

NFT Marketplace Smart Contract Development

Name: Yang Yu, yy3269

Code: https://github.com/hjahdv/blockchain_project

1. ERC-721 Introduction

ERC-721 is a token standard for Ethereum smart contracts that introduced a new type of token: Non-Fungible Tokens (NFTs). Unlike Fungible Tokens, each ERC-721 token is unique and can have different values and attributes. The ERC-721 standard defines a set of interface methods that allow standardized operations on tokens based on this standard. For example, tokens can be created, destroyed, transferred, and so on through interfaces defined by ERC-721. In addition, ERC-721 also specifies the metadata information that each token should include, such as name, description, and image.

Compared to other token standards, the main advantage of ERC-721 is its applicability in the digital asset field. It can be used to represent various unique and non-fungible assets, such as original artworks, virtual land, etc. Cryptokitties is a popular example of a blockchain game built on the Ethereum platform using ERC-721 tokens. It allows players to purchase, collect, breed, and trade virtual cats, with each cat being unique and represented by its own NFT token.

For ERC-721 metadata, a JSON object can be used to represent the metadata information such as the name, description, image, and other attributes. The JSON object can have its own properties and constraints based on the needs of the application. The following is the NFT JSON schema definition:

```
{
  "name": "Identifies the asset to which this NFT represents",
  "description": "Describes the asset to which this NFT represents",
  "image": "A URI pointing to a resource with mime type image/*
representing the asset to which this NFT represents. Consider making any
images at a width between 320 and 1080 pixels and aspect ratio between
1.91:1 and 4:5 inclusive."
}
```

Names, descriptions, and attributes are easy to store on-chain, but images pose a challenge. Many people choose AWS or any other cloud service to host information, but it goes against the decentralized nature of blockchain. If we can store images on the chain, they won't be susceptible to downtime or hacking. However, on-chain storage takes up a lot of space, and requires a significant amount of gas.

IPFS (InterPlanetary File System) is a better choice, a distributed storage system. IPFS utilizes content-addressing, a technique that ensures that the content of a file can be verified and hasn't been tampered with. Moreover, In contrast to centralized storage systems, IPFS utilizes a decentralized network of nodes to store and share files. This ensures that data stored on IPFS is more reliable, secure, and resistant to censorship and failure.

2. Work Summary

2.1 Implement NFT Smart Contract

Create a new file in the contracts directory, e.g. NFTMarketplace.sol. Write the smart contract code for creating, transferring, listing, and selling NFTs. Next I'll explain what each piece of code means.

```
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
contract NFTMarketplace is ERC721, Ownable {
}
```

This code declares a contract called NFTMarketplace which inherits from the ERC721 and Ownable contracts. The ERC721 is a standard contract for creating non-fungible digital assets (NFTs), while the Ownable contract allows certain functions of the contract to only be called by the owner. The import statements specify that the code is utilizing other pre-existing contracts from the OpenZeppelin library, which have already been implemented and tested.

```
struct NFT {
    string name;
    string description;
    bool forSale;
    uint256 price;
}
mapping(uint256 => NFT) private nfts;
```

This code declares a struct named NFT, and uses a mapping type to associate instances of the NFT struct with uint256 identifiers. This means that we can look up and access the corresponding NFT struct instance by using the identifier. The struct contains properties such as the name, description, whether it is for sale, and the price. The `private` keyword indicates that this Mapping can only be accessed internally within the contract.

```
constructor() ERC721("NFT Marketplace", "HW") {}
```

This line of code is the constructor for the contract, which is executed when the contract is deployed. The `ERC721` keyword indicates that this contract is inheriting from the ERC721 smart contract, which is a standardized token contract for NFT. The parentheses contain values for the name and symbol of the NFT. The `NFT Marketplace` is the name of NFTs being traded on this marketplace and `HW` is the symbol for these NFTs.

```
function _baseURI() internal pure override returns (string memory) {
    return "ipfs://QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea/";
}
```

The function returns a string literal which represents a universal resource identifier (URI) for the base location of the metadata of a NFT collection.

```
function createNFT(uint256 tokenId, string memory name, string memory
description) public onlyOwner {
    require(!_exists(tokenId), "NFT with this ID already exists");
    _safeMint(msg.sender, tokenId);
    nfts[tokenId] = NFT(name, description, false, 0);
}
```

- The function takes three input parameters: `tokenId`, `name`, and `description`. It's worth noting that the `onlyOwner` modifier restricts access to this function only to the contract owner, ensuring that only authorized parties can create new NFTs.
- First, the function checks if an NFT with the given `tokenId` already exists. If so, it will return an error message and exit the function.
- If the `tokenId` is unique, the function will use the `_safeMint` function to create a new NFT token and mint it to the contract owner. `_safeMint` is a built-in function of OpenZeppelin's ERC-721 implementation, which is a safe way to mint new tokens that ensures compliance with the ERC-721 standard.
- Finally, the function adds the newly created NFT's metadata to a mapping called `nfts`.

```
function transferNFT(uint256 tokenId, address to) public {
    require(_exists(tokenId), "NFT with this ID does not exist");
    require(ownerOf(tokenId) == msg.sender, "Only the owner can transfer
this NFT");
    safeTransferFrom(msg.sender, to, tokenId);
}
```

- The function takes two inputs, `tokenId` and `to`, representing the unique identifier of the NFT and the new owner's address, respectively.
- First, the function checks if the NFT with the given `tokenId` exists. If it doesn't exist, the function returns an error message and exits.
- Next, the function verifies that the sender is the current owner of the NFT. If not, the function returns an error message and exits.
- Assuming the NFT exists and the sender is the current owner, the function then calls a built-in function called `safeTransferFrom` to transfer ownership of the NFT from the sender to the new owner at the specified `to` address.

```
function listNFTForSale(uint256 tokenId, uint256 price) public {
    require(_exists(tokenId), "NFT with this ID does not exist");
    require(ownerOf(tokenId) == msg.sender, "Only the owner can list this
NFT for sale");
    nfts[tokenId].forSale = true;
    nfts[tokenId].price = price;
}
```

- The function takes two inputs, `tokenId` and `price`, representing the unique identifier of the NFT and the sale price in wei, respectively.
- First, the function checks if the NFT with the given `tokenId` exists. If it doesn't exist, the function returns an error message and exits.
- Next, the function verifies that the sender is the current owner of the NFT. If not, the function returns an error message and exits.
- Assuming the NFT exists and the sender is the current owner, the function sets the corresponding instance in the `nfts` mapping to mark the NFT as `forSale` with the given `price`.

```
function removeNFTFromSale(uint256 tokenId) public {
    require(_exists(tokenId), "NFT with this ID does not exist");
    require(ownerOf(tokenId) == msg.sender, "Only the owner can remove
this NFT from sale");
    nfts[tokenId].forSale = false;
    nfts[tokenId].price = 0;
}
```

- The function takes one input, `tokenId`, representing the unique identifier of the NFT.
- First, the function checks if the NFT with the given `tokenId` exists. If it doesn't exist, the function returns an error message and exits.
- Next, the function verifies that the sender is the current owner of the NFT. If not, the function returns an error message and exits.
- Assuming the NFT exists and the sender is the current owner, the function sets the corresponding instance in the `nfts` mapping to mark the NFT as `not forSale` by setting `forSale` to false and `price` to 0.

```
function purchaseNFT(uint256 tokenId) public payable {
    require(_exists(tokenId), "NFT with this ID does not exist");
    require(nfts[tokenId].forSale, "NFT is not for sale");
    require(msg.value == nfts[tokenId].price, "Incorrect Ether amount");
    address payable owner = payable(ownerOf(tokenId));
    _transfer(owner, msg.sender, tokenId);
    owner.transfer(msg.value);
    nfts[tokenId].forSale = false;
    nfts[tokenId].price = 0;
}
```

- The function takes one input, `tokenId`, representing the unique identifier of the NFT. The buyer must also send Ether equal to the sale price of the NFT.
- First, the function checks if the NFT with the given `tokenId` exists. If it doesn't exist, the function returns an error message and exits.
- Next, the function verifies that the NFT is actually for sale by checking the `forSale` flag in the `nfts` mapping. If the NFT is not for sale, the function returns an error message and exits.
- Assuming the NFT exists and is for sale, the function verifies that the amount of Ether sent by the buyer matches the sale price of the NFT. If the amounts do not match, the function returns an error

message and exits.

- If all requirements are met, the function transfers ownership of the NFT from the current owner to the buyer using the built-in `_transfer` function. It also transfers the Ether sent by the buyer to the original owner.
- Finally, the function clears the `forSale` and `price` metadata for the NFT.

```
function getNFT(uint256 tokenId) public view returns (NFT memory nft) {
    require(_exists(tokenId), "NFT with this ID does not exist");
    return nfts[tokenId];
}
```

- The function takes one input, `tokenId`, representing the unique identifier of the NFT.
- First, the function checks if the NFT with the given `tokenId` exists. If it doesn't exist, the function returns an error message and exits.
- Assuming the NFT exists, the function returns the corresponding instance of the `NFT` struct stored in the `nfts` mapping. The returned `NFT` struct contains metadata such as `forSale` and `price` associated with the given `tokenId`.

2.2 Test NFT Smart Contract

Create a new file in the test directory, e.g. `test_nft_x.js`. Write test cases for each function in the smart contract. `NFTMarketplace` is the NFT contract instance we created, and subsequent test cases will be written according to this template:

```
const NFTMarketplace = artifacts.require("NFTMarketplace");
contract("NFTMarketplace", accounts => {
    it("Test that the smart contract can create a new NFT", async function
    () {
        // Create a new instance of the smart contract
        const nftMarketplaceInstance = await NFTMarketplace.deployed();
        // Create two user accounts
        const seller = accounts[0];
        const buyer = accounts[1];
        // Create a new NFT with the first user account
        const tokenId = 1;
        const name = "Test NFT";
        const description = "This is a test NFT";
        .....
    });
});
```

Case1

Call the `createNFT` function with a unique ID, name, and description.

```
await nftMarketplaceInstance.createNFT(tokenId, name, description, { from: accounts[0] });
```

Assert that the NFT was created and its properties match the input values.

```
const nft = await nftMarketplaceInstance.getNFT(tokenId);  
assert.equal(nft[0], name);  
assert.equal(nft[1], description);
```

Case2

Transfer ownership of the NFT to the second user account using the transferNFT function.

```
await nftMarketplaceInstance.transferNFT(tokenId, buyer, {from: seller});
```

Assert that the NFT is now owned by the second user account.

```
const nowOwner = await nftMarketplaceInstance.ownerOf(tokenId);  
assert.equal(nowOwner, buyer);
```

Case3

List the NFT for sale using the listNFTForSale function with a sale price.

```
const salePrice = web3.utils.toWei("1", "ether");  
await nftMarketplaceInstance.listNFTForSale(tokenId, salePrice, { from: owner });
```

Assert that the NFT is now listed for sale and its sale price matches the input value.

```
const nft = await nftMarketplaceInstance.getNFT(tokenId);  
assert.equal(nft.forSale, true);  
assert.equal(nft.price, salePrice);
```

Case4

Remove the NFT from sale using the removeNFTFromSale function.

```
await nftMarketplaceInstance.removeNFTFromSale(tokenId, {from: owner});
```

Assert that the NFT is no longer listed for sale.

```
const nft = await nftMarketplaceInstance.getNFT(tokenId);
assert.equal(nft.forSale, false);
assert.equal(nft.price, 0);
```

Case5

Purchase the NFT using the purchaseNFT function with the second user account and the correct amount of Ether.

```
const balanceSellerBefore = web3.utils.toBN(await
web3.eth.getBalance(seller));
await nftMarketplaceInstance.purchaseNFT(tokenId, { from: buyer, value:
salePrice });
const balanceSellerAfter = web3.utils.toBN(await
web3.eth.getBalance(seller));
```

Assert that the NFT is now owned by the second user account and the correct amount of Ether was transferred to the first user account.

```
const newOwner = await nftMarketplaceInstance.ownerOf(tokenId);
assert.equal(newOwner, buyer);
assert.equal(balanceSellerAfter.sub(balanceSellerBefore), salePrice);
```

Case6

Attempt to purchase the NFT using the purchaseNFT function with the second user account but with an incorrect amount of Ether.

```
const balanceSellerBefore = web3.utils.toBN(await
web3.eth.getBalance(seller));
const incorrectPrice = salePrice / 2;
try {
    await nftMarketplaceInstance.purchaseNFT(tokenId, { from: buyer,
value: incorrectPrice });
} catch (err) {
    assert(err.message.includes("Incorrect Ether amount"), "Purchase
should fail with incorrect Ether amount");
}
const balanceSellerAfter = web3.utils.toBN(await
web3.eth.getBalance(seller));
```

Assert that the NFT ownership remains with the first user account and no Ether was transferred.

```
const newOwner = await nftMarketplaceInstance.ownerOf(tokenId);
assert.equal(newOwner, seller);
assert.equal(balanceSellerAfter.sub(balanceSellerBefore), 0);
```

Result

```
truffle test ./test/test_nft_*.js
```

Contract: NFTMarketplace

✓ Test that the smart contract can create a new NFT (202ms)

Contract: NFTMarketplace

✓ Test that the smart contract can transfer ownership of an NFT (231ms)

Contract: NFTMarketplace

✓ Test that the smart contract can list an NFT for sale (231ms)

Contract: NFTMarketplace

✓ Test that the smart contract can remove an NFT from sale (250ms)

Contract: NFTMarketplace

✓ Test that the smart contract can execute a successful NFT purchase (256ms)

Contract: NFTMarketplace

✓ Test that the smart contract can execute an unsuccessful NFT purchase (332ms)

3. Result Summary

After the smart contract and test cases are written, contract deployment and NFT generation will start below.




3.1 Prepare Image

Step1. Prepare three pictures and upload them to IPFS.

Step2. According to the ERC721 specification, write 3 JSON files named 0, 1, and 2 respectively.

Step3. Create a folder data, put three JSON files in it, and upload it to the IPFS.

Get the CID of the folder: QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea

Index of /ipfs/QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea			
QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea			
	0	QmVo...jn7g	141 B
	1	QmYu...tiXx	141 B
	2	QmeG...eNDR	141 B

These three pictures can be indexed through this folder CID, that is, three NFTs:

<https://ipfs.io/ipfs/QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea/0>

<https://ipfs.io/ipfs/QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea/1>

<https://ipfs.io/ipfs/QmPi6AabevKz4EBRBDh2MXei2QrnQgoPK5R1sZyZkYHoea/2>

3.2 Perpare Account

Step1. Installation command: `npm install -g ganache`

Step2. Start command: `ganache`

Step3. After startup, you can see 10 accounts created by default, and each account has a test balance of 1000ETH.

```
ganache v7.7.7 (@ganache/cli: 0.8.6, @ganache/core: 0.8.6)
Starting RPC server
```

Available Accounts

=====

```
(0) 0x976e5C59170555d6bB72B76674c5EDDd143aEEE1 (1000 ETH)
(1) 0xB497490da9C6d73d19f9357bC1676C44b39eA03d (1000 ETH)
(2) 0xa2C0fE2b11c468040bb7eA756786F110C6C1686C (1000 ETH)
(3) 0x4805ec21aC60fe350EB7c7e0eDfc6bE603B0bD45 (1000 ETH)
(4) 0x223a50a490CD1fB4E4EAf461e6838e3512Ae771d (1000 ETH)
(5) 0xe776b64A6Cb4949FC39889304CDAcc548216bf45 (1000 ETH)
(6) 0x39d56d96dee37B1460701Adca4351eCCA32f06E1 (1000 ETH)
(7) 0x94aCD117740a8536E93b2fc99b3C1d233e70b542 (1000 ETH)
(8) 0x938cdA75E1997aFbCdc398d8797C2Ec1020Fe4CB (1000 ETH)
(9) 0xd3A365cFac2F502e8658E22AC8675518fd8aa05b (1000 ETH)
```

3.3 Perpare Truffle

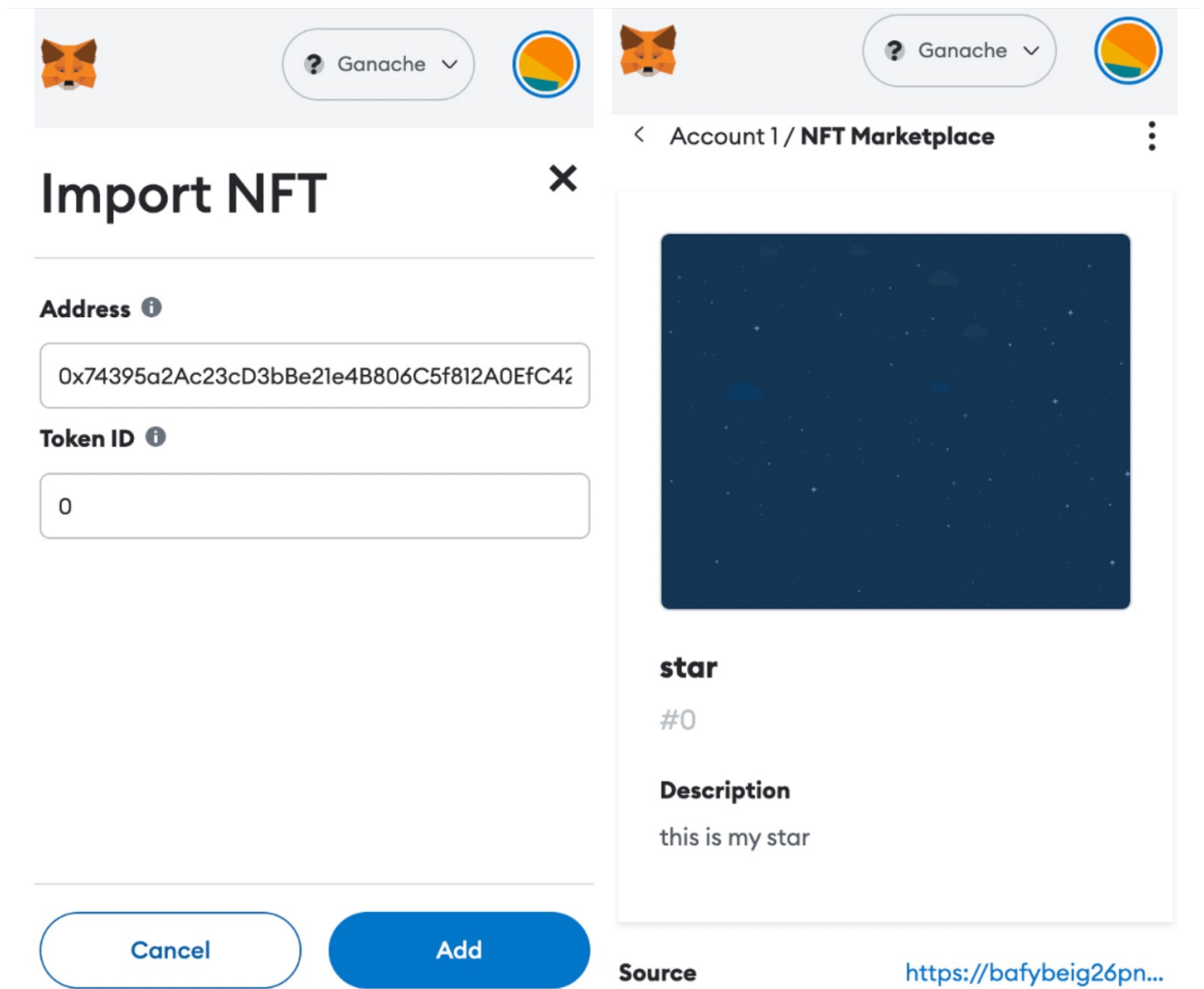
Step1. Installation command: `npm install -g truffle`

Step2. Initialization command: `truffle init`

Step3. Edit `truffle-config.js`, use `ganache` as our develop environment.

```
networks: {
  // Useful for testing. The `development` name is special - truffle uses it by default
  // if it's defined here and no other network is specified at the command line.
  // You should run a client (like ganache, geth, or parity) in a separate terminal
  // tab if you use this network and you must also set the `host`, `port` and `network_id`
  // options below to some value.
  //
  development: {
    host: "127.0.0.1",      // Localhost (default: none)
    port: 8545,            // Standard Ethereum port (default: none)
    network_id: "*",       // Any network (default: none)
  },
},
```

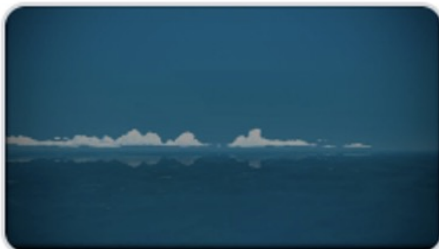
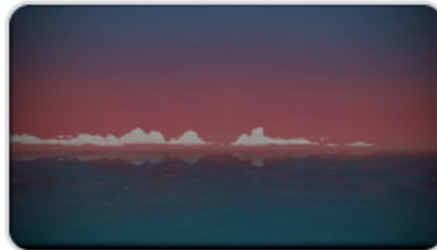
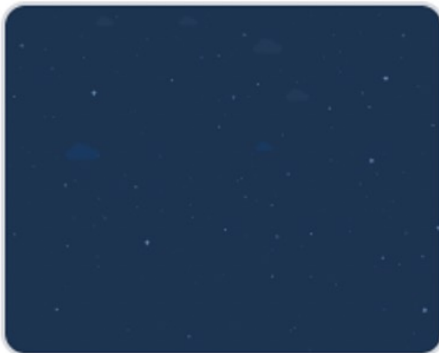

Step3. Copy the contract address and tokenId to MetaMask, you can see that the contract we implemented is compatible with ERC-721, and MetaMask successfully recognized it.



Step4. Continue to add, you can see that the NFTs we originally uploaded on IPFS have been successfully generated.



NFT Marketplace (3)



4. Conclusion

This article analyzes ERC721 with the default implementation provided by OpenZeppelin. The following is a summary:

1. Learn the difference between Fungible tokens and non-Fungible tokens: Fungible tokens are interchangeable and identical, meaning each token has the same value as another. For example, currency is fungible since one dollar has the same value as another dollar. On the other hand, NFT are unique and not interchangeable. Each NFT contains distinct information, making it impossible to exchange one NFT for another. NFT have a wide range of applications, including in collectibles, gaming items, digital art, event tickets, domain names, and even ownership rights of physical assets.
2. Disadvantages of ERC-721: So expensive to deploy and use, particularly for smaller-scale projects. Each NFT requires its own smart contract, which can result in higher gas fees. Additionally, since each NFT is unique, storing and retrieving large numbers of them can be resource-intensive.
3. Learn the difference between on-chain storage and off-chain storage. For large-volume files that cannot be stored in the blockchain, IPFS storage can be used.
4. Where the code can be optimized: TokenID needs to be unique, otherwise it cannot be created. Therefore, in practice, we can maintain the TokenID in an auto-incremental way without artificially passing parameters.

5. References

<https://github.com/ethereum/eips/issues/721>

<https://eips.ethereum.org/EIPS/eip-721>

<https://www.cryptokitties.co/>

<https://trufflesuite.com/docs/truffle/how-to/debug-test/write-tests-in-solidity/>

<https://www.dappuniversity.com/articles/blockchain-developer-toolkit>