MeloManiac Music Market – Sharing, Listening and more

Siddharth Sankaran
50421657
sankara2@buffalo.edu

Shriram Ravi
50419944
shriramr@buffalo.edu

1. Introduction

Melomaniacs is a blockchain powered independent music buying and selling marketplace. It is a web application with its own personalized Ethereum based token (MELO) that helps independent artists to share their work in a peer-to-peer marketplace. Artists can upload their original or cover songs with relevant tags to the platform. Recommendations are sent to listeners with matching tastes. Artists can set the own prices, free sample duration and even advertise their content on the platform. By eliminating the need for third parties and labels Melomaniacs ensures that artists receive 100% of the streaming revenue. The company's blockchain foundation allows for maximum security in the peer-to-peer transfer of music. Listeners are also rewarded for creating trending playlists (colabs) that attract new users and increase the popularity of the application.

Since it is expensive and impractical to store songs on the blockchain this system has been built to make use of the Inter-Planetary File System (IPFS) to store and retrieve music files. The IPFS is a truly decentralized storage that is maintained through a peer-to-peer network of partner computer which each store small chunks of the data. The system is reliable and fault tolerant. (For this project we are using the free version which has limitations on size and speed)

2. System Design

The project is implemented with four components

2.1 Solidity Smart Contract

The core of any blockchain Dapp project is the smart contract. For this system, the smart contract was written in solidity language. The

2.2 JavaScript Interface

The user interface is built using Javascript and in specific, NodeJS in combination with HTML and CSS. The packages and dependencies needed for the successful execution of this system is listed in the package.json file. The script communicates with the smart contract by using the smart contract address and ABI (Application Binary Interface), both of which are dynamically generated using truffle.

2.3 Ganache Truffle suite

Ganache is used to deploy a personal Ethereum blockchain on localhost run tests, execute commands, and inspect state while controlling how the chain operates. Truffle is used to generate the ABI from the smart contract and for deploying it from a unique address. This address is obtained from one of the accounts generated from Ganache. The *truffle* and *truffle-contract* libraries of javascript are deployed and used for this purpose

2.4 Music Storage – Inter-Planetary File System (IPFS)

The storage mechanism used for this system is the truly decentralized Inter-Planetary File System (IPFS). It approximately costs 20,000 gas to transacting 256 bit (8 bytes) of information to the blockchain. In simpler words, 8 bytes is generally one word or string. The price of gas is ~43 gwei/gas as per Etherscan which mean that transacting over the blockchain is not very cheap.

The obvious solution is to store the data on a data storage system like AWS, Azure and only pass the transaction info to the blockchain. But this again makes the system dependent on a centralized system. To avoid this, our project makes use of IPFS. When storing files on the IPFS, they are hidden behind a cryptic hash and stored across multiple servers across the network. When looking up files, we are asking the network to obtain the file by providing the unique file hash. In this way, we can store large amounts of data on the IPFS and store the immutable, permanent IPFS hash into a blockchain transaction. This secures the content without having to put data into the blockchain itself saving money.

For this project, we have used the ipfs.infura.io node to connect to the IPFS. Infura IPFS module is a fast and reliable way to connect and interact with IPFS. It is employed free storage, high connectivity and quick response times making it apt for this project. Since it is the free version, there may be limits on upload size of the file.

The ipfs-api module in javascript was used to implement the connection and interaction with IPFS through Infura. The module has an add() method which can add any file stored as a buffer to the IPFS. The add operation returns the unique IPFS has which can be used to retrieve the file again. The file can be obtained by visiting “https://gateway.ipfs.io/ipfs/”+ “IPFS hash#”

The transaction is added to the blockchain with just the important and immutable information like the unique IPFS hash, userID, songID, etc.

Once the user purchases a song, the song’s unique hash is added under the user’s account and listed under owned songs.

IPFS’s unique hash is also used to avoid piracy and duplicacy. If a file’s hash is already present in the blockchain, the user is not allowed to upload the song again. This would be a great way to stop reposting, song theft and piracy.

