CS 48001 Project Final Report

Cavit Çakır, Gökberk Yar
Görkem Köse, Kaya Kapağan

June 2, 2021

Abstract

NFT Suits is a game centered around collectible unique items that are used to create one-of-a-kind avatars where the uniqueness is guaranteed by ERC721 protocol. No items can be replicated, taken away, or destroyed, since the proof of ownerships are stored on Ethereum blockchain. In order to see the items and interact with the marketplace, user has to install MetaMask extension to her/his browser. Then, by connecting his/her wallet; the user can buy, bid on, offer items for auction and sell the items via the marketplace. As tree required and complementary pieces are purchased and worn, an avatar is created and displayed to the community.

Keywords— Blockchain, Ethereum, NFT, Marketplace, Auction, Decentralized Application

Contents

1 Introduction **3**

2 Features **4**
- 2.1 Mint NFT . . . 4
- 2.2 Set Username . . . 5
- 2.3 Put On Sale . . . 6
- 2.4 Cancel Sale . . . 7
- 2.5 Buy Item . . . 8
- 2.6 Put On Auction . . . 9
- 2.7 Cancel Auction . . . 10
- 2.8 Accept Highest Bid . . . 11
- 2.9 Bid on an Item: . . . 12
- 2.10 Withdraw Bid . . . 13
- 2.11 Withdraw Money . . . 14
- 2.12 Wear Item . . . 15
- 2.13 Unwear Item . . . 16
- 2.14 Wear Items . . . 17
- 2.15 Display Avatars . . . 18
- 2.16 List her/his items with certain filtering options . . . 19
- 2.17 See the statistics . . . 20
- 2.18 List all items with certain filtering options . . . 21
- 2.19 Sample Item Page . . . 22

3 Security Analysis **22**
- 3.1 Testing And Unauthorized Access . . . 22
- 3.2 Overflow/Underflow Attacks . . . 26
- 3.3 Re-entrance Attacks . . . 27
- 3.4 Visualization of the Contract . . . 28
- 3.5 Static Analysis . . . 29
  - 3.5.1 Securify v2.0 . . . 29
  - 3.5.2 Securify Slither . . . 32

4 Gas Analysis **32**

5 Extra Cool Stuff: IPFS **33**

6 Conclusion **33**

7 Future Work **33**

8 References **34**

9 Appendix **35**
- 9.1 NFT Contract . . . 35
- 9.2 NFT Contract Test Code . . . 38

1 Introduction

We developed a decentralized marketplace application inspired by the synthesis of the Cryptopunks[2] idea and famous dress up games in the late 90s and early 2000s. CryptoPunk [2] is an inspirational application for our project since they include features like selling avatar(head) digital human like pieces. In addition, CryptoKitties[1] or other decentralized game costume marketplaces are also inspirational since they include sale of a digital costume/avatar/pet. Yet other platforms do not include customization of the personal avatar which consist of many parts, for example a different head, body and low body. Other projects address this customization problem with completely changing the token which leads the less configurable personal avatars. As a novel idea, our project will combine many parts of the body and provide better customization. One can image our project as a combination of CryptoPunks[2] and costume stores for online games with additional features that user can customize their overall avatar.

NFT is a new form of cryptocurrency that uses technology to exhibit, buy and sell digital collectibles, including art, music, video games and more. NFTs contain assets like art and music, instead of holding money. These are effectively digital certificates of ownership that can be attached to digital art or anything else that comes in digital form; audio files, video clips, animated stickers, or even this article. NFTs confirm the ownership of an item by recording it in a digital ledger known as a blockchain, whose details are publicly available and stored on computers on the internet, making it effectively impossible to lose or destroy.

10 heads, 10 tops and 10 bottom wears are pixelized and tokenized as Non-fungible Tokens (NFTs) and stored with proof of ownership on the Ethereum blockchain. Using a single head, top and bottom wear; users create their one-of-a-kind avatar and the completed avatars are displayed within the Avatars page. All the individual items with different rarities are exhibited in the All Items section, where the users can select an item and display its details. The transactions that involve the particular item can be examined via the transaction table of the item. Depending of the ownership status, a user can buy and bid on an item, or create auction and a sale for the item.

2 Features

2.1 Mint NFT

```python
def mint(clothType, name, cid, rarity):
  print(name)
  transaction = contract.functions.mint(clothType, name, cid, rarity).buildTransaction({
      'gas': 800000,
      'gasPrice': w3.toWei('10', 'gwei'),
      'from': '0xFFc867A24F9cf53ed5514D94Cd0992329987132A',
      'nonce': w3.eth.getTransactionCount('0xFFc867A24F9cf53ed5514D94Cd0992329987132A'),
      #'value': w3.toWei(415992086870360064,'wei')
      })
  private_key = "c47c2537e651e436836c48ce252fe4ceabae53364e72f78f09c4592a33872bf6"
  signed_txn = w3.eth.account.signTransaction(transaction, private_key=private_key)
  w3.eth.sendRawTransaction(signed_txn.rawTransaction)

```

```python
import time
for item in items_to_mint:
  mint(**item)
  time.sleep(60)
```

```
Elon Musk
Simba
Black Pants
Green Pants
Blue Pants
Dark Pink Pants
Light Pink Pants
Short Skirt
Striped Pants
Short
Long Skirt
Jeans
Striped Shirt
```

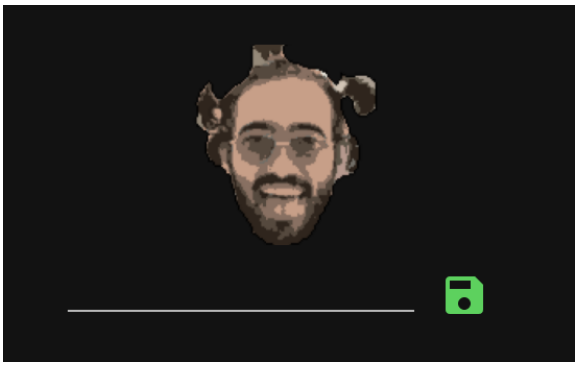
(a)

```solidity
function mint(
    uint256 _clothType,
    string memory _name,
    string memory _cid,
    string memory _rarity
) public {
    require(isExist[_cid] == false, "Item link should be unique, for you to mint it");
    require(this.totalSupply() < maxSupply, "You cannot mint any more item since you already reached the maximum supply.");
    require(msg.sender ==owner, "Only owner can of this contract can mint, you are trying to some fraud.");
    require(_clothType == 1 || _clothType == 2 || _clothType == 3, "Invalid cloth type.");
    uint256 _id =
        nfts.push(
            nftData(
                _clothType,
                _name,
                _cid,
                _rarity,
                false,
                0,
                false,
                0,
                address(0x0),
                false
            )
        );
    _mint(msg.sender, _id);
    isExist[_cid] = true;


    emit nftTransaction(_id, "claimed", address(0x0), msg.sender, 0);

    putOnSale(_id, 10000000000000000);
}
```

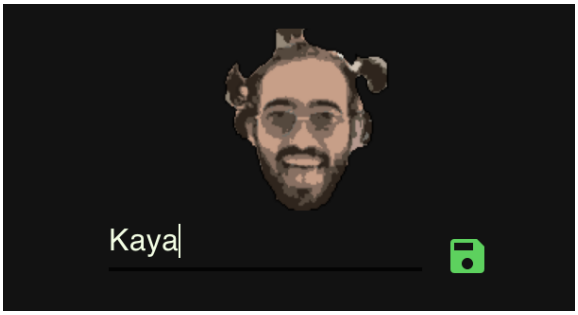
(b)

In order to mint an item, msg.sender should be owner of the contract. The item cid should be unique, total supply should not exceed the maximum supply. After minting process, item is automatically set to on sale by fixed price 0.01 ether.

2.2 Set Username



(a)



(b)

(c)

(d)

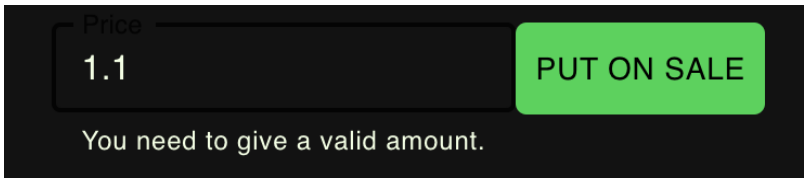
```
function setUsername(string memory _username) public {
    users[msg.sender].username = _username;
}
```

(e)

Figure 2: (a) Username area (b) Username is entered (c) Respective fields containing the username is updated (d) Username is set (e) setUsername function

User can set his/her username after entering the desired username by clicking the save button. When the save button is hit, setUsername function of the contract is called, and the respective user's username is set.

2.3 Put On Sale



(a)

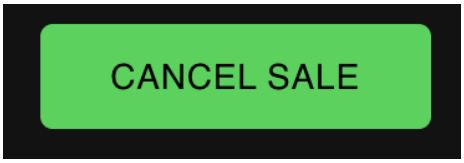
```
function putOnSale(uint256 _tokenId, uint256 _sellPrice) public {

    require(msg.sender == this.ownerOf(_tokenId), "You cannot put this item on sale, because you are not the owner of it.");
    require(nfts[_tokenId - 1].isOnSale == false, "Item is already on sale!");
    require(nfts[_tokenId - 1].isWearing == false, "You must unwear it first, then you can sell it." );

    nfts[_tokenId - 1].isOnSale = true;
    nfts[_tokenId - 1].sellPrice = _sellPrice;
    approve(address(this), _tokenId);

    emit nftTransaction(
        _tokenId,
        "On Sale",
        msg.sender,
        address(0x0),
        _sellPrice
    );
}
```

(b)

Figure 3: (a) Textbox for the price and put on sale button (b) putOnSale function

In order to put an item into sale, the user must be the actual owner of the item, therefore only the actual owner can call putOnSale function. The item shouldn't be already on sale and if the owner wears that item, s/he must unwear it first to put on sale. If all of these conditions hold, then the state of the item is updated and the approve function of the ERC721 contract is called in order to sell the item to any buyer on behalf of the owner. Lastly an event is emitted to display the transaction in transaction table.

2.4 Cancel Sale



(a)

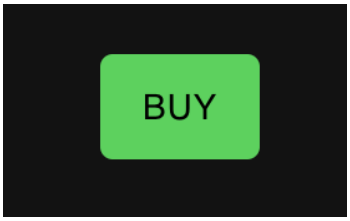
```
function cancelSale(uint256 _tokenId) public {
    require(nfts[_tokenId - 1].isOnSale == true, "Item should be on sale first, to be cancelled.");
    require(msg.sender == this.ownerOf(_tokenId), "You cannot cancel the sale of this item, because you are not the owner.");

    nfts[_tokenId - 1].isOnSale = false;
    nfts[_tokenId - 1].sellPrice = 0;

    emit nftTransaction(
        _tokenId,
        "Sale Cancelled",
        msg.sender,
        address(0x0),
        0
    );
}
```

(b)

Figure 4: (a) Button to cancel the sale for the item (b) calcelSale function

In order to cancel the sale for an item, the user must be the actual owner of the item, therefore only the actual owner can call calcelSale function. The item should be on sale to cancel it. If both of these conditions hold, then the state of the item is updated and an event is emitted to display the transaction in transaction table.

2.5 Buy Item



(a)

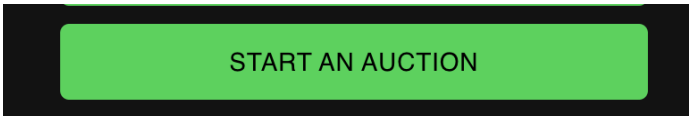
```
function buyFromSale(uint256 _tokenId) public payable {

    require(msg.sender != this.ownerOf(_tokenId), "You cannot buy your own item!");

    require(nfts[_tokenId - 1].isOnSale == true, "Item should be on sale for you to buy it.");

    require (nfts[_tokenId - 1].sellPrice <= msg.value, "The amount you tried to buy, is less than price.");

    require(this.getApproved(_tokenId) == address(this), "Seller did not give the allowance for us to sell this item, contact with seller.");
    address sellerAddress = this.ownerOf(_tokenId);
    this.safeTransferFrom(sellerAddress, msg.sender, _tokenId);
    nfts[_tokenId - 1].isOnSale = false;
    nfts[_tokenId - 1].sellPrice = 0;
    users[sellerAddress].userBalance = add256(users[sellerAddress].userBalance, msg.value);


    if (nfts[_tokenId - 1].maxBid > 0) {
        users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance ,  nfts[_tokenId - 1].maxBid);
    }
    nfts[_tokenId - 1].maxBid = 0;
    nfts[_tokenId - 1].maxBidder = address(0x0);
    nfts[_tokenId - 1].isBiddable = false;

    emit nftTransaction(
        _tokenId,
        "sold",
        sellerAddress,
        msg.sender,
        msg.value
    );
}
```

(b)

Figure 5: (a) Button to buy the item at the price fixed by seller (b) buy function

As the items are placed in the marketplace, a user who is not the owner of a particular item can purchase the item at the price fixed by the seller. The item must be on sale to buy it at a fixed price and the seller must give approval to the contract for the contract to transfer the item. Approval is called within putOnSale function already, but still this is one of the requirements.If above condition hold, safeTransferFrom function of the ERC721 contract is called to safely transfer the item from the seller to buyer. State updated of the item is performed and the balance of the seller is incremented by the price. If there was an auction for the item, then the balance of the highest bidder must be incremented by his/her bid, because from now on, the item is owned by someone else. All the bidding history also refreshed, and an event is emitted to display the purchase on the transaction table.

