## **The Generative Art NFT Hub Powered by AI**

## **Index**

1. [Introduce](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#1-gi%E1%BB%9Bi-thi%E1%BB%87u)
2. [System architecture](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#2-ki%E1%BA%BFn-tr%C3%BAc-h%E1%BB%87-th%E1%BB%91ng)
3. [Frontend part](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#3-ph%E1%BA%A7n-frontend)
   * 3.1 [Wallet Management](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#31-qu%E1%BA%A3n-l%C3%BD-v%C3%AD-wallet-management)
   * 3.2 [Create and manage NFTs](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#32-t%E1%BA%A1o-v%C3%A0-qu%E1%BA%A3n-l%C3%BD-nft)
   * 3.3 [Marketplace](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#33-marketplace)
   * 3.4 [User interface](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#34-giao-di%E1%BB%87n-ng%C6%B0%E1%BB%9Di-d%C3%B9ng)
4. [Backend part](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#4-ph%E1%BA%A7n-backend)
   * 4.1 [API handling](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#41-x%E1%BB%AD-l%C3%BD-api)
   * 4.2 [Database](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#42-c%C6%A1-s%E1%BB%9F-d%E1%BB%AF-li%E1%BB%87u)
   * 4.3 [Create photos and videos with AI](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#43-sinh-%E1%BA%A3nh-v%C3%A0-video-v%E1%BB%9Bi-ai)
5. [Smart Contract](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#5-smart-contract)
   * 5.1 [Overview](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#51-t%E1%BB%95ng-quan)
   * 5.2 [Main functions](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#52-c%C3%A1c-ch%E1%BB%A9c-n%C4%83ng-ch%C3%ADnh)
6. [Smart chatbots](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#6-chatbot-th%C3%B4ng-minh)
   * 6.1 [Architecture](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#61-ki%E1%BA%BFn-tr%C3%BAc)
   * 6.2 [Message processing](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#62-x%E1%BB%AD-l%C3%BD-tin-nh%E1%BA%AFn)
   * 6.3 [Integrated vector search](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#63-t%C3%ADch-h%E1%BB%A3p-vector-search)
7. [NFT ranking algorithm](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#7-thu%E1%BA%ADt-to%C3%A1n-x%E1%BA%BFp-h%E1%BA%A1ng-nft)
8. [Optimization and performance](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#8-t%E1%BB%91i-%C6%B0u-h%C3%B3a-v%C3%A0-hi%E1%BB%87u-su%E1%BA%A5t)
9. [Conclude](https://claude.ai/chat/2a7d5b2d-523d-46cc-b95e-7ef0bcf11f3d#9-k%E1%BA%BFt-lu%E1%BA%ADn)

## **1. Introduction**

Generative Art Hub is a blockchain-integrated platform that allows users to create, buy, sell and exchange AI-generated art in the form of NFT (Non-Fungible Token) on the Forma blockchain. This system combines AI art generation technology with blockchain smart contracts, creating a complete digital art market.

The system is built based on modern technologies:

* **Frontend**: HTML, CSS, JavaScript with libraries like Bootstrap, Web3.js
* **Backend**: Python with FastAPI, MongoDB
* **AI Models**: Stable Diffusion, AnimateDiff
* **Blockchain**: Forma blockchain (based on Tendermint) with Smart Contract written in Solidity
* **Vector Database**: FAISS for semantic search

The project aims to create an ecosystem where users can easily create AI art, turn them into NFTs, and trade them on the decentralized market.

## **2. System architecture**

The Generative Art Hub system is built according to a microservices architecture, with the following components:

1. **Frontend**: Web user interface, handling interaction and display, connecting to digital wallets and blockchain.
2. **Backend API**: FastAPI server handles requests from the frontend, interacting with databases and other services.
3. **AI Generation Service**: Image and video generation service using AI models (Stable Diffusion and AnimateDiff).
4. **Blockchain Interface**: Connect to Forma blockchain via Web3.js and smart contract.
5. **Database Layer**: MongoDB stores user information, NFTs, and transactions.
6. **Vector Search**: Uses FAISS and Sentence Transformers to support semantic search.
7. **WebSocket Service**: Supports real-time interaction and chatbots.

Basic operating flow of the system:

1. User connects wallet and authenticates
2. Users use the tool to generate art from text descriptions
3. The created art is saved on IPFS and metadata is written to the blockchain
4. NFTs are created and users can trade them on the marketplace

## **3. Frontend part**

### **3.1 Wallet Management**

The frontend uses Web3.js to interact with Ethereum wallets like MetaMask. Main functions include:

* **Wallet connection**: Supports MetaMask, Leap Wallet and Keplr Wallet.

async function connectMetaMask() {

if (typeof window.ethereum !== "undefined") {

try {

const accounts = await window.ethereum.request({ method: "eth\_requestAccounts" });

const account = accounts[0];

console.log("MetaMask wallet connected:", account);

// Add Forma Testnet to MetaMask

try {

await window.ethereum.request({

method: 'wallet\_addEthereumChain',

params: [{

chainId: '0xF043B',

chainName: 'Forma Sketchpad',

nativeCurrency: {

name: 'Test TIA',

symbol: 'TIA',

decimals: 18

},

rpcUrls: ['https://rpc.sketchpad-1.forma.art/'],

blockExplorerUrls: ['https://explorer.sketchpad-1.forma.art/']

}]

});

} catch (switchError) {

console.error("Error adding/switching network:", switchError);

}

localStorage.setItem("walletAddress", account);

window.location.href = "index.html";

} catch (error) {

console.error("Error connecting MetaMask:", error);

