2022

Web 3.0 Application: Music NFTs Marketplace

BLOCKCHAIN & CRYPTOCURRENCIES

IFRAH, KÉVIN

TECHNION, ISRAEL

Abstract

The music industry has faced changes since the beginning of the century. “Consuming without owning” are the main pillar of the Web 2.0 streaming platforms. This model leads to trust a third party to put in relation artists and fans. It also leads to have an unequal competition between artists, an unequal distribution of royalties and inability to buy music content online.

My solution is Music is to create an NFT Marketplace with a streaming feature. This Web 3.0 Application will allow the artists to mint their music content as NFT. Fans will be able to stream, buy and resell music content while reversing royalty fee to the deployer’s collection. The solution will ensure Data Availability and Authenticity thank to the use of IPFS Storage. The final Frontend will have at first a Home Page where all Minted Music will be available to be streamed or buy, a Tokens Page where each Music NFT buy by an user will be available on its dashboard, and a Resell Page where a seller will be able to track its in-sale and sold NFTs. The solution use Solidity to write its Smart Contract, Hardhat as a Development Framework, IPFS for Metadata Storage and can be implemented directly in Ethereum Network or in one of its layers 2 such as Polygon Network.

This paper also provides you a Smart Contracts framework that you can implement on other Ethereum Layer 2 that doesn’t use Solidity such as StartNet which uses Cairo Programming Language.

Problem

Streaming is a type of multimedia technology that delivers video and audio content to a device connected to the Internet. This allows you to access content (TV, movies, music, podcasts) at any time, on a computer or mobile device, regardless of a broadcast schedule.

Consume without owning. That's how to define streaming in three words. Streaming is the streaming of content on the Internet. This technology allows you to watch a video or listen to music online, without owning it. The data is stored in the server of a platform and can be accessed at any time. Therefore, the user only reads the files.

Subscribing to a music streaming service allows you to discover new artists, explore different genres or listen again to great classics. Among them, you will find Spotify, Amazon Prime Music, Apple Music, YouTube Music Premium, Deezer, Google Play Music, Tidal or even Qobuz.

As innovative as it may seems, music streaming has turned the music industry upside down by rethinking the wealth distribution systems between the company and the artists it hosts.

Today, 4 key problems are observable in this model:

- The need to trust a third party;
- Unequal competition between artists;
- Unequal distribution of royalties;

- Inability to buy music content online.

Web2.0 Based App & Third-Party Reliability For Data Storage

Web2.0 refers to the second generation of internet services, which focused on enabling users to interact with content on the web. Web 2.0 fostered the growth of user-generated content alongside interoperability and usability for end users. The second-generation web does not focus on modifying any technical specifications. On the contrary, it emphasizes changing the design of web pages and the ways of using them. Web 2.0 encouraged collaboration and interaction among users in P2P transactions, thereby setting the stage for e-commerce and social media platforms.

Web 2.0 has been accompanied by several failures:

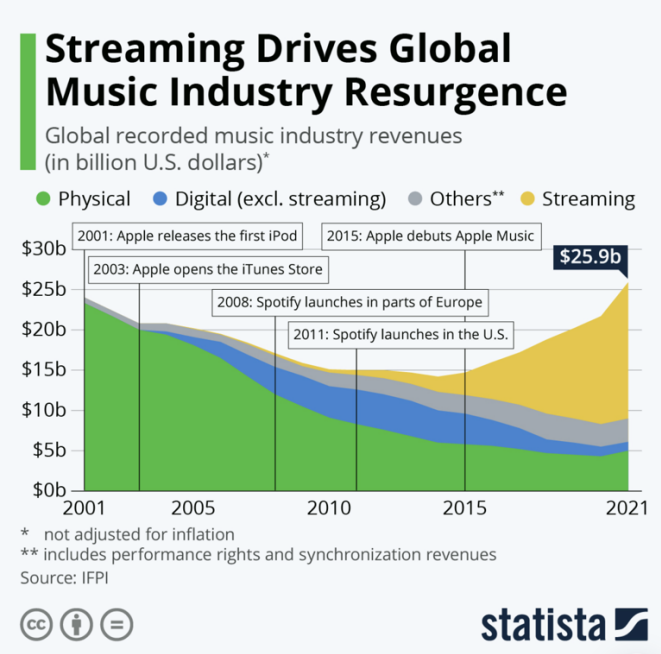
1. The lack of web governance due to the centralisation of power in the hands of large platforms: GAFAMs own, exploit and monetise the digital identities of individuals, sometimes without their knowledge;
2. Digital identity management systems are vulnerable: the increasing number of security breaches and other obvious failures is a clear symptom of the limitations of the centralised digital identity management model;
3. Multiple digital identities mean multiple points of failure: with an average of 150 Internet accounts for each user, multiple digital identities create real headaches both for users - who often use the same passwords to access multiple accounts - and for corporate information systems, which have to work harder to secure user data.

