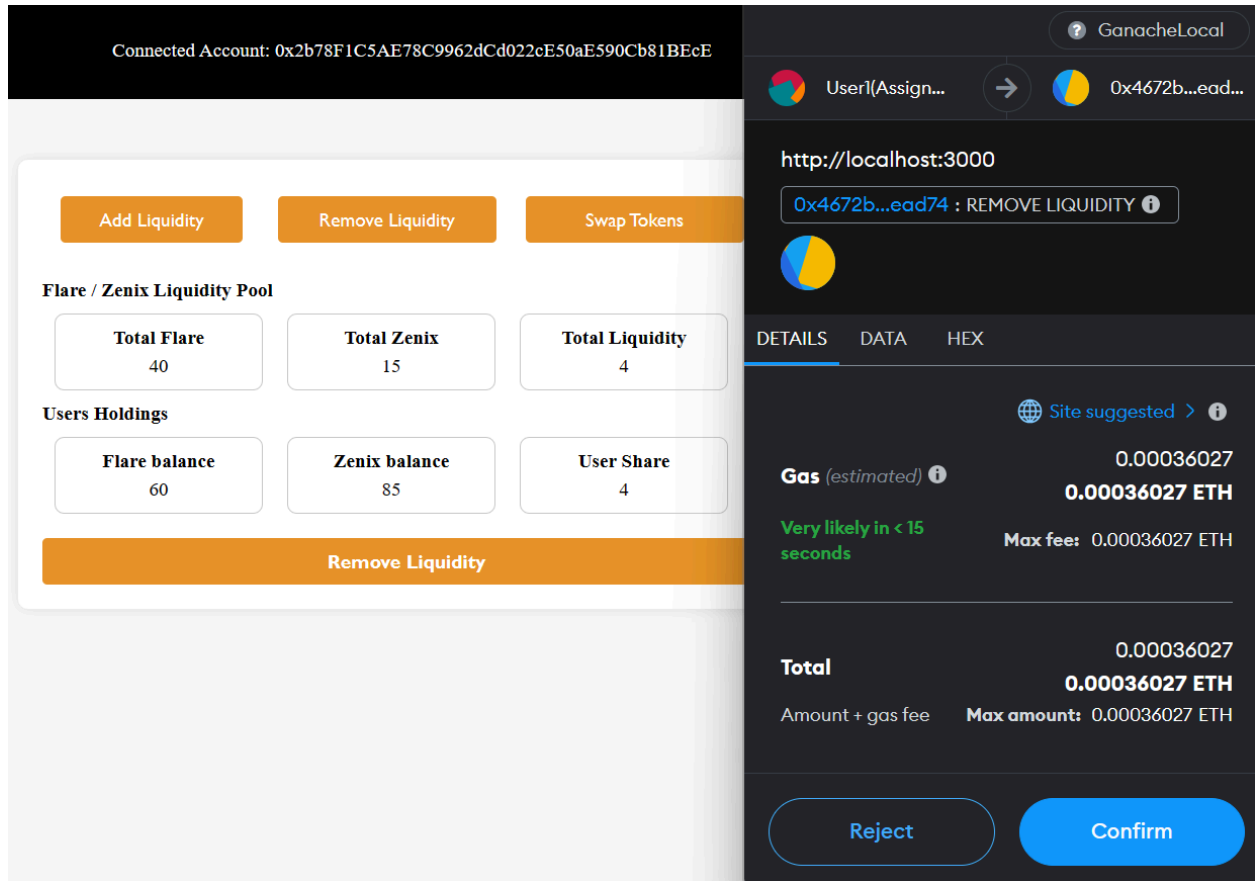


Diagram 7.12: MetaMask notified user to confirm the liquidity removal transaction.

The liquidity pool and users holdings are updated immediately. The user share contributed to the liquidity pool previously reverted to 0, the liquidity pool deducted the liquidity provided by the user previously, and, the tokens amount added to the liquidity is withdrawn back to the user's wallet.