}

} else {

alert("Please install MetaMask!");

}

}

* **Session management**: Save wallet address and session information in localStorage to maintain logged in status.

function checkWalletConnection() {

const savedAddress = localStorage.getItem("walletAddress");

const userProfile = localStorage.getItem("userProfile");

if (savedAddress) {

const shortAddress = `${savedAddress.substring(0, 6)}...${savedAddress.substring(savedAddress.length - 4)}`;

// Display logged in user information

} else {

showConnectButton();

}

}

* **Blockchain event handling**: Listen for account and network change events.

if (typeof window.ethereum !== "undefined") {

window.ethereum.on("accountsChanged", function (accounts) {

if (accounts.length === 0) {

WalletManager.disconnect();

} else {

localStorage.setItem("walletAddress", accounts[0]);

window.location.reload();

}

});

}

* **Network Switch**: Make sure the user is connected to the correct Forma network.

async function checkNetwork() {

try {

await window.ethereum.request({

method: "wallet\_switchEthereumChain",

params: [{ chainId: web3.utils.toHex(FORMA\_CHAIN\_ID) }],

});

} catch (switchError) {

if (switchError.code === 4902) {

// Add network if not already there

}

throw switchError;

}

}

### **3.2 Create and manage NFTs**

The frontend provides tools to create and manage NFTs:

* **Born art AI**: Allows users to create photos or videos from the prompt.

async function handleGenerateImage() {

const loadingSpinner = document.getElementById("loadingSpinner");

const previewPlaceholder = document.getElementById("preview-placeholder");

const previewImageContainer = document.querySelector(".preview-image");

const prompt = document.getElementById("prompt").value.trim();

if (!prompt) {

alert("Please enter a prompt to generate the image");

return;

}

try {

// Show loading spinner and hide placeholder

document.getElementById("loadingSpinner").classList.add("show");

document.getElementById("preview-placeholder").style.display = "none";

const formData = {

num\_inference\_steps: parseInt(document.getElementById("inferenceSteps").value),

guidance\_scale: parseFloat(document.getElementById("guidanceScale").value),

width: parseInt(document.getElementById("width").value || 512),

height: parseInt(document.getElementById("height").value || 512),

user\_prompt: prompt,

};

const response = await fetch("http://127.0.0.1:8000/api/generate-art", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify(formData),

});

if (!response.ok) {

throw new Error(`HTTP error! status: ${response.status}`);

}

const result = await response.json();

// Display the resulting image

if (result.image) {

previewImageContainer.innerHTML = "";

const previewImage = document.createElement("img");

previewImage.src = result.image;

previewImage.style.width = "100%";

previewImage.style.height = "100%";

previewImage.style.objectFit = "cover";

previewImageContainer.appendChild(previewImage);

}

} catch (error) {

console.error("Error generating image:", error);

alert("Failed to generate image. Please try again.");

}

}

* **Like NFT**: Create NFTs from generated AI art.

async function handleCreateNFT(name, description, file) {

try {

// Upload file to IPFS

updateLoadingStep("image", "loading");

const fileHash = await uploadToIPFS(file);

updateLoadingStep("image", "success");

// Prepare metadata

const metadata = {

name: name.trim(),

description: description ? description.trim() : "",

image: fileHash,

// Other metadata

};

// Upload metadata to IPFS

updateLoadingStep("metadata", "loading");

const metadataHash = await uploadToIPFS(metadata);

updateLoadingStep("metadata", "success");

// Mint NFT on blockchain

updateLoadingStep("mint", "loading");

const tx = await nftContract.methods.publicMint(metadataHash).send({

from: userAddress,

value: web3.utils.toWei(MINT\_PRICE, "ether"),

});

// Get tokenId from transfer event

const mintEvent = tx.events.Transfer;

const tokenId = mintEvent.returnValues.tokenId;

updateLoadingStep("mint", "success");

// Save NFT to database

await saveNFTToServer(userAddress, {

metadataUri: metadataHash,

tokenId: parseInt(tokenId),

fileType: fileType,

});

return {

tx,

fileHash,

metadataHash,

metadata,

tokenId,

fileType,

};

} catch (error) {

console.error("Mint error:", error);

throw error;

}

}

* **Manage NFT listings**: Display and manage the list of created NFTs.

async function loadCollectedNFTs() {

const collectedNftsDiv = document.getElementById("collected-nfts");

const loadingDiv = document.getElementById("loading-nfts");

const noNftsDiv = document.getElementById("no-nfts");

try {

loadingDiv.style.display = "block";

collectedNftsDiv.innerHTML = "";

noNftsDiv.style.display = "none";

const web3 = new Web3(window.ethereum);

const accounts = await window.ethereum.request({

method: "eth\_requestAccounts",

});

const userAddress = accounts[0];

// Initialize contract

const nftContract = new web3.eth.Contract(contractABI, contractAddress);

// Get the total number of NFTs of the user

const balance = await nftContract.methods.balanceOf(userAddress).call();

console.log("NFT balance:", balance);

if (balance == 0) {

loadingDiv.style.display = "none";

noNftsDiv.style.display = "block";

return;

}

// Get a list of user's token IDs

const tokenIds = await nftContract.methods.tokensOfOwner(userAddress).call();

// Load metadata for each NFT

const nftPromises = tokenIds.map(async (tokenId) => {

try {

const tokenURI = await nftContract.methods.tokenURI(tokenId).call();

const metadata = await fetchMetadataWithTimeout(tokenURI);

const listingInfo = await nftContract.methods.getListingInfo(tokenId).call();

return {

tokenId,

metadata,

isListed: listingInfo.isListed,

price: listingInfo.price,

};

