ELENE6883 Project Report

Topic 1: Vtuber Marketplace Smart Contact Development

Haomiao Li (hl3619), Haichun Zhao(yz2833), Yue Niu(yn2433), Yufei Jin(yj2725), Tianrui Wang (tw2896)
Github: https://github.com/hzhao233/Vtuber_Marketplace

1 Introduction

Over the past hundred years, centralized marketplaces have dominated the world. Centralized marketplaces are controlled by a single entity, which can result in limitations such as obligatory fees for listing and selling items, lack of privacy, and inadequate transaction security. However, with the aid of blockchain and Ethereum, decentralized online marketplaces can be developed and implemented in the real world, which has better privacy, greater transparency, and lower fees.NFT (Non-Fungible Token), is a digital asset that represents ownership of a unique item on a blockchain. Compared with cryptocurrencies such as Bitcoin or Ethereum, NFTs are not interchangeable, as each one has its own distinct values. Recent years have witnessed rapid growth in the NFT market, with some NFTs selling for millions of dollars.

In the meanwhile, Vtuber has become a major heat in recent years in East Asia. Vtuber is a type of online personality in Japan and China who uses a digital avatar or character to interact with their audience. They typically use live streaming platforms such as YouTube, Twitch, or Bilibili to broadcast their activities and engage with viewers in real-time. With the growing followers of Vtubers, the industry has generated over 10 billion dollars of market value in 2022. Our team has combined these two popular concepts together and developed a decentralized Vtuber NFT marketplace based on Ethereum technology with Truffle evolvement framework with solidity language. The application follows ERC-721standred, which provides a framework for creating and managing unique digital assets that can be easily verified, bought, sold, or traded.

2 Procedure, methodology, and results

In this section, the design procedure methodology and results are presented.

2.1 Procedure

2.1.1 Build the NFT Smart Contract:

To create a BoredPetsNFT smart contract, we first need to edit the SimpleStorage.sol contract under contracts/ethereum by changing the file name and contract name to BoredPetsNFT.sol. Additionally, we need to install OpenZeppelin and switch back into the nft-marketplace directory. In the smart contract, we define a few variables, including the address of the Marketplace contract and the event NFTMinted, which is emitted every time a NFT is minted. The parameters of the event are stored in the transaction's log, and we will need the tokenId later when building out the web app.
The mint function in the smart contract takes a string memory _tokenURI as its only parameter, which points to the JSON metadata on IPFS that stores the NFT's metadata (i.e., image, name, description). This function mints an NFT with an increasing, unique token Id. We use setApprovalForAll to grant approver access to our Marketplace contract for transferring ownership of the NFT between various addresses.

2.1.2 Build the Marketplace Contract

In this section, we will introduce a new contract called Marketplace.sol under the contracts/ethereum directory, which will provide all the necessary marketplace functionality. The Marketplace contract inherits ReentrancyGuard to prevent reentrancy attacks.

2.1.3 Deploy the Smart Contracts Locally

To deploy the smart contracts, modifications must be made to the migrations/1_deploy_contracts.js file. In our project, the graphical user interface will be used. A workspace was created and saved in Ganache. This will create a running Ganache instance at HTTP://127.0.0.1:7545. The development network in the truffle-config.js file was edited to match the port number. The compiled contracts can be found under ./client/contracts/ethereum-contracts.

2.1.4 Wite a script

In order to automate common tasks, we utilized the "truffle exec" command to execute scripts. To begin, we created a new file named "run.js" under a new scripts folder. Within this file, we accessed the contract abstractions using "artifacts.require". We then interacted with the contracts using the "@truffle/contracts" convenience library. To write unit tests in Truffle in JavaScript or TypeScript, we utilized the functionality provided by the "@truffle/contracts" library. If using TypeScript, it was necessary to

create a "tsconfig.json" file and use "tsc" to compile down to JavaScript. After completing the script, we ran "truffle exec scripts/run.js" to execute it successfully. Lastly, if one wishes to deploy contracts on a populated blockchain, it is possible to use Ganache to fork the main net with zero configuration.

2.1.5 Testing

In this section, we describe the testing procedures and results of the Marketplace contract. We used the Truffle framework to test the contract.

2.2 Methodology

In this design, the ERC721 standard is introduced to provide a blueprint for creating and managing unique, indivisible digital assets on the Ethereum blockchain. ERC-721 tokens are non-fungible and unique, which helps identify and differentiate it from other tokens. It is also ownership taken and supports metadata, which makes it widely adopted for creating and managing NFT marketplaces. In terms of the code structure, there are two main sections in this design: contracts, and tests.

In the contract file, there are two .sol files. The VtuberNFT.sol uses OpenZeppelin's contracts library to handle the implementation of the ERC-721 standard and related functionality. The code firstly imports the necessary contracts and libraries from the OpenZeppelin package. Then it defines the contract inherited from ERC721URIStorage. After that, The 'using Counters for Counters.Counter' statement enables the contract to use the Counters library to handle the token ID counter. Finally, the mint function, allows users to mint a new NFT with a specified token URI (metadata). The marketplace.sol uses smart contract to allow users to list, buy, and resell their ERC-721 based NFTs. The contract employs the OpenZeppelin library and utilizes ReentrancyGuard to prevent reentrancy attacks. There are six functions in this file, including listNft(), buyNft(), resellNft(), getListedNfts(), getMyNfts(), getMyListedNfts(), which is responsible for

(1) Listing an NFT on the marketplace by specifying its contract address, token ID, and price.
(2) Allowing a user to buy an NFT listed on the marketplace by providing its contract address and token ID
(3) Enableing a user to resell an NFT purchased from the marketplace by specifying its contract address, token ID, and new price.
(4) Returning an array of NFT structs representing all the NFTs currently listed on the marketplace
(5) Returning an array of NFT structs representing all the NFTs owned by the caller.
(6) Returning an array of NFT structs representing all the NFTs listed by the caller.

In terms of the testing folder, the marketplace.js file is a test script for the Marketplace contract using the Truffle framework. It tests various aspects of the contract, such as listing, buying, and reselling NFTs.

(1) It firstly checks if the contract correctly validates the listing conditions.
(2) Then it Tests the functionality of listing an NFT, ensuring that the listing fee is transferred, and the appropriate events are emitted.
(3) After that, it verifies that the contract correctly validates the buying conditions, such as checking if the buyer has sent enough ether to cover the asking price.
(4) It tests the behavior of the contract when an NFT is bought, checking if the NFT is removed from the listed NFTs, and if the payment is transferred to the seller.
(5) It checks if the contract validates the reselling conditions correctly and the reselling functionality
(6) It ensures that the NFT is relisted with the updated seller, owner, and listing status.

3 Results and Showcase

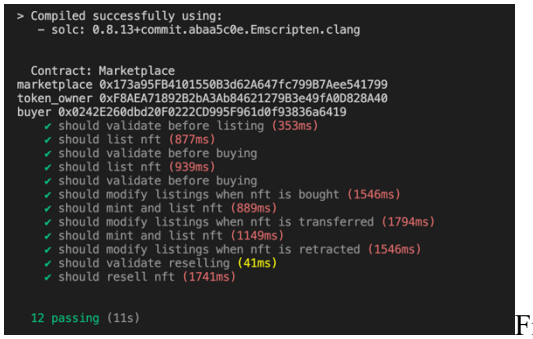


Fig.1 Pass all test cases

3.1 Test that the smart contract can create a new NFT and can list an NFT for sale

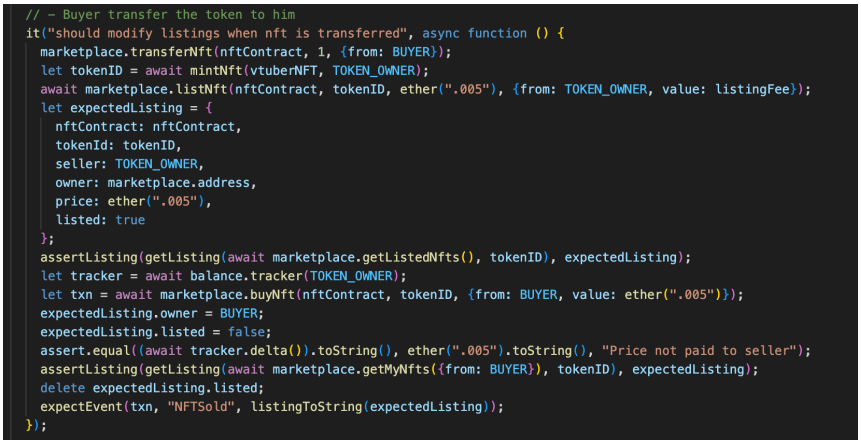
This test case satisfies the first and third testing criteria in design requirements, creating a new NFT and listing it. The test case firstly mints an NFT with the mintNft function, providing the vtuberNFT contract and TOKEN_OWNER as arguments. Then it creates a balance tracker for the marketplace owner's account. After that, it lists the newly minted NFT on the marketplace using the listNft function by specifying the contract address, token ID, price, and the listing fee and asserting that the listing fee has been transferred from the NFT owner to the marketplace owner. It then defines the expected listing object, which includes the NFT contract, token ID, seller, owner (marketplace contract address), price, and listed status (true). Finally, it retrieves the NFT from the getListedNfts function and checks that it matches the expected listing object and retrieves the NFT from the getMyListedNfts function (as the token owner) and checks that it matches the expected listing object. It removes the 'listed' property from the expected listing object and check if the "NFTListed" event has been emitted with the correct parameters.t

```
// Mint and list nft
it("should list nft", async function () {
  let tokenID = await mintNft(vtuberNFT, TOKEN_OWNER);
  let tracker = await balance.tracker(MARKETPLACE_OWNER);
  await tracker.get();
  let txn = await marketplace.listNft(nftContract, tokenID, ether(".005"), {from: TOKEN_OWNER, value: listingFee});
  assert.equal(await tracker.delta(), listingFee, "Listing fee not transferred");
  let expectedListing = {
    nftContract: nftContract,
    tokenId: tokenID,
    seller: TOKEN_OWNER,
    owner: marketplace.address,
    price: ether(".005"),
    listed: true
  };
  assertListing(getListing(await marketplace.getListedNfts(), tokenID), expectedListing);
  assertListing(getListing(await marketplace.getMyListedNfts({from: TOKEN_OWNER}), tokenID), expectedListing);
  delete expectedListing.listed;
  expectEvent(txn, "NFTListed", listingToString(expectedListing));
});
```

Fig. 2 Testing code for creating and listing NFT

3.2 Test that the smart contract can transfer ownership of an NFT

The test case for transferring NFT is shown in Figure 2. It first transfers the NFT with token ID 1 to the buyer. It mints a new NFT with the mintNft function, providing the vtuberNFT contract and TOKEN_OWNER as arguments. Then it lists the newly minted NFT on the marketplace using the listNft function by specifying the contract address, token ID, price, and the listing fee. It also defines the expected listing object, which includes the NFT contract, token ID, seller, owner (marketplace contract address), price, and listed status (true). After that, it retrieves the NFT from the getListedNfts function and checks that it matches the expected listing object. Then, it creates a balance tracker for the token owner's account. Then, it initiates the purchase of the NFT using the buyNft function by specifying the contract address, token ID, and the buyer's payment. It then updates the expected listing object with the new owner (buyer) and set the listed status to false and asserts that the token owner has received the correct payment for the NFT. Finally, it retrieves the NFT from the getMyNfts function (as the buyer) and checks that it matches the updated expected listing object and removes the 'listed' property from the expected listing object and checks if the "NFTSold" event has been emitted with the correct parameters.



```
// - Buyer transfer the token to him
it("should modify listings when nft is transferred", async function () {
  marketplace.transferNft(nftContract, 1, {from: BUYER});
  let tokenID = await mintNft(vtuberNFT, TOKEN_OWNER);
  await marketplace.listNft(nftContract, tokenID, ether(".005"), {from: TOKEN_OWNER, value: listingFee});
  let expectedListing = {
    nftContract: nftContract,
    tokenId: tokenID,
    seller: TOKEN_OWNER,
    owner: marketplace.address,
    price: ether(".005"),
    listed: true
  };
  assertListing(getListing(await marketplace.getListedNfts(), tokenID), expectedListing);
  let tracker = await balance.tracker(TOKEN_OWNER);
  let txn = await marketplace.buyNft(nftContract, tokenID, {from: BUYER, value: ether(".005")});
  expectedListing.owner = BUYER;
  expectedListing.listed = false;
  assert.equal((await tracker.delta()).toString(), ether(".005").toString(), "Price not paid to seller");
  assertListing(getListing(await marketplace.getMyNfts({from: BUYER}), tokenID), expectedListing);
  delete expectedListing.listed;
  expectEvent(txn, "NFTSold", listingToString(expectedListing));
});
```

Fig. 3 Testing code for transferring NFT

3.3 Test that the smart contract can remove an NFT from sale

For testing the retracting nft, we need two test cases. The first test case mints an NFT, lists it on the marketplace, checks the correct listing status, and then removes it from sale. The second test case examines the marketplace's behavior when an NFT is bought after it has been retracted, ensuring proper ownership transfer and event emission.

```
// Retract a sale
it("should mint and list nft", async function () {
  let tokenID = await mintNft(vtuberNFT, TOKEN_OWNER);
  let tracker = await balance.tracker(MARKETPLACE_OWNER);
  await tracker.get();
  let txn = await marketplace.listNft(nftContract, tokenID, ether(".005"), {from: TOKEN_OWNER, value: listingFee});
  assert.equal(await tracker.delta(), listingFee, "Listing fee not transferred");
  let expectedListing = {
    nftContract: nftContract,
    tokenId: tokenID,
    seller: TOKEN_OWNER,
    owner: marketplace.address,
    price: ether(".005"),
    listed: true
  };
  assertListing(getListing(await marketplace.getListedNfts(), tokenID), expectedListing);
  assertListing(getListing(await marketplace.getMyListedNfts({from: TOKEN_OWNER}), tokenID), expectedListing);
  delete expectedListing.listed;
  expectEvent(txn, "NFTListed", listingToString(expectedListing));
  marketplace.removeNFTfromSale(nftContract, tokenID, {from: TOKEN_OWNER});
});
it("should modify listings when nft is retracted", async function () {
  let tokenID = await mintNft(vtuberNFT, TOKEN_OWNER);
  await marketplace.listNft(nftContract, tokenID, ether(".005"), {from: TOKEN_OWNER, value: listingFee});
  let expectedListing = {
    nftContract: nftContract,
    tokenId: tokenID,
    seller: TOKEN_OWNER,
    owner: marketplace.address,
    price: ether(".005"),
    listed: true
  };
  assertListing(getListing(await marketplace.getListedNfts(), tokenID), expectedListing);
  let tracker = await balance.tracker(TOKEN_OWNER);
  let txn = await marketplace.buyNft(nftContract, tokenID, {from: BUYER, value: ether(".005")});
  expectedListing.owner = BUYER;
  expectedListing.listed = false;
  assert.equal((await tracker.delta()).toString(), ether(".005").toString(), "Price not paid to seller");
  assertListing(getListing(await marketplace.getMyNfts({from: BUYER}), tokenID), expectedListing);
  delete expectedListing.listed;
  expectEvent(txn, "NFTSold", listingToString(expectedListing));
});
```

Fig. 4 Testing code for retracting NFT listing

3.4 Test that the smart contract can execute a successful NFT purchase

Testing buying NFT contains two steps. The first step, "should validate before buying," checks that the buyer has enough ether to cover the asking price of the NFT. If not, the test expects the transaction to revert with the message "Not enough ether to cover asking price." The second step, "should modify listings when nft is bought," checks that the marketplace behaves correctly when a buyer purchases an NFT. This test firstly Mints a new NFT with the mintNft function and lists the NFT on the marketplace using the listNft function. It then defines the expected listing object with details such as the NFT contract, token ID, seller, owner, price, and listed status. After that, it asserts that the NFT is listed correctly on the marketplace using the getListedNfts function. Then it creates a balance tracker for the token owner's account. It then Executes the buyNft function to purchase the NFT, providing the NFT contract, token ID, and buyer's payment and updates the expected listing object with the new owner (buyer) and sets the listed status to false. Then, it asserts that the token owner has received the correct payment for the NFT and the NFT is now owned by the buyer using the getMyNfts function. Finally, it removes the 'listed' property from the expected listing object and checks if the "NFTSold" event has been emitted with the correct parameters.

```
// Buying
it("should validate before buying", async function () {
  await expectRevert(
    marketplace.buyNft(nftContract, 1, {from: BUYER}),
    "Not enough ether to cover asking price"
  );
});
it("should modify listings when nft is bought", async function () {
  let tokenID = await mintNft(vtuberNFT, TOKEN_OWNER);
  await marketplace.listNft(nftContract, tokenID, ether(".005"), {from: TOKEN_OWNER, value: listingFee});
  let expectedListing = {
    nftContract: nftContract,
    tokenId: tokenID,
    seller: TOKEN_OWNER,
    owner: marketplace.address,
    price: ether(".005"),
    listed: true
  };
  assertListing(getListing(await marketplace.getListedNfts(), tokenID), expectedListing);
  let tracker = await balance.tracker(TOKEN_OWNER);
  let txn = await marketplace.buyNft(nftContract, tokenID, {from: BUYER, value: ether(".005")});
  expectedListing.owner = BUYER;
  expectedListing.listed = false;
  assert.equal((await tracker.delta()).toString(), ether(".005").toString(), "Price not paid to seller");
  assertListing(getListing(await marketplace.getMyNfts({from: BUYER}), tokenID), expectedListing);
  delete expectedListing.listed;
  expectEvent(txn, "NFTSold", listingToString(expectedListing));
});
```

Fig. 5 Testing code for buying NFT successfully

3.5 Test that the smart contract can execute an unsuccessful NFT purchase

By slightly change the test case of 3.4 we can verify that if the purchased amount is larger than the balance, there will be an error occurred.

```
// Unsuccessful buying
it("should list nft", async function () {
  let tokenID = await mintNft(vtuberNFT, TOKEN_OWNER);
  let tracker = await balance.tracker(MARKETPLACE_OWNER);
  await tracker.get();
  let txn = await marketplace.listNft(nftContract, tokenID, ether("100000000000000.0"), {from: TOKEN_OWNER, value: listingFee});
  assert.equal(await tracker.delta(), listingFee, "Listing fee not transferred");
  let expectedListing = {
    nftContract: nftContract,
    tokenId: tokenID,
    seller: TOKEN_OWNER,
    owner: marketplace.address,
    price: ether("100000000000000.0"),
    listed: true
  };
  assertListing(getListing(await marketplace.getListedNfts(), tokenID), expectedListing);
  assertListing(getListing(await marketplace.getMyListedNfts({from: TOKEN_OWNER}), tokenID), expectedListing);
  delete expectedListing.listed;
  expectEvent(txn, "NFTListed", listingToString(expectedListing));
});
it("should validate before buying", async function () {
  await expectRevert(
    marketplace.buyNft(nftContract, 1, {from: BUYER}),
    "Not enough ether to cover asking price"
  );
});
```

Fig. 6 Testing code for buying NFT unsuccessfully

3.6 **Assert the NFT ownership remains with the first user account and no Ether was transferred**

When an unsuccessful buying happens, buyNFT function will immediately fail before it can reach transferring Ether and reassign nft.owner. In marketplace.sol, buying function has a *require* statement to first make sure the buyer has enough funds to buy the NFT before anything could happen.

```
// Buy an NFT
function buyNft(address _nftContract, uint256 _tokenId) public payable nonReentrant {
  NFT storage nft = _idToNFT[_tokenId];
  require(msg.value >= nft.price, "Not enough ether to cover asking price");

  address payable buyer = payable(msg.sender);
  payable(nft.seller).transfer(msg.value);
  IERC721(_nftContract).transferFrom(address(this), buyer, nft.tokenId);
  nft.owner = buyer;
  nft.listed = false;

  _nftsSold.increment();
  emit NFTSold(_nftContract, nft.tokenId, nft.seller, buyer, msg.value);
}
```

Fig. 7 Buying code

4 Conclusion

In this project, we have successfully developed a decentralized Vtuber NFT marketplace based on Ethereum technology with Truffle evolvement framework with solidity language. We passed the test cases which satisfies the 7 criteria made in the requirement.