3. Design Diagrams

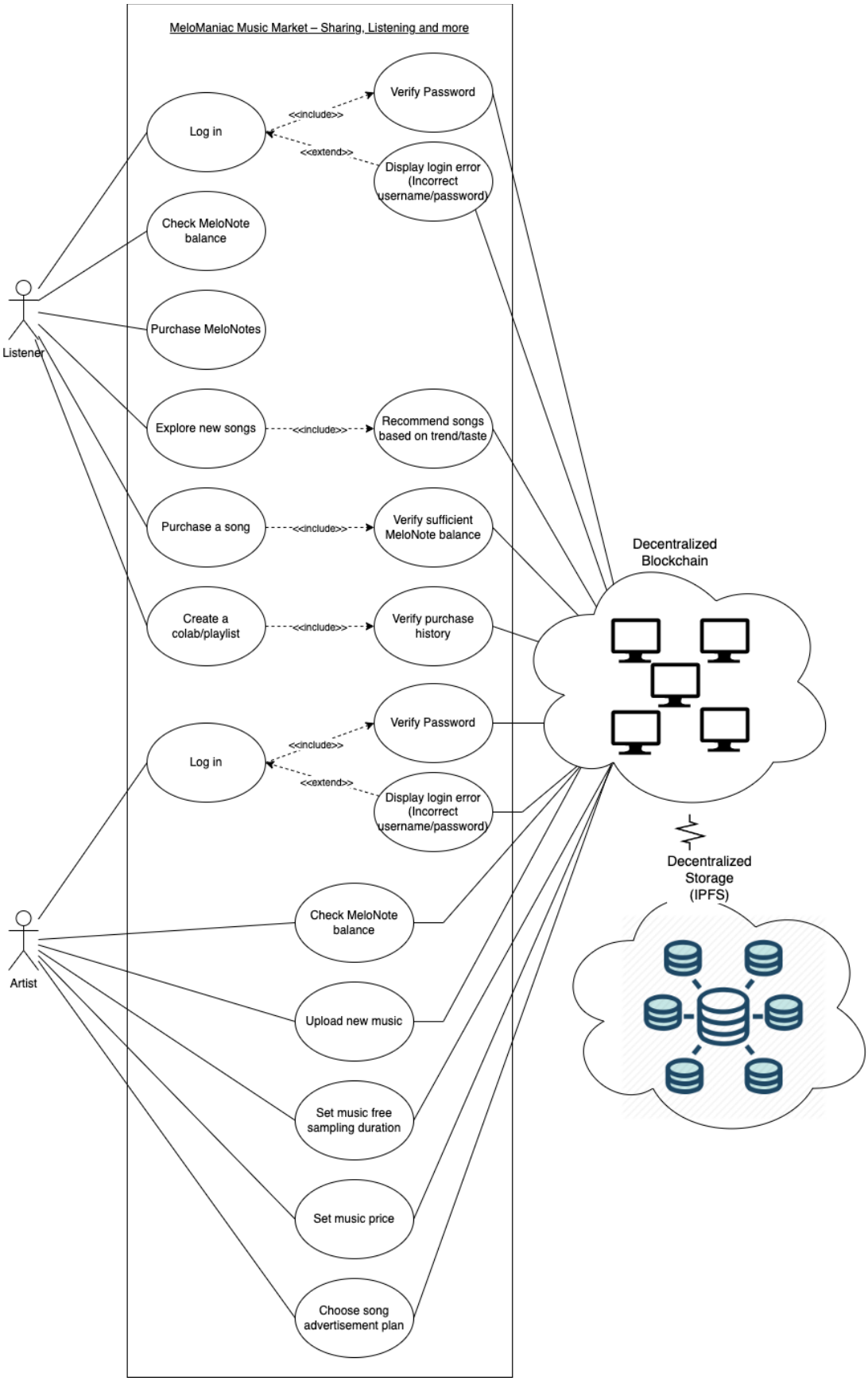
3.1 Quad Chart

The quad chart highlights the problem that is being addressed in this system. It underlines the cons of the existing systems and explains how the proposed blockchain-based solution overcomes the shortcomings. The chart also highlights the benefits of the system to all its users.