} catch (error) {

console.error(`Error loading NFT ${tokenId}:`, error);

return null;

}

});

// Wait for all promises to complete

const nftResults = await Promise.allSettled(nftPromises);

// Filter valid results

allNFTs = nftResults.filter((result) => result.status === "fulfilled" && result.value).map((result) => result.value);

// Render NFTs

await renderNFTs(allNFTs);

} catch (error) {

console.error("Error loading NFTs:", error);

} finally {

loadingDiv.style.display = "none";

}

}

* **Post NFTs to the marketplace**: Allows users to sell NFTs on the marketplace.

async function handleListNFT(tokenId, modalElement) {

// Get price from input

const priceInput = modalElement.querySelector("#listing-price");

const price = parseFloat(priceInput.value);

// Check for valid prices

if (!price || price <= 0) {

showNotification("Please enter a valid price", "error");

return;

}

try {

// Display processing messages

const confirmButton = modalElement.querySelector(".confirm-listing-btn");

confirmButton.textContent = "Processing...";

confirmButton.disabled = true;

// Initialize contract

const web3 = new Web3(window.ethereum);

const accounts = await window.ethereum.request({

method: "eth\_requestAccounts",

});

const userAddress = accounts[0];

const nftContract = new web3.eth.Contract(contractABI, contractAddress);

// Convert price from TIA to Wei

const priceInWei = web3.utils.toWei(price.toString(), "ether");

// Check for approval and set up if necessary

const isApprovedForAll = await nftContract.methods.isApprovedForAll(userAddress, contractAddress).call();

if (!isApprovedForAll) {

await nftContract.methods.setApprovalForAll(contractAddress, true).send({

from: userAddress

});

}

// Call function listNFTForSale from smart contract

const tx = await nftContract.methods.listNFTForSale(tokenId, priceInWei).send({

from: userAddress,

});

// Update MongoDB database

const apiBaseUrl = window.location.hostname.includes('localhost') ||

window.location.hostname.includes('127.0.0.1')

? 'http://localhost:8000' : '';

await fetch(`${apiBaseUrl}/api/update-nft-listing/${tokenId}`, {

method: 'PUT',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ price: price }),

});

// Close the modal and display the notification

modalElement.classList.add("closing");

setTimeout(() => {

document.body.removeChild(modalElement);

}, 300);

showNotification("NFT has been successfully listed!", "success");

loadCollectedNFTs();

} catch (error) {

console.error("Error listing NFT:", error);

showNotification("An error has occurred: " + (error.message || "NFTs cannot be listed for sale"), "error");

}

}

### **3.3 Marketplace**

Marketplace is an important part of the system, allowing users to buy and sell NFTs:

* **List of NFTs for sale**: Shows NFTs being sold with detailed information.

async function loadListedNFTs() {

const nftGrid = document.getElementById("nft-grid");

const loading = document.getElementById("loading");

const noNfts = document.getElementById("no-nfts");

try {

loading.style.display = "block";

nftGrid.innerHTML = "";

noNfts.style.display = "none";

const web3 = new Web3(window.ethereum);

const contract = new web3.eth.Contract(contractABI, contractAddress);

// Get past NFTListed events

const events = await contract.getPastEvents("NFTListed", {

fromBlock: 0,

toBlock: "latest",

});

if (events.length === 0) {

noNfts.style.display = "block";

return;

}

// Get information and render NFTs

const batch = new web3.BatchRequest();

const nftPromises = events.map((event) => {

const tokenId = event.returnValues.tokenId;

return new Promise((resolve) => {

batch.add(

contract.methods.getListingInfo(tokenId).call.request(async (error, listing) => {

if (error || !listing.isListed) {

resolve(null);

return;

}

try {

const tokenURI = await contract.methods.tokenURI(tokenId).call();

const metadata = await fetchMetadata(tokenURI);

const block = await web3.eth.getBlock(event.blockNumber);

const nftData = {

tokenId,

price: listing.price,

seller: listing.seller,

listed\_time: block.timestamp \* 1000,

metadata: {

...metadata,

isListed: true,

},

};

resolve(nftData);

} catch (err) {

resolve(null);

}

})

);

});

});

batch.execute();

const nfts = (await Promise.all(nftPromises)).filter((nft) => nft !== null);

globalNFTs = nfts;

renderNFTs(nfts);

} catch (error) {

noNfts.textContent = "Unable to load NFTs. Please check your connection and try again.";

noNfts.style.display = "block";

} finally {

loading.style.display = "none";

}

}

* **First NFT**: Function to buy NFTs from sellers.

async function buyNFT(tokenId) {

try {

const web3 = new Web3(window.ethereum);

const accounts = await web3.eth.requestAccounts();

const userAddress = accounts[0];

const contract = new web3.eth.Contract(contractABI, contractAddress);

const listing = await contract.methods.getListingInfo(tokenId).call();

if (!listing.isListed) {

throw new Error("NFT is not listed for sale");

}

window.notificationManager.addNotification({

message: "Processing your purchase...",

type: "info",

icon: "bi bi-hourglass-split",

});

const buyTx = await contract.methods.buyNFT(tokenId).send({

from: userAddress,

value: listing.price,

gasPrice: "25000000000",

gas: "250000",

});

if (buyTx.status) {

try {

await fetch("http://localhost:8000/api/record-nft-purchase", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({

tokenId: parseInt(tokenId),

buyer\_address: userAddress,

}),

});

window.notificationManager.addNotification({

message: "NFT purchased successfully!",

type: "success",

icon: "bi bi-check-circle",

});

