Some features of Commune AI Frontend

(The guide how to implement)

In this docs, we want to describe about the some features of commune ai frontend and the steps to implement these features such as tracking money, substrate balance for transferring money, user roles (admin, friend, and user).

1. Track money

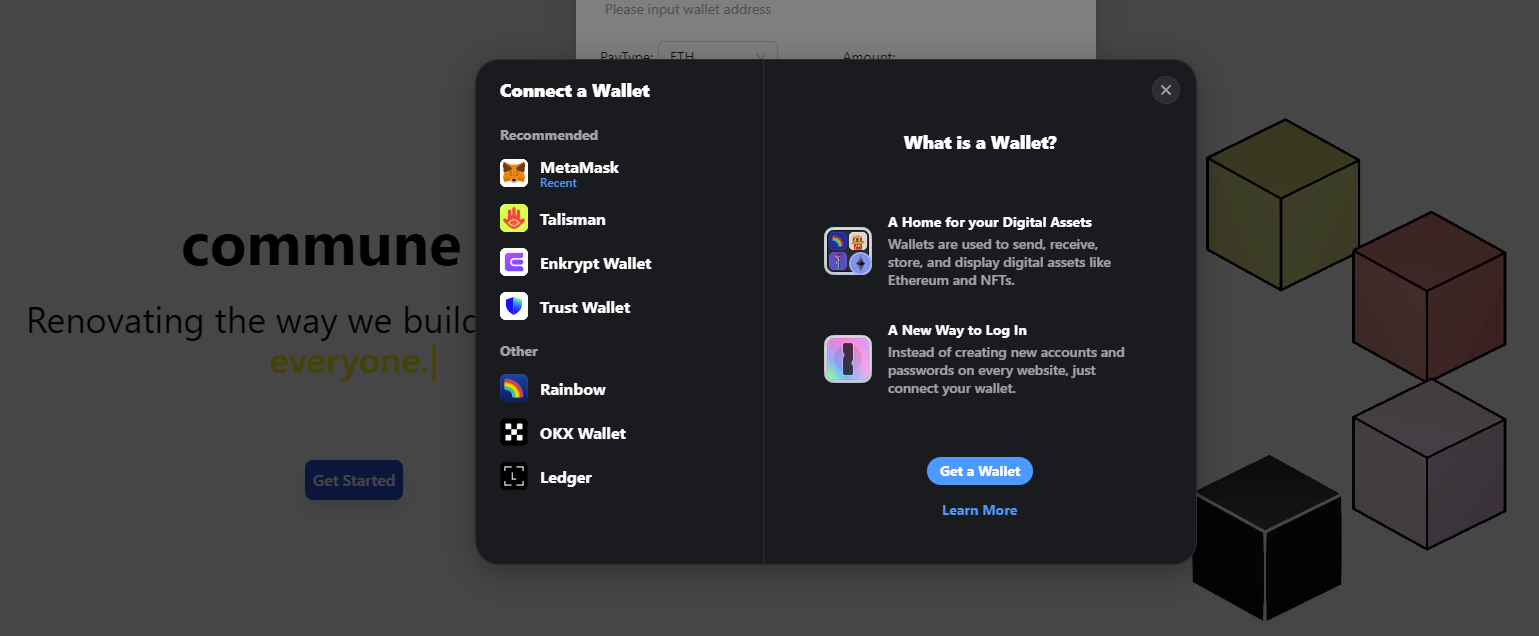
* This feature lets you see which users made transactions and how much they sent.
* 2 payment methods we used: pay with **stripe**, and then pay with **wallet.**

Users can make a transaction with ETH, MATIC, USDT, and USDC when they use wallet.

For this feature:

* In Frontend:

We used *rainbow wallet provider* for wallet connection and then every transaction when user made is saved to backend.

Using rainbow wallet, users can select their own chain. We added *Polkadot* *wallet* as a custom wallet.

* Backend:

We used Django as a backend framework.