Connected Account: 0x2b78F1C5AE78C9962dCd022cE50aE590Cb81BEcE

Add Liquidity

Remove Liquidity

Swap Tokens

**Flare / Zenix Liquidity Pool**

<b>Total Flare</b> 0	<b>Total Zenix</b> 0	<b>Total Liquidity</b> 0
-------------------------	-------------------------	-----------------------------

**Users Holdings**

<b>Flare balance</b> 100	<b>Zenix balance</b> 100	<b>User Share</b> 0
-----------------------------	-----------------------------	------------------------

Remove Liquidity

Diagram 7.13: Liquidity pool and users holdings are updated after successful liquidity withdrawal. If a user has no user share, which means the user is not a liquidity provider, then the user will not be able to remove liquidity.

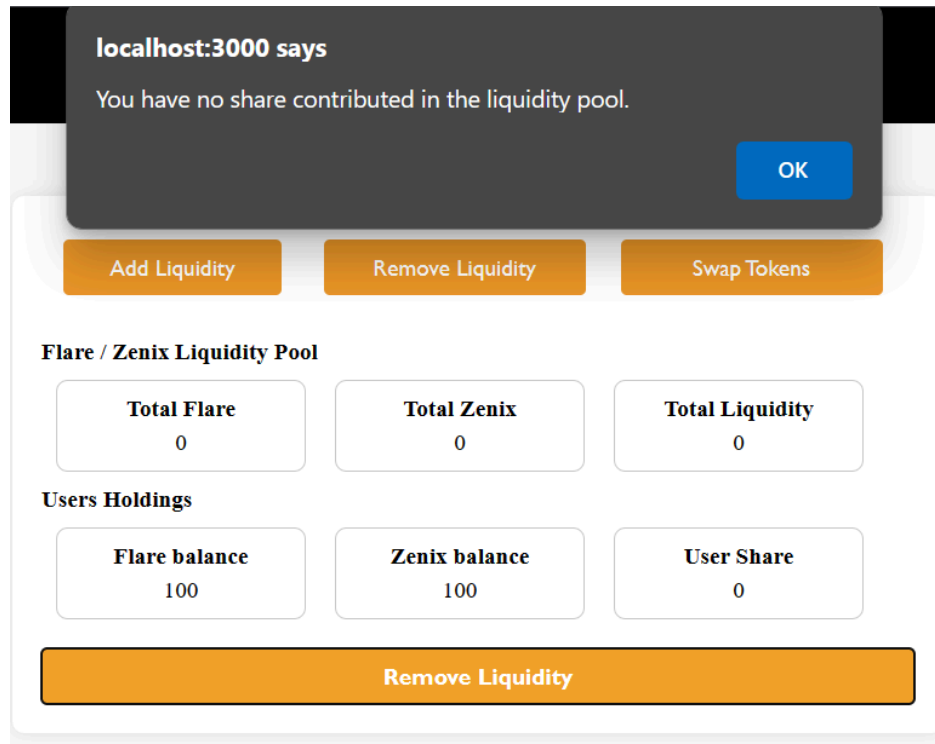
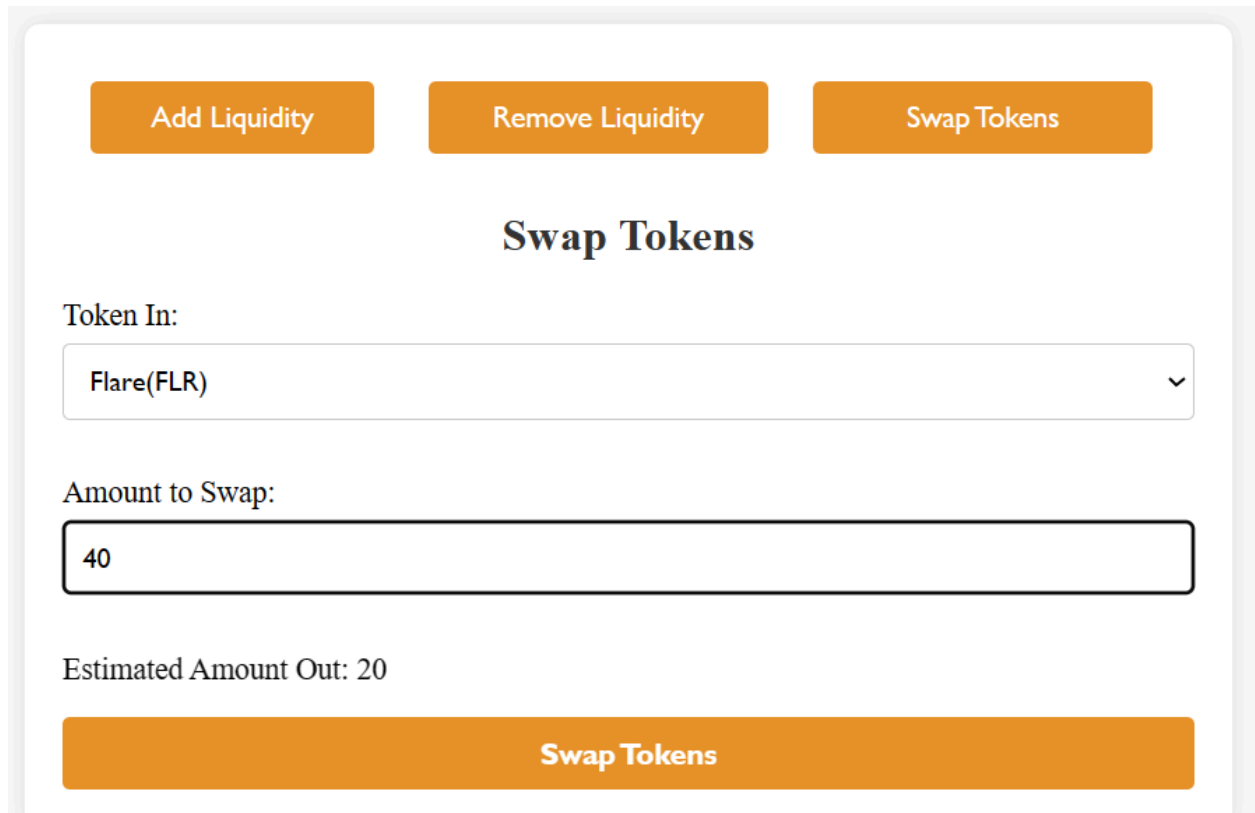