} catch (apiError) {

// Silent fail for API error

}

}

} catch (error) {

window.notificationManager.addNotification({

message: error.message,

type: "error",

icon: "bi bi-x-circle",

});

}

}

* **Shopping cart**: Manage shopping cart for multiple NFT purchases.

class CartManager {

constructor() {

this.cart = JSON.parse(localStorage.getItem("cart")) || [];

this.badge = document.querySelector(".cart-badge");

this.cartList = document.querySelector(".cart-list");

this.totalAmount = document.querySelector(".total-amount");

this.clearCartBtn = document.querySelector(".clear-cart");

this.checkoutBtn = document.querySelector(".checkout-btn");

this.init();

}

init() {

this.updateCartBadge();

this.renderCart();

this.setupEventListeners();

}

addToCart(item) {

this.cart.push({

...item,

id: Date.now(),

timestamp: new Date(),

});

this.saveCart();

this.updateCartBadge();

this.renderCart();

}

// Other methods of CartManager

}

* **Filter and search**: Filter NFTs by criteria and search.

function searchNFTs(searchTerm) {

searchTerm = searchTerm.trim().toLowerCase();

if (searchTerm === "") {

renderNFTs(globalNFTs);

return;

}

const filteredNFTs = globalNFTs.filter((nft) =>

nft.metadata.name.toLowerCase().includes(searchTerm)

);

renderNFTs(filteredNFTs);

}

### **3.4 User interface**

Frontend is designed with interface components:

* **Profile and account**: Manage user information.

async function updateProfile(event) {

event.preventDefault();

const username = document.getElementById("username").value;

const avatarFile = document.getElementById("avatar").files[0];

const bio = document.getElementById("bio").value;

const walletAddress = localStorage.getItem("walletAddress");

const formData = new FormData();

formData.append("username", username);

formData.append("walletAddress", walletAddress);

formData.append("was", was);

if (avatarFile) {

formData.append("avatar", avatarFile);

}

try {

const response = await fetch("http://localhost:8000/api/update-profile", {

method: "POST",

body:formData,

});

if (!response.ok) {

throw new Error("Failed to update profile");

}

const result = await response.json();

document.getElementById("navbarUsername").textContent = result.username;

if (result.avatar) {

document.querySelector(".wallet-avatar").src = result.avatar;

}

alert("Profile updated successfully!");

} catch (error) {

console.error("Error updating profile:", error);

alert("Failed to update profile");

}

}

* **Carousel and page layout**: Featured NFT display.

async function loadCarouselNFTs() {

try {

const response = await fetch('http://localhost:8000/api/carousel-nfts');

const nfts = await response.json();

const carouselInner = document.querySelector('.carousel-inner');

const carouselIndicators = document.querySelector('.carousel-indicators');

if (!nfts || !Array.isArray(nfts) || nfts.length === 0) {

console.log('No NFTs found for carousel');

return;

}

// Clear existing content

carouselInner.innerHTML = '';

carouselIndicators.innerHTML = '';

// Add NFTs to carousel

nfts.forEach((nft, index) => {

// Add indicator and carousel item

});

// Initialize carousel

const carouselElement = document.querySelector('#carouselExampleIndicators');

const carousel = new bootstrap.Carousel(carouselElement, {

interval: 5000,

wrap: true

});

} catch (error) {

console.error('Error loading carousel NFTs:', error);

}

}

* **Notification**: The system notifies the user.

class NotificationManager {

constructor() {

this.notifications = JSON.parse(localStorage.getItem("notifications")) || [];

this.badge = document.querySelector(".notification-badge");

this.notificationList = document.querySelector(".notification-list");

this.markAllReadBtn = document.querySelector(".mark-all-read");

this.apiUrl = "http://localhost:8000/api";

this.init();

}

init() {

this.updateNotificationBadge();

this.renderNotifications();

this.setupEventListeners();

this.checkSellerNotifications();

}

addNotification(notification) {

const newNotification = {

id: Date.now(),

message: notification.message,

type: notification.type,

icon: notification.icon,

timestamp: new Date(),

read: false,

};

this.notifications.unshift(newNotification);

this.saveNotifications();

this.updateNotificationBadge();

this.renderNotifications();

}

// Other methods of NotificationManager

}

* **Search**: System-wide search function.

function displaySearchResults(nfts, users, query) {

let dropdown = document.querySelector(".search-results");

if (!dropdown) {

dropdown = document.createElement("div");

dropdown.className = "search-results";

document.querySelector(".search-box").appendChild(dropdown);

}

if ((!nfts || nfts.length === 0) && (!users || users.length === 0)) {

dropdown.innerHTML = '<div class="no-results">No results found</div>';

dropdown.style.display = "block";

return;

}

let html = "";

if (users && users.length > 0) {

html += '<div class="search-section"><h6>Users</h6>';

html += users

.map(user => `

<a href="profile.html?address=${user.walletAddress}" class="search-result-item">

<img src="${user.avatar || "../assets/user.png"}"

alt="${user.username || "User"}"

onerror="this.src='../assets/user.png'">

<div class="search-result-info">

<div class="search-result-name">${user.username || "Unnamed User"}</div>

<div class="search-result-address">${user.walletAddress}</div>

</div>

</a>

`)

.join("");

html += "</div>";