Usecase: MeloManiac Music Market Problem Statement: To build a truly decentralized blockchain-based marketplace for buying and selling independent music	**Issues with existing system:** • Few big players control access for artists getting record contracts and deals • Opaque and heavily delayed royalty payments cause artists to earn meagre amounts from album sales • Songs heavily promoted by big players dominate most of the market. Difficult for independent music artists to make the music reach listeners • Artist have to go through multiple authorities to reach listeners • Song piracy reduces the earning of music artists
Proposed blockchain-based solution: • Artists are individually responsible for getting deals or sales of their music • Artists get 99% of the earnings made from sales (minor transaction costs) • Recommendations based on what the user searches for and what the community likes. Not skewed by promotional activities of central authorities • Artists can directly interact with listeners with no middlemen • All songs are hashed and stored reducing piracy and re-uploading of songs drastically	**Benefits:** • Better user experience. Users listen to songs that they like and are not skewed by promotions • Better experience for artists as well. They benefit the most from the music sales • Saves time, effort, cost for artists by providing a platform to directly transact with customers instead of going through middlemen • Reduction in piracy and song downloading • Better music - The best music circulated by the community rises to the top

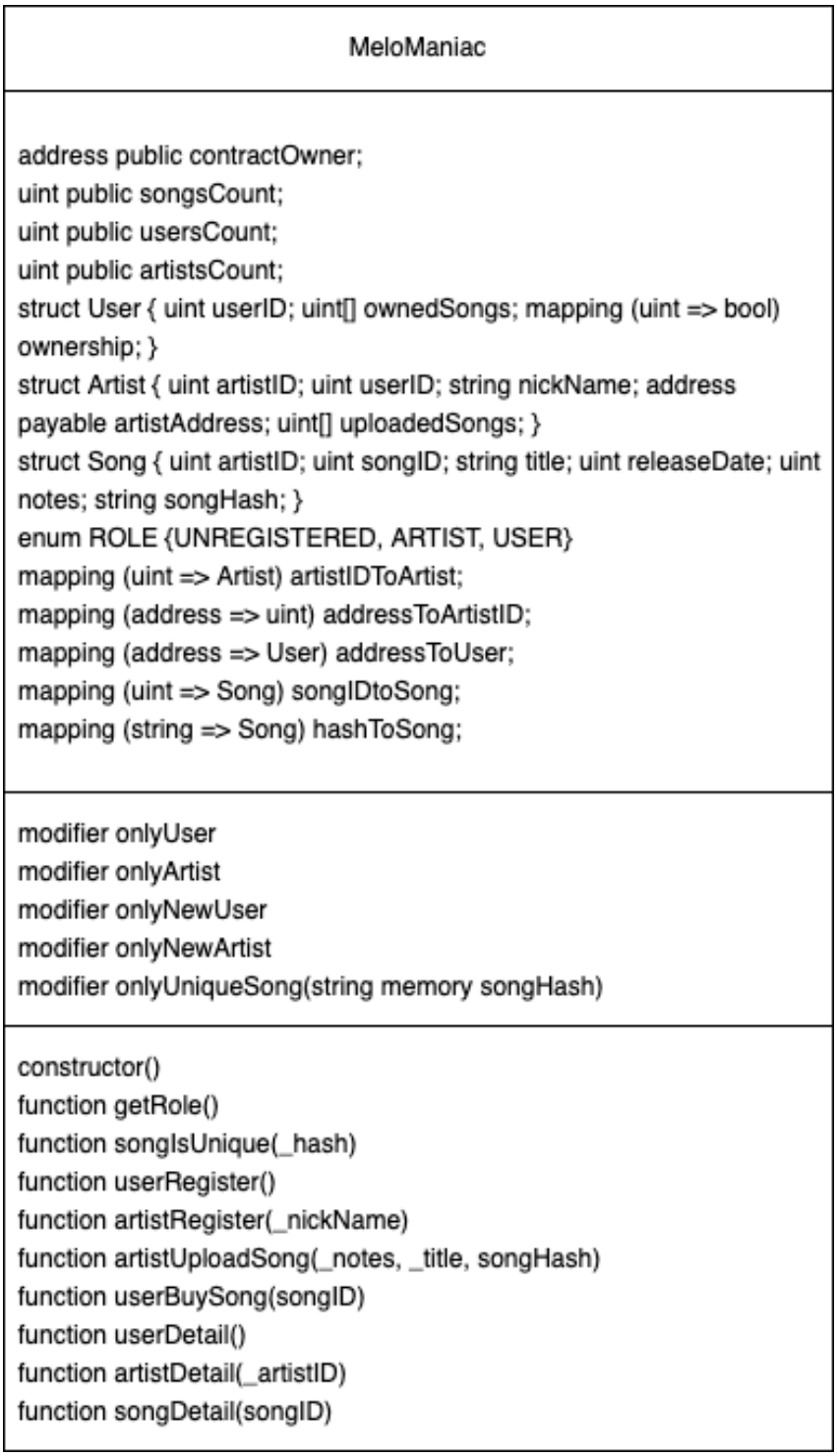
3.2 Use-case Diagram

The use-case diagram has been slightly modified from the previous phase. The right-hand side is now shown to depict the decentralized architecture. The users interact with the decentralized blockchain verification, validation and payment. The decentralized storage of IPFS is also depicted to indicate that the bulk storage of files is done through a decentralized system.


MeloManiac Music Market – Sharing, Listening and more
Log in
Verify Password
<<include>>
<<extend>>
Display login error (Incorrect username/password)
Check MeloNote balance
Purchase MeloNotes
Listener
Explore new songs
<<include>>
Recommend songs based on trend/taste
Purchase a song
<<include>>
Verify sufficient MeloNote balance
Create a colab/playlist
<<include>>
Verify purchase history
Log in
Verify Password
<<include>>
<<extend>>
Display login error (Incorrect username/password)
Artist
Check MeloNote balance
Upload new music
Set music free sampling duration
Set music price
Choose song advertisement plan
Decentralized Blockchain
Decentralized Storage (IPFS)