The transaction model is like that:

class TransactionRecordModel(models.Model):

    transactionId = models.AutoField(primary\_key=True)

    payType = models.CharField(max\_length=255)

    amount = models.FloatField()  # Change from Decimal to FloatField

    destinationAddress = models.CharField(max\_length=100)

    txHash = models.CharField(max\_length=100)

    created\_at = models.DateTimeField(auto\_now\_add=True)

    updated\_at = models.DateTimeField(auto\_no3/5/2024w=True)

    deleted\_at = models.DateTimeField(blank=True, null=True)

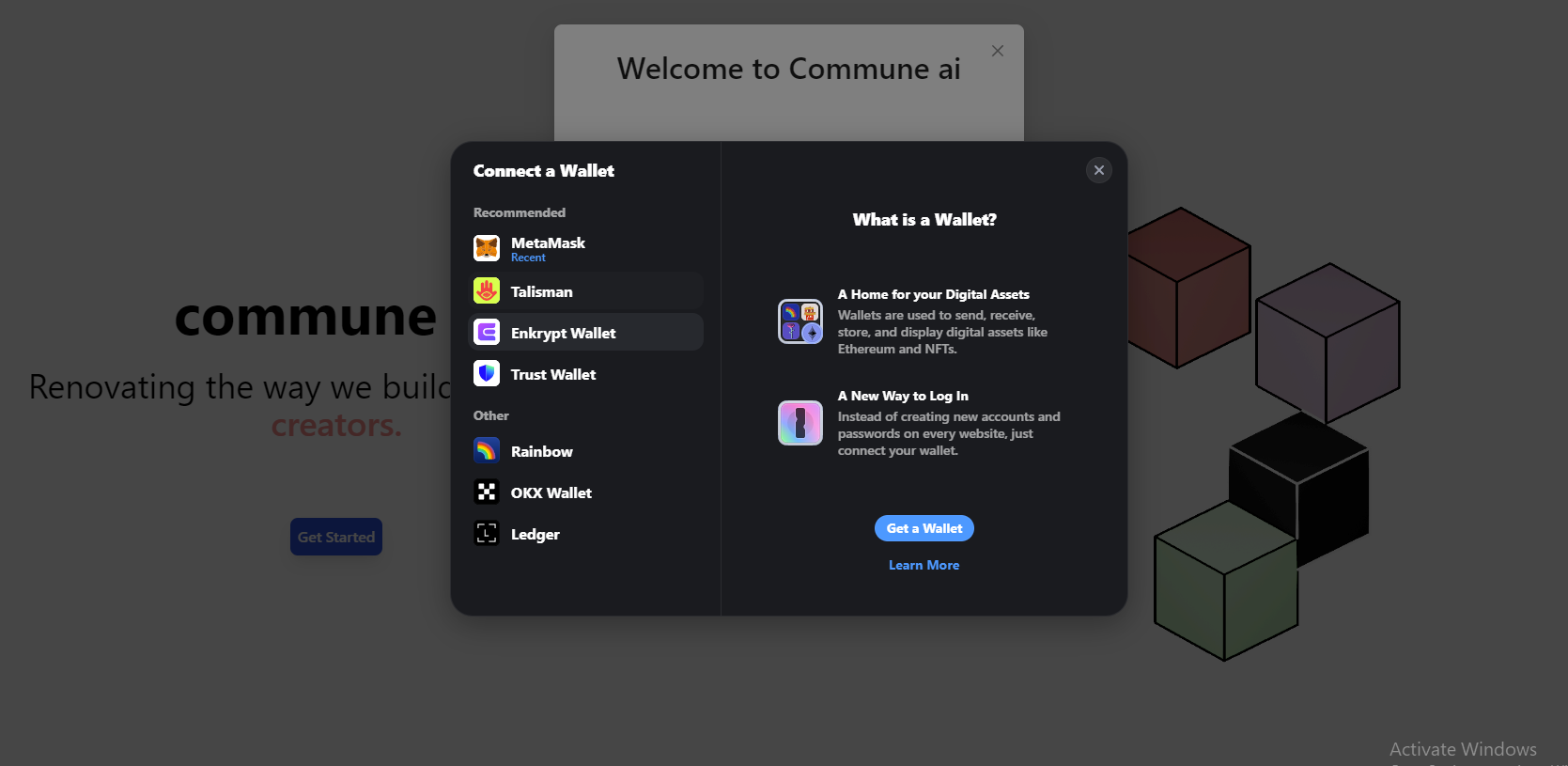
    def \_\_str\_\_(self):

        return str(self.transactionId)

1. Login with metamask login

We used *rainbow wallet provider* and *wagmi wallet connector* to authenticate users.

User info – pub key will save to database.

This figure shows how to login with wallet in commune ai.

1. Substrate balance

We calculate the amount of USD someone has given me as the website owner.

For this we made a user role with ‘user’, ‘friend’, ‘admin’ like a social network.

We assume friend and owner get unlimited calls and users to have to pay.

urls:

onfinality:

ws: wss://commune.api.onfinality.io/public-ws

http: <https://commune.api.onfinality.io/public-http>

We start with transfer and receive as well as sign, and encrypt.