A streaming application like Spotify builds its business model like many Web 2.0 applications:

1. You can access the app thanks to a browser or a mobile device.
2. The browser/mobile device talks to a web server where all the code user data, artist data, and audio files are stored.
3. Whenever you hit play on one of your playlist, a request is made to serve the corresponding audio files back from the server to your mobile device.

All user information (artists and customers) is stored in private company databases. It is difficult to ensure the security of the data stored. No guarantee can be brought as for the impossibility of having a breach and thus leaks of sensitive data. An ethical question can also be raised about the way the platform uses these different data for personal purposes.

No Purchase Possibility on Streaming Platform



The chart above highlights the global recorded music industry revenues (in billion U.S. Dollars). It compares the revenues that generate different format: Physical, Digital, Streaming and Others (included performance rights and synchronisation revenues). The graph clearly highlights the drop in sales of physical and digital formats, stremaing exlusive since the years 2001. On the other hand, music consumption via streaming has undergone a clear evolution and is now the leading consumption method.

Streaming has replaced a model of selling artistic pieces with a model of generating clicks as a source of income. Illegal downloading is one of the reasons why this model was created. Anyway, if a user can get free and unlimited music content through downloading, we might as well bring this database to him in a simpler and more centralized way. Platforms such as Spotify are therefore these centralizers of music data. However, their strategy only stops at consumption without possession. An artist will not be able to sell his works on the platform because no service is implemented for that. He will have to do this through a marketplace provided for this purpose.

Unfair competition between artists

The click replaces the sale. As we have stated, the income distributed to artists is proportional to the amount of clicks they receive and since selling is not possible on these platforms. The applications are looking for trends rather than talents. Indeed, the music streaming platforms can analyze the trends to make them have a more important weight in their referencing. But giving so much power to a pinch of actors can create avoidance behavior. Indeed, these platforms can also decide (often linked to the highest bidder) who will make the front page of their feed tomorrow.

If you discover an up-and-coming artist that you think has a chance of being a big hit in the future, Spotify or any other streaming platform doesn't offer you a convenient way to bet on that artist's future success. Fans have to settle for collecting physical merchandise sold by the artist or attending their concerts to support them. From the artist's point of view, their revenue from Spotify is based solely on the number of streams they receive, not on the amount of support they receive from their fans. For example, the number of streams for two artists may be equal, but the number of fans who buy signed copies of their work may be significantly different.

Unfair Royalty Fee Distribution

It is easy to understand with the above mentioned issues that the power given to these platforms and the pay-per-click model creates a certain oppacity as to the way of distributing revenues to artists. In addition, the removal of the ability to purchase tracks and unlimited use of these works effectively removes the copyright fees that an artist is supposed to receive.

Solution

CRS Music Player is an Ethereum Web 3.0-based music player app solution that you can access through a web browser with a blockchain wallet extension like Metamask.

Web 3.0 is the decentralized Internet - based on peer-to-peer technologies like blockchain - that allows every Internet user to have full control over their personal data and, more broadly, to actively participate in the governance of the web. By enshrining blockchain protocols, Web 3.0 aims to give Internet users back control over their personal data.

The project remains on the construction of a Web 3.0 Music Player that allows artists to publish their musical works on the platform. These works can be streamed by its users. If they feel like supporting the artist, they will have the possibility to buy and resell music titles in the form of NFT. Each collection will represent a set of tracks from an album. The artist of the songs represented as NFTs will receive a royalty for each NFT sold. He will have the possibility to mint his works in limited quantity which will allow to come back on a system of rarity which was known at the time of the releases of physical copies (Vinyl, Cassette, CD). Indeed, each NFT Musical purchased will contain key information that will allow to attest the authenticity of each work and thus avoid the illegal reproduction of these. The website will be written in JavaScript, HTML and CSS and the Smart Contracts will be written in Solidity.

The interface will allow :

- To read and write data directly on the Blockchain where our code will be written in Ethereum Smart Contracts deployed on the Ethereum Blockchain ;
- All audio files to be stored on a decentralized file storage system called IPFS ;
- Personal collection dashboard (with buy and sell tracks).

Data Storage

One of the problems with blockchain technology is data storage. It is difficult to store large files. Indeed, the transaction cost to perform this kind of action would be too high.

Wishing to store relatively heavy music files, the CRS Music Player project uses different systems:

- Google Drive and web3.storage to store music files and images.
- NFT.storage to store the metadatas of the music files and images.

Web3.storage is an iteration of the gateways used to interact with the IPFS network and the Filecoin blockchain.

The IPFS protocol is a distributed pairwise file system that does not depend on centralized servers. Its purpose is to connect a set of computing devices with the same file system. In other words, IPFS provides a high-capacity, content-addressable block storage model using hyperlinks for access.

The Filecoin network is a decentralized marketplace for data storage. Users can store and retrieve their data on the network. Storage space is provided by the storage provided on the network. Traditionally, online storage providers and protocols store data on centralized servers using IP addresses. While IPFS stores data but does not ensure its availability, Filecoin is a paid service on top of IPFS Protocol that does.