Diagram 7.14: Alert message when non-liquidity provider attempts to withdraw liquidity

Users can swap tokens by switching the current container to Swap Tokens. Users can select the token to be swapped and input the amount of the token to be swapped. The default token selected is Flare. The estimated amount out is the estimated amount of another token after swapping



The diagram shows a user interface for token swapping. At the top, there are three orange buttons: "Add Liquidity", "Remove Liquidity", and "Swap Tokens". Below these is a section titled "Swap Tokens" in bold. Under the title, there is a label "Token In:" followed by a dropdown menu showing "Flare(FLR)" with a downward arrow. Below that is a label "Amount to Swap:" followed by a text input field containing the number "40". Underneath the input field is the text "Estimated Amount Out: 20". At the bottom of the section is a large orange button labeled "Swap Tokens".

Add Liquidity Remove Liquidity Swap Tokens

### Swap Tokens

Token In:

Flare(FLR) ✓

Amount to Swap:

40

Estimated Amount Out: 20

Swap Tokens

Diagram 7.15: Interface of token swapping

The token amount to be swapped must not exceed the reserve of the token in the liquidity pool, else alert message is shown.

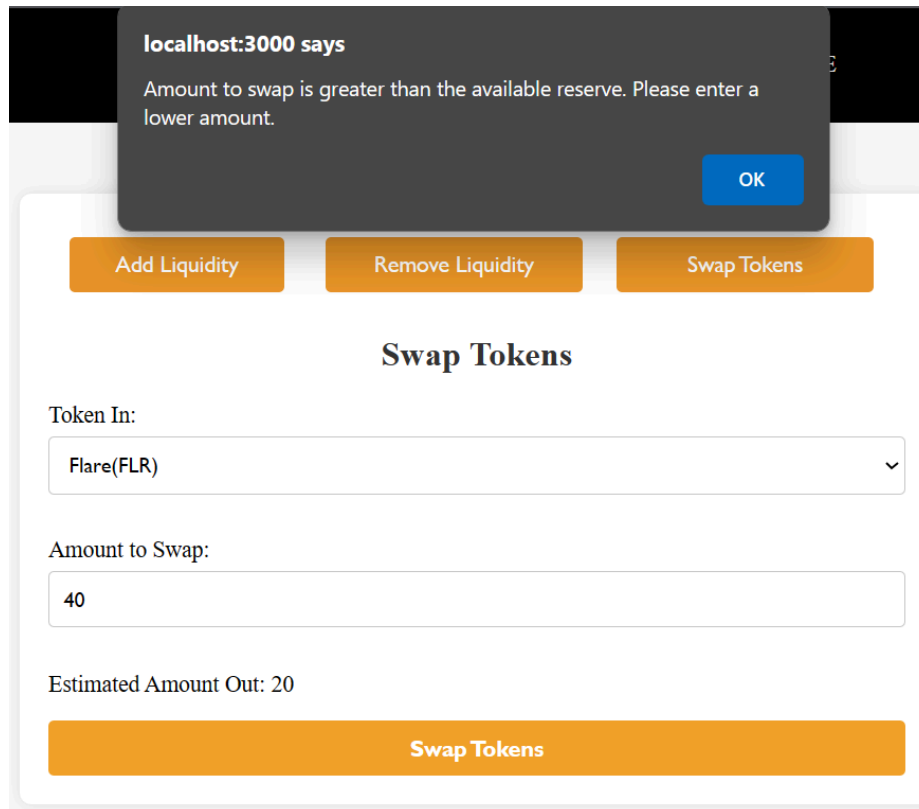


Diagram 7.16: Alert message when token amount to swap is too large for the token reserve.

When user enter an smaller token amount to swap and click swap token button, MetaMask notified user to set spending cap for the selected token in. This is to prevent unauthorized or unintentional large transactions when you interact with a decentralized application (DApp) that requires spending tokens on your behalf. User have to confirm the following needed transactions including approving the spending cap set and swap function execution. After that, the liquidity pool and user holdings will be updated immediately. User can check the details of liquidity pool by switching to add liquidity or remove liquidity container.