This code shows you how we can transfer and receive with substrate

import { ApiPromise, WsProvider } from '@polkadot/api';

export async function transferBalance(sender: string, recipient: string, amount: number): Promise<string> {

    const provider = new WsProvider('wss://commune.api.onfinality.io/public-ws');

    const api = await ApiPromise.create({ provider });

    const transfer = api.tx.balances.transfer(recipient, amount);

    const hash = await transfer.signAndSend(sender);

    return hash.toString();

}

1. Interacts with substrate

We made a component that interacts with substrate chain.

This code shows you how we can interacts with substrate chain.

import React, { useState, useEffect } from 'react';

import { ApiPromise, WsProvider } from '@polkadot/api';

const SubstrateComponent: React.FC = () => {

    const [api, setApi] = useState<ApiPromise | null>(null);

    const [chainInfo, setChainInfo] = useState('');

    const [nodeName, setNodeName] = useState('');

    useEffect(() => {

        const connectToSubstrate = async () => {

            const provider = new WsProvider('wss://rpc.polkadot.io');

            const substrateApi = await ApiPromise.create({ provider });

            setApi(substrateApi);

        };

        connectToSubstrate();

    }, []);

    const getChainInfo = async () => {

        if (api) {

            const chain = await api.rpc.system.chain();

            setChainInfo(chain.toString())

            const nodeName = await api.rpc.system.name();

            setNodeName(nodeName.toString())

            console.log(`Connected to chain ${chain} using ${nodeName}`);

        }

    };

    return (

        <div>

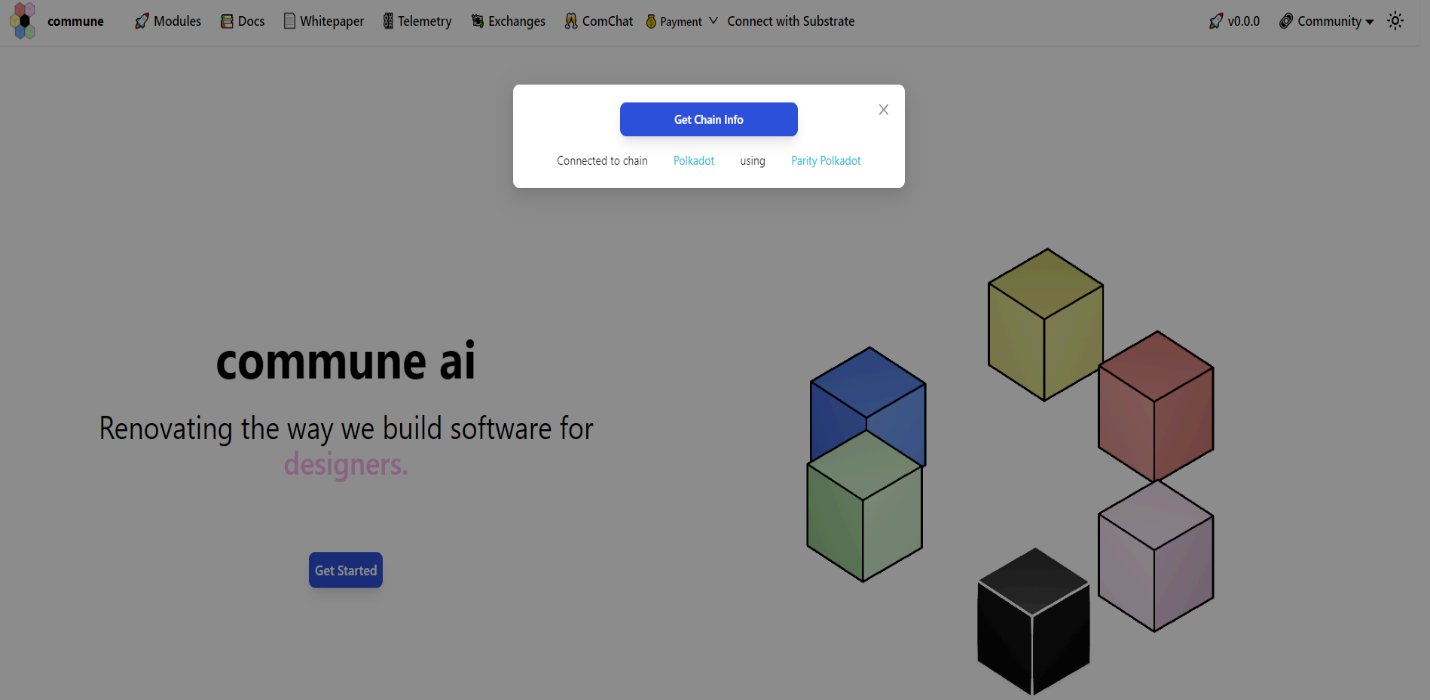
            <button onClick={getChainInfo}>Get Chain Info</button>