NFT.Storage is a long-term storage service designed for NFT off-chain data (such as metadata, images, and other assets) up to 31GiB in size per individual download. Data is addressed by its content using IPFS, which means that the URI pointing to a data item ("ipfs://...") is completely unique for that data (using a content identifier or CID). IPFS URLs and CIDs can be used in NFTs and metadata to ensure that the NFT actually refers to the intended data (eliminating things like rug pulls and reliably verifying what content an NFT is associated with);

Unfortunately, these services are not optimal for data recovery. In fact, the recovery time is too long for the user to have a pleasant experience with the CRS Music Player. Most of the time, no music can be played because the pointer is not able to retrieve the data properly.

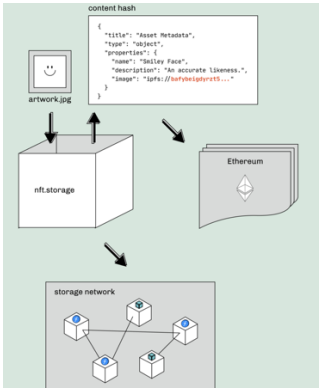
In order to define a solution that ensures data playback at a correct retrieval rate on the one hand and ensures data decentralization on the other, the process used is as follows:

1. Store the entire music collection and associated photos on Google Cloud;
2. Store the same data set on web3.0.storage. This will allow us to keep track of our download database and have automatic pinning from the IPFS network and FileCoin. This automation will allow us to ensure data recovery;
3. Create a .json file with the metadata (TokenID, Name, IPFS Image File Pointer, Google Drive Music File pointer, IPFS Music File pointer) of each song;

```
{
    "tokenId": 0,
    "name": "A Guy in an Office - Ocean Cloud (Tribal Vocal Work)",
    "audio": "https://storage.cloud.google.com/crs_soundsystem_storage/AGiaO%20-%20Ocean%20Cloud%20(Tribal%20Work%20-
%20Vocal).wav",
    "image": "https://storage.cloud.google.com/crs_soundsystem_storage/Ocean%20Cloud%20EP.png",
    "cryptoID": "https://ipfs.io/ipfs/QmbVTRt5tg8fpDVH8CG7FsHGwardJe3hnnUh1hJHEVRYVt?filename=AGiaO%20-
%20Ocean%20Cloud%20(Tribal%20Work%20-%20Vocal).wav"
}
```


4. Format the file as a .CAR file;
5. Store the .CAR file in the IPFS network using NFT.storage;
6. Insert the IPFS link NFT.storage into our smart contract as a metadata pointer that will retrieve the music content and all relevant information.

Duplicating the storage method allows to quickly retrieve the data for the streaming part and have an original NFT copy of the same data in a decentralized storage network. The URL metadata pointer is linked to the hash of a specific smart contract that allows to prove the authenticity of each artwork.




```
# Copy of the NFT.storage IPFS Link used to store the metadata of our NFT
Collection
<https://bafybeid2qyl54rqkkcnd5dcqdifubttxv5iicnnz3dsdojpnucvik7hgc4.ipfs.n
ftstorage.link/>
```

Network & Scalability

For our preliminary creation, the deployment of the contract was done on the local hardhat testnet representing the economics of the Ethereum network. Although the deployment of this one was a success, we know the problems of scalability of the transactions as well as the important gas costs that the Ethereum network knows. To overcome this, the application will have to be deployed on the Mainet Polygon (PoS) or on the Mainet Starknet (ZK-Rollups). In addition to avoiding the evolution in an energy-intensive Proof-of-Stake system, this will allow the user to avoid indescendent gas costs.

The following demo in the "Demo & UX" part will take place on the local hardhat testnet. Indeed, we will focus on the proper functioning of our smart contract. The implementation of

our contract on the Starnet network will require to translate our Solidity contract into a Cairo contract. However, the implementation of our contract on the Polygon network will only require to modify our`hardhat.config.js` and add the network to our Metamask.

For the Polygon Mumbai Testnet

```
require('dotenv').config();
require("@nomiclabs/hardhat-ethers");
require("@nomiclabs/hardhat-etherscan");
require("@nomiclabs/hardhat-waffle");

module.exports = {
  defaultNetwork: "matic",
  networks: {
    hardhat: {
    },
    matic: {
      url: "<https://rpc-mumbai.maticvigil.com>",
      accounts: [process.env.PRIVATE_KEY]
    }
  },
  etherscan: {
    apiKey: process.env.POLYGONSCAN_API_KEY
  },
  solidity: {
    version: "0.8.4",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
  paths: {
    artifacts: "./src/backend/artifacts",
    sources: "./src/backend/contracts",
    cache: "./src/backend/cache",
    tests: "./src/backend/test"
  },
}
```

For the mainet Polygon, the same script will be used however the `defaultNetwork` and `url` parts will have to be modified accordingly. The Main and Polygon will have to be added to the Wallet Metamask used.

It is also possible to use Alchemy in order to deploy with more ease and a pleasant user interface our Smart Contracts on different networks.