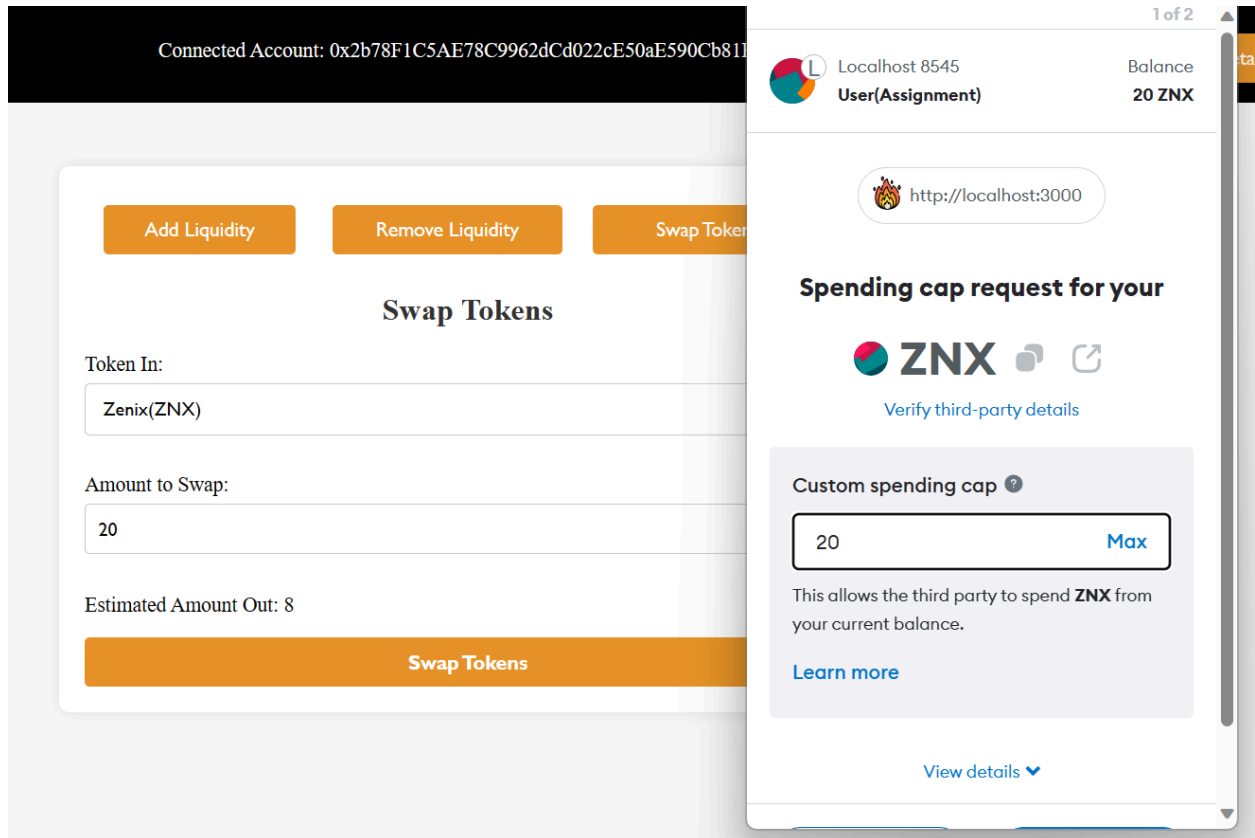


Diagram 7.17: MetaMask notified user to set spending cap for the selected token in.