        </div>

    );

};

export default SubstrateComponent;

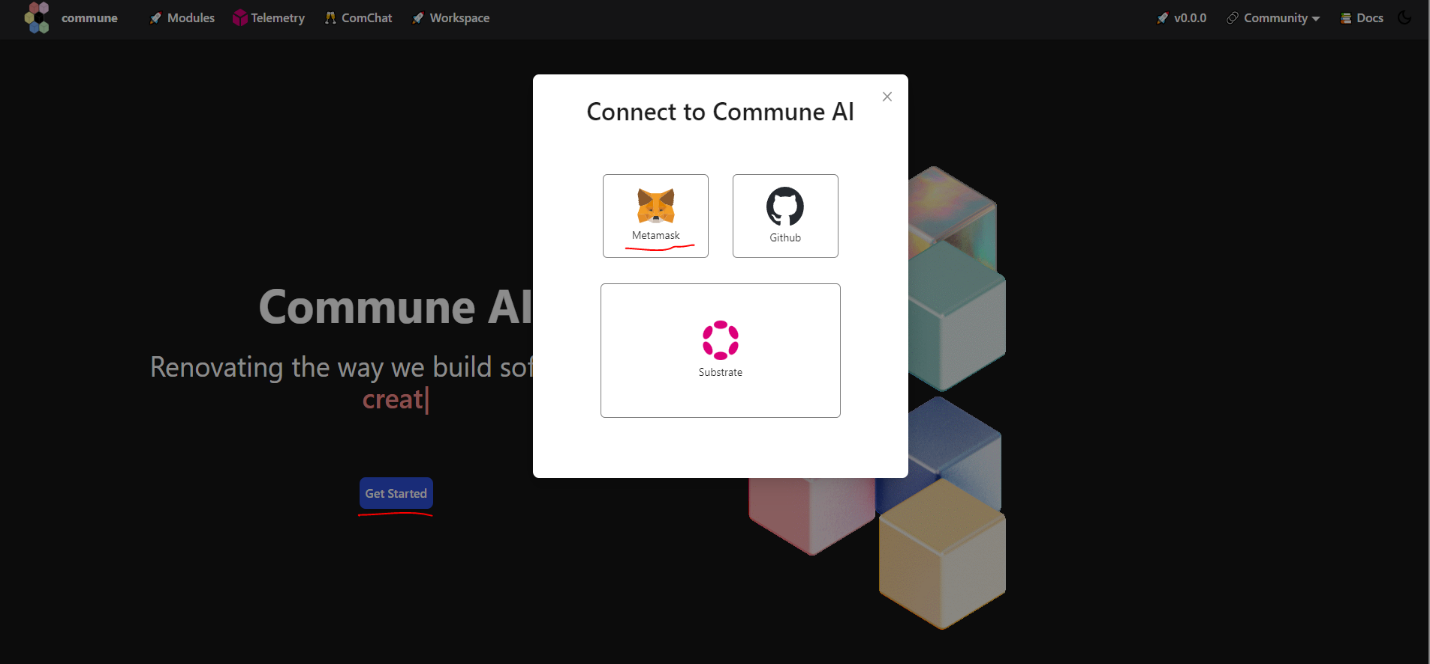


This figure shows you the chain and node name we connect to substrate.

1. User Profile

After login with metamask, then we can go to user profile page.

* Login process

When user login with metamask, we can get the public key of user who signed with metamask.

We save this public key to database (postgresql).