2.6 Put On Auction



(a)

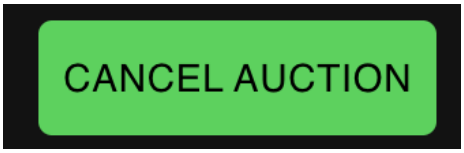
```
function putOnAuction(uint256 _tokenId) public {


    require(msg.sender == this.ownerOf(_tokenId), "Only owner of this item can put on sale" );
    require(nfts[_tokenId - 1].isWearing == false,  "You must unwear it first, then you can put it on auction.");
    require(nfts[_tokenId - 1].isBiddable == false, "This item is already on auction!");

    nfts[_tokenId - 1].isBiddable = true;
    nfts[_tokenId - 1].maxBid = 0;
    approve(address(this), _tokenId);

    emit nftTransaction(
        _tokenId,
        "Auction Starts",
        msg.sender,
        address(0x0),
        0
    );
}
```

(b)

Figure 6: (a) Button to create an auction (b) putOnAuction function

In order to create an auction for an item, the user must be the actual owner of the item, therefore only the actual owner can call putOnAuction function. There shouldn't be an already created auction for the item and if the owner wears that item, s/he must unwear it first. If all of these conditions hold, then the state of the item is updated and the approve function of the ERC721 contract is called in order to sell the item to any buyer on behalf of the owner. Lastly an event is emitted to display the auction creation in transaction table.

2.7 Cancel Auction



(a)

```
function cancelAuction(uint256 _tokenId) public {
    require(msg.sender == this.ownerOf(_tokenId), "You cannot cancel the auction of this item, because you are not the owner.");
    require(nfts[_tokenId - 1].isBiddable == true, "Item must be on auction before it can be canceled, currently it is not!");


    if (nfts[_tokenId - 1].maxBid > 0) {
        users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId - 1].maxBid);
    }
    nfts[_tokenId - 1].isBiddable = false;
    nfts[_tokenId - 1].maxBid = 0;
    nfts[_tokenId - 1].maxBidder = address(0x0);

    emit nftTransaction(
        _tokenId,
        "Auction Cancelled",
        msg.sender,
        address(0x0),
        0
    );
}
```

(b)

Figure 7: (a) Button to cancel the auction (b) cancelAuction function

In order to cancel the auction for an item, the user must be the actual owner of the item, therefore only the actual owner can call cancelAuction function. There should be an already created auction for the item to cancel it. If both of these conditions hold, then the state of the item is updated and an event is emitted to display the cancellation of the auction in transaction table.

2.8 Accept Highest Bid

(a)

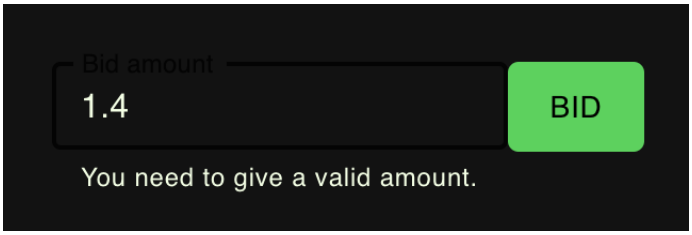
```
function acceptHighestBid(uint256 _tokenId) public {

    require(msg.sender == this.ownerOf(_tokenId), "You need to be owner of this item, to accept its highest bid.");
    require(nfts[_tokenId - 1].isBiddable == true , "Item should be biddable for you to accept its highest bid.");
    require(nfts[_tokenId - 1].maxBid > 0 ,"Max bid must be more than 0 to accept it, currently it is not!");
    require(nfts[_tokenId - 1].maxBidder != msg.sender, "Max bidder cannot be the same person as seller!");

    address buyer = nfts[_tokenId - 1].maxBidder;
    uint256 soldValue = nfts[_tokenId - 1].maxBid;
    this.safeTransferFrom(
        msg.sender,
        nfts[_tokenId - 1].maxBidder,
        _tokenId
    );
    users[msg.sender].userBalance = add256(users[msg.sender].userBalance, nfts[_tokenId - 1].maxBid);
    nfts[_tokenId - 1].maxBid = 0;
    nfts[_tokenId - 1].maxBidder = address(0x0);
    nfts[_tokenId - 1].isBiddable = false;
    nfts[_tokenId - 1].isOnSale = false;
    nfts[_tokenId - 1].sellPrice = 0;

    emit nftTransaction(
        _tokenId,
        "Sold From Auction",
        msg.sender,
        buyer,
        soldValue
    );
}
```

(b)

Figure 8: (a) Button to accept the highest bid in the auction (b) acceptHighestBid function

As in the Put On Auction section, only the actual owner of the item can click on the Accept Highest Bid button and call the respective function to initiate the transaction and receive the money. There should be an auction for the item. The highest bid can't be from the seller and should be greater than zero. If above requirements hold, the ownership of the item is safely transferred from the seller to the highest bidder by the safeTransferFrom function. Balance of the seller is incremented by the highest bid and state of the item is refreshed with default settings.

2.9 Bid on an Item:



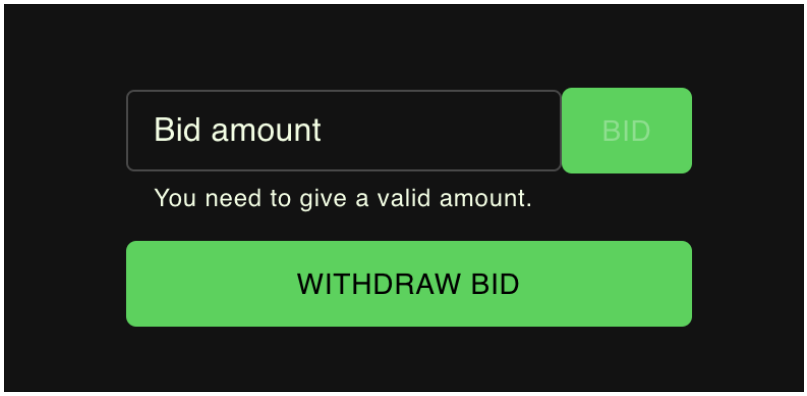
(a)

```
function bid(uint256 _tokenId) public payable {


    require(msg.value > 0, "You did not send any money");
    require(nfts[_tokenId - 1].isBiddable == true,"Item you tried to bid, is not biddable!");
    require(msg.value >= nfts[_tokenId - 1].maxBid, "The amount you tried to bid, is less than current max bid.");
    require(msg.sender != this.ownerOf(_tokenId), "You cannot bid your own item.");

    if (nfts[_tokenId - 1].maxBid > 0) {
        users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId - 1].maxBid);
    }
    nfts[_tokenId - 1].maxBid = msg.value;
    nfts[_tokenId - 1].maxBidder = msg.sender;

    emit nftTransaction(
        _tokenId,
        "Bidded",
        msg.sender,
        ownerOf(_tokenId),
        msg.value
    );
}
```

(b)

In order to bid on an item, the item should be biddable and shouldn't owned by bidder. In addition to these requirements, bidder should bid more than the current maximum bid.
If the requirements are satisfied, the old maximum bid is added to old maximum bidders balance, and maximum bidder is set to new bidder.

2.10 Withdraw Bid



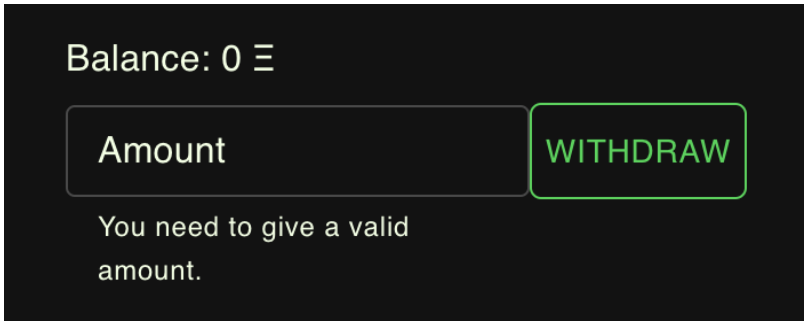
(a)

```
function withdrawBid(uint256 _tokenId) public {

    require(msg.sender == nfts[_tokenId - 1].maxBidder, "You must be the max bidder to withdraw your bid!");
    uint256 withdrawnValue = nfts[_tokenId - 1].maxBid;
    users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId - 1].maxBid)
    nfts[_tokenId - 1].maxBid = 0;
    nfts[_tokenId - 1].maxBidder = address(0x0);

    emit nftTransaction(
        _tokenId,
        "Bid Withdrawn",
        msg.sender,
        address(0x0),
        withdrawnValue
    );
}
```

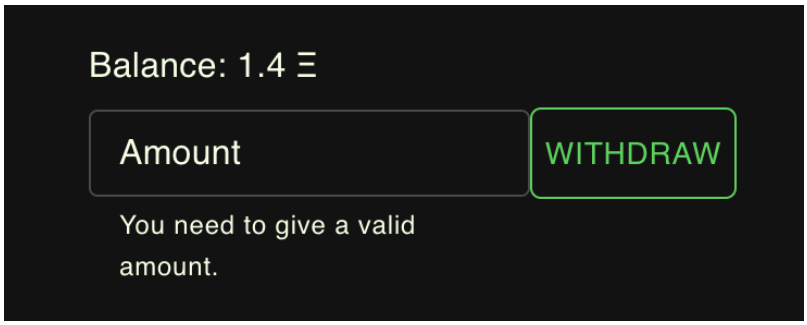
(b)

To withdraw bid from biddable item, the user who withdraws should be the current maximum bidder. After user withdraws his/her bid, current maximum bidder is set to address 0 and maximum bid set to 0.

2.11 Withdraw Money



(a)



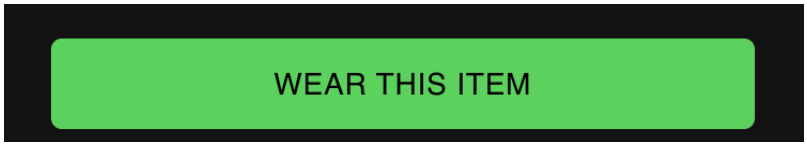
(b)

```
function withdrawMoney(uint256 _amount) public {

    require(users[msg.sender].userBalance >= _amount, "You do not have enough balance to withdraw this amount");

    uint initialBalance = users[msg.sender].userBalance;
    users[msg.sender].userBalance = sub256(initialBalance, _amount);
    msg.sender.transfer(_amount);

}
```

(c)

The user can withdraw his/her balance from the profile page. Contract checks if user have more balance than he/she wants to withdraw. If requirement satisfied, then user's balance is set to 0, then requested amount of money is sent to the user.

2.12 Wear Item



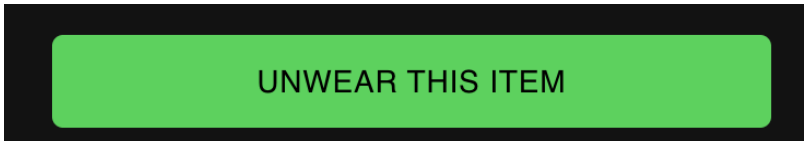
(a)

```
function wearItem(uint256 _tokenId) public {
    require(this.ownerOf(_tokenId)        == msg.sender, "You are not the owner of this item, so you cannot wear it.");
    require(nfts[_tokenId - 1].isOnSale   == false,      "You cannot wear an item while it is on sale.");
    require(nfts[_tokenId - 1].isBiddable == false,      "You cannot wear an item while it is on auction.");


    if (nfts[_tokenId - 1].clothType == 1) {
        if (users[msg.sender].head != 0)
        {
            nfts[users[msg.sender].head - 1].isWearing = false;
        }
        users[msg.sender].head = _tokenId;
    } else if (nfts[_tokenId - 1].clothType == 2) {
        if (users[msg.sender].middle != 0)
        {
            nfts[users[msg.sender].middle - 1].isWearing = false;
        }
        users[msg.sender].middle = _tokenId;
    } else if (nfts[_tokenId - 1].clothType == 3) {
        if (users[msg.sender].bottom != 0)
        {
            nfts[users[msg.sender].bottom - 1].isWearing = false;
        }
        users[msg.sender].bottom = _tokenId;
    }
     nfts[_tokenId - 1].isWearing = true;
}
```

(b)

To wear an item, the item should be owned, not on sale and not on auction. If these requirements are met, type of the item is determined, if the user is currently wearing an item, contract removes it first and wears requested one.

2.13 Unwear Item



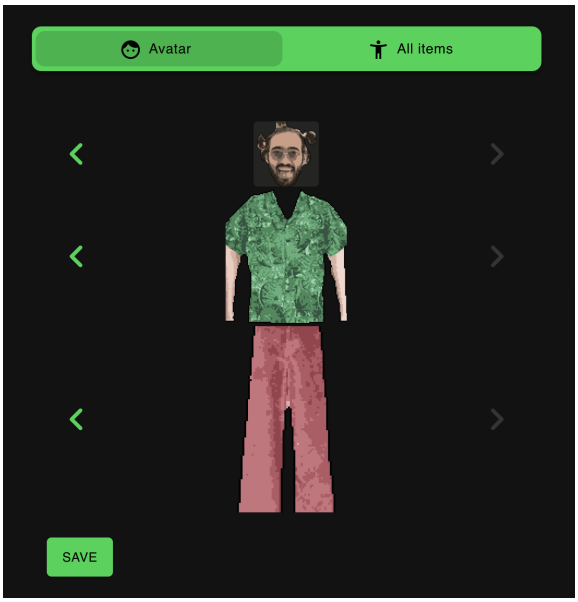
(a)

```
function unWearItem(uint256 _clothType) public {
    require(_clothType == 1 || _clothType == 2 || _clothType == 3, "Invalid cloth type.");

    if (_clothType == 1) {
        require(users[msg.sender].head !=0, "You must wear a head item first to unwear.");
        nfts[users[msg.sender].head - 1].isWearing = false;
        users[msg.sender].head = 0;
    } else if (_clothType == 2) {
        require(users[msg.sender].middle !=0, "You must wear a middle item first to unwear.");

        nfts[users[msg.sender].middle - 1].isWearing = false;
        users[msg.sender].middle = 0;
    } else if (_clothType == 3) {
        require(users[msg.sender].bottom !=0, "You must wear a bottom item first to unwear.");
        nfts[users[msg.sender].bottom - 1].isWearing = false;
        users[msg.sender].bottom = 0;
    }
}
```

(b)

To wear an item, the item should be already worn by the user. The type of the item should be valid. After these checks, item is simply unworn from the user.

2.14 Wear Items



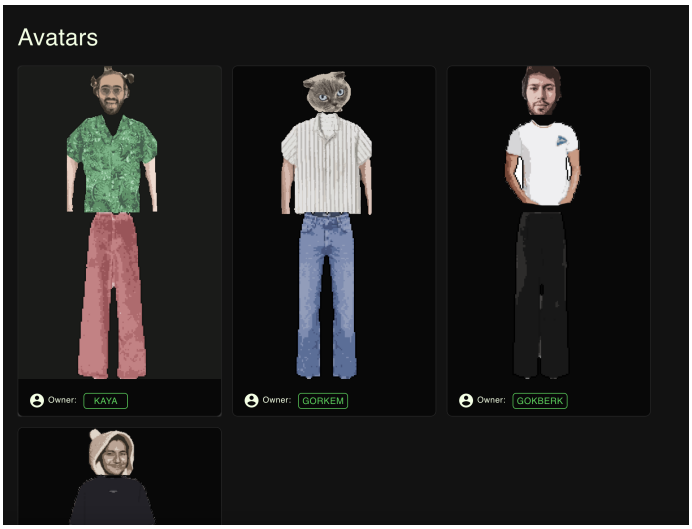
(a)

```
function wearItems(
    uint256 _headTokenId,
    uint256 _middleTokenId,
    uint256 _bottomTokenId
) public {
    require(
        _headTokenId == 0 ||
            (this.ownerOf(_headTokenId) == msg.sender &&
                nfts[_headTokenId - 1].clothType == 1)
    ,"you must be the owner or you tried not head item on head");
    require(
        _middleTokenId == 0 ||
            (this.ownerOf(_middleTokenId) == msg.sender &&
                nfts[_middleTokenId - 1].clothType == 2)
    ,"you must be the owner or you tried not middle item on middle");
    require(
        _bottomTokenId == 0 ||
            (this.ownerOf(_bottomTokenId) == msg.sender &&
                nfts[_bottomTokenId - 1].clothType == 3)
    ,"you must be the owner or you tried not bottom item on bottom");

    require(_headTokenId   == 0 || nfts[_headTokenId   - 1].isOnSale   == false, "head on sale, you cannot wear it!");
    require(_headTokenId   == 0 || nfts[_headTokenId   - 1].isBiddable == false, "head on bid, you cannot wear it!");
    require(_middleTokenId == 0 || nfts[_middleTokenId - 1].isOnSale   == false, "middle on sale, you cannot wear it!");
    require(_middleTokenId == 0 || nfts[_middleTokenId - 1].isBiddable == false, "middle on bid, you cannot wear it!");
    require(_bottomTokenId == 0 || nfts[_bottomTokenId - 1].isOnSale   == false, "bottom on sale, you cannot wear it!");
    require(_bottomTokenId == 0 || nfts[_bottomTokenId - 1].isBiddable == false, "middle on bid, you cannot wear it!");

    if (users[msg.sender].head != 0) {
        nfts[users[msg.sender].head - 1].isWearing = false;
    }
    if (users[msg.sender].middle != 0) {
        nfts[users[msg.sender].middle - 1].isWearing = false;
    }
    if (users[msg.sender].bottom != 0) {
        nfts[users[msg.sender].bottom - 1].isWearing = false;
    }


    if (_headTokenId != 0) {
        nfts[_headTokenId - 1].isWearing = true;
    }
    if (_middleTokenId != 0) {
        nfts[_middleTokenId - 1].isWearing = true;
    }
    if (_bottomTokenId != 0) {
        nfts[_bottomTokenId - 1].isWearing = true;
    }

    users[msg.sender].head   = _headTokenId;
    users[msg.sender].middle = _middleTokenId;
    users[msg.sender].bottom = _bottomTokenId;
}
```

(b)

All the checks are the same as "Wear Item" and "Unwear Item". However, user can wear and also unwear multiple items in the profile page.

2.15 Display Avatars

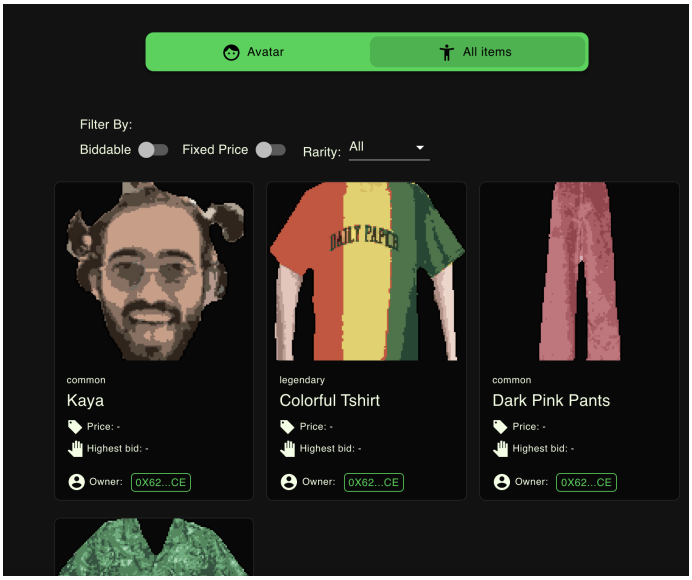


(a)

Figure 15: (a) Avatars page to display the avatars of the users

In Avatars page, the avatars of the users who wear 3 complementary pieces -head, top, and middle- is displayed. If a piece is missing, i.e. naked in this context, the avatar is not displayed.

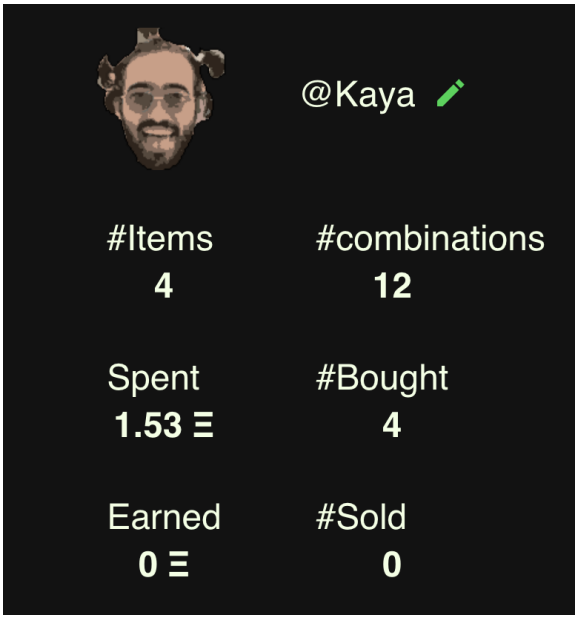
2.16 List her/his items with certain filtering options



(a)

In the profile page, the user can display all the items, only heads, tops or middles individually. For each tab, user can filter the ones that are on sale at a fixed price, have an auction and the rarity levels of the items can also be used to filter the items.

2.17 See the statistics

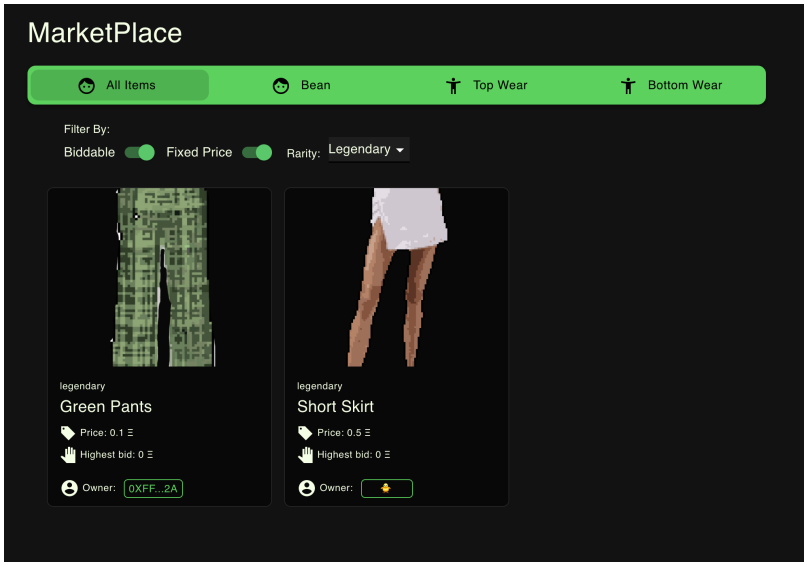


(a)

Figure 17: (a) Statistics of the user displayed in the profile

In the statistics section, the number of items that this owner owns, the number of items that are sold and bought by the user and the money that is spent and earned are displayed, as well as the number of possible combinations that can be created with the items of the user, including the naked option.

2.18 List all items with certain filtering options

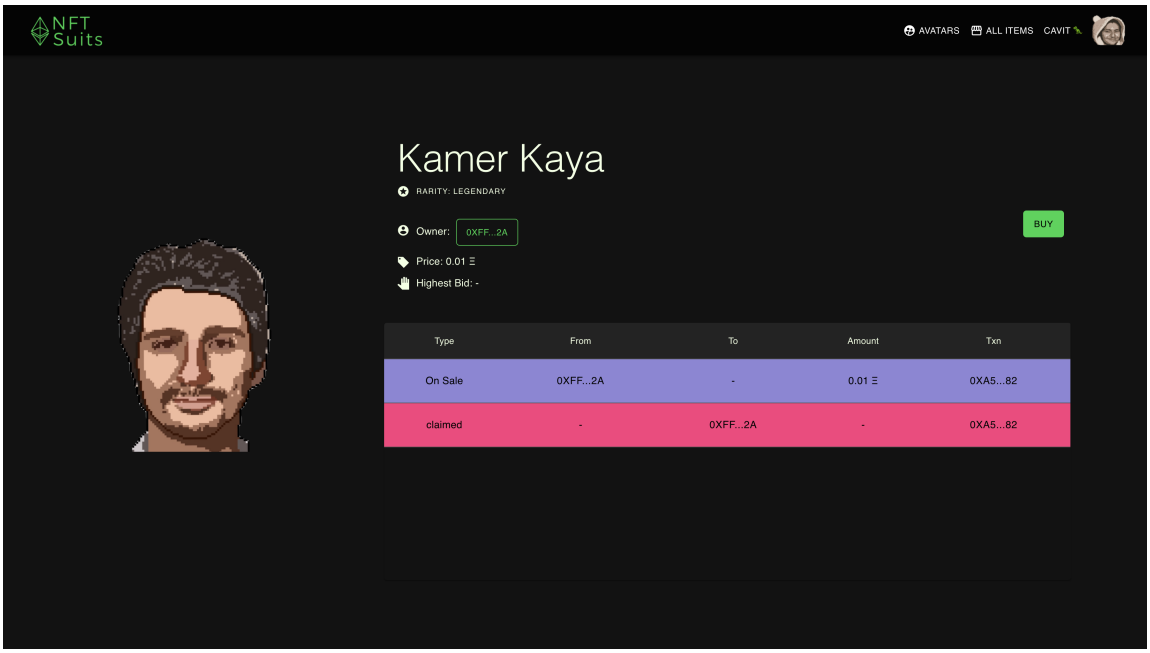


(a)

Figure 18: (a) Filtering options in the marketplace

In the marketplace, the user can display all the items, only heads, tops or middles individually. For each tab, user can filter the ones that are on sale at a fixed price, have an auction and the rarity levels of the items can also be used to filter the items.

2.19 Sample Item Page



(a)

In the item page, user can buy, bid, sell a item, and open auction, and see the transaction history of specific item. All the addresses in the table are clickable. If address refers to a user, the user is redirected to the his/her profile page, if the address is transaction txn then user is redirected to the etherscan.

3 Security Analysis

We consider security analysis in 5 different sections. In the following sections one can find testing results of the contract, precaution taken for overflow/underflow and re-entrance attacks, visualization and the static analysis of the contract.

3.1 Testing And Unauthorized Access

Contracts that are deployed to the chain cannot changed. So, testing the contract is becoming more important then testing a normal back-end server. We developed scenarios and tested those different scenarios. One of the most important case is the authorization of the item, the ownership. There are many cases that user should not handle features of the items that are not himself/herself. We have 76 test cases that covers a lot of different problematic cases. During the contract development process, we realized our mistakes with the help of our test cases and when we changed a part from contract we easily run our test over the changed contract and be sure that everything is fine. The outputs of the test cases can be seen from Figure 20 to Figure 23.

```
  Contract: NFT
before called parent
    DEPLOYMENT
      ✓ deploys succesfully
      ✓ deployer is valid (57ms)
    MINT
      ✓ deployer try mint (1079ms)
      ✓ not deployer cannot mint (784ms)
      ✓ deployer cannot mint the same link (92ms)
    OWNER FEATURES
      ✓ owner cannot putOnSale if it is sale (95ms)
      ✓ not owner cannot cancelSale (94ms)
      ✓ owner can cancelSale (259ms)
      ✓ owner can wear item 1 (259ms)
      ✓ owner cannot wear another item 2 if it is on sale (83ms)
      ✓ owner can wear another item 2 if it is not on sale (570ms)
      ✓ not owner cannot wear (103ms)
```

Figure 20: Testing Output - 1

```
    ✓ owner can unwear (164ms)
    ✓ owner cannot unwear not wear item  (51ms)
    ✓ someone cannot wear not existing token (163ms)
    ✓ owner cannot cancelSale if not on sale (91ms)
    ✓ not owner cannot putOnSale (92ms)
    ✓ someone cannot sell if item does not exist (163ms)
    ✓ owner cannot putOnSale if wearing the item (263ms)
    ✓ owner can putOnSale while item is not wearing (241ms)
    ✓ owner cannot buy his own item (116ms)
    ✓ someone cannot buy if item does not exist (143ms)
    ✓ buyer cannot buy if item not on sale (75ms)
    ✓ buyer cannot buy if sending value is less then sellPrice (114ms)
    ✓ buyer can buy owner's item (285ms)
    ✓ owner cannot withdrawMoney if withdraw amount is higher then owner has (90ms)
    ✓ owner cannot withdrawMoney if owner trys to trick us with negative values
    ✓ owner can withdrawMoney (101ms)
    ✓ owner cannot cancelAuction if not biddable (139ms)
    ✓ someone cannot cancelAuction if item not exists (94ms)
    ✓ owner cannot putOnAuction if wearing (289ms)
    ✓ someone cannot putOnAuction random one (95ms)
```

Figure 21: Testing Output - 2

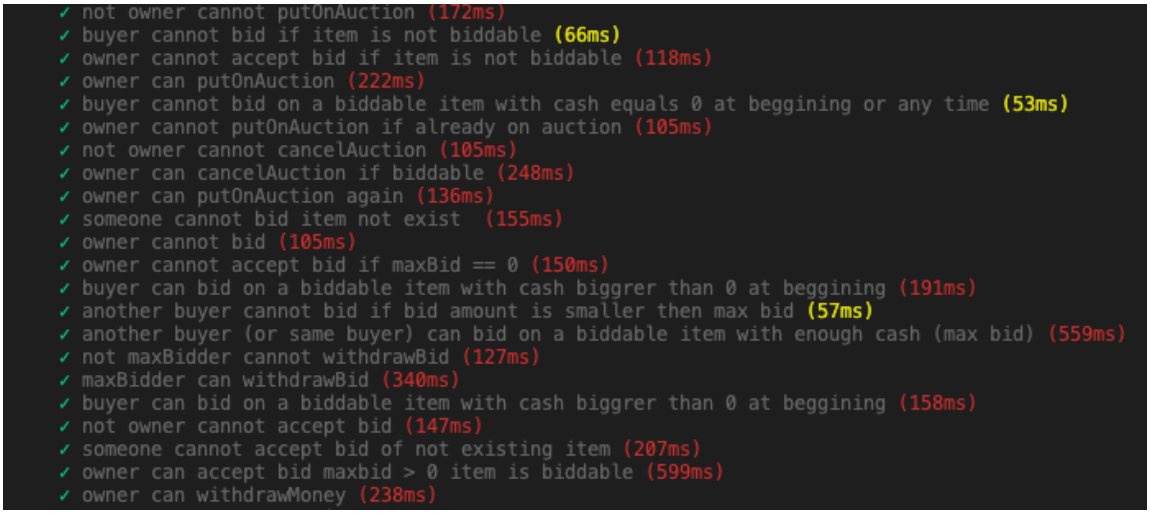


Figure 22: Testing Output - 3

Figure 23: Testing Output - 4

From Figure 24 to Figure 28 some sample test cases are showed, the rest can be found in appendix (Appendix test cases). At Figure 24 there is a sample case that we check the authorization. Address

buyer is not the owner of the tokenId_1 so it should be rejected.

```
it("not owner cannot wear", async () => {
    await nftContract.wearItem(tokenId_1, {from: buyer}).should.be.rejected;
});
```

Figure 24: Sample Testing Code - 1

```
it("owner can unwear", async () => {
    await nftContract.unWearItem(1, {from: deployer});
    const userData = await nftContract.users.call(deployer);
    assert.equal(userData.head, 0);
    const oldClothData = await nftContract.nfts.call(nftId_2);
    assert.equal(oldClothData.isWearing, false);
});
```

Figure 25: Sample Testing Code - 2

```
it("owner cannot unwear not wear item ", async () => {
    await nftContract.unWearItem(1, {from: deployer}).should.be.rejected;
});
```

Figure 26: Sample Testing Code -3

Another sample is Figure 27, it shows, testing done for owner's attempt to wear his/her item. Since at that point owner is not wearing any item and item is not previously worn thus, required checks are limited. (Checking item is set as wearing and owner head should be the item tried to be wear) Figure 28, is an item wearing which involve un-wearing the previously worn item, so more testing is required on item trying to be wear, item trying to unwear and user itself.

```
    it("owner can wear item 1", async () => {
        await nftContract.wearItem(tokenId_1, {from: deployer,});
        const userData = await nftContract.users.call(deployer);
        assert.equal(userData.head, tokenId_1);
        const newClothData = await nftContract.nfts.call(nftId_1);
        assert.equal(newClothData.isWearing, true);

    });
```

Figure 27: Sample Testing Code - 4

```
it("owner can wear another item 2 if it is not on sale", async () => {
    await nftContract.cancelSale(tokenId_2, {from: deployer});
    await nftContract.wearItem(tokenId_2, {from: deployer,});
    const userData = await nftContract.users.call(deployer);
    assert.equal(userData.head, tokenId_2);
    const oldClothData = await nftContract.nfts.call(nftId_1);
    assert.equal(oldClothData.isWearing, false);
    const newClothData = await nftContract.nfts.call(nftId_2);
    assert.equal(newClothData.isWearing, true);
});
```

Figure 28: Sample Testing Code - 5

3.2 Overflow/Underflow Attacks

In Ethereum Blockchain, integers/unsigned integers have boundaries and they overflow/underflow when those boundaries are exceeded. For example if you subtract 5 from a unsigned integer with value 3 then it will underflow. Those cases will cause problems like losing the contracts money. To prevent those cases we add256 and sub256 functions that can be seen at Figure 29. We check if the addition or subtraction finished as expected, otherwise we revert the transaction with the appropriate messages. One can see some sample places where we use those functions in our contract in Figure 30 and 31. We used those functions in all the necessary places during the contract.

```
function add256(uint256 a, uint256 b) internal pure returns (uint) {
    uint c = a + b;
    require(c >= a, "addition overflow");
    return c;
}

function sub256(uint256 a, uint256 b) internal pure returns (uint) {
    require(b <= a, "subtraction underflow");
    return a - b;
}
```

Figure 29: Overflow Attack Protection

```
users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId - 1].maxBid);
```

Figure 30: Overflow Attack Usage - 1

```
users[sellerAddress].userBalance = add256(users[sellerAddress].userBalance, msg.value);
```

Figure 31: Overflow Attack Usage - 2

3.3 Re-entrance Attacks

Re-entrance attacks are one of the common pitfalls in smart contract development. Adversery take advantage of external calls which send money to his wallet. Since money transfers call the fallback function of the receiver if receivers call the same function again and again recursively. It will spent all available gas in the victim contract. To resolve, this problem state variables should be set before making external calls and/or transfer function should be used. We applied both techniques in our implementation below Figure 33 is an example for the usage. Also, transfer of NFTs from previous owner to new owner requires similar protection mechanism. For this purpose, we used inherited class safeTransfer function which is implemented for just this problem, below Figure 32 is an example.

```
this.safeTransferFrom(sellerAddress, msg.sender, _tokenId);
```

Figure 32: Protection from Re-Entrance by Inherited Contract

```
function withdrawMoney(uint256 _amount) public {

    require(users[msg.sender].userBalance >= _amount, "You do not have enough balance to withdraw this amount");

    uint initialBalance = users[msg.sender].userBalance;
    users[msg.sender].userBalance = sub256(initialBalance, _amount);
    msg.sender.transfer(_amount);

}
```

Figure 33: Protection from Re-Entrance by Using transfer and Setting State First

3.4 Visualization of the Contract

We used Sūrya tool to visualize the contracts function call's graph to see the dependencies. One can see the Figure 34 for details.For example mint function uses _mint, totalSupply and putOnSale functions.

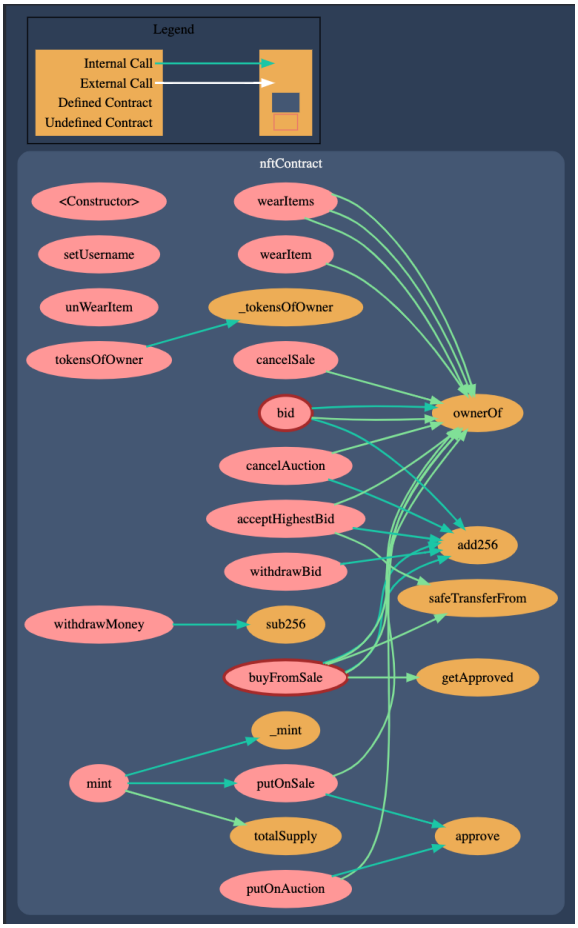


Figure 34: Visualization of the Contract as Graph

3.5 Static Analysis

Static Analysis is analysis of smart contract without running and testing extensively, common pitfalls and patterns are checked during this process. For static analysis we tried many alternatives some of them were not installing properly and some were not producing meaningful results. Finally, we decided on two static analysis tools which are Securify v2.0 and Slither.

3.5.1 Securify v2.0

"Securify[4] 2.0 is a security scanner for Ethereum smart contracts supported by the Ethereum Foundation and ChainSecurity. The core research behind Securify was conducted at the Secure, Reliable, and Intelligent Systems Lab at ETH Zurich." - Securify

In below Figure 35, static analysis show that we could use external keyword instead public, this could be done if we further work on this contract.

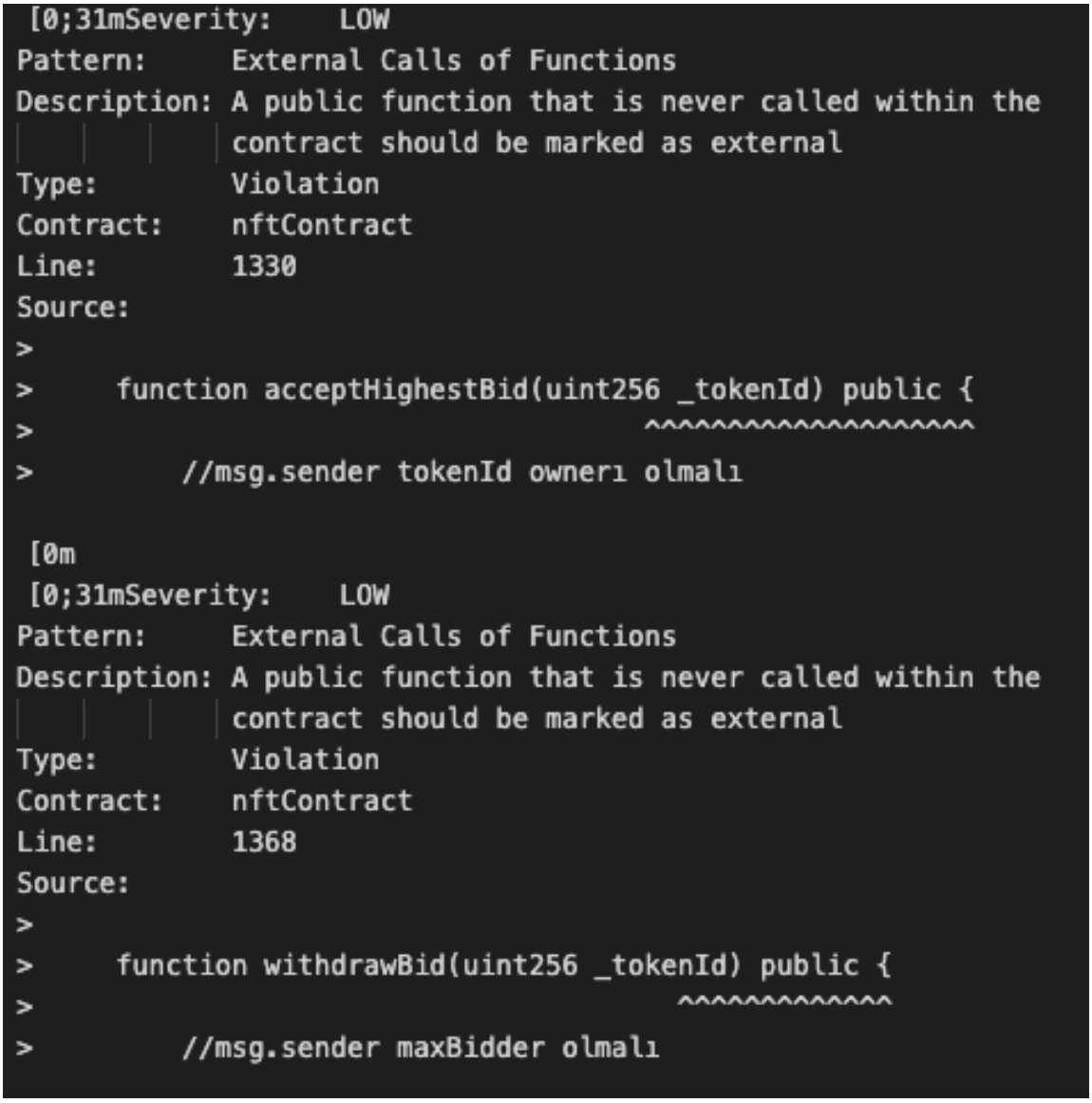


Figure 35: Static Analysis Securify - 1

In below Figure 36, static analysis show unchecked exceptions however only case where an exception could occur is an item without an owner in such scenario owner of function will return 0, and this will never be equal to msg.sender so this warning and severity do not exist.

```
 [0m
 [0;33mSeverity:    HIGH
Pattern:     Unhandled Exception
Description: The return value of statements that may return error
             values must be explicitly checked.
Type:        Warning
Contract:    nftContract
Line:        1063
Source:
>             _middleTokenId == 0 ||
>                 (this.ownerOf(_middleTokenId) == msg.sender &&
>                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>                     nfts[_middleTokenId - 1].clothType == 2)

 [0m
 [0;33mSeverity:    HIGH
Pattern:     Unhandled Exception
Description: The return value of statements that may return error
             values must be explicitly checked.
Type:        Warning
Contract:    nftContract
Line:        1068
Source:
>             _bottomTokenId == 0 ||
>                 (this.ownerOf(_bottomTokenId) == msg.sender &&
>                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>                     nfts[_bottomTokenId - 1].clothType == 3)
```

Figure 36: Static Analysis Securify - 2

In below Figure 37, static analysis warns about possible re-entrance attack however the function that is called is the one of the functions of the inherited class and it is not calling the contract one more time so this warning is also not a valid problem.

```
[0;33mSeverity:    HIGH
Pattern:     Unhandled Exception
Description: The return value of statements that may return error
             values must be explicitly checked.
Type:        Warning
Contract:    nftContract
Line:        1402
Source:
>         require(isExist[_cid] == false, "Item link should be unique, for you to mint it");
>         require(this.totalSupply() < maxSupply, "You cannot mint any more item since you already reached the maximum supply.");
>                                                 ^^^^^^^^^^^^^^^^^
>         require(msg.sender ==owner, "Only owner can of this contract can mint, you are trying to some fraud.");
```

Figure 37: Static Analysis Securify - 3

3.5.2 Securify Slither

" Slither [3] is a Solidity static analysis framework written in Python 3. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses." - Slither

In below Figure 38, static analysis show warns about possible re-entrance attack due to external call before state variable update however, safeTransferFrom's implementation, is very similar to transfer, so it is not causing re-entrancy attack. This thread is also not valid.

```
ethsec@499d804af9cf:/home/trufflecon$ slither FullContract_v2.sol
INFO:Detectors:
Reentrancy in nftContract.acceptHighestBid(uint256) (FullContract_v2.sol#1330-1361):
        External calls:
        - this.safeTransferFrom(msg.sender,nfts[_tokenId - 1].maxBidder,_tokenId) (FullContract_v2.sol#1342-1346)
        State variables written after the call(s):
        - nfts[_tokenId - 1].maxBid = 0 (FullContract_v2.sol#1348)
        - nfts[_tokenId - 1].maxBidder = address(0x0) (FullContract_v2.sol#1349)
        - nfts[_tokenId - 1].isBiddable = false (FullContract_v2.sol#1350)
        - nfts[_tokenId - 1].isOnSale = false (FullContract_v2.sol#1351)
        - nfts[_tokenId - 1].sellPrice = 0 (FullContract_v2.sol#1352)
Reentrancy in nftContract.buyFromSale(uint256) (FullContract_v2.sol#1227-1257):
        External calls:
        - this.safeTransferFrom(sellerAddress,msg.sender,_tokenId) (FullContract_v2.sol#1237)
        State variables written after the call(s):
        - nfts[_tokenId - 1].isOnSale = false (FullContract_v2.sol#1238)
        - nfts[_tokenId - 1].sellPrice = 0 (FullContract_v2.sol#1239)
        - nfts[_tokenId - 1].maxBid = 0 (FullContract_v2.sol#1246)
        - nfts[_tokenId - 1].maxBidder = address(0x0) (FullContract_v2.sol#1247)
        - nfts[_tokenId - 1].isBiddable = false (FullContract_v2.sol#1248)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

Figure 38: Static Analysis Slither - 1

4 Gas Analysis

Gas amounts are important since Ethereum price and transaction fees are so high nowadays so we analyzed our contract's gas amounts for each function.

Functionalities	Gas Cost
Set Username	29874
Mint	404000
Put On Sale	81517
Cancel Sale	20088
Buy Item	91989
Put On Auction	61455
Accept Hightest Bid	25088
Bid on an item	66343
Withdraw Bid	30751
Withdraw Money	18912
Wear Item	78555
Unwear Item	17105
Wear Items	108235

Table 1: Each function and their respected required gas amount.

In above Table 1, one can observe some of the functions requires a lot of gas. In production, this could dis-attract people so we need to try to re-write same functionalities with less checks to reduce our gas consumption. Also, we should simplify our contract.

5 Extra Cool Stuff: IPFS

IPFS is a distributed system for storing and accessing images. Images are stored in a network of nodes all over the world. In pinata[6], each image has exact cid which refers to that image. Specific cid is cannot be changed, cannot be deleted which is very suitable for storing nft images. We used Pinata as IPFS service.

6 Conclusion

We implemented discussed features in the proposal of the project. We managed to validate authorization functions with writing testing code. We tried to eliminate re-entrancy attacks by following best practise.Furthurmore, conducted static analysis on our code to eliminate common pitfalls and used well-tested libraries like Open Zeppelin [5]. For the overflow attacks, we used custom functions with overflow/underflow detection. Also, we managed to produce a gui for the user to interact with our contract.

7 Future Work

Next stage of this project could be deployment to mainnet. In order to achieve this, gas requirements should be decreased. For example, some public functions could be re-written as external which is suggested by static analysis. Also, even more extensive testing is required before publishing the product. Also, since minting all NFTs would require a lot of money, a randomization/lottery like

event could be added to project. This project will take the cost of minting from the creator to clients of the tool(minter of NFT).

8 References

[1] *CryptoKitties*. https://www.cryptokitties.co/. URL: `https://www.cryptokitties.co/`.

[2] *CryptoPunks*. www.larvalabs.com. URL: `https://www.larvalabs.com/cryptopunks`.

[3] *crytic/slither*. GitHub, June 2021. URL: `https://github.com/crytic/slither` (visited on 06/02/2021).

[4] *eth-sri/securify2*. GitHub, May 2021. URL: `https://github.com/eth-sri/securify2` (visited on 06/02/2021).

[5] *OpenZeppelin*. OpenZeppelin. URL: `https://openzeppelin.com/` (visited on 06/02/2021).

[6] *Pinata*. pinata.cloud. URL: `https://pinata.cloud/` (visited on 06/02/2021).

9 Appendix

9.1 NFT Contract

```
pragma solidity 0.5.0;
//import "https://github.com/OpenZeppelin/openzeppelin-contracts/tree/docs-org/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721Full.sol";

contract nftContract is ERC721Full {
    struct nftData {
        uint256 clothType; //--> 1 --> head, 2 --> middle, 3 --> bottom
        string name;
        string cid;
        string rarity;
        bool isOnSale;
        uint256 sellPrice;
        bool isBiddable;
        uint256 maxBid;
        address maxBidder;
        bool isWearing;
    }

    struct userData {
        uint256 head;
        uint256 middle;
        uint256 bottom;
        string username;
        uint256 userBalance;
    }

    event nftTransaction(
        uint256 indexed id,
        string transactionType,
        address fromAddress,
        address toAddress,
        uint256 value
    );

    mapping(address => userData) public users;
    mapping(string => bool) isExist;
    nftData[] public nfts;

    address  public owner;
    uint public maxSupply;

    constructor() public ERC721Full("nftContract", "NFTC") {
        owner = msg.sender;
        maxSupply=100;
    }

    function setUsername(string memory _username) public {
        users[msg.sender].username = _username;
    }

    function wearItems(
        uint256 _headTokenId,
        uint256 _middleTokenId,
        uint256 _bottomTokenId
    ) public {
        require(
            _headTokenId == 0 ||
                (this.ownerOf(_headTokenId) == msg.sender &&
                    nfts[_headTokenId - 1].clothType == 1)
        ,"you must be the owner or you tried not head item on head");
        require(
            _middleTokenId == 0 ||
                (this.ownerOf(_middleTokenId) == msg.sender &&
                    nfts[_middleTokenId - 1].clothType == 2)
        ,"you must be the owner or you tried not middle item on middle");
        require(
            _bottomTokenId == 0 ||
                (this.ownerOf(_bottomTokenId) == msg.sender &&
                    nfts[_bottomTokenId - 1].clothType == 3)
        ,"you must be the owner or you tried not bottom item on bottom");

        require(_headTokenId   == 0 || nfts[_headTokenId   - 1].isOnSale   == false, "head on sale, you cannot wear it!");
        require(_headTokenId   == 0 || nfts[_headTokenId   - 1].isBiddable == false, "head on bid, you cannot wear it!");
        require(_middleTokenId == 0 || nfts[_middleTokenId - 1].isOnSale   == false, "middle on sale, you cannot wear it!");
        require(_middleTokenId == 0 || nfts[_middleTokenId - 1].isBiddable == false, "middle on bid, you cannot wear it!");
        require(_bottomTokenId == 0 || nfts[_bottomTokenId - 1].isOnSale   == false, "bottom on sale, you cannot wear it!");
        require(_bottomTokenId == 0 || nfts[_bottomTokenId - 1].isBiddable == false, "middle on bid, you cannot wear it!");

        if (users[msg.sender].head != 0) {
            nfts[users[msg.sender].head - 1].isWearing = false;
        }
        if (users[msg.sender].middle != 0) {
            nfts[users[msg.sender].middle - 1].isWearing = false;
        }
        if (users[msg.sender].bottom != 0) {
            nfts[users[msg.sender].bottom - 1].isWearing = false;
        }

        if (_headTokenId != 0) {
            nfts[_headTokenId - 1].isWearing = true;
        }
        if (_middleTokenId != 0) {
            nfts[_middleTokenId - 1].isWearing = true;
        }
        if (_bottomTokenId != 0) {
            nfts[_bottomTokenId - 1].isWearing = true;
        }
```

```
        users[msg.sender].head   = _headTokenId;
        users[msg.sender].middle = _middleTokenId;
        users[msg.sender].bottom = _bottomTokenId;
    }

    function wearItem(uint256 _tokenId) public {
        require(this.ownerOf(_tokenId)        == msg.sender, "You are not the owner of this item, so you cannot wear it.");
        require(nfts[_tokenId - 1].isOnSale   == false,      "You cannot wear an item while it is on sale.");
        require(nfts[_tokenId - 1].isBiddable == false,      "You cannot wear an item while it is on auction.");

        if (nfts[_tokenId - 1].clothType == 1) {
            if (users[msg.sender].head != 0)
            {
                nfts[users[msg.sender].head - 1].isWearing = false;
            }
            users[msg.sender].head = _tokenId;
        } else if (nfts[_tokenId - 1].clothType == 2) {
            if (users[msg.sender].middle != 0)
            {
                nfts[users[msg.sender].middle - 1].isWearing = false;
            }
            users[msg.sender].middle = _tokenId;
        } else if (nfts[_tokenId - 1].clothType == 3) {
            if (users[msg.sender].bottom != 0)
            {
                nfts[users[msg.sender].bottom - 1].isWearing = false;
            }
            users[msg.sender].bottom = _tokenId;
        }
         nfts[_tokenId - 1].isWearing = true;
    }

    function unWearItem(uint256 _clothType) public {
        require(_clothType == 1 || _clothType == 2 || _clothType == 3, "Invalid cloth type.");

        if (_clothType == 1) {
            require(users[msg.sender].head !=0, "You must wear a head item first to unwear.");
            nfts[users[msg.sender].head - 1].isWearing = false;
            users[msg.sender].head = 0;
        } else if (_clothType == 2) {
            require(users[msg.sender].middle !=0, "You must wear a middle item first to unwear.");

            nfts[users[msg.sender].middle - 1].isWearing = false;
            users[msg.sender].middle = 0;
        } else if (_clothType == 3) {
            require(users[msg.sender].bottom !=0, "You must wear a bottom item first to unwear.");
            nfts[users[msg.sender].bottom - 1].isWearing = false;
            users[msg.sender].bottom = 0;
        }
    }

    function tokensOfOwner(address _owner)
        public
        view
        returns (uint256[] memory)
    {
        return _tokensOfOwner(_owner);
    }

    function putOnSale(uint256 _tokenId, uint256 _sellPrice) public {

        require(msg.sender == this.ownerOf(_tokenId), "You cannot put this item on sale, because you are not the owner of it.");
        require(nfts[_tokenId - 1].isOnSale == false, "Item is already on sale!");
        require(nfts[_tokenId - 1].isWearing == false, "You must unwear it first, then you can sell it." );

        nfts[_tokenId - 1].isOnSale = true;
        nfts[_tokenId - 1].sellPrice = _sellPrice;
        approve(address(this), _tokenId);

        emit nftTransaction(
            _tokenId,
            "On Sale",
            msg.sender,
            address(0x0),
            _sellPrice
        );
    }

    function cancelSale(uint256 _tokenId) public {
        require(nfts[_tokenId - 1].isOnSale == true, "Item should be on sale first, to be cancelled.");
        require(msg.sender == this.ownerOf(_tokenId), "You cannot cancel the sale of this item, because you are not the owner.");

        nfts[_tokenId - 1].isOnSale = false;
        nfts[_tokenId - 1].sellPrice = 0;

        emit nftTransaction(
            _tokenId,
            "Sale Cancelled",
            msg.sender,
            address(0x0),
            0
        );
    }

    function buyFromSale(uint256 _tokenId) public payable {

        require(msg.sender != this.ownerOf(_tokenId), "You cannot buy your own item!");

        require(nfts[_tokenId - 1].isOnSale == true, "Item should be on sale for you to buy it.");

        require (nfts[_tokenId - 1].sellPrice <= msg.value, "The amount you tried to buy, is less than price.");

        require(this.getApproved(_tokenId) == address(this), "Seller did not give the allowance for us to sell this item, contact with
            seller.");
        address sellerAddress = this.ownerOf(_tokenId);
        this.safeTransferFrom(sellerAddress, msg.sender, _tokenId);
        nfts[_tokenId - 1].isOnSale = false;
```

```
        nfts[_tokenId - 1].sellPrice = 0;
        users[sellerAddress].userBalance = add256(users[sellerAddress].userBalance, msg.value);

        if (nfts[_tokenId - 1].maxBid > 0) {
            users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance ,  nfts[_tokenId -
                1].maxBid);
        }
        nfts[_tokenId - 1].maxBid = 0;
        nfts[_tokenId - 1].maxBidder = address(0x0);
        nfts[_tokenId - 1].isBiddable = false;

        emit nftTransaction(
            _tokenId,
            "sold",
            sellerAddress,
            msg.sender,
            msg.value
        );
    }

    function putOnAuction(uint256 _tokenId) public {

        require(msg.sender == this.ownerOf(_tokenId), "Only owner of this item can put on sale" );
        require(nfts[_tokenId - 1].isWearing == false,  "You must unwear it first, then you can put it on auction.");
        require(nfts[_tokenId - 1].isBiddable == false, "This item is already on auction!");

        nfts[_tokenId - 1].isBiddable = true;
        nfts[_tokenId - 1].maxBid = 0;
        approve(address(this), _tokenId);

        emit nftTransaction(
            _tokenId,
            "Auction Starts",
            msg.sender,
            address(0x0),
            0
        );
    }

    function cancelAuction(uint256 _tokenId) public {
        require(msg.sender == this.ownerOf(_tokenId), "You cannot cancel the auction of this item, because you are not the owner.");
        require(nfts[_tokenId - 1].isBiddable == true, "Item must be on auction before it can be canceled, currently it is not!");

        if (nfts[_tokenId - 1].maxBid > 0) {
            users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId -
                1].maxBid);
        }
        nfts[_tokenId - 1].isBiddable = false;
        nfts[_tokenId - 1].maxBid = 0;
        nfts[_tokenId - 1].maxBidder = address(0x0);

        emit nftTransaction(
            _tokenId,
            "Auction Cancelled",
            msg.sender,
            address(0x0),
            0
        );
    }

    function bid(uint256 _tokenId) public payable {

        require(msg.value > 0, "You did not send any money");
        require(nfts[_tokenId - 1].isBiddable == true,"Item you tried to bid, is not biddable!");
        require(msg.value >= nfts[_tokenId - 1].maxBid, "The amount you tried to bid, is less than current max bid.");
        require(msg.sender != this.ownerOf(_tokenId), "You cannot bid your own item.");

        if (nfts[_tokenId - 1].maxBid > 0) {
            users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId -
                1].maxBid);
        }
        nfts[_tokenId - 1].maxBid = msg.value;
        nfts[_tokenId - 1].maxBidder = msg.sender;

        emit nftTransaction(
            _tokenId,
            "Bidded",
            msg.sender,
            ownerOf(_tokenId),
            msg.value
        );
    }

    function acceptHighestBid(uint256 _tokenId) public {

        require(msg.sender == this.ownerOf(_tokenId), "You need to be owner of this item, to accept its highest bid.");
        require(nfts[_tokenId - 1].isBiddable == true , "Item should be biddable for you to accept its highest bid.");
        require(nfts[_tokenId - 1].maxBid > 0 ,"Max bid must be more than 0 to accept it, currently it is not!");
        require(nfts[_tokenId - 1].maxBidder != msg.sender, "Max bidder cannot be the same person as seller!");

        address buyer = nfts[_tokenId - 1].maxBidder;
        uint256 soldValue = nfts[_tokenId - 1].maxBid;
        this.safeTransferFrom(
            msg.sender,
            nfts[_tokenId - 1].maxBidder,
            _tokenId
        );
        users[msg.sender].userBalance = add256(users[msg.sender].userBalance, nfts[_tokenId - 1].maxBid);
        nfts[_tokenId - 1].maxBid = 0;
        nfts[_tokenId - 1].maxBidder = address(0x0);
        nfts[_tokenId - 1].isBiddable = false;
        nfts[_tokenId - 1].isOnSale = false;
        nfts[_tokenId - 1].sellPrice = 0;

        emit nftTransaction(
            _tokenId,
            "Sold From Auction",
            msg.sender,
```

```
            buyer,
            soldValue
        );
    }

    function withdrawBid(uint256 _tokenId) public {

        require(msg.sender == nfts[_tokenId - 1].maxBidder, "You must be the max bidder to withdraw your bid!");
        uint256 withdrawnValue = nfts[_tokenId - 1].maxBid;
        users[nfts[_tokenId - 1].maxBidder].userBalance = add256(users[nfts[_tokenId - 1].maxBidder].userBalance, nfts[_tokenId -
            1].maxBid);
        nfts[_tokenId - 1].maxBid = 0;
        nfts[_tokenId - 1].maxBidder = address(0x0);

        emit nftTransaction(
            _tokenId,
            "Bid Withdrawn",
            msg.sender,
            address(0x0),
            withdrawnValue
        );
    }

    function withdrawMoney(uint256 _amount) public {

        require(users[msg.sender].userBalance >= _amount, "You do not have enough balance to withdraw this amount");

        uint initialBalance = users[msg.sender].userBalance;
        users[msg.sender].userBalance = sub256(initialBalance, _amount);
        msg.sender.transfer(_amount);

    }

    function mint(
        uint256 _clothType,
        string memory _name,
        string memory _cid,
        string memory _rarity
    ) public {
        require(isExist[_cid] == false, "Item link should be unique, for you to mint it");
        require(this.totalSupply() < maxSupply, "You cannot mint any more item since you already reached the maximum supply.");
        require(msg.sender ==owner, "Only owner can of this contract can mint, you are trying to some fraud.");
        require(_clothType == 1 || _clothType == 2 || _clothType == 3, "Invalid cloth type.");
        uint256 _id =
            nfts.push(
                nftData(
                    _clothType,
                    _name,
                    _cid,
                    _rarity,
                    false,
                    0,
                    false,
                    0,
                    address(0x0),
                    false
                )
            );
        _mint(msg.sender, _id);
        isExist[_cid] = true;

        emit nftTransaction(_id, "claimed", address(0x0), msg.sender, 0);

        putOnSale(_id, 10000000000000000);
    }


    function add256(uint256 a, uint256 b) internal pure returns (uint) {
        uint c = a + b;
        require(c >= a, "addition overflow");
        return c;
    }

    function sub256(uint256 a, uint256 b) internal pure returns (uint) {
        require(b <= a, "subtraction underflow");
        return a - b;
    }
}
```

9.2 NFT Contract Test Code

```
const { assert } = require('chai');

const NFT = artifacts.require("./nftContract");

require('chai')
    .use(require('chai-as-promised'))
    .should()


contract("NFT", ([deployer, buyer, seller]) => {
    let nftContract

    before(async() => {
        nftContract = await NFT.deployed();
        console.log("before called parent");

    })

    describe("DEPLOYMENT", async () => {
```

```
    it("deploys succesfully", async () => {
        const address = await nftContract.address;
        assert.notEqual(address,"");
        assert.notEqual(address,null);
        assert.notEqual(address,undefined);
        assert.notEqual(address,0x0);
    })
    it("deployer is valid", async() =>{
        const address = await nftContract.owner.call();
        assert.equal(address,deployer);
    })
})

describe("MINT", async () => {
    it("deployer try mint", async () => {
        const totalSupplyBefore = await nftContract.totalSupply.call();
        const maxSupply = await nftContract.maxSupply.call();

        if(totalSupplyBefore>=maxSupply){
            await nftContract.mint(1,"name","link","rarity",{from:buyer}).should.be.rejected;
        }
        else{
            await nftContract.mint(1,"name","link","rarity",{from:deployer}).should.be.fulfilled;
            const totalSupplyAfter = await nftContract.totalSupply.call();
            assert.equal(totalSupplyBefore.toNumber()+1,totalSupplyAfter);
            const newMinted = await nftContract.nfts.call(totalSupplyAfter.toNumber() - 1);
            assert.equal(newMinted.clothType,1);
            assert.equal(newMinted.name,"name");
            assert.equal(newMinted.cid,"link");
            assert.equal(newMinted.rarity,"rarity");
            assert.equal(newMinted.isOnSale,true);
            assert.equal(newMinted.sellPrice,10000000000000000);
            assert.equal(newMinted.isBiddable,false);
            assert.equal(newMinted.maxBid,0);
            assert.equal(newMinted.maxBidder,0x0);
            assert.equal(newMinted.isWearing,false);
            const newMintedOwner = await nftContract.ownerOf.call(totalSupplyAfter);
            assert.equal(newMintedOwner, deployer);
        }
    })

    it("not deployer cannot mint", async () => {
        await nftContract.mint(0,"name","link","rarity",{from: buyer,}).should.be.rejected;
    })
    it("deployer cannot mint the same link", async () => {
         await nftContract.mint(0,"name","link","rarity",{from: deployer}).should.be.rejected;
    })

})
describe("OWNER FEATURES", async () => {
    let tokenId_1;
    let nftId_1;
    let tokenId_2;
    let nftId_2;

    before(async() => {
        await nftContract.mint(1,"for owner features","special owner features 1","rarity",{from:deployer});

        const totalSupply_1 = await nftContract.totalSupply.call();

        nftId_1 = totalSupply_1.toNumber() - 1;
        tokenId_1 = totalSupply_1.toNumber();

        await nftContract.mint(1,"for owner features", "special owner features 2","rarity",{from:deployer});

        const totalSupply_2 = await nftContract.totalSupply.call();

        nftId_2 = totalSupply_2.toNumber() - 1;
        tokenId_2 = totalSupply_2.toNumber();
    })
    // initial state owner has tokenId and it is on sale
    ////////////////////////////////////////////////////
    //          PUT ON SALE // CANCEL SALE STARTS
    ////////////////////////////////////////////////////


    it("owner cannot putOnSale if it is sale", async () => {
        await nftContract.putOnSale(tokenId_1, 48001, {from: deployer}).should.be.rejected;

    });
    it("not owner cannot cancelSale", async () => {
        await nftContract.cancelSale(tokenId_1, {from: buyer}).should.be.rejected;
    });
    it("owner can cancelSale", async () => {
        await nftContract.cancelSale(tokenId_1, {from: deployer});

        const nftData = await nftContract.nfts.call(nftId_1);
        assert.equal(nftData.isOnSale, false);
        assert.equal(nftData.sellPrice, 0);
    });

    ////////////////////////////////////////////////////
    //                    WEAR STARTS
```

```
//                  State: deployer has item1 and item2
//                   item 1 and item 2 are on sale
//                    owner does not wear and items are not being weared.
//
/////////////////////////////////////////////////////

it("owner can wear item 1", async () => {
    await nftContract.wearItem(tokenId_1, {from: deployer,});
    const userData = await nftContract.users.call(deployer);
    assert.equal(userData.head, tokenId_1);
    const newClothData = await nftContract.nfts.call(nftId_1);
    assert.equal(newClothData.isWearing, true);

});

it("owner cannot wear another item 2 if it is on sale", async () => {
    await nftContract.wearItem(tokenId_2, {from: deployer,}).should.be.rejected;

});

it("owner can wear another item 2 if it is not on sale", async () => {
    await nftContract.cancelSale(tokenId_2, {from: deployer});
    await nftContract.wearItem(tokenId_2, {from: deployer,});
    const userData = await nftContract.users.call(deployer);
    assert.equal(userData.head, tokenId_2);
    const oldClothData = await nftContract.nfts.call(nftId_1);
    assert.equal(oldClothData.isWearing, false);
    const newClothData = await nftContract.nfts.call(nftId_2);
    assert.equal(newClothData.isWearing, true);
});

it("not owner cannot wear", async () => {
    await nftContract.wearItem(tokenId_1, {from: buyer}).should.be.rejected;
});

it("owner can unwear", async () => {
    await nftContract.unWearItem(1, {from: deployer});
    const userData = await nftContract.users.call(deployer);
    assert.equal(userData.head, 0);
    const oldClothData = await nftContract.nfts.call(nftId_2);
    assert.equal(oldClothData.isWearing, false);
});

it("owner cannot unwear not wear item ", async () => {
    await nftContract.unWearItem(1, {from: deployer}).should.be.rejected;
});

it("someone cannot wear not existing token", async () => {
    const totalSupply = await nftContract.totalSupply.call();
    await nftContract.wearItem(totalSupply.toNumber() + 1, {from: deployer}).should.be.rejected;
});


/////////////////////////////////////////////////////
//                   WEAR ENDS
/////////////////////////////////////////////////////

it("owner cannot cancelSale if not on sale", async () => {
    await nftContract.cancelSale(tokenId_1 ,{from: deployer}).should.be.rejected;
});

it("not owner cannot putOnSale", async () => {
    await nftContract.putOnSale(tokenId_1, 48001, {from: buyer}).should.be.rejected;
});

it("someone cannot sell if item does not exist", async() =>{
    const totalSupply = await nftContract.totalSupply.call();
    await nftContract.putOnSale(totalSupply.toNumber() + 1, 48001, {from: deployer}).should.be.rejected;
});

it("owner cannot putOnSale if wearing the item", async () => {
    await nftContract.wearItem(tokenId_1, {from: deployer}).should.be.fulfilled;
    await nftContract.putOnSale(tokenId_1, 48001, {from: deployer}).should.be.rejected;
    await nftContract.unWearItem(1, {from: deployer});
});

it("owner can putOnSale while item is not wearing", async () => {
    await nftContract.putOnSale(tokenId_1, 48001, {from: deployer});
    const nftData = await nftContract.nfts.call(nftId_1);
    assert.equal(nftData.isOnSale, true);
    assert.equal(nftData.sellPrice, 48001);
    const approvedAddress = await nftContract.getApproved(tokenId_1);
    assert.equal(approvedAddress, nftContract.address);
});


//
//          deploy have item1 and item2 both Items are not work
//          item1 and item2 are not worn

/////////////////////////////////////////////////////
//            PUT ON SALE  // CANCEL SALE ENDS
/////////////////////////////////////////////////////

/////////////////////////////////////////////////////
//                   BUYS ITEM STARTS
/////////////////////////////////////////////////////

it("owner cannot buy his own item", async() =>{
    await nftContract.buyFromSale(tokenId_1, {from: deployer, value: 48001}).should.be.rejected;
});

it("someone cannot buy if item does not exist", async() =>{
    const totalSupply = await nftContract.totalSupply.call();
    await nftContract.buyFromSale(totalSupply.toNumber() + 1, {from: deployer, value: 48001}).should.be.rejected;
});
```

```
it("buyer cannot buy if item not on sale", async() =>{
    //tokenId_2 is not on sale
    await nftContract.buyFromSale(tokenId_2, {from: buyer, value: 48001}).should.be.rejected;
});

it("buyer cannot buy if sending value is less then sellPrice", async() =>{
    await nftContract.buyFromSale(tokenId_1, {from: buyer, value: 310}).should.be.rejected;
});

it("buyer can buy owner's item", async() =>{
    await nftContract.buyFromSale(tokenId_1, {from: buyer, value: 48001});

    const clothData = await nftContract.nfts.call(nftId_1);
    const newOwnerAddress = await nftContract.ownerOf.call(tokenId_1);
    assert.equal(newOwnerAddress, buyer);
    assert.equal(clothData.isOnSale, false);
    assert.equal(clothData.sellPrice, 0);
    assert.equal(clothData.isBiddable, false);
    assert.equal(clothData.maxBid, 0);
    assert.equal(clothData.maxBidder, 0x0);
});

////////////////////////////////////////////////////
//                  BUYS ITEM ENDS
////////////////////////////////////////////////////

////////////////////////////////////////////////////
//                  WITHDRAW MONEY STARTS
////////////////////////////////////////////////////

it("owner cannot withdrawMoney if withdraw amount is higher then owner has", async () => {
    await nftContract.withdrawMoney(40600000, {from: deployer}).should.be.rejected;
});

it("owner cannot withdrawMoney if owner trys to trick us with negative values", async () => {
    await nftContract.withdrawMoney(-2, {from: deployer}).should.be.rejected;
});

it("owner can withdrawMoney", async () => {
    const startingUserData = await nftContract.users.call(deployer);
    await nftContract.withdrawMoney(48001, {from: deployer});
    const endingUserData = await nftContract.users.call(deployer);
    assert.equal(startingUserData.userBalance.toNumber() - 48001, endingUserData.userBalance.toNumber());
});

////////////////////////////////////////////////////
//                  WITHDRAW MONEY ENDS
////////////////////////////////////////////////////

// state buyer has item1  deployer has item2
// no one wears anything


////////////////////////////////////////////////////
//                  AUCTION STARTS
////////////////////////////////////////////////////

it("owner cannot cancelAuction if not biddable", async () => {
    await nftContract.cancelAuction(tokenId_2, {from: deployer}).should.be.rejected;
})

it("someone cannot cancelAuction if item not exists", async () => {
    const totalSupply = await nftContract.totalSupply.call();
    await nftContract.cancelAuction(totalSupply.toNumber() + 1, {from: deployer}).should.be.rejected;
})

it("owner cannot putOnAuction if wearing", async () => {
    //console.log("aaa");
    //const nftData = await nftContract.nfts.call(tokenId_2);
    //console.log("nftData", nftData.isBiddable);
    await nftContract.wearItem(tokenId_2, {from: deployer});
    await nftContract.putOnAuction(tokenId_2, {from: deployer}).should.be.rejected;
    await nftContract.unWearItem(1, {from: deployer});
})

it("someone cannot putOnAuction random one", async () => {
    const totalSupply = await nftContract.totalSupply.call();
    await nftContract.putOnAuction(totalSupply.toNumber() + 1, {from: deployer}).should.be.rejected;
})

it("not owner cannot putOnAuction", async () => {
    await nftContract.putOnAuction(tokenId_2, {from: buyer}).should.be.rejected;
})

it("buyer cannot bid if item is not biddable", async () => {
    await nftContract.bid(tokenId_2, {from: buyer, value: 48001}).should.be.rejected;

})

it("owner cannot accept bid if item is not biddable", async () => {
    await nftContract.acceptHighestBid(tokenId_2, {from: deployer}).should.be.rejected;
})

it("owner can putOnAuction", async () => {
    await nftContract.putOnAuction(tokenId_2, {from: deployer});
    const clothData = await nftContract.nfts.call(nftId_2);

    assert.equal(clothData.isBiddable, true);
    assert.equal(clothData.maxBid, 0);
    assert.equal(clothData.maxBidder, 0x0);
    assert.equal(clothData.isWearing, false);

    const approvedAddress = await nftContract.getApproved(tokenId_2);
    assert.equal(approvedAddress, nftContract.address);
})
```

```
    it("buyer cannot bid on a biddable item with cash equals 0 at beggining or any time", async () => {
        await nftContract.bid(tokenId_2, {from: buyer, value: 0}).should.be.rejected;
    })

    it("owner cannot putOnAuction if already on auction", async () => {
        await nftContract.putOnAuction(tokenId_2, {from: deployer}).should.be.rejected;
    })

    it("not owner cannot cancelAuction", async () => {
        await nftContract.cancelAuction(tokenId_2, {from: buyer}).should.be.rejected;
    })

    it("owner can cancelAuction if biddable", async () => {
        const clothDataBefore = await nftContract.nfts.call(nftId_2);
        const maxBidder = clothDataBefore.maxBidder;
        const maxBid = clothDataBefore.maxBid;
        const maxBidderDataBefore = await nftContract.users.call(maxBidder);
        const maxBidderBalanceBefore = maxBidderDataBefore.userBalance;

        await nftContract.cancelAuction(tokenId_2, {from: deployer});

        const maxBidderDataAfter= await nftContract.users.call(maxBidder);
        const maxBidderBalanceAfter = maxBidderDataAfter.userBalance;

        const clothData = await nftContract.nfts.call(nftId_2);

        assert.equal(maxBid.toNumber() + maxBidderBalanceBefore.toNumber() ,maxBidderBalanceAfter.toNumber());
        assert.equal(clothData.isBiddable, false);
        assert.equal(clothData.maxBid, 0);
        assert.equal(clothData.maxBidder, 0x0);
        assert.equal(clothData.isWearing, false);
    })

    //burda bidibblea çevrilmeli AGAIN
    it("owner can putOnAuction again", async () => {
        await nftContract.putOnAuction(tokenId_2, {from: deployer});

        const clothData = await nftContract.nfts.call(nftId_2);
        assert.equal(clothData.isBiddable, true);
        assert.equal(clothData.maxBid, 0);
        assert.equal(clothData.maxBidder, 0x0);
        assert.equal(clothData.isWearing, false);

        const approvedAddress = await nftContract.getApproved(tokenId_2);
        assert.equal(approvedAddress, nftContract.address);
    })

    it("someone cannot bid item not exist ", async () => {
        const totalSupply = await nftContract.totalSupply.call();
        await nftContract.bid(totalSupply.toNumber() + 1, {from: deployer, value: 48001}).should.be.rejected;
    })

    it("owner cannot bid", async () => {
        await nftContract.bid(tokenId_2, {from: deployer, value: 48001}).should.be.rejected;
    })

    it("owner cannot accept bid if maxBid == 0", async () => {
        await nftContract.acceptHighestBid(tokenId_2, {from: deployer}).should.be.rejected;
    })

    it("buyer can bid on a biddable item with cash biggrer than 0 at beggining", async () => {
        await nftContract.bid(tokenId_2, {from: buyer, value: 406});
        const clothDataBefore = await nftContract.nfts.call(nftId_2);

        const clothData = await nftContract.nfts.call(nftId_2);
        assert.equal(clothData.isBiddable, true);
        assert.equal(clothData.maxBid, 406);
        assert.equal(clothData.maxBidder, buyer);
        assert.equal(clothData.isWearing, false);
    })

    it("another buyer cannot bid if bid amount is smaller then max bid", async () => {
        await nftContract.bid(tokenId_2, {from: seller, value: 306}).should.be.rejected;
    })

    it("another buyer (or same buyer) can bid on a biddable item with enough cash (max bid)", async () => {
        const clothDataBefore = await nftContract.nfts.call(nftId_2);
        const maxBidderBefore = clothDataBefore.maxBidder;
        const maxBidBefore = clothDataBefore.maxBid;
        const maxBidderDataBefore = await nftContract.users.call(maxBidderBefore);
        const maxBidderBalanceBefore = maxBidderDataBefore.userBalance;

        await nftContract.bid(tokenId_2, {from: seller, value: 48001});

        const clothDataAfter = await nftContract.nfts.call(nftId_2);

        const previousMaxBidderDataAfter = await nftContract.users.call(maxBidderBefore);
        const previousMaxBidderBalanceAfter = previousMaxBidderDataAfter.userBalance;

        assert.equal(maxBidBefore.toNumber() + maxBidderBalanceBefore.toNumber(), previousMaxBidderBalanceAfter.toNumber());

        assert.equal(clothDataBefore.isBiddable, true);
        assert.equal(clothDataBefore.maxBid, 406);
        assert.equal(clothDataBefore.maxBidder, buyer);
        assert.equal(clothDataBefore.isWearing, false);

        assert.equal(clothDataAfter.isBiddable, true);
        assert.equal(clothDataAfter.maxBid, 48001);
        assert.equal(clothDataAfter.maxBidder, seller);
        assert.equal(clothDataAfter.isWearing, false);
    })

    it("not maxBidder cannot withdrawBid", async () => {
        //current maxBidder is seller so trying withdraw with buyer
        await nftContract.withdrawBid(tokenId_2, {from: buyer,}).should.be.rejected;
    })

    it("maxBidder can withdrawBid", async () => {
        const clothDataBefore = await nftContract.nfts.call(nftId_2);
        const maxBidderBefore = clothDataBefore.maxBidder;
        const maxBidBefore = clothDataBefore.maxBid;
```

```
        const maxBidderDataBefore = await nftContract.users.call(maxBidderBefore);
        const maxBidderBalanceBefore = maxBidderDataBefore.userBalance;

        await nftContract.withdrawBid(tokenId_2, {from: seller});

        const previousMaxBidderDataAfter = await nftContract.users.call(maxBidderBefore);
        const previousMaxBidderBalanceAfter = previousMaxBidderDataAfter.userBalance;

        const clothData = await nftContract.nfts.call(nftId_2);

        assert.equal(maxBidBefore.toNumber() + maxBidderBalanceBefore.toNumber(), previousMaxBidderBalanceAfter.toNumber());

        assert.equal(clothData.isBiddable, true);
        assert.equal(clothData.maxBid, 0);
        assert.equal(clothData.maxBidder, 0x0);
        assert.equal(clothData.isWearing, false);
    })

    it("buyer can bid on a biddable item with cash biggrer than 0 at beggining", async () => {
        await nftContract.bid(tokenId_2, {from: buyer, value: 204});
        const clothDataBefore = await nftContract.nfts.call(nftId_2);

        const clothData = await nftContract.nfts.call(nftId_2);
        assert.equal(clothData.isBiddable, true);
        assert.equal(clothData.maxBid, 204);
        assert.equal(clothData.maxBidder, buyer);
        assert.equal(clothData.isWearing, false);
        assert.equal(clothData.isBiddable, true);

    })

    it("not owner cannot accept bid", async () => {
        await nftContract.acceptHighestBid(tokenId_2, {from: buyer}).should.be.rejected;
    })

    it("someone cannot accept bid of not existing item", async () => {
        const totalSupply = await nftContract.totalSupply.call();
        await nftContract.acceptHighestBid(totalSupply.toNumber() + 1, {from: deployer}).should.be.rejected;
    })

    it("owner can accept bid maxbid > 0 item is biddable", async () => {

        const ownerBefore = await nftContract.ownerOf.call(tokenId_2);
        const ownerDataBefore = await nftContract.users.call(ownerBefore);

        const clothDataBefore = await nftContract.nfts.call(nftId_2);
        const buyer = clothDataBefore.maxBidder;
        const price = clothDataBefore.maxBid;


        await nftContract.acceptHighestBid(tokenId_2, {from: deployer});
        const previous_ownerDataAfter = await nftContract.users.call(ownerBefore);

        // previous owner in parası arttı mı
        assert.equal(ownerDataBefore.userBalance.toNumber() + price.toNumber(), previous_ownerDataAfter.userBalance.toNumber());

        // item default durumda mı

        const clothData = await nftContract.nfts.call(nftId_2);
        assert.equal(clothData.isBiddable, false);
        assert.equal(clothData.isOnSale, false);
        assert.equal(clothData.maxBid, 0);
        assert.equal(clothData.sellPrice, 0);
        assert.equal(clothData.maxBidder, 0x0);
        assert.equal(clothData.isWearing, false);
        // alıcı sahibi mi

        const ownerAfter = await nftContract.ownerOf.call(tokenId_2);
        assert.equal(buyer,ownerAfter);
    })

    it("owner can withdrawMoney", async () => {
        const startingUserData = await nftContract.users.call(deployer);
        await nftContract.withdrawMoney(204, {from: deployer});
        const endingUserData = await nftContract.users.call(deployer);
        assert.equal(startingUserData.userBalance.toNumber() - 204, endingUserData.userBalance.toNumber());
    })

    ////////////////////////////////////////////////////////
    //                    AUCTION ENDS
    ////////////////////////////////////////////////////////
})

describe("WEAR/UNWEAR MULTIPLE ITEMS", async () => {
    let  tokenId_1;
    let  tokenId_2;
    let  tokenId_3;
    let  nftId_1;
    let  nftId_2;
    let  nftId_3;
    let  tokenId_4;
    let  tokenId_5;
    let  tokenId_6;
    let  nftId_4;
    let  nftId_5;
    let  nftId_6;

    before(async() => {
        await nftContract.mint(1,"for owner features", "wear 1","rarity",{from:deployer});

        const totalSupply_1 = await nftContract.totalSupply.call();

        nftId_1 = totalSupply_1.toNumber() - 1;
        tokenId_1 = totalSupply_1.toNumber();

        await nftContract.mint(2,"for owner features", "wear 2","common",{from:deployer});

        const totalSupply_2 = await nftContract.totalSupply.call();

        nftId_2 = totalSupply_2.toNumber() - 1;
```

```
        tokenId_2 = totalSupply_2.toNumber();

        await nftContract.mint(3,"for owner features", "wear 3","legendary",{from:deployer});

        const totalSupply_3 = await nftContract.totalSupply.call();

        nftId_3 = totalSupply_3.toNumber() - 1;
        tokenId_3 = totalSupply_3.toNumber();

        await nftContract.mint(1,"for owner features", "wear 4","legendary",{from:deployer});

        const totalSupply_4 = await nftContract.totalSupply.call();

        nftId_4 = totalSupply_4.toNumber() - 1;
        tokenId_4 = totalSupply_4.toNumber();

        await nftContract.mint(2,"for owner features", "wear 5","legendary",{from:deployer});

        const totalSupply_5 = await nftContract.totalSupply.call();

        nftId_5 = totalSupply_5.toNumber() - 1;
        tokenId_5 = totalSupply_5.toNumber();

        await nftContract.mint(3,"for owner features", "wear 6","legendary",{from:deployer});

        const totalSupply_6 = await nftContract.totalSupply.call();

        nftId_6 = totalSupply_6.toNumber() - 1;
        tokenId_6 = totalSupply_6.toNumber();

    })


    it("owner cannot wear head on sale", async()=>{
        await nftContract.wearItems(0, tokenId_2, 0, {from: deployer,}).should.be.rejected;
    })
    it("owner cannot wear bottom on sale ", async()=>{
        await nftContract.wearItems(0, 0, tokenId_3, {from: deployer,}).should.be.rejected;
    })
    it("owner cannot wear all if on sale", async () => {
        await nftContract.wearItems(tokenId_1, tokenId_2, tokenId_3, {from: deployer,}).should.be.rejected;
    })
    it("cancel sales to wear items", async () => {
        await nftContract.cancelSale(tokenId_1, {from: deployer});
        await nftContract.cancelSale(tokenId_2, {from: deployer});
        await nftContract.cancelSale(tokenId_3, {from: deployer});
    })
    it("put on auction to show cannot wear items", async () => {
        await nftContract.putOnAuction(tokenId_1, {from: deployer});
        await nftContract.putOnAuction(tokenId_2, {from: deployer});
        await nftContract.putOnAuction(tokenId_3, {from: deployer});
    })
    it("owner cannot wear head on auction", async()=>{
        await nftContract.wearItems(tokenId_1, 0, 0, {from: deployer,}).should.be.rejected;
    })
    it("owner cannot wear middle on auction", async()=>{
        await nftContract.wearItems(0, tokenId_2, 0, {from: deployer,}).should.be.rejected;
    })
    it("owner cannot wear bottom on auction", async()=>{
        await nftContract.wearItems(0, 0, tokenId_3, {from: deployer,}).should.be.rejected;
    })
    it("owner cannot wear all if on auction", async () => {
        await nftContract.wearItems(tokenId_1, tokenId_2, tokenId_3, {from: deployer,}).should.be.rejected;
    })
    it("cancel auction to test other cases without affected by auction", async () => {
        await nftContract.cancelAuction(tokenId_1, {from: deployer});
        await nftContract.cancelAuction(tokenId_2, {from: deployer});
        await nftContract.cancelAuction(tokenId_3, {from: deployer});
    })
    it("not owner cannot wear head", async()=>{
        await nftContract.wearItems(tokenId_1, 0, 0, {from: buyer,}).should.be.rejected;
    })
    it("not owner cannot wear middle", async()=>{
        await nftContract.wearItems(0, tokenId_2, 0, {from: buyer,}).should.be.rejected;
    })
    it("not owner cannot wear bottom", async()=>{
        await nftContract.wearItems(0, 0, tokenId_3, {from: buyer,}).should.be.rejected;
    })
    it("not owner cannot wear all", async () => {
        await nftContract.wearItems(tokenId_1, tokenId_2, tokenId_3, {from: buyer,}).should.be.rejected;
    })
    it("not head cannot be weared as head", async () => {
        await nftContract.wearItems(tokenId_2, 0, 0, {from: deployer,}).should.be.rejected;
    })
    it("not middle cannot be weared as middle", async () => {
        await nftContract.wearItems(0, tokenId_1, 0, {from: deployer,}).should.be.rejected;
    })
    it("not bottom cannot be weared  bottom", async () => {
        await nftContract.wearItems(0, 0, tokenId_1, {from: deployer,}).should.be.rejected;
    })
    it("not matching cannot wear", async () => {
        await nftContract.wearItems(tokenId_2, tokenId_3, tokenId_1, {from: deployer,}).should.be.rejected;
    })
```

```
        it("wear all", async () => {
            await nftContract.wearItems(tokenId_1, tokenId_2, tokenId_3, {from: deployer,});
            const userData = await nftContract.users.call(deployer);
            // user açısından
            const userHead = userData.head;
            const userMiddle = userData.middle;
            const userBottom = userData.bottom;
            assert.equal(userHead, tokenId_1);
            assert.equal(userMiddle, tokenId_2);
            assert.equal(userBottom, tokenId_3);
            // yeni ürünler açısından
            const new_item_1 = await nftContract.nfts.call(nftId_1);
            const new_item_2 = await nftContract.nfts.call(nftId_2);
            const new_item_3 = await nftContract.nfts.call(nftId_3);
            assert.equal(new_item_1.isWearing,true);
            assert.equal(new_item_2.isWearing,true);
            assert.equal(new_item_3.isWearing,true);
        })

        it("cancel sales to wear other items", async () => {
            await nftContract.cancelSale(tokenId_4, {from: deployer});
            await nftContract.cancelSale(tokenId_5, {from: deployer});
            await nftContract.cancelSale(tokenId_6, {from: deployer});
        })

        it("wear another all", async () => {
            await nftContract.wearItems(tokenId_4, tokenId_5, tokenId_6, {from: deployer,});
            const userData = await nftContract.users.call(deployer);
            // user açısından
            const userHead = userData.head;
            const userMiddle = userData.middle;
            const userBottom = userData.bottom;
            assert.equal(userHead, tokenId_4);
            assert.equal(userMiddle, tokenId_5);
            assert.equal(userBottom, tokenId_6);
            // eski ürünler açısından
            const old_item_1 = await nftContract.nfts.call(nftId_1);
            const old_item_2 = await nftContract.nfts.call(nftId_2);
            const old_item_3 = await nftContract.nfts.call(nftId_3);
            assert.equal(old_item_1.isWearing,false);
            assert.equal(old_item_2.isWearing,false);
            assert.equal(old_item_3.isWearing,false);
            // yeni ürünler açısından
            const new_item_1 = await nftContract.nfts.call(nftId_4);
            const new_item_2 = await nftContract.nfts.call(nftId_5);
            const new_item_3 = await nftContract.nfts.call(nftId_6);
            assert.equal(new_item_1.isWearing,true);
            assert.equal(new_item_2.isWearing,true);
            assert.equal(new_item_3.isWearing,true);
        })

        it("owner can unwear all", async () => {
            await nftContract.wearItems(0, 0, 0, {from: deployer,});

            const userData = await nftContract.users.call(deployer);
            // user açısından
            const userHead = userData.head;
            const userMiddle = userData.middle;
            const userBottom = userData.bottom;
            assert.equal(userHead, 0);
            assert.equal(userMiddle, 0);
            assert.equal(userBottom, 0);

            const new_item_1 = await nftContract.nfts.call(nftId_4);
            const new_item_2 = await nftContract.nfts.call(nftId_5);
            const new_item_3 = await nftContract.nfts.call(nftId_6);
            assert.equal(new_item_1.isWearing, false);
            assert.equal(new_item_2.isWearing, false);
            assert.equal(new_item_3.isWearing, false);
        })
    })
})
```