}

if (nfts && nfts.length > 0) {

html += '<div class="search-section"><h6>NFTs</h6>';

html += nfts

.map(nft => `

<a href="nft-details.html?tokenId=${nft.tokenId}" class="search-result-item">

<img src="${nft.image || "../assets/placeholder.jpg"}"

alt="${nft.name}"

onerror="this.src='../assets/placeholder.jpg'">

<div class="search-result-info">

<div class="search-result-name">${nft.name || "Unnamed NFT"}</div>

<div class="search-result-price">${nft.price ? nft.price + " TIA" : "No price"}</div>

</div>

</a>

`)

.join("");

html += "</div>";

}

dropdown.innerHTML = html

}

## **3.4 User interface (continued)**

* **Search**: System-wide search function (continued).

dropdown.innerHTML = html;

dropdown.style.display = "block";

}

document.addEventListener("click", function (e) {

const searchBox = document.querySelector(".search-box");

const dropdown = document.querySelector(".search-results");

if (!searchBox?.contains(e.target) && dropdown) {

dropdown.style.display = "none";

}

});

* **Show NFT details**: View NFT details and transaction history.

async function loadNFTDetails() {

const urlParams = new URLSearchParams(window.location.search);

const tokenId = urlParams.get("tokenId");

if (!tokenId) {

window.location.href = "marketplace.html";

return;

}

try {

await fetch(`http://localhost:8000/api/update-views/${tokenId}`, {

method: "POST",

headers: {

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

},

});

const web3 = new Web3(window.ethereum);

const contract = new web3.eth.Contract(contractABI, contractAddress);

const tokenURI = await contract.methods.tokenURI(tokenId).call();

const metadata = await fetchMetadata(tokenURI);

const listing = await contract.methods.getListingInfo(tokenId).call();

const owner = await contract.methods.ownerOf(tokenId).call();

// Update UI with NFT information

document.getElementById("nftName").textContent = metadata.name;

document.getElementById("nftDescription").textContent = metadata.description || "No description available";

document.getElementById("contractAddress").textContent = `${contractAddress.substring(0, 6)}...${contractAddress.substring(38)}`;

document.getElementById("tokenId").textContent = tokenId;

document.getElementById("nftPrice").innerHTML = `

<img src="../assets/celestia-icon.png" alt="Celestia" class="celestia-icon">

${web3.utils.fromWei(listing.price, "ether")}

<span class="currency"></span>

`;

// Load transaction history

loadTransactionHistory(contract, tokenId, web3);

} catch (error) {

showNotification("Error loading NFT details: " + error.message, "error");

}

}

* **Lazy Loading**: Optimize image data loading when needed.

function initLazyLoading() {

if (!('IntersectionObserver' in window)) return;

const imageObserver = new IntersectionObserver(

(entries) => {

entries.forEach(entry => {

if (entry.isIntersecting) {

const img = entry.target;

if (img.dataset.src) {

img.src = img.dataset.src;

img.removeAttribute('data-src');

imageObserver.unobserve(img);

}

}

});

},

{ rootMargin: '50px 0px', threshold: 0.1 }

);

document.querySelectorAll('img.lazy-image').forEach(img => {

if (img.dataset.src) {

imageObserver.observe(img);

}

});

}

## **4. Backend part**

### **4.1 API handling**

The backend is built with FastAPI to handle requests from the frontend:

* **API Endpoints**: Endpoints serving interaction with data.

@app.post("/api/save-nft")

async def save\_nft(request: CreateNFTRequest):

try:

existing\_nft = nft\_collection.find\_one({"tokenId": request.tokenId})

if existing\_nft:

update\_data = {

"$set": {

"metadataUri": request.metadataUri,

"last\_updated": datetime.now(),

}

}

nft\_collection.update\_one({"tokenId": request.tokenId}, update\_data)

return {

"message": "NFT updated successfully",

"nft\_id": str(existing\_nft["\_id"]),

}

else:

nft\_doc = {

"walletAddress": request.walletAddress.lower(),

"metadataUri": request.metadataUri,

"tokenId": request.tokenId,

"fileType": request.fileType,

"created\_at": datetime.now(),

"price": None,

"is\_listed": False,

"views": 0,

"listed\_time": None,

"score": 0,

}

result = nft\_collection.insert\_one(nft\_doc)

return {

"message": "NFT saved successfully",

"nft\_id": str(result.inserted\_id),

}

except Exception as e:

print("Error saving NFT:", str(e))

raise HTTPException(status\_code=500, detail=str(e))

* **Save user information**: Manage user information.

@app.post("/api/update-profile")

async def update\_profile(

username: str = Form(...),

walletAddress: str = Form(...),

avatar: Optional[UploadFile] = File(None),

bio: Optional[str] = Form(None)

):

try:

# Check if the user exists

existing\_user = collection.find\_one({"walletAddress": walletAddress.lower()})

if not existing\_user:

raise HTTPException(status\_code=404, detail="User not found")

# Prepare update data

update\_data = {

"username": username,

"bio": bio or "", # If there is no bio, store the empty string

"last\_updated": datetime.now()

}

# Handle avatar if any

if avatar and avatar.filename:

try:

# Check file type

if not avatar.content\_type.startswith('image/'):

raise HTTPException(status\_code=400, detail="File must be an image")

# Read and encode images

contents = await avatar.read()

encoded\_image = base64.b64encode(contents).decode('utf-8')

update\_data["avatar"] = f"data:{avatar.content\_type};base64,{encoded\_image}"

except Exception as e:

print(f"Error processing avatar: {str(e)}")

# Update user information

result = collection.update\_one(

{"walletAddress": walletAddress.lower()},

{"$set": update\_data}

)

if result.modified\_count == 0:

raise HTTPException(status\_code=400, detail="Profile update failed")

return {

"message": "Profile updated successfully",

"username": username,

"bio": bio or "",

"avatar": update\_data.get("avatar", existing\_user.get("avatar", ""))

}

except Exception as e:

print(f"Error updating profile: {str(e)}")

raise HTTPException(status\_code=500, detail=str(e))

* **WebSocket cho Chatbot**: Real-time interactive processing.