Alchemy Supernode is the most widely used blockchain API for Ethereum, Polygon, Solana, Arbitrum, Optimism, Flow and Crypto.org. Get all the functionality of a node, including JSON-RPC support, but with the supercharged reliability, data accuracy and scalability needed to run world-class applications on the blockchain.

Contracts

We are going to have a single Solidity smart contract that powers the whole application. Deployment and testing of this contract will be done from Hardhat.

Hardhat is a development environment for compiling, deploying, testing and debugging your Ethereum software. It helps developers manage and automate the recurring tasks that are inherent in the process of creating smart contracts and dApps, as well as easily introduce more functionality around this workflow. This means compiling, executing, and testing smart contracts at the core.

As stated above, the deployment of the contract was done on our local network using hardhat nodes representing the Ethereum network economy. The smart contract will serve 2 purposes:

1. It will represent the collection of NFTs from the music.
2. It will represent a marketplace where users can buy and resell them.

The idea of using NFTs is to keep track of the metadata of the NFT (which in this case is an audio file deployed on IPFS), a corresponding ID and the address of the NFT owner. These NFTs will have a special feature that will allow you to transfer your NFT to a different account and hit new NFTs. This master agreement will have a royalty attached to it so that every time a music NFT is sold, the music artist will receive a royalty. The contract will represent a collection of audio files as NFTs. The audio files are the songs of an album. We already know which songs make up that album, so we can monetize them all together. This contract will also work as a marketplace. After we hit them, we will list them for sale as marketplace items.

After writing the code that handles this initial contract setup (deployment), we'll create the functions that fans can interact with:

- The buy function;
- The resell function.

We want to code these functions so that the artist selling the album receives royalties for each sale.

The Solidity Smart Contract Script Overview

The Smart Contract use OpenZeppelin ERC721 and Ownable contract which are open-source standards develop by OpenZeppelin Company.

Our MusicNFTMarketplace take all the specifies of those in order to create non fungible tokens thanks to ERC721 Standard and grant exclusive access to specific functions thanks to Ownable Standard.

The memory of our contract stores IPFS Metadata, .json scripts, artists, royalty fee value, market items array, which is structured for each NFTs with tokenID, payable seller and a price. The storage also stores 2 events while someone buy an item and relisted an item, and a

constructor that initalize royalty fee, artist address and prices of music nfts when the contract is deployed.

Six functions are implemented in the contract:

1. `updateRoyaltyFee` - Updates the royalty fee of the contract
2. `buyToken` - Creates the sale of a music nft listed on the marketplace & Transfers ownership of the nft, as well as funds between parties
3. `resellToken` - Allows someone to resell their music nft
4. `getAllUnsoldTokens` - Fetches all the tokens currently listed for sale
5. `getMyTokens` - Fetches all the tokens owned by the user
6. `_baseURI` - Internal function that gets the baseURI initialized in the constructor

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract MusicNFTMarketplace is ERC721("DAppFi", "DAPP"), Ownable {
    string public baseURI =

"<https://bafybeid2qyl54rqkkcnd5dcqdifubttxv5iicnnz3dsdojpnucvik7hgc4.ipfs.
nftstorage.link/>";
    string public baseExtension = ".json";
    address public artist;
    uint256 public royaltyFee;

    struct MarketItem {
        uint256 tokenId;
        address payable seller;
        uint256 price;
    }
    MarketItem[] public marketItems;

    event MarketItemBought(
        uint256 indexed tokenId,
        address indexed seller,
        address buyer,
        uint256 price
    );
    event MarketItemRelisted(
        uint256 indexed tokenId,
        address indexed seller,
        uint256 price
    );

    /* In constructor we initalize royalty fee, artist address and prices
of music nfts*/
    constructor(
        uint256 _royaltyFee,
        address _artist,
        uint256[] memory _prices
    ) payable {
        require(
            _prices.length * _royaltyFee <= msg.value,
```

```
            "Deployer must pay royalty fee for each token listed on the
marketplace"
        );
        royaltyFee = _royaltyFee;
        artist = _artist;
        for (uint8 i = 0; i < _prices.length; i++) {
            require(_prices[i] > 0, "Price must be greater than 0");
            _mint(address(this), i);
            marketItems.push(MarketItem(i, payable(msg.sender),
_prices[i]));
        }
    }

    /* Updates the royalty fee of the contract */
    function updateRoyaltyFee(uint256 _royaltyFee) external onlyOwner {
        royaltyFee = _royaltyFee;
    }

    /* Creates the sale of a music nft listed on the marketplace */
    /* Transfers ownership of the nft, as well as funds between parties */
    function buyToken(uint256 _tokenId) external payable {
        uint256 price = marketItems[_tokenId].price;
        address seller = marketItems[_tokenId].seller;
        require(
            msg.value == price,
            "Please send the asking price in order to complete the
purchase"
        );
        marketItems[_tokenId].seller = payable(address(0));
        _transfer(address(this), msg.sender, _tokenId);
        payable(artist).transfer(royaltyFee);
        payable(seller).transfer(msg.value);
        emit MarketItemBought(_tokenId, seller, msg.sender, price);
    }