The code is like below:

Frontend Side

export const saveMetaMaskAddress = (address: string) => async (dispatch: any) => {

    const body = JSON.stringify(

        {

            address

        }

    )

    dispatch({ type: LOADING })

    try { // const token = window.localStorage.getItem('token');

        const res = await fetch(`${API\_URL}/api/data-analysis/saveMetamask/`, {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

                Accept: 'application/json',

            },

            body: body

        })

        const data = await res.json()

        if (data.res === 'success') {

            dispatch({ type: SAVE\_METAMASK\_SUCCESS, payload: address })

        }

        dispatch({ type: DONE })

    }

    catch (e) {

        dispatch({ type: SAVE\_METAMASK\_FAILED })

        dispatch({ type: DONE })

    }

}

Backend Side:

class SaveMetamask(APIView):

    def post(self, request):

        address = request.data.get("address")

        if not address:

            return Response({"error": "Address is required"}, status=status.HTTP\_400\_BAD\_REQUEST)

        try:

            # Assuming you have the model defined in models.py

            wallet\_obj, created = LoginUserWalletAddress.objects.get\_or\_create(walletAddress=address)

            if created:

                return Response({"message": "Address saved successfully"}, status=status.HTTP\_201\_CREATED)

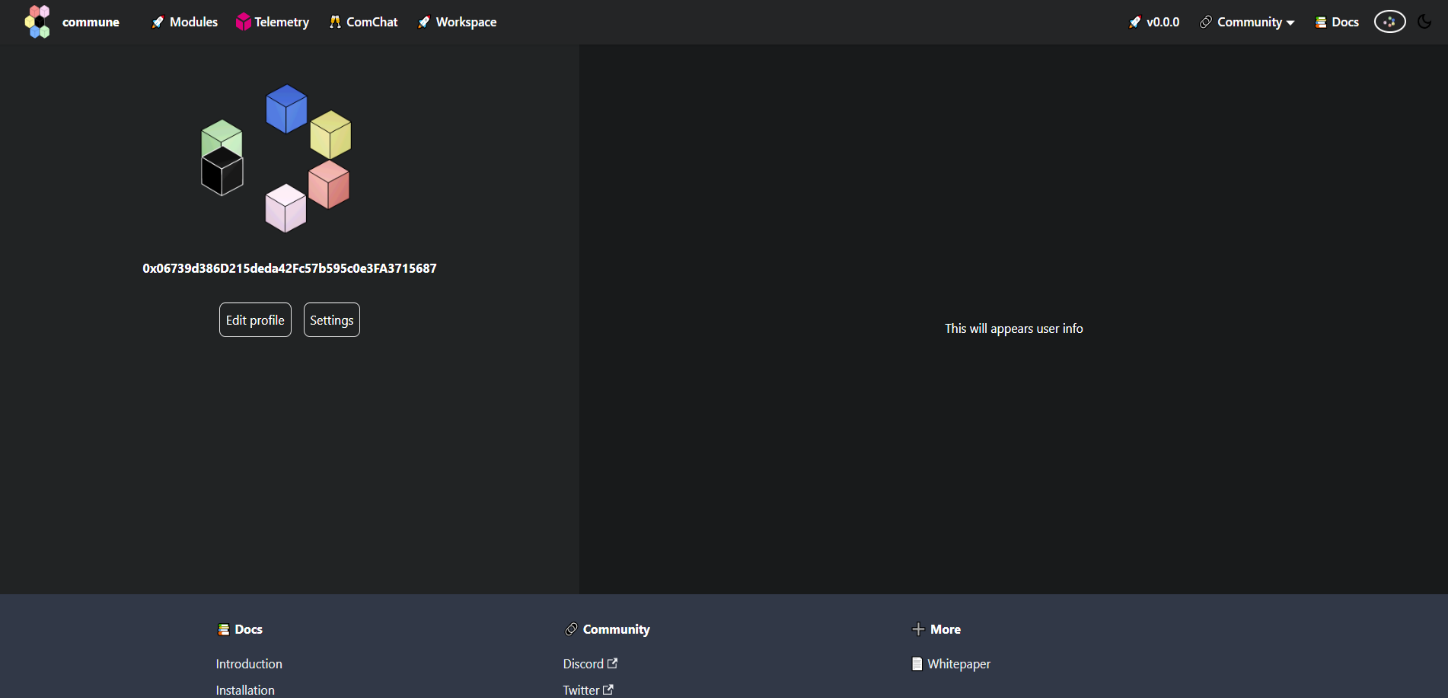
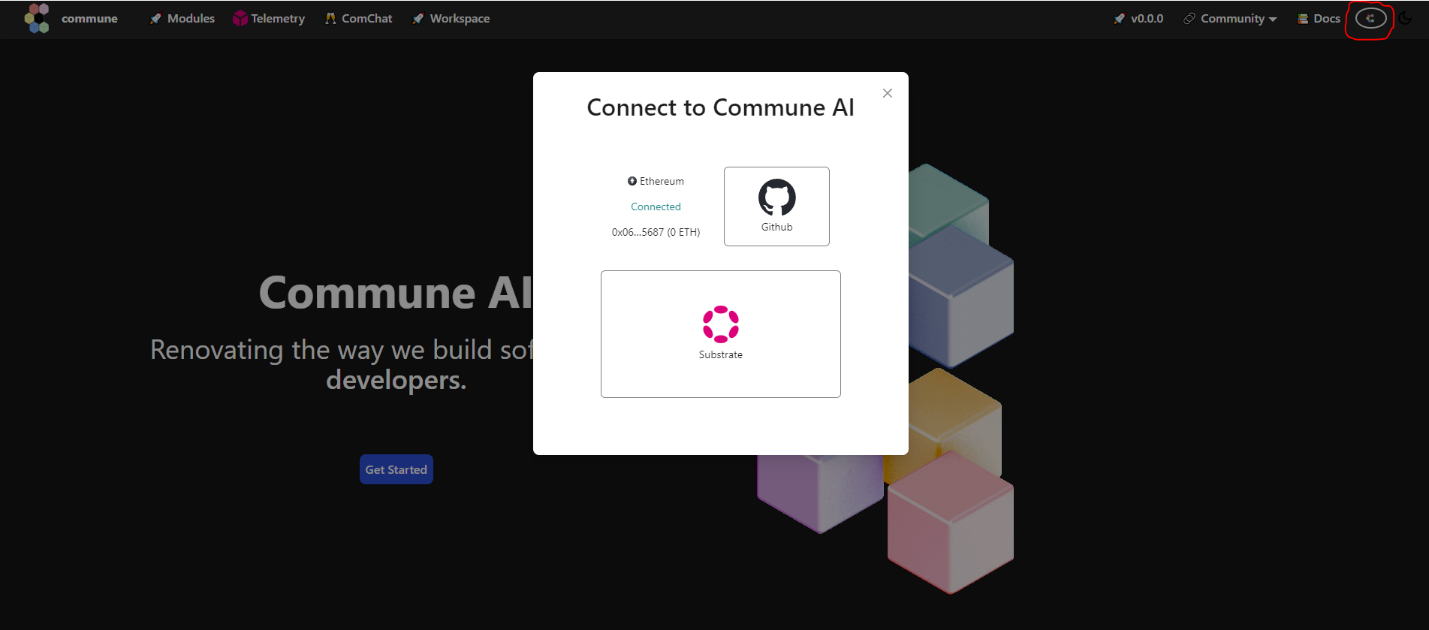
            else:

                return Response({"message": "Address already exists"}, status=status.HTTP\_200\_OK)

        except Exception as e:

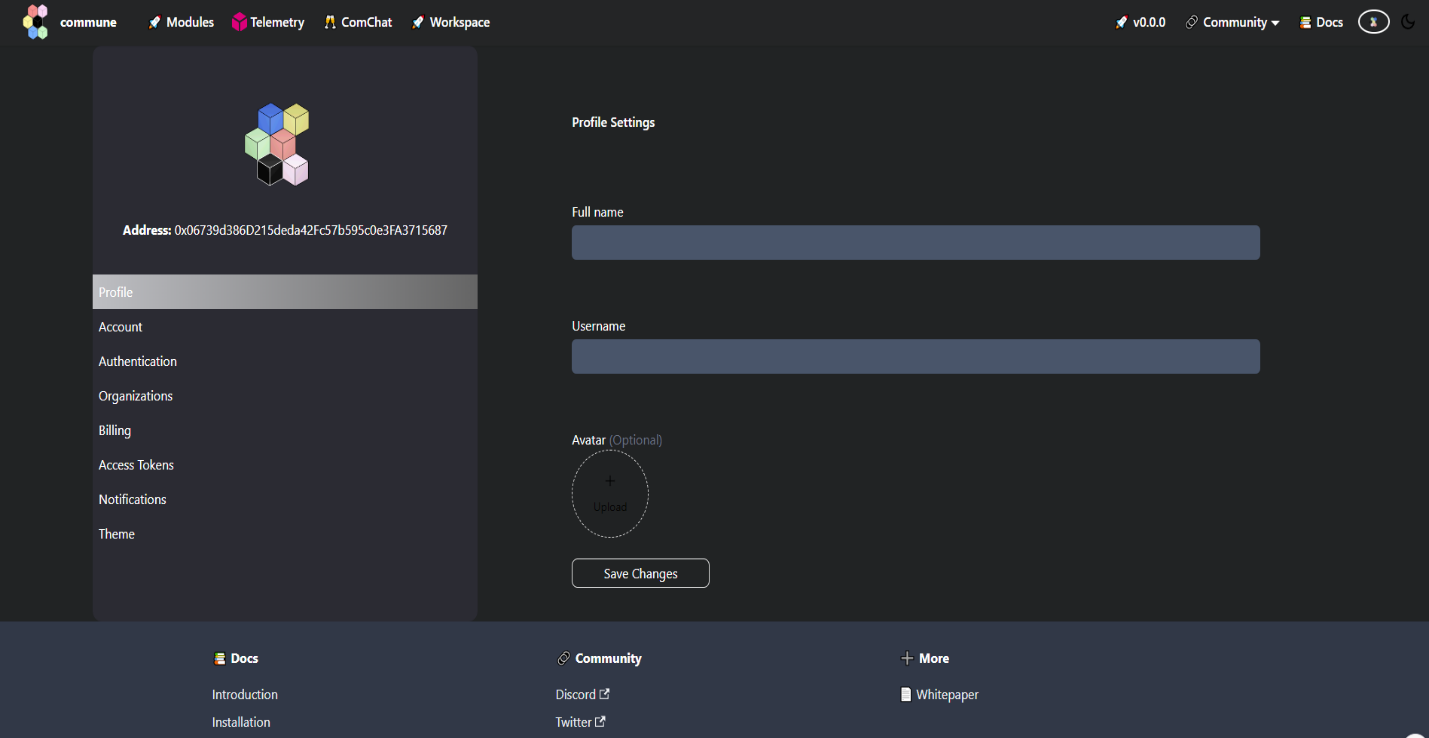
            return Response({"error": str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