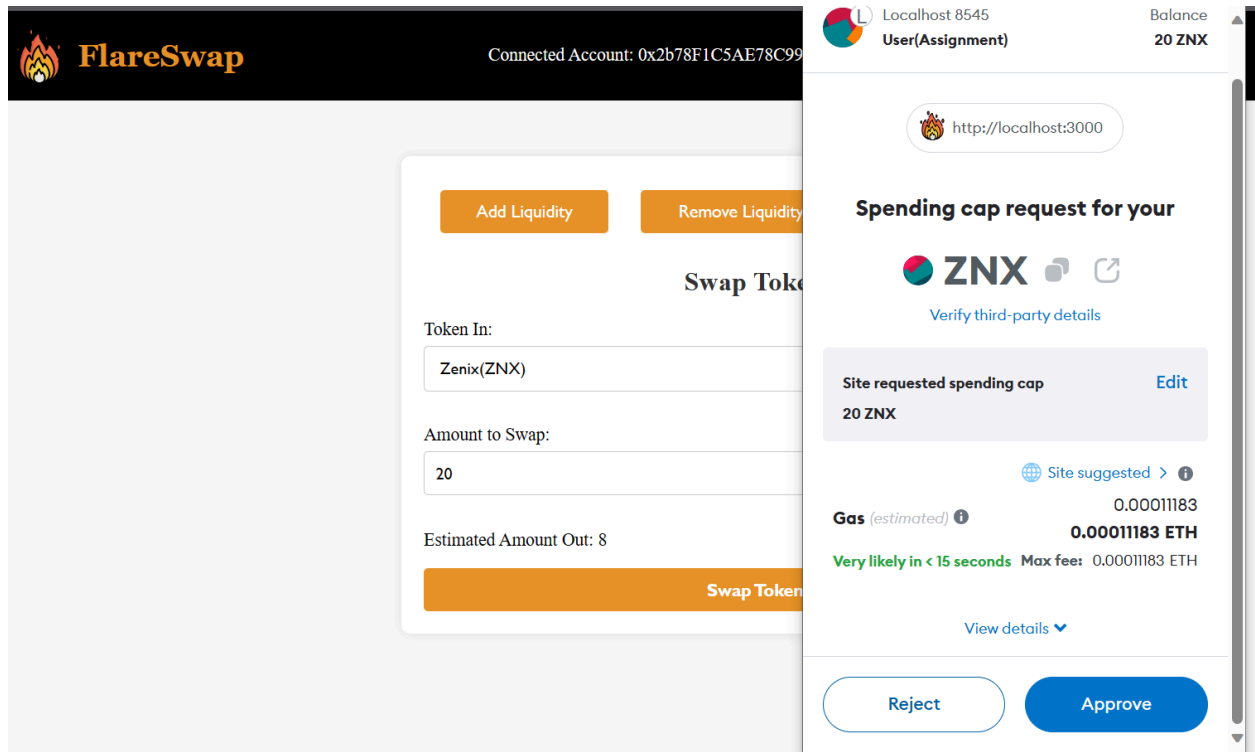


Diagram 7.18: MetaMask requesting approval of the spending cap set.