    /* Allows someone to resell their music nft */
    function resellToken(uint256 _tokenId, uint256 _price) external payable
{
        require(msg.value == royaltyFee, "Must pay royalty");
        require(_price > 0, "Price must be greater than zero");
        marketItems[_tokenId].price = _price;
        marketItems[_tokenId].seller = payable(msg.sender);

        _transfer(msg.sender, address(this), _tokenId);
        emit MarketItemRelisted(_tokenId, msg.sender, _price);
    }

    /* Fetches all the tokens currently listed for sale */
    function getAllUnsoldTokens() external view returns (MarketItem[]
memory) {
        uint256 unsoldCount = balanceOf(address(this));
        MarketItem[] memory tokens = new MarketItem[](unsoldCount);
        uint256 currentIndex;
        for (uint256 i = 0; i < marketItems.length; i++) {
            if (marketItems[i].seller != address(0)) {
                tokens[currentIndex] = marketItems[i];
                currentIndex++;
            }
        }
        return (tokens);
    }
```

```
    /* Fetches all the tokens owned by the user */
    function getMyTokens() external view returns (MarketItem[] memory) {
        uint256 myTokenCount = balanceOf(msg.sender);
        MarketItem[] memory tokens = new MarketItem[](myTokenCount);
        uint256 currentIndex;
        for (uint256 i = 0; i < marketItems.length; i++) {
            if (ownerOf(i) == msg.sender) {
                tokens[currentIndex] = marketItems[i];
                currentIndex++;
            }
        }
        return (tokens);
    }

    /* Internal function that gets the baseURI initialized in the
constructor */
    function _baseURI() internal view virtual override returns (string
memory) {
        return baseURI;
    }
}
```

Demo - Deploy, Test & UX

1. Install NodeJS;
2. Install Hardhat;
3. Connect Hardhat Development Blockchain to your Metamask;
4. Install dependencies inside the project folder - `npm install`;
5. Start Local Development Blockchain with hardhat - `npx node`;
6. Deploy Smart Contract - `npm run deploy`;
7. Test Smart Contract - `npx hardhat test`;
8. Launch Web3.0 Page - `npm run start`.

The Deploy Script Overview

The script allows the deployment of the contract with specify royalty fee amount (here 0.1ETH), specify listed prices for each NFTs of the collection (here 1ETH) and the amount of the deployment fees. When the deployer signed with its Metamask, the contract is deployed and the console print the deployer account, the account balance and the NFTs Collection Smart Contract address. Finally, for each contract deployed, a function passes the contract and name to save a copy of the contract ABI and address to the front end. Note that the ABI stands for Application Binary Interface. In the context of computer science, it is an interface between two program modules.

```
async function main() {
  const toWei = (num) => ethers.utils.parseEther(num.toString())
  let royaltyFee = toWei(0.01);
  let prices = [toWei(1), toWei(1), toWei(1), toWei(1), toWei(1), toWei(1),
toWei(1), toWei(1), toWei(1)]
  let deploymentFees = toWei(prices.length * 0.01)
  const [deployer, artist] = await ethers.getSigners();

  console.log("Deploying contracts with the account:", deployer.address);
  console.log("Account balance:", (await
deployer.getBalance()).toString());

  // deploy contracts here:
  const NFTMarketplaceFactory = await
ethers.getContractFactory("MusicNFTMarketplace");
  nftMarketplace = await NFTMarketplaceFactory.deploy(
    royaltyFee,
    artist.address,
    prices,
    { value: deploymentFees }
  );

  console.log("Smart contract address:", nftMarketplace.address)

  // For each contract, pass the deployed contract and name to this
function to save a copy of the contract ABI and address to the front end.
  saveFrontendFiles(nftMarketplace, "MusicNFTMarketplace");
}

function saveFrontendFiles(contract, name) {
  const fs = require("fs");
  const contractsDir = __dirname + "/../../frontend/contractsData";

  if (!fs.existsSync(contractsDir)) {
    fs.mkdirSync(contractsDir);
  }

  fs.writeFileSync(
    contractsDir + `/${name}-address.json`,
    JSON.stringify({ address: contract.address }, undefined, 2)
  );

  const contractArtifact = artifacts.readArtifactSync(name);

  fs.writeFileSync(
    contractsDir + `/${name}.json`,
    JSON.stringify(contractArtifact, null, 2)
  );
}