* Go to User profile page

After login successful, then user can go to his profile page.

Then you can see your wallet address.

This figure is the user profile page.



* The items of user profile page

There are 8 items in user profile page such as profile and account, authentication, organizations, billing, access tokens, notifications, theme

The current updates are profile and account items

1. Profile Item

The above figure is also the screenshot of profile item.

Here, users can set their full name and user name and also avatar as an optional.

This is the backend code.

class UpdateUserProfile(APIView):

    permission\_classes = (AllowAny,)

    authentication\_classes = ()

    def post(self, request):

        user\_id = request.user.id  # Assuming the user is authenticated

        # Fetch the user instance

        user = get\_object\_or\_404(User, id=user\_id)

        # Assuming you have the required fields in your user model

        full\_name = request.data.get("full\_name")

        username = request.data.get("username")

        avatar = request.data.get("avatar")

        # Update user fields if provided

        if full\_name:

            user.full\_name = full\_name

        if username:

            user.username = username

        if avatar:

            # Handle avatar upload or save the file path to the user's profile

            # You may want to use a serializer to handle file upload and save logic

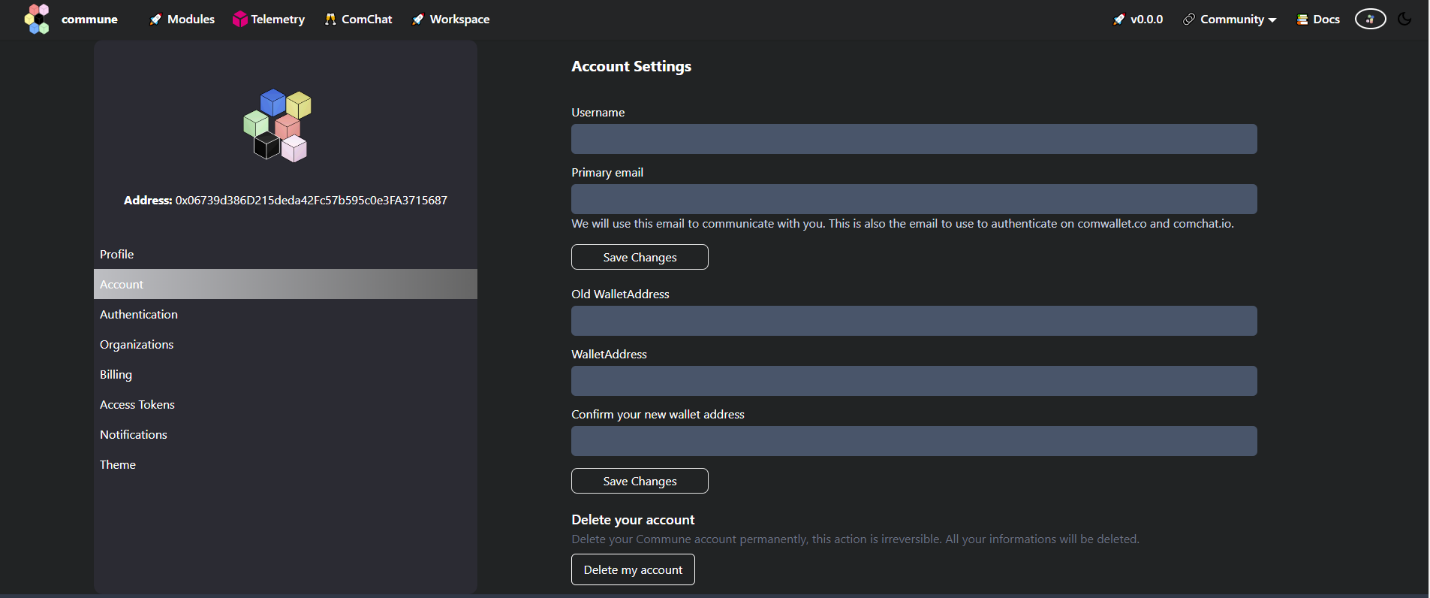
            user.avatar = avatar

            # Save the user instance

            user.save()

        return Response({"message": "Profile updated successfully"})

1. Account Item



In this item, users can set their account settings such as user name and primary email and also update their wallet address and delete their own account.

Below, I will show you the frontend and backend code

. Update the user name and primary email

export const updateWalletAddress = (oldWalletAddress: string, walletAddress: string) => async (dispatch: any) => {

    const body = JSON.stringify({

        oldWalletAddress,

        walletAddress

    })

    try { // const token = window.localStorage.getItem('token');

        const res = await fetch(`${API\_URL}/api/data-analysis/saveNewWalletAddress/`, {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

                Accept: 'application/json',

            },

            body: body

        })

        const data = await res.json()

        if (data.res === 'success') {

            dispatch({ type: UPDATE\_METAMASK\_SUCCESS })

        }

    }

    catch (e) {

        dispatch({ type: UPDATE\_METAMASK\_FAILED })

    }

}

class SaveNewWalletAddress(APIView):

    permission\_classes = (permissions.AllowAny,)

    authentication\_classes = ()

    def post(self, request):

        walletAddress = request.data.get("walletAddress")

        oldWalletAddress = request.data.get("oldWalletAddress")

        if not walletAddress or not oldWalletAddress:

            return Response(

                {"error": "Both walletAddress and oldWalletAddress are required"}, status=status.HTTP\_400\_BAD\_REQUEST

            )

        try:

            # Retrieve the old wallet object

            old\_wallet\_obj = LoginUserWalletAddress.objects.get(walletAddress=oldWalletAddress)

            # Update the wallet address

            old\_wallet\_obj.walletAddress = walletAddress

            old\_wallet\_obj.save()

            return Response({"message": "Wallet address updated successfully"}, status=status.HTTP\_200\_OK)

        except LoginUserWalletAddress.DoesNotExist:

            return Response({"error": "Old wallet address not found"}, status=status.HTTP\_404\_NOT\_FOUND)

        except Exception as e:

            return Response({"error": str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

. Delete user account

export const deleteUserAccount = (address: string) => async (dispatch: any) => {

    const body = JSON.stringify(

        {

            address

        }

    )

    try {

        const res = await fetch(`${API\_URL}/api/data-analysis/deleteAccount/`, {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

                Accept: 'application/json',

            },

            body: body

        })

        const data = await res.json()

        if (data.res === 'success') {

            dispatch({ type: DELETE\_ACCOUNT\_SUCCESS, payload: address })

        }

    }

    catch (e) {

        dispatch({ type: DELETE\_ACCOUNT\_FAILED, payload: address })

    }

}

class DeleteAccount(APIView):

    permission\_classes = (permissions.AllowAny,)

    authentication\_classes = ()

    def post(self, request):

        address = request.data.get("address")

        if not address:

            return Response({"error": "Address is required"}, status=status.HTTP\_400\_BAD\_REQUEST)

        try:

            # Retrieve the wallet object

            wallet\_obj = LoginUserWalletAddress.objects.get(walletAddress=address)

            # Delete the wallet object

            wallet\_obj.delete()

            return Response({"message": "Account deleted successfully"}, status=status.HTTP\_200\_OK)

        except LoginUserWalletAddress.DoesNotExist:

            return Response({"error": "Wallet address not found"}, status=status.HTTP\_404\_NOT\_FOUND)

        except Exception as e:

            return Response({"error": str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)