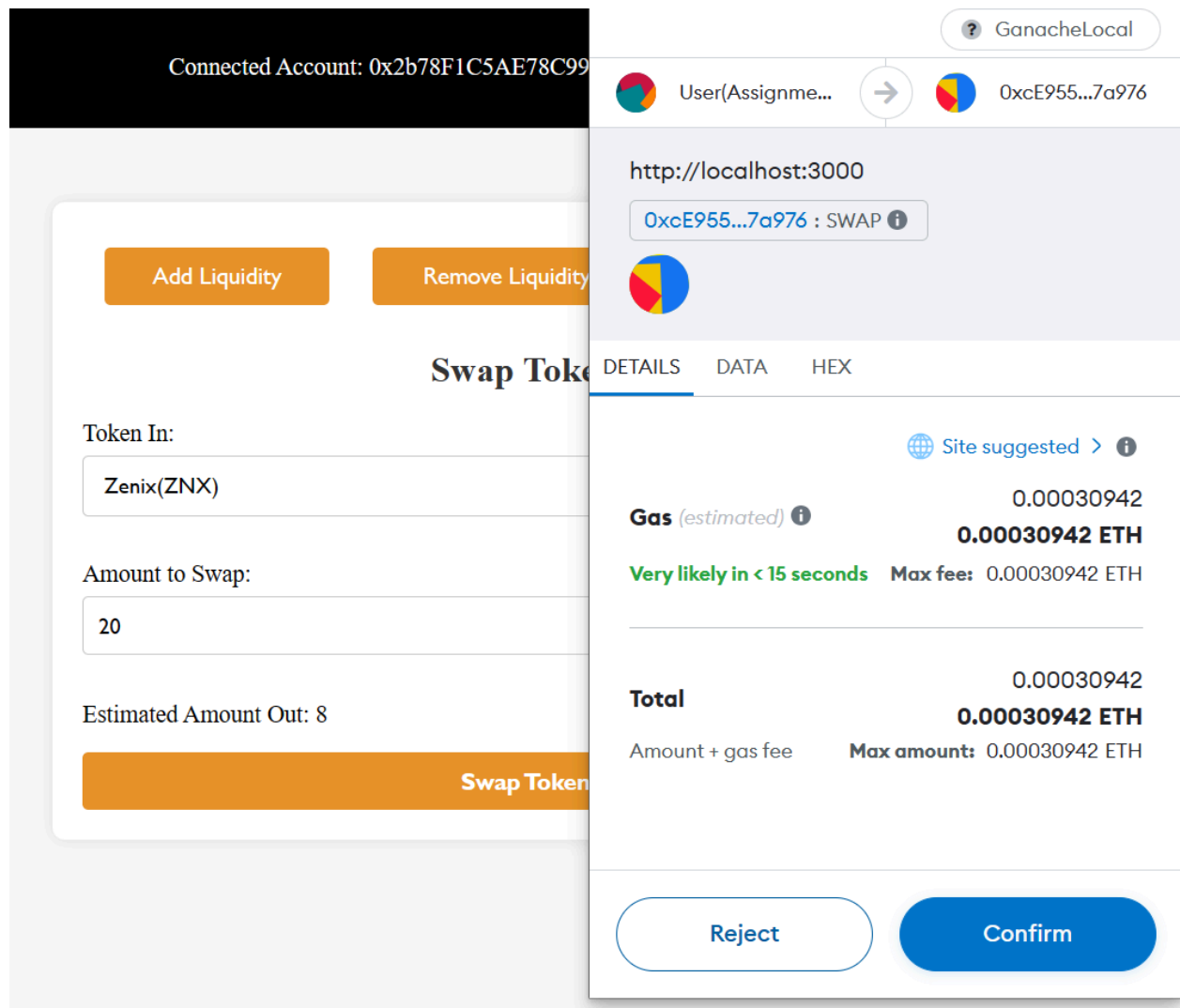


Diagram 7.19: User have to confirm the swap transaction.

### Flare / Zenix Liquidity Pool

<b>Total Flare</b> 6	<b>Total Zenix</b> 50	<b>Total Liquidity</b> 24
-------------------------	--------------------------	------------------------------

### Users Holdings

<b>Flare balance</b> 60	<b>Zenix balance</b> 50	<b>User Share</b> 0
----------------------------	----------------------------	------------------------

Diagram 7.20: The state before token swapping

### Flare / Zenix Liquidity Pool

<b>Total Flare</b> 4	<b>Total Zenix</b> 70	<b>Total Liquidity</b> 24
-------------------------	--------------------------	------------------------------

### Users Holdings

<b>Flare balance</b> 62	<b>Zenix balance</b> 30	<b>User Share</b> 0
----------------------------	----------------------------	------------------------

Diagram 7.21: The state after token swapping is updated accordingly

## References

Adams, H. (2020) 🦄 Uniswap Whitepaper.

<https://hackmd.io/@HaydenAdams/HJ9jLsfTz?type=view#%F0%9F%A6%84-Uniswap-Whitepaper>

*Creating ERC20 Supply - OpenZeppelin Docs.* (2017). Openzeppelin.com.

<https://docs.openzeppelin.com/contracts/2.x/erc20-supply>

*ERC20 - OpenZeppelin Docs.* (2017). Openzeppelin.com.

<https://docs.openzeppelin.com/contracts/2.x/erc20>

*From Idea to Minimum Viable Dapp - How to use Ganache to enhance your auction dapp -*

*Truffle Suite.* (2020). Trufflesuite.com.

<https://trufflesuite.com/blog/from-idea-to-minimum-viable-dapp-how-to-use-ganache-to-enhance-your-auction-dapp/>

Lacapra, E. (2023). What are liquidity provider and how do they work?. Cointelegraph.com

<https://cointelegraph.com/explained/what-are-liquidity-provider-lp-tokens-and-how-do-they-work>

Solidity. (2016). *Solidity 0.8.20 documentation.* Soliditylang.org.

<https://docs.soliditylang.org/en/v0.8.20/>

*Ownership - OpenZeppelin Docs.* (2017). Openzeppelin.com.

<https://docs.openzeppelin.com/contracts/2.x/api/ownership>

Viera, A. (2021, April 20). *What Is Uniswap?* Moonbeam Network; Moonbeam Network.