main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

The Test Script Overview

```
const { expect } = require("chai");

const toWei = (num) => ethers.utils.parseEther(num.toString())
const fromWei = (num) => ethers.utils.formatEther(num)

describe("MusicNFTMarketplace", function () {

  let nftMarketplace
  let deployer, artist, user1, user2, users;
  let royaltyFee = toWei(0.01); // 1 ether = 10^18 wei
  let URI =
"<https://bafybeid2qyl54rqkkcnd5dcqdifubttxv5iicnnz3dsdojpnucvik7hgc4.ipfs.
nftstorage.link/>"
  let prices = [toWei(1), toWei(2), toWei(3), toWei(4), toWei(5), toWei(6),
toWei(7), toWei(8), toWei(9)]
  let deploymentFees = toWei(prices.length * 0.01)
  beforeEach(async function () {
    // Get the ContractFactory and Signers here.
    const NFTMarketplaceFactory = await
ethers.getContractFactory("MusicNFTMarketplace");
    [deployer, artist, user1, user2, ...users] = await ethers.getSigners();

    // Deploy music nft marketplace contract
    nftMarketplace = await NFTMarketplaceFactory.deploy(
      royaltyFee,
      artist.address,
      prices,
      { value: deploymentFees }
    );

  });

  describe("Deployment", function () {

    it("Should track name, symbol, URI, royalty fee and artist", async
function () {
      const nftName = "DAppFi"
      const nftSymbol = "DAPP"
      expect(await nftMarketplace.name()).to.equal(nftName);
      expect(await nftMarketplace.symbol()).to.equal(nftSymbol);
      expect(await nftMarketplace.baseURI()).to.equal(URI);
      expect(await nftMarketplace.royaltyFee()).to.equal(royaltyFee);
      expect(await nftMarketplace.artist()).to.equal(artist.address);
    });

    it("Should mint then list all the music nfts", async function () {
      expect(await
nftMarketplace.balanceOf(nftMarketplace.address)).to.equal(9);
      // Get each item from the marketItems array then check fields to
ensure they are correct
      await Promise.all(prices.map(async (i, indx) => {
        const item = await nftMarketplace.marketItems(indx)
        expect(item.tokenId).to.equal(indx)
        expect(item.seller).to.equal(deployer.address)
        expect(item.price).to.equal(i)
      }))
    });
    it("Ether balance should equal deployment fees", async function () {
```

```
      expect(await
ethers.provider.getBalance(nftMarketplace.address)).to.equal(deploymentFees
)
    });

  });
  describe("Updating royalty fee", function () {

    it("Only deployer should be able to update royalty fee", async function
() {
      const fee = toWei(0.02)
      await nftMarketplace.updateRoyaltyFee(fee)
      await expect(
        nftMarketplace.connect(user1).updateRoyaltyFee(fee)
      ).to.be.revertedWith("Ownable: caller is not the owner");
      expect(await nftMarketplace.royaltyFee()).to.equal(fee)
    });

  });
  describe("Buying tokens", function () {
    it("Should update seller to zero address, transfer NFT, pay seller, pay
royalty to artist and emit a MarketItemBought event", async function () {
      const deployerInitalEthBal = await deployer.getBalance()
      const artistInitialEthBal = await artist.getBalance()
      // user1 purchases item.
      await expect(nftMarketplace.connect(user1).buyToken(0, { value:
prices[0] }))
        .to.emit(nftMarketplace, "MarketItemBought")
        .withArgs(
          0,
          deployer.address,
          user1.address,
          prices[0]
        )
      const deployerFinalEthBal = await deployer.getBalance()
      const artistFinalEthBal = await artist.getBalance()
      // Item seller should be zero addr
      expect((await
nftMarketplace.marketItems(0)).seller).to.equal("0x0000000000000000000000000
0000000000000000")
      // Seller should receive payment for the price of the NFT sold.
      expect(+fromWei(deployerFinalEthBal)).to.equal(+fromWei(prices[0]) +
+fromWei(deployerInitalEthBal))
      // Artist should receive royalty
      expect(+fromWei(artistFinalEthBal)).to.equal(+fromWei(royaltyFee) +
+fromWei(artistInitialEthBal))
      // The buyer should now own the nft
      expect(await nftMarketplace.ownerOf(0)).to.equal(user1.address);
    })
    it("Should fail when ether amount sent with transaction does not equal
asking price", async function () {
      // Fails when ether sent does not equal asking price
      await expect(
        nftMarketplace.connect(user1).buyToken(0, { value: prices[1] })
      ).to.be.revertedWith("Please send the asking price in order to
complete the purchase");
    });
  })
  describe("Reselling tokens", function () {
    beforeEach(async function () {
      // user1 purchases an item.
```

```
      await nftMarketplace.connect(user1).buyToken(0, { value: prices[0] })
    })

    it("Should track resale item, incr. ether bal by royalty fee, transfer
NFT to marketplace and emit MarketItemRelisted event", async function () {
      const resaleprice = toWei(2)
      const initMarketBal = await
ethers.provider.getBalance(nftMarketplace.address)
      // user1 lists the nft for a price of 2 hoping to flip it and double
their money
      await expect(nftMarketplace.connect(user1).resellToken(0,
resaleprice, { value: royaltyFee }))
        .to.emit(nftMarketplace, "MarketItemRelisted")
        .withArgs(
          0,
          user1.address,
          resaleprice
        )
      const finalMarketBal = await
ethers.provider.getBalance(nftMarketplace.address)
      // Expect final market bal to equal inital + royalty fee
      expect(+fromWei(finalMarketBal)).to.equal(+fromWei(royaltyFee) +
+fromWei(initMarketBal))
      // Owner of NFT should now be the marketplace
      expect(await
nftMarketplace.ownerOf(0)).to.equal(nftMarketplace.address);
      // Get item from items mapping then check fields to ensure they are
correct
      const item = await nftMarketplace.marketItems(0)
      expect(item.tokenId).to.equal(0)
      expect(item.seller).to.equal(user1.address)
      expect(item.price).to.equal(resaleprice)
    });

    it("Should fail if price is set to zero and royalty fee is not paid",
async function () {
      await expect(
        nftMarketplace.connect(user1).resellToken(0, 0, { value: royaltyFee
})
      ).to.be.revertedWith("Price must be greater than zero");
      await expect(
        nftMarketplace.connect(user1).resellToken(0, toWei(1), { value: 0
})
      ).to.be.revertedWith("Must pay royalty");
    });
  });
  describe("Getter functions", function () {
    let soldItems = [0, 1, 4]
    let ownedByUser1 = [0, 1]
    let ownedByUser2 = [4]
    beforeEach(async function () {
      // user1 purchases item 0.
      await (await nftMarketplace.connect(user1).buyToken(0, { value:
prices[0] })).wait();
      // user1 purchases item 1.
      await (await nftMarketplace.connect(user1).buyToken(1, { value:
prices[1] })).wait();
      // user2 purchases item 4.
      await (await nftMarketplace.connect(user2).buyToken(4, { value:
prices[4] })).wait();
    })
```

```
    it("getAllUnsoldTokens should fetch all the marketplace items up for
sale", async function () {
      const unsoldItems = await nftMarketplace.getAllUnsoldTokens()
      // Check to make sure that all the returned unsoldItems have filtered
out the sold items.
      expect(unsoldItems.every(i => !soldItems.some(j => j ===
i.tokenId.toNumber()))).to.equal(true)
      // Check that the length is correct
      expect(unsoldItems.length === prices.length -
soldItems.length).to.equal(true)
    });
    it("getMyTokens should fetch all tokens the user owns", async function
() {
      // Get items owned by user1
      let myItems = await nftMarketplace.connect(user1).getMyTokens()
      // Check that the returned my items array is correct
      expect(myItems.every(i => ownedByUser1.some(j => j ===
i.tokenId.toNumber()))).to.equal(true)
      expect(ownedByUser1.length === myItems.length).to.equal(true)
      // Get items owned by user2
      myItems = await nftMarketplace.connect(user2).getMyTokens()
      // Check that the returned my items array is correct
      expect(myItems.every(i => ownedByUser2.some(j => j ===
i.tokenId.toNumber()))).to.equal(true)
      expect(ownedByUser2.length === myItems.length).to.equal(true)
    });
  });
})
```

Our Test Script run a suit of define action Solidity Smart Contract and output the success of each step.

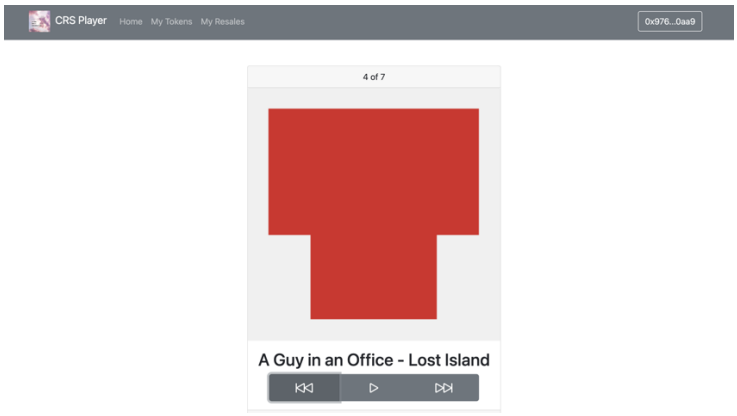
First, the script tests the deployment of our contract with different information such as the royalty fee, the URI link, the price for each item, the deployment fee and so on.

Then the script test 10 phases and test if the steps that are passing or not.

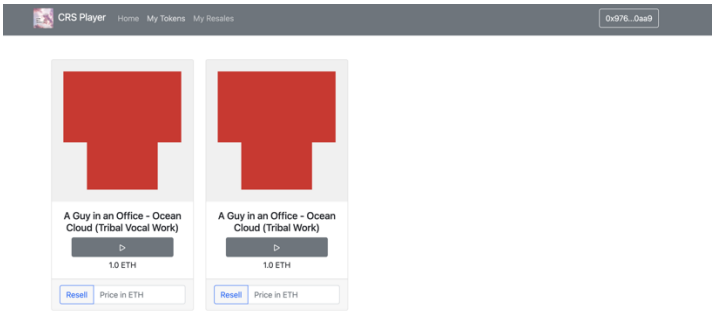
```
(base) kevinifrah@Kevins-MacBook-Pro music_nfts-main 2 % npx hardhat test


  MusicNFTMarketplace
    Deployment
      ✓ Should track name, symbol, URI, royalty fee and artist
      ✓ Should mint then list all the music nfts (40ms)
      ✓ Ether balance should equal deployment fees
    Updating royalty fee
      ✓ Only deployer should be able to update royalty fee (61ms)
    Buying tokens
      ✓ Should update seller to zero address, transfer NFT, pay seller, pay royalty to artist and emit a MarketItemBought event (45ms)
      ✓ Should fail when ether amount sent with transaction does not equal asking price
    Reselling tokens
      ✓ Should track resale item, incr. ether bal by royalty fee, transfer NFT to marketplace and emit MarketItemRelisted event
      ✓ Should fail if price is set to zero and royalty fee is not paid (40ms)
    Getter functions
      ✓ getAllUnsoldTokens should fetch all the marketplace items up for sale
      ✓ getMyTokens should fetch all tokens the user owns


  10 passing (2s)
```

Frontend Overview

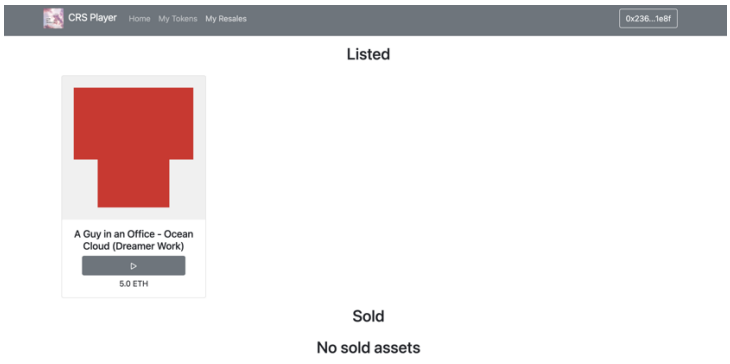
Home Page - Marketplace/Streaming Platform



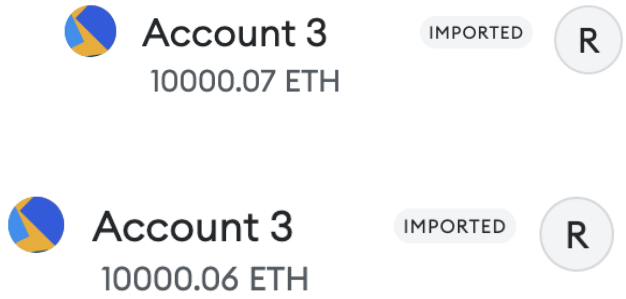
My Tokens Page - All Item that an account has bought



My Resales - All Resales & Sold NFTs



Royalty Fees after each resell to the original artist



Recommendation, Thought & Future Work

The application has a smart contract with limitations and a rather trivial user interface. However, it is a good starting point to make this application evolve into a useful tool set for the user, whether he is an artist or a client.

Here is the development of several Smart Contracts that could be implemented around this application:

1. A Smart Contract that allows to stream all the musics deployed on the platform if the listener has a Subscription Token. In the same way as a regular streaming platform, the user will have access to a tool that will allow him to access the content of the platform and create his own playlists. A Subscriber will be able to obtain reduction on all the content of the Marketplace.
2. A Smart Contract of honest redistribution of the revenue from the real clicks recorded by the script on the streaming platform.
3. A Smart Contract that will act as a general DAO for the whole platform. It will allow to create an honest vote for the favorite artists.
4. A Smart Contract that acts as a DAO for each artist, collective or label. A user who is part of a DAO will be able to participate in its common decisions. His participation in the community will allow him to benefit from NFT, access to real or live events and workshops around the music of the DAO in question.
5. A Smart Contract acting as an event ticket. Each ticket will be an NFT that will serve as an entry ticket to an artist's events.

The evolution of the Frontend will also have to evolve to make it easier to use:

1. Separate the marketplace from the streaming platform
2. Gather all the collections using our contracts on the platform.
3. To make the web page more autonomous for the artist so that the site can create the process of mining and storing data independently.
4. To make the page more user friendly and gather all the tools stated above to act as a musical operating system. This tool could be a sharing ground between music lovers and artists.

5. Extend the music marketplace to other forms of music files (samples, stems, etc.).

Conclusion

The realization of this project was such interesting and stimulating. However, I've faced some challenges to develop it:

First, I've learned programming from scratch to realize it. I had to learn at different points Solidity, Hardhat, JavaScript, React and so on to see the contract deployed and the final interface to born.

Second, I've faced some deployment and Metamask Interaction problematic that stick me for days in my project. This problem was due to the use of wrong nodes on the Hardhat Local Development Blockchain Environment. It wasn't obvious to figure it out.

Third, think to a relevant way to store data, ensure decentralization, ensure their availability, their security and their authenticity was challenging. However, it leads me to learn a lot concerning Blockchain Data Storage technology.