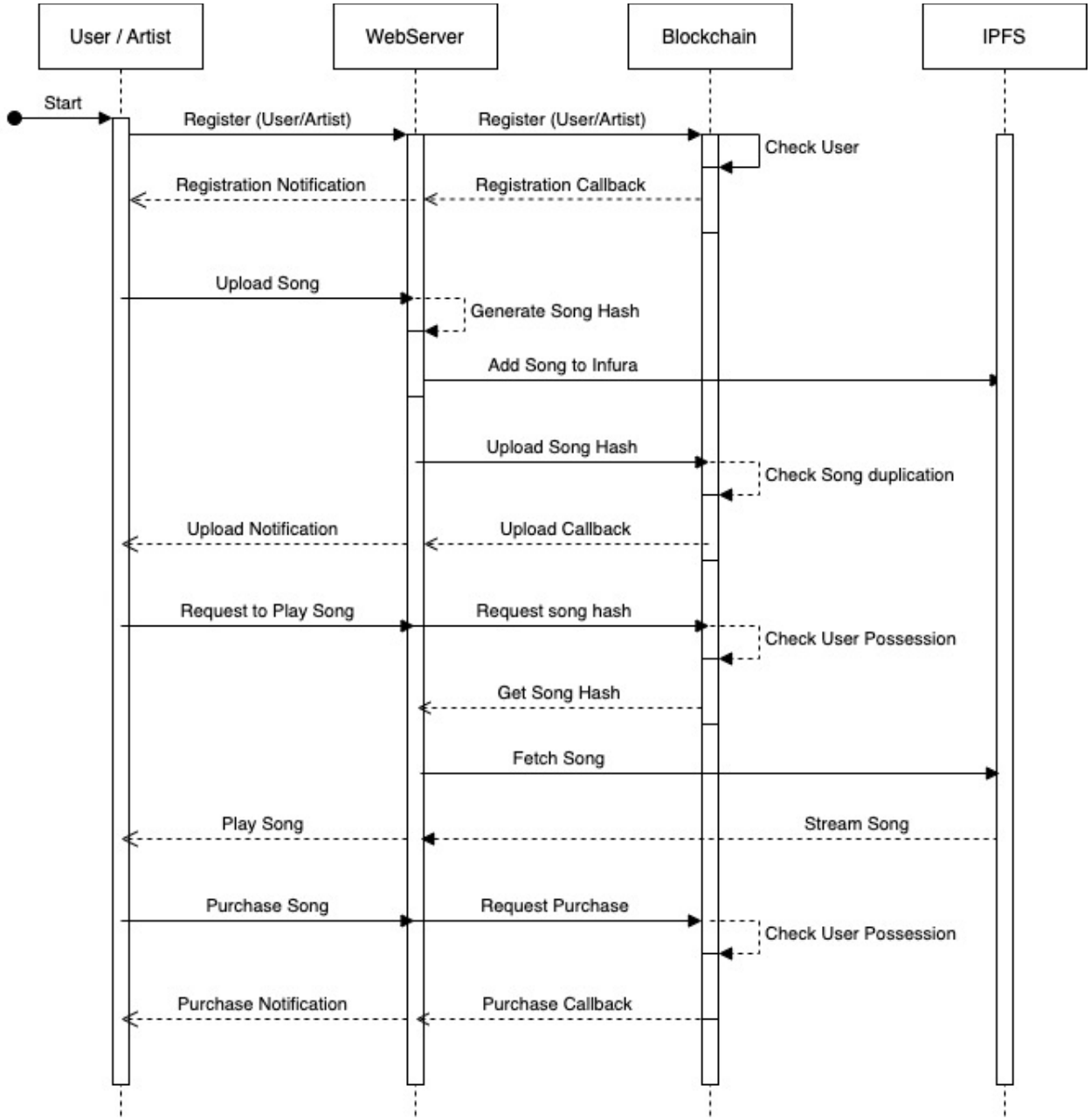
3.3 Contract Diagram

The contract diagram gives an overview of the smart contract detailing every variable, function and modifier that is being used. An intuitive naming of variables and methods is followed to allow the user to easily understand the functionality of each component.



3.4 Sequence Diagram

The sequence diagram shows in detail the order of operation followed in this system.



4. Smart Contract Code

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.9.0;

// Deployed at <Enter address starting with 0x> on Ganache
```

```
contract MeloManiac {
    address public contractOwner;
    uint public songsCount;
    uint public usersCount;
    uint public artistsCount;

    struct User {
        uint userID;
        uint[] ownedSongs;
        mapping (uint => bool) ownership;
    }

    struct Artist {
        uint artistID;
        uint userID;
        string nickName;
        address payable artistAddress;
        uint[] uploadedSongs;
    }

    struct Song {
        uint artistID;
        uint songID;
        string title;
        uint releaseDate;
        uint notes;
        string songHash;
    }

    enum ROLE {UNREGISTERED, ARTIST, USER}

    mapping (uint => Artist) artistIDToArtist;
    mapping (address => uint) addressToArtistID;
    mapping (address => User) addressToUser;
    mapping (uint => Song) songIDtoSong;
    mapping (string => Song) hashToSong;

    modifier onlyUser {
        require(addressToUser[msg.sender].userID != 0, "Not a user");
        _;
    }

    modifier onlyArtist {
        require(addressToArtistID[msg.sender] != 0, "Not an artist");
        _;
    }
```

```
    modifier onlyNewUser {
        require(addressToUser[msg.sender].userID == 0, "User Already registered!");
        _;
    }

    modifier onlyNewArtist {
        require(addressToArtistID[msg.sender] == 0, "Artist Already registered!");
        _;
    }

    modifier onlyUniqueSong(string memory songHash) {
        require(hashToSong[songHash].songID == 0, "Can't upload duplicate");
        _;
    }

    constructor() public {
        contractOwner = msg.sender;
        songsCount = 0;
        usersCount = 0;
        artistsCount = 0;
    }

    function getRole() external view returns(ROLE) {
        return ((addressToArtistID[msg.sender] != 0) ? ROLE.ARTIST:
(addressToUser[msg.sender].userID != 0) ? ROLE.USER: ROLE.UNREGISTERED);
    }

    function songIsUnique(string calldata _hash) external view returns(uint) {
        return hashToSong[_hash].songID;
    }

    function userRegister() public onlyNewUser {
        usersCount += 1;

        User memory newUser = User(usersCount, new uint[](0));
        addressToUser[msg.sender] = newUser;
    }

    function artistRegister(string calldata _nickName) external onlyNewArtist payable
{
        require(msg.value == 0.05 ether,"Artist registration fee");
        artistsCount += 1;

        if (addressToUser[msg.sender].userID == 0) {
            userRegister();
        }
```

```
        Artist memory newArtist =
Artist(artistsCount,addressToUser[msg.sender].userID, _nickName, msg.sender, new
uint[](0));

        addressToArtistID[msg.sender] = artistsCount;
        artistIDToArtist[artistsCount] = newArtist;
    }

    function artistUploadSong(uint _notes, string calldata _title, string calldata
songHash) external onlyArtist onlyUniqueSong(songHash) {
        songsCount += 1;

        Artist storage artistInstance =
artistIDToArtist[addressToArtistID[msg.sender]];
        artistInstance.uploadedSongs.push(songsCount);

        songIDtoSong[songsCount] = Song(artistInstance.artistID, songsCount, _title,
now, _notes, songHash);
        hashToSong[songHash] = songIDtoSong[songsCount];
    }

    function userBuySong(uint songID) external onlyUser payable {
        Song storage song = songIDtoSong[songID];
        require(song.songID != 0, "Song does not exist!");
        require(msg.value == song.notes, "Check if song notes is paid");

        User storage user = addressToUser[msg.sender];
        require(!user.ownership[songID], "Can't buy twice");

        user.ownedSongs.push(songID);
        user.ownership[songID] = true;

        artistIDToArtist[song.artistID].artistAddress.transfer(msg.value);
    }

    function userDetail() external view returns(uint, uint, uint[] memory) {
        return (addressToUser[msg.sender].userID, addressToArtistID[msg.sender],
addressToUser[msg.sender].ownedSongs);
    }

    function artistDetail(uint _artistID) external view returns(string memory, uint[]
memory) {
        return (artistIDToArtist[_artistID].nickName,
artistIDToArtist[_artistID].uploadedSongs);
    }

    function songDetail(uint songID) external view returns(uint artistID, uint id,
string memory title, uint notes, uint releaseDate, string memory songHash) {
```

```
        Song memory song = songIDtoSong[songID];
        id = song.songID;
        artistID = song.artistID;
        title = song.title;
        notes = song.notes;
        releaseDate = song.releaseDate;
        songHash = song.songHash;
    }