<https://moonbeam.network/education/what-is-uniswap/>

Uniswap. (2018). Trade Tokens. <https://docs.uniswap.org/contracts/v1/guides/trade-tokens>

**BMIS2003 Blockchain Application Development**  
**Assignment Rubric / Evaluation (Part A)**

Programme : RDS,RSD

Group :G3,G1

CLO2: Produce one Blockchain-based application using a selected Blockchain platform. (P4, PLO3)

**Application Developed (70%)**

Criteria	Poor	Average	Good	Excellent	Group mark
<b>Quantity of work (20%)</b>	Not able to complete most of the required functions in Dapp (FrontEnd & BackEnd). (1 – 4)	Completed more than 40% the required functions in Dapp (FrontEnd & BackEnd). (5 - 9)	Completed more than 60% of the required functions in Dapp (FrontEnd & BackEnd). (10 -14)	Completed more than 80% of the required functions in Dapp (FrontEnd & BackEnd). (15 -20)	
<b>Quality of work (20%)</b>	Not able to Complete workable functions (contain errors). UIs are designed – very poor user interface. (1 - 4)	Completed more than 40% functions are working and implemented with some errors. UIs are designed – some not suitable/difficult to use. (5 -9)	Completed more than 60% functions are working and implemented some error detection and handling methods with minor errors. UIs are designed -good user interface. (10 -14)	Completed more than 80% functions are working and implemented with proper error detection and handling methods. UIs are designed - very good and intuitive user interface. (15 -20)	
<b>Originality and understanding (10%)</b> [Explanation During Presentation / Q&A]	No citations are included for works/codes contributed by others. AND Not able to explanations of the system logic during presentation/ Q&A. (0 -2)	No citations are included for works/codes contributed by others. AND unclear & inconcise explanations of the system logic during presentation/ Q&A. (3 -5)	Clear citations are included for works/codes contributed by others. OR Clear & concise explanations of the system logic during presentation/ Q&A. (6 -7)	Clear citations are included for works/codes contributed by others. AND Clear & concise explanations of the system logic during presentation/ Q&A. (8 -10)	
<b>Innovativeness (10%)</b>	Inappropriate proposed application and solution to the problem(s). (0 - 2)	Appropriate proposed application and solution to the problem(s). (3 - 5)	Good proposed application and creative solution to the problem(s). (6 -7)	Very good proposed application and very creative solution to the problem(s). (8 -10)	
<b>Design documentatio</b>	Report is too brief, poorly	Report is averagely written	Report length is adequate and	Report length is reasonable and	

<b>n (10%)</b>	written and/or lack of explanation. (0 -2)	with some explanation. (3 -5)	overall content is good with adequate explanation. (6 -7)	overall content is well written and explained. (8 -10)	
<b>Sub Total</b>					

Remark:

**BMIS2003 Blockchain Application Development**  
**Assignment Rubric / Evaluation (Part B)**

Programme : RDS,RSD

Group :G3,G1

CLO3: Cooperate to build one Blockchain-based application in a team. (A2, PLO4)

**Team work and Presentation (30%)**

Criteria	Individual mark				
	Member 1 name:	Member 2 name:	Member 3 name:	Member 4 name:	Member 5 name:
<b>Project involvement (10%)</b> <ul style="list-style-type: none"> <li>Actively involving and contributing to the assignment</li> </ul>					
<b>Teamwork (10%)</b> <ul style="list-style-type: none"> <li>Be able to work in a team to accomplish the assignment</li> </ul>					
<b>Quality of presentation (10%)</b> The presenter is well prepared and language use is appropriate. *Read from source without looking at the audience. (1-2m) *Read from source and occasionally facing the audience. (3-4m) *Present (not read) facing the audience with eye contact. (5-6m) *Engage the audience through eye contact and asking/answering questions. (7-8m) *Engaging and being engaged by the audience making the session interactive. (9-10m)					
Sub Total					
<b>Final Total (A+B)</b>					

Remark:

Marked by:

Date: