Nescrow

Permissionless web3 Marketplace Documentation

While0x1
CATALYST FUND 11

Contents

Forward

The following documentation is to be viewed in conjunction with the Nescrow codebase repository viewable @ https://github.com/while0x1/Nescrow-Catalyst .

It would not have been possible to build Nescrow without the help of many people, projects and resources made available through the Cardano Community. Jerry from pycardano https://pycardano.readthedocs.io/en/latest/ and Niels from Opshin deserve special mention for their support during Nescrow's development. I believe pycardano and Opshin provide the easiest pathway to developing decentralised applications (dApps) on Cardano.

Please note Nescrow was released with one of the very early revisions of Opshin, previously called Eopsin, and there have been substantial design revisions since the language's early releases. There will be new design patterns and more efficient ways to achieve similar results to those in our contract. Please see https://github.com/OpShin/opshin for the most up-to-date documentation.

This document requires a working knowledge of the underlying concepts of the Cardano Blockchain. If you have no knowledge of the mechanics of Cardano, then I recommend first doing some background reading or using some of the amazing resources available such as a course from the Cardano Foundation https://academy.cardanofoundation.org/ .

This document is also not intended to be a substitute for a programming or framework course. Some basic knowledge of computer science is required but links to programming concepts are provided throughout.

Introduction

Nescrow is a permissionless web3 marketplace running on the Cardano Blockchain. Nescrow uses three core software components to function – an on-chain validation contract written with Opshin (Python), a front-end written with NextJS (typescript), and offchain processing code written with Python which heavily leverages the brilliant pycardano python library. Nescrow was the first Cardano Mainnet application written and deployed with Opshin smart contracts. The original intent of Nescrow was to provide a secure, trust less platform to arrange NFT escrow services without relying on Discord or untrustworthy middlemen. I quickly realized with a few smart-contract adjustments Nescrow could be a full-blown order-book decentralized exchange as-well as an escrow service. Nescrow was also the first Cardano marketplace to allow NFTs to be traded for *any* Cardano native asset or ADA.

Validator Security Considerations

Double Satisfaction

'Double Satisfaction' is a common security vulnerability that needs to be considered when formulating smart contract logic. See https://plutus.readthedocs.io/en/latest/reference/writing-scripts/common-weaknesses/double-satisfaction.html .

This exploit allows a threat actor to satisfy the conditions of the validator for one contract locked UTxO but spend *more* than one UTxO after passing the initial validation. The first iteration of the

Nescrow contract contained logic which stipulated only 1 UTxO could be released from the contract at a time which prevented a situation where a double satisfaction exploit could be used. To allow permissionless batchers to match order-book transactions this logic was changed to allow < 3 UTxO's to be released from the contract in one invocation of the validator. Without any other contract checks this opened the possibility for a double-satisfaction exploit. If two identical sell orders were locked in the contract an attacker could satisfy the first order but spend two order UTxO's. To mitigate the attack introduced by the batcher logic, all orders received unique datums. The datums for the UTxO's locked into the contract were updated to introduce a field that contained a unique field generated from the python secrets library by calling secrets.token_hex(28) thereby making all locked UTxO's unique and preventing a double spend.

Example of initial contract double spend prevention logic:

```
def check_single_utxo_spent(txins: List[TxInInfo], addr: Address) -> None:
    """To prevent double spending, count how many UTxOs are unlocked from the contract address"""
    count = 0
    res = False
    for txi in txins:
        if txi.resolved.address == addr:
            count += 1
    if count == 1:
        res = True
    assert res, "Only 1 contract utxo allowed"
```

Figure 1 Original Double Satisfaction Logic

We can use the transactions context to resolve the smart contracts address and determine what UTxO's are being spent from within the contract. The pictured check_single_utxo_spent function iterates over the inputs of the transaction and checks that only 1 UTxO from the contract is spent. If more than one UTxO from the contract address was included in the transaction inputs the validator would fail and prevent the user from unlocking any funds.

Mangled Address Attacks

If your smart contract or offchain code only checks the stake address, or only the payment credential of a user then under certain circumstances you can be vulnerable to a mangled address attack – see https://medium.com/adamant-security/multi-sig-concerns-mangled-addresses-and-the-dangers-of-using-stake-keys-in-your-cardano-project-94894319b1d8 .

Users could assume staking control of your funds by mangling their staking key with another user's payment credential, or they could mangle someone else's staking key with a payment credential they control to withdraw funds – if the validator or off-chain code does not prevent these actions by checking *both* the staking key and the payment credential of the user. Under certain circumstances a user may have no staking key (see enterprise addresses) in which case you must contain validator provisions to allow addresses without staking keys without opening an attack vector.

Validator Logic

The Order Datum

Cardano contracts require any UTxO locked in a contract to have a datum attached. Any UTxO sent to a Cardano smart contract without a datum will be locked permanently and cannot be released from the contract. A datum is just data attached to the UTxO that a validator contract can use to verify conditions for releasing that UTxO. Datums and redeemers that are declared in your Opshin contract code can be easily imported into pycardano to simplify your off-chain code - see the off-chain documentation for examples of importing the Listing Datum and redeemers in the off-chain code.

```python
@dataclass()
class Listing(PlutusData):
    # the NFT which can unlock the escrowed NFT
    unlock_policy: bytes
    unlock_name: bytes
    amount: int
    # whoever is allowed to withdraw the listing
    owner: bytes
    stake_cred: bytes
    # marketplace fees
    fee: int
    cancel_fee: int
    fee_hash: bytes
    # unique order
    rnd: bytes
```

Figure 2 Listing Datum

When an order is placed by a user through the front-end the user will specify a Native asset (token or NFT) or $ADA that they are willing to trade for the $ADA/Native assets(s) they lock into the contract. This native asset and its magnitude are referenced in the datum as the unlock_policy, unlock_name and amount. $ADA is identified by an empty policy id and asset name. We don't need to record the asset the user offers as this will be locked into the contract and easily identified when we query all UTxO's locked in the contract. We list the owners publish key hash credential in the datum and their staking credential (if they have one) – this allows the contract to ensure the owner gets the tokens they requested or to allow only this user to cancel the order. The fee can be set dynamically by the offchain code, so in future fees can be increased or decreased, the fee_hash is an encoding of the address the fees must be paid to. The last piece of the datum is rnd - this refers to a random hex character sequence used to ensure each order datum is unique.

Order Redeemers

Redeemers are used to separate actions within a smart contract when you try to spend or 'redeem' a UTxO in the contract. For Nescrow there are only 2 actions, Buy or Cancel. A redeemer is not needed when loading a UTxO into the contract, but a datum is required for that UTxO to be spendable(redeemed). The validator logic for cancelling an order is different from buying an existing order – when we cancel an order, we only need to check the user has the rights (the correct public key hash) and that any fees are paid. The redeemer is used to set which validation logic the contract invokes.

Buy Redeemer

```python
@dataclass()
class Buy(PlutusData):
    # Redeemer to buy the listed values
    CONSTR_ID = 0
```

Figure 3 Buy Redeemer

Unlist Redeemer

```
@dataclass()
class Unlist(PlutusData):
    # Redeemer to unlist the values
    CONSTR_ID = 1
```

Figure 4 Unlist Redeemer

Functions

correct_buyer

The correct_buyer function iterates over the transaction outputs to ensure that when a user attempts to claim an order locked into the contract that the conditions specified in the claimed orders datum are honoured. The user who locked the order being claimed must receive the amount of the token they requested in return, specifically to their address (payment and or stake credentials), and the unique datum random secret must be the same. This is the logic that allows Nescrow users to lock any token into the contract and request any token (including ADA) in return. The user must receive the correct amount of the token specified by the datum.unlock_policy and datum.unlock_name which was set when the user placed the order through the front-end. If the unlock_policy and unlock_name are empty (b"") the user is requesting payment in ADA.

```
if isinstance(redeemer, Buy):
    correct_buyer(tx_info.outputs,datum)
    has_paid(tx_info.outputs,datum.fee_hash,datum.fee)
```

Figure 5 Buy Redeemer Instance

```
def correct_buyer(txouts: List[TxOut],datum: Listing) -> None:
    pk_ok = False
    stake_ok = False
    for txo in txouts:
        if txo.value.get(datum.unlock_policy, {b"": 0}).get(datum.unlock_name, 0) == datum.amount:
            if txo.address.payment_credential.credential_hash == datum.owner:
                if txo.datum == SomeOutputDatum(datum.rnd):
                        pk_ok = True
            stake_cred = txo.address.staking_credential
            if isinstance(stake_cred, SomeStakingCredential):
                some_sc = stake_cred.staking_credential
                if isinstance(some_sc,StakingHash):
                    if some_sc.value.credential_hash == datum.stake_cred:
                        stake_ok = True
            if isinstance(stake_cred, NoStakingCredential):
                stake_ok = True
    assert pk_ok and stake_ok, "Swap assets not found or bad credentials"
```

Figure 6 correct_buyer function

has_paid

This function ensures the fee to the marketplace has been paid. We iterate over the outputs of the transaction and look for an output containing ADA. Cardano cleverly accommodates native tokens on the ledger by using a policy id and name. When the policy id and name of an output are empty – ***txo.value.get(b"", {b"":0}.get(b"",0)*** - we know this output contains only ADA. When we find an ADA only output in the transaction outputs that is equal to the fee (as declared by the datum) we then record the public key credential hash (pkh). If the fee output amount is being paid to the fee_hash

specified by the datum when locked into the contract, we can reliably say the user has paid the correct fee to the address specified in the original order.

```
if isinstance(redeemer, Buy):
    correct_buyer(tx_info.outputs,datum)
    has_paid(tx_info.outputs,datum.fee_hash,datum.fee)
```

Figure 7 Buy Fees Instance

```
def has_paid(txouts: List[TxOut], payhash: PubKeyHash, fee: int) -> None:
    res = False
    for txo in txouts:
        if txo.value.get(b"", {b"":0}).get(b"",0) == fee:
            cred = txo.address.payment_credential
            pkh = cred.credential_hash
            if pkh == payhash:
                res = True
    assert res, "Incorrect Address/Amount Used"
```

Figure 8 has_paid function

check_spent_utxos

In this function we look at the transaction context purpose to determine if we are *Spending* from the script address; if this is the case, we look at the transaction inputs to determine the script address. Knowing the script address we can then iterate over the transaction inputs to count how many UTxO's from the contract address are being spent. Every time a transaction input address matches the script address we previously determined we add to the count. For the validator to successfully execute this count must be less than 3. This allows for the front-end to formulate a transaction for a user to match 1 UTxO out of the contract, or for a batching bot to match 2 UTxO's from the contract.

```
def validator(datum: Listing, redeemer: ListingAction, context: ScriptContext) -> None:
    purpose = context.purpose
    tx_info = context.tx_info
    if isinstance(purpose, Spending):
        own_utxo = resolve_spent_utxo(tx_info.inputs, purpose)
        own_addr = own_utxo.address
    else:
        assert False, "Wrong script purpose"

    check_spent_utxos(tx_info.inputs, own_addr)
```

Figure 9 validator function

```
def check_spent_utxos(txins: List[TxInInfo], addr: Address) -> None:
    count = 0
    res = False
    for txi in txins:
        if txi.resolved.address == addr:
            count += 1
    if count < 3:
        res = True
    assert res, "Max 2 contract utxos allowed"
```

Figure 10 check_spent_utxos function

check_owner_signed

This function is called when a user wants to cancel their order. When the off-chain code provides the Unlist redeemer in the transaction this function is checked. Here we only need to check if the key of the person signing the cancel transaction is the same as the key recorded in the datum of the original order and invoke the has_paid function to ensure the cancel fee set by the original order datum is present in the outputs.

```
def check_owner_signed(signatories: List[PubKeyHash], owner: PubKeyHash) -> None:
    assert owner in signatories, "Owner did not sign transaction"
```