@app.websocket("/ws")

async def websocket\_endpoint(websocket: WebSocket):

await websocket.accept()

print("✅ WebSocket connection established.")

try:

while True:

data = await websocket.receive\_text()

print(f"📩 Receive message: {data}")

try:

response = chatbot\_process\_message(data, model, index, nft\_collection)

formatted = format\_response(response)

print(f"🔄 Send feedback: {formatted}")

await websocket.send\_text(formatted)

except Exception as e:

print(f"❌ Error processing message: {str(e)}")

await websocket.send\_text("Sorry, I don't understand your request.")

except Exception as e:

print(f"❌ Lỗi WebSocket: {str(e)}")

finally:

print("⚠ WebSocket connection closed.")

* **Born art AI**: Endpoints for photo and video generation with AI.

@app.post("/api/generate-art")

async def generate\_art\_endpoint(request: GenerateArtRequest):

try:

image = generate\_art(

num\_inference\_steps=request.num\_inference\_steps,

guidance\_scale=request.guidance\_scale,

width=request.width,

height=request.height,

user\_prompt=request.user\_prompt,

)

buffered = BytesIO()

image.save(buffered, format="PNG")

img\_str = base64.b64encode(buffered.getvalue()).decode()

response\_data = {"image": f"data:image/png;base64,{img\_str}"}

return response\_data

except Exception as e:

print("Error occurred:", str(e))

raise HTTPException(status\_code=500, detail=str(e))

### **4.2 Database**

The system uses MongoDB to store data:

* **Collections**:  
  + users: User information
  + nfts: NFT information
  + notifications: Notification
* **NFT data model**:

nft\_doc = {

"walletAddress": request.walletAddress.lower(),

"metadataUri": request.metadataUri,

"tokenId": request.tokenId,

"fileType": request.fileType,

"created\_at": datetime.now(),

"price": None,

"is\_listed": False,

"views": 0,

"listed\_time": None,

"score": 0,

}

* **User data model**:

user\_doc = {

"walletAddress": wallet,

"username": request\_data.username or f"User\_{wallet[:6]}",

"avatar": request\_data.avatar,

"bio": request\_data.bio or "",

"created\_at": datetime.now(),

"last\_updated": datetime.now(),

}

* **Message data model**:

class Notification(BaseModel):

recipient: str

message: str

type: str

icon: str

read: bool = False

timestamp: datetime = datetime.now()

### **4.3 Create photos and videos with AI**

The backend uses the Stable Diffusion model to generate images and AnimateDiff to generate videos:

* **Imaging with Stable Diffusion**:

def generate\_art(num\_inference\_steps, guidance\_scale, width, height, user\_prompt):

"""

Generate images based on input parameters.

"""

base\_prompt = (

f"mdjrny-v4 style {user\_prompt}, masterpiece, 8k uhd, "

"ultra-realistic, hyper detailed, volumetric lighting, cinematic composition, "

"dramatic lighting, ray tracing, subsurface scattering, octane render, unreal engine 5, "

"trending on artstation, award winning, professional photography, highly detailed, "

"sharp focus, rich colors, intricate details, elegant, luxurious, ethereal atmosphere, "

"perfect composition, color grading, post-processing, artistic masterpiece, featured on behance, "

"featured on artstation, NFT art, digital art"

)

negative\_prompt = (

"ugly, deformed, noisy, blurry, low quality, duplicate, mutated, extra limbs, "

"poorly drawn face, poorly drawn hands, distorted, underexposed, overexposed, "

"bad art, beginner art, amateur, watermark, signature, text"

)

try:

if device == "cuda":

with autocast(device):

output = pipe(

prompt=base\_prompt,

negative\_prompt=negative\_prompt,

num\_inference\_steps=num\_inference\_steps,

guidance\_scale=guidance\_scale,

width=width,

height=height

)

else:

output = pipe(

prompt=base\_prompt,

negative\_prompt=negative\_prompt,

num\_inference\_steps=num\_inference\_steps,

guidance\_scale=guidance\_scale,

width=width,

height=height

)

return output.images[0]

except Exception as e:

raise RuntimeError(f"Error during image generation: {e}")

* **Generate videos with AnimateDiff**:

def generate\_video(num\_inference\_steps, guidance\_scale, user\_prompt):

"""

Generate video based on input parameters.

"""

# Device configuration

device = "cuda" if torch.cuda.is\_available() else "cpu"

dtype = torch.float16 if torch.cuda.is\_available() else torch.float32

# Model information

step = num\_inference\_steps

repo = "ByteDance/AnimateDiff-Lightning"

ckpt = f"animatediff\_lightning\_{step}step\_diffusers.safetensors"

base = "prompthero/openjourney"

try:

# Download motion adapter

adapter = MotionAdapter().to(device, dtype)

adapter.load\_state\_dict(load\_file(hf\_hub\_download(repo, ckpt), device=device))

# Load main pipeline

pipe = AnimateDiffPipeline.from\_pretrained(base, motion\_adapter=adapter, torch\_dtype=dtype).to(device)

pipe.scheduler = EulerDiscreteScheduler.from\_config(pipe.scheduler.config, timestep\_spacing="trailing", beta\_schedule="linear")

# Create base prompt

base\_prompt = (

f"mdjrny-v4 style {user\_prompt}, masterpiece, 8k uhd, "

"ultra-realistic, hyper detailed, volumetric lighting, cinematic composition, "

"dramatic lighting, ray tracing, subsurface scattering, octane render, unreal engine 5, "

"trending on artstation, award winning, professional photography, highly detailed, "

"sharp focus, rich colors, intricate details, elegant, luxurious, ethereal atmosphere, "

"perfect composition, color grading, post-processing, artistic masterpiece, featured on behance, "

"featured on artstation, NFT art, digital art"

)

negative\_prompt = (

"ugly, deformed, noisy, blurry, low quality, duplicate, mutated, extra limbs, "

"poorly drawn face, poorly drawn hands, distorted, underexposed, overexposed, "

"bad art, beginner art, amateur, watermark, signature, text"

)

# Create animation

output = pipe(

prompt=base\_prompt,

negative\_prompt=negative\_prompt,

guidance\_scale=guidance\_scale,

num\_inference\_steps=step

)

# Save video to buffer

video\_buffer = io.BytesIO()

imageio.mimsave(video\_buffer, frames, format='mp4', fps=fps)

video\_buffer.seek(0)

# Base64 encoding

video\_base64 = base64.b64encode(video\_buffer.read()).decode('utf-8')

return video\_base64

except Exception as e:

raise RuntimeError(f"Error during video generation: {e}")

## **5. Smart Contract**

### **5.1 Overview**

Smart contract for the Generative Art Hub system is built based on the ERC-721 standard with additional functions for the marketplace:

const contractAddress = "0xDeE9B116D1D712B4147A8a85A5B960f048D99ff8";

Contract is deployed on Forma blockchain with the above address.

### **5.2 Main functions**

Smart contract provides the following functions:

* **Like NFT**: Create new NFT tokens.

function publicMint(string memory metadataURI) public payable {

// Ask the caller to pay the correct mint price

require(msg.value == mintPrice, "Incorrect payment amount");

// New Mint NFT for caller

\_mint(msg.sender, \_nextTokenId());

// Set URI metadata for token

\_setTokenURI(currentTokenId, metadataURI);

// Broadcast event

emit MetadataUpdate(currentTokenId);

}

* **List of NFTs for sale**: Bringing NFTs to market.

function listNFTForSale(uint256 tokenId, uint256 price) public {

// Check owner

require(ownerOf(tokenId) == msg.sender, "Not the owner");

// Check price

require(price > 0, "Price must be greater than zero");

// Save listing information

listings[tokenId] = Listing({

price: price,

seller: msg.sender,

isListed: true

});

// Broadcast event

emit NFTListed(tokenId, msg.sender, price);

}

* **First NFT**: Buy NFTs from the marketplace.

function buyNFT(uint256 tokenId) public payable {

Listing memory listing = listings[tokenId];

// Check if NFT is listed

require(listing.isListed, "NFT not listed for sale");

// Check the amount sent

require(msg.value >= listing.price, "Insufficient payment");

// Get seller information and price

address seller = listing.seller;

uint256 price = listing.price;

// Delete listing

delete listings[tokenId];

// Calculate transaction fees

uint256 fee = (price \* transactionFeePercentage) / 100;

uint256 sellerAmount = price - fee;

// Transfer money to the seller

payable(seller).transfer(sellerAmount);

// Transfer the NFT to the buyer

\_safeTransfer(seller, msg.sender, tokenId, "");

// Broadcast event

emit NFTSold(tokenId, seller, msg.sender, price);

}

* **Cancel listing**: Cancel NFT listing.

function cancelListing(uint256 tokenId) public {

// Check if the caller is a seller

require(listings[tokenId].seller == msg.sender, "Not the seller");

// Delete listing

delete listings[tokenId];

// Broadcast event

emit ListingCanceled(tokenId, msg.sender);

}

* **Update listing price**: Change the price of NFTs being sold.

function updateListingPrice(uint256 tokenId, uint256 newPrice) public {

// Check if the caller is a seller

require(listings[tokenId].seller == msg.sender, "Not the seller");

// Check new price

require(newPrice > 0, "Price must be greater than zero");

// Update price

listings[tokenId].price = newPrice;

// Broadcast event

emit NFTListed(tokenId, msg.sender, newPrice);

}

## **6. Smart chatbots**

### **6.1 Architecture**

Chatbot is built with an architecture including:

* **WebSocket**: Real-time connection to the frontend.
* **Vector Search**: Use FAISS and Sentence Transformers for semantic search.
* **MongoDB**: Store and retrieve NFT and user data.

Main ingredients:

# Load model Embedding

model = SentenceTransformer("intfloat/multilingual-e5-large")

# Initialize FAISS Index

vector\_dim = 1024 # vector size of the model

index = faiss.IndexFlatL2(vector\_dim)

# Function to load data from MongoDB into FAISS

def load\_data\_to\_faiss():

print("🔄 Loading data into FAISS...")

nfts = list(nft\_collection.find({}, {"\_id": 0, "metadataUri": 1}))

users = list(collection.find({}, {"\_id": 0, "username": 1, "walletAddress": 1}))

all\_texts = []

all\_ids = []

# Process data for inclusion in FAISS

...

all\_vectors = model.encode(all\_texts, convert\_to\_numpy=True)

index.add(all\_vectors)

print(f"✅ FAISS đã index {index.ntotal} vectors.")

### **6.2 Message processing**

Chatbot handles these types of messages from users:

def process\_message(message: str, model, index, nft\_collection, selenium\_utils=None) -> str:

"""

Process user messages based on keyword list.

"""

message = message.lower().strip()

if not message:

return "Please enter your question."

#1. Greetings

if any(greet in message for greet in KEYWORDS.get("greetings", [])):

return ("Hello! How can I help you today? (You can ask about NFTs, pricing, marketplace, art creation, user information, reporting, etc.)")

#2. Check the request for users list in index

if "what users are in this index page" in message:

return "SHOW\_TOP\_ARTISTS"

#3. Marketplace with search requests

marketplace\_keywords = KEYWORDS.get("marketplace", [])

search\_keywords = KEYWORDS.get("search", [])

if any(word in message for word in marketplace\_keywords) and any(word in message for word in search\_keywords):

try:

# Get search keywords from messages

if "keyword" in message:

keyword = message.split("keyword")[-1].strip()

elif "search" in message:

keyword = message.split("search")[-1].strip()

elif "search" in message:

keyword = message.split("search")[-1].strip()

else:

keyword = ""

if not keyword:

keyword = "CHECK"

print(f"Keyword extracted: '{keyword}'")

return f"REDIRECT:marketplace:search:{keyword}"

except Exception as e:

print(f"Error processing marketplace search: {str(e)}")

return "Sorry, I can't perform searches at this time."

# Other treatments...

# Fallback: if not recognized, use vector search

return vector\_search(message, model, index, nft\_collection)

### **6.3 Integrating vector search**

Chatbot uses FAISS for semantic search:

def vector\_search(query, model, index, nft\_collection, k=5):

"""

Search NFTs based on query content using FAISS.

"""

if not query:

return "Please provide search keywords."

try:

query\_vec = model.encode([query])

except Exception as e:

return f"Error while encoding query: {e}"

try:

distances, indices = index.search(np.array(query\_vec, dtype=np.float32), k=k)

except Exception as e:

return f"Error performing search in FAISS: {e}"

if indices[0][0] == -1:

return "⚠ No matching results found."

nft\_ids = [int(idx) for idx in indices[0] if idx != -1]

if not nft\_ids:

return "⚠ No matching NFTs found."

try:

matched\_nfts = list(nft\_collection.find(

{"tokenId": {"$in": nft\_ids}},

{"\_id": 0, "tokenId": 1, "metadataUri": 1, "price": 1}

))

except Exception as e:

return f"Error retrieving NFT data from database: {e}"

if not matched\_nfts:

return "⚠ No matching NFTs found after querying the database."

return {"nfts": matched\_nfts}

## **7. NFT ranking algorithm**

The system uses a special algorithm to rank NFTs based on popularity, listing time, and user selling history:

def calculate\_nft\_score(

views: int, hours\_since\_listed: float, user\_sales: int

) -> float:

try:

# Constant

lambda\_ = 1 # λ = 1 instead of 0.01

k = 2 # k = 2

g = 1.8 # g = 1.8

# Calculate user factor: 1 + λ × userSales

user\_factor = 1 + (lambda\_ \* user\_sales)

# Calculate the time denominator: (hours\_since\_listed + k)^g

time\_denominator = pow(hours\_since\_listed + k, g)

# Tính score: (views × user\_factor) / time\_denominator

score = (views \* user\_factor) / time\_denominator

return score

except Exception as e:

print(f"Error calculating score: {e}")

return 0

This formula is based on the principle:

* NFTs with more views are ranked higher
* NFTs from sellers with many successful transactions are given priority
* Newly listed NFTs have priority over old NFTs

The algorithm is applied in the system to:

* Rank NFTs on the marketplace
* Choose featured NFTs for the carousel
* Recommend NFTs to users

## **8. Optimization and performance**

The system applies many performance optimization techniques:

* **Lazy Loading**: Load images when needed.

function initializeLazyLoading() {

if (!('IntersectionObserver' in window)) return;

const imageObserver = new IntersectionObserver(

(entries, observer) => {

entries.forEach((entry) => {

if (entry.isIntersecting) {

const img = entry.target;

if (img.dataset.src) {

// Preload image

const preloadImg = new Image();

preloadImg.onload = () => {

img.src = img.dataset.src;

img.removeAttribute("data-src");

img.classList.remove("loading");

};

preloadImg.onerror = () => {

img.src = "../assets/placeholder.jpg";

img.removeAttribute("data-src");

img.classList.remove("loading");

};

preloadImg.src = img.dataset.src;

observer.unobserve(img);

}

}

});

},

{

rootMargin: "200px 0px", // Preload images 200px before they come into view

threshold: 0.01,

}

);

document.querySelectorAll(".explore-nft-preview[data-src]").forEach((img) => {

imageObserver.observe(img);

});

}

* **Infinite Scrolling**: Load data by page as user scrolls.

function setupInfiniteScroll() {

// Delete old sentinel if any

const oldSentinel = document.getElementById("users-sentinel");

if (oldSentinel) {

oldSentinel.remove();

}

}

## **9. Security & Performance**

### **9.1. Security measures**

* **Verify digital signature** to ensure valid access.
* **Use HTTPS and CORS security** for APIs.
* **Authentication on smart contract** to prevent transaction manipulation.

### **9.2. Optimize performance**

* **Offload blockchain data** using MongoDB cache.
* **Use lazy loading** for NFT images.
* **Optimize smart contracts** to reduce gas fees.

## **10. Conclude**

Generative Art Hub is a powerful NFT platform that integrates blockchain to ensure transparency and ownership of artwork. The system has been optimally designed for security, performance and user experience.