    // function donate(uint artistID) public payable {
    //     artistIDToArtist[artistID].artistAddress.transfer(msg.value);
    // }
}
```

5. Smart Contract Description

The variables used in the contract are:

- **address public contractOwner** – General contract owner
- **uint public songsCount** – Total number of songs in the system
- **uint public usersCount** – Total number of users in the system
- **uint public artistsCount** – Total number of artists in the system
- **struct User { uint userID; uint[] ownedSongs; mapping (uint => bool) ownership; }** – Information about the listener
- **struct Artist { uint artistID; uint userID; string nickName; address payable artistAddress; uint[] uploadedSongs; }** - Information about the song uploader
- **struct Song { uint artistID; uint songID; string title; uint releaseDate; uint notes; string songHash; }** - Information about the song
- **enum ROLE {UNREGISTERED, ARTIST, USER}** – Current state of the user
- **mapping (uint => Artist) artistIDToArtist** – Mapping each ID to the Artist struct
- **mapping (address => uint) addressToArtistID**– Mapping each address to the Artist ID
- **mapping (address => User) addressToUser**– Mapping each address to the User struct
- **mapping (uint => Song) songIDtoSong**– Mapping each ID to the Song struct
- **mapping (string => Song) hashToSong**– Mapping each unique IPFS hash to the Song struct

These are the modifiers that were used:

- **modifier onlyUser**– To check if the current id is a user
- **modifier onlyArtist**– To check if the current id is a Artist
- **modifier onlyNewUser**– To check if the current id is a New User
- **modifier onlyNewArtist**– To check if the current id is a New Artist
- **modifier onlyUniqueSong(string memory songHash)** – To check if the song already exists in the system

These are the functions that were used:

- **constructor()** – Initialize variables in the contract
- **function getRole()**– Get role of current user
- **function userRegister()**– Register current user as a USER (listener) in the system
- **function artistRegister(_nickName)** – Register current user as a ARTIST (song uploader) in the system
- **function artistUploadSong(_notes, _title, songHash)** – Upload song to IPFS and generate hash. Send hash to the smart contract to record on blockchain
- **function userBuySong(songID)** – User to purchase song and add song hash to list of owned songs
- **function userDetail()**– Get user's details
- **function artistDetail(_artistID)** – Get artist's details
- **function songDetail(songID)** – Get song's details

6. Important functions

6.1 registerUser()

This function is used in app.js to trigger the user registration process. This calls the userRegister() method from the smart contract. This takes in the user's address and assigns a user id. User id starts at 1 and is incremented once for each new user. Being a user ensure that the person can interact with the system. A user is allowed to listen to songs and buy songs. A user can only listen to a song for 10 seconds if it is not purchased. Once user purchases it, he can listen to the entire song. The list of songs owned by the user is shown in the user tab in the interface. Once a user purchases a song, the song's hash is stored under the user's name. Whenever the user wishes to listen to the song, the song is fetched from IPFS using the unique hash.

6.2 registerArtist(nickName)

This function is used in app.js to trigger the artist registration process. This calls the artistRegister() method from the smart contract. This takes in the address and assigns a artist id. Artist id starts at 1 and is incremented once for each new artist. A user can also register as an artist if he wishes to upload songs as well. Each registration process has a small fee involved. An artist is allowed to upload new songs to the system. This is pushed to IPFS and the unique IPFS hash is pushed to the blockchain and stored as a transaction. The artist is allowed to set the song's price for other's listeners. Each and every buy of the song, adds money to the artist's wallet. All the songs uploaded by the artist can be viewed under the artist tab in the UI. To ensure DRM compliance multiple uploads of the same song is checked and prohibited.

6.3 purchaseSong (songID)

A user can listen to all the songs available in the network for free, but with a limit of 10 seconds. If the user likes the song, he will need to purchase the full version to listen to it completely. For this the user needs to click on the buy button next to each song. This triggers the purchaseSong function in JS which in turn calls the userBuySong(songId) on the smart contract. The user address and song id and the necessary amount (song price) are passed to the smart contract. The smart contract extracts the song hash and adds it to this user's address as an owned song. The user's wallet is decremented by the amount and it is added to the artist's wallet.

6.4 uploadSong()

This is a function in JS which is used to upload a new song into the system. It takes it the file uploaded by the user and the purchase price. It generates a buffer of the file using Buffer.from(). This is done so that the buffer can be passed to the IPFS through Infura node. This is done by calling ipfs.files.add(). This send the buffered song to IPFS and returns the unique hash behind which the song is stored in IPFS. The generated hash, song name, and price are then sent to blockchain. This is recorded as a transaction on the block and this is saved as one of the songs in the system.

7. Unique Features of System

7.1 Sample Duration

The music that is available on the system can be sampled by any user for how many ever times they want. But a limit is set of the duration of the song. Currently, a listener can listen to any song for free for 10 seconds. For listening further, the listener has to purchase the song. The sample duration can be increased or even be made as a user input, but that will be done during the next phase

7.2 Decentralized IPFS Storage

The system uses a truly decentralized storage system IPFS for storing music files. IPFS has been explained in detail above. It allows the system to be truly decentralized without relying on a central storage service (like Amazon S3 or Azure). Most applications in the market currently implement the blockchain based music marketplace often rely on a central server/ storage service. Using the IPFS storage makes our system fast, reliable and completely peer-to-peer.

7.3 DRM Compliance

To ensure that only original artists get credit for their work, we have to make sure that there is no reposting of content. For this we have again utilized the functionality of IPHS's unique hash. If a user tries to upload a song that already exists, the system hits him with a toast message informing that the song already exists. This avoid data duplicacy and ensures DRM compliance.

8. Steps to Implement Code

MeloManiac
Melomaniacs is a blockchain powered independent music buying and selling marketplace.

To run the MeloManiac system

- Make sure your system has Ganache installed. Open Ganache and quickstart
- Make sure your browser has MetaMask extension installed
- Create a network for the local system and import Ganache accounts with their private keys

Migrate contract and host in Local host
truffle migrate --reset

Install browserify to generate bundle.js
npm install -g browserify
cd src
browserify app.js -o bundle.js

Start the webserver
npm install
npm start

9. Citation and References

- https://www.youtube.com/watch?v=coQ5dg8wM2o&t=1128s
- https://itnext.io/build-a-simple-ethereum-interplanetary-file-system-ipfs-react-js-dapp-23ff4914ce4e
- https://getbootstrap.com/docs/5.1/getting-started/introduction/
- Infura
- https://docs.npmjs.com/