Figure 11 check_owner_signed function

```
elif isinstance(redeemer, Unlist):
    check_owner_signed(tx_info.signatories, datum.owner)
    has_paid(tx_info.outputs,datum.fee_hash,datum.cancel_fee)
```

Figure 12 Unlist redeemer Instance

dApp Stack

Applications often rely on a technology 'stack' to function. The stack is simply the collective term used to denote the different parts of the application. With respect to a Cardano decentralized application there are typically three distinct components:

1. A Smart Contract – sometimes referred to as an on-chain validator.
2. Offchain code – software often used to query on-chain data, proprietary databases, oracles and can be used to craft transactions to deliver to the front-end.
3. Front-end – the user interface that will connect to a user's wallet using the CIP30 Wallet connector standard.

In our case the Nescrow front-end allows us to query a user's wallet after they connect to determine the UTxO's they contain and to find important information such as public keys. The graphical user interface (GUI) is used to display all orders locked in the contract, the user's orders locked in the contract, and allows them to place, accept and cancel orders.

The front-end works in close concert with the off-chain code via an API. Functions in the offchain code are triggered by API calls from the front-end as a user traverses the site and interacts with its various components.

Ancillary services

There are various supplemental services or applications used to create a working application. To keep website traffic private and encrypted via the https protocol you require a domain-name and an SSL certificate for that domain.

Certbot

Nescrow utilises certbot to generate and manage SSL certificates for both the offchain API and the front-end. The offchain API requires a certificate to satisfy the CORS security requirements of many browsers. See https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS for more information on Cross-Origin Resource Sharing (CORS)

Nginx

Both the off-chain python code and the NextJS front-end use Nginx as a reverse proxy to manage SSL connections. Nginx listens for incoming traffic on https port 443 and can forward those requests to applications running on the server on other ports. See https://docs.nginx.com/nginx/admin-guide/security-controls/securing-http-traffic-upstream/ .

Pm2

Once you have tested and built your front-end you want a service to manage the built application. This is to ensure if the service crashes or the machine running the application restarts, that the service is restarted accordingly. Nescrow uses pm2 - an advanced process manager for production JavaScript applications. I used the following guide to learn how to setup and manage the pm2 daemon: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-node-js-application-for-production-on-ubuntu-20-04

Gunicorn

The offchain web server API software is served via Gunicorn a Python WSGI HTTP Server for UNIX see https://wsgi.readthedocs.io/en/latest/what.html . The python application itself is called through Gunicorn and Gunicorn is in turn called by a Linux service manager daemon – SystemD. Gunicorn is sufficient for Nescrow's traffic but a more performant server like FastAPI could be considered for higher traffic applications.

Off-chain Code

The offchain code is written with Python and relies on two main libraries – pycardano and flask. Pycardano is used to manage Cardano-specific transaction querying and creation while flask is used to create an API that listens for HTTP POST requests from the front-end to facilitate offer placement, acceptance, or cancellations. There are 6 routes / POST endpoints the front-end can engage with:

- flaskOffer
- flaskWitnessed
- flaskFindOffers
- flaskAcceptOffer
- flaskCancelOrder
- flaskProcessUtxos

Coin Selection

A major part of all offchain software on Cardano concerns techniques used to correctly identify UTxO's (coins) required to create successful transactions. Pycardano provides an automatic coin-selection algorithm when creating transactions using Transaction builder. Transaction builder can select the UTxO's required to satisfy the outputs of a transaction when you provide it an input address. This poses an issue for users that have multi-address wallets. For single address wallets you can simply provide the one and only address to transaction builder and the coin-selection algorithm can operate. To allow transaction builder to select the correct UTxO inputs for multi address wallets we need to provide more information to its coin-selection algorithm. For multi-address wallets a function called multi_address has been created for this purpose and it is defined in multi_address.py. This function parses over all the assets in the wallet (found and supplied to us by the front-end) and identifies all the UTxO's in the wallet required to fulfill the order, including any offered assets, fees, and collateral. The function appends all the identified UTxO's into an array called utxos_to_use. We

then use the koios chain indexer API to resolve the addresses that own these UTxO's and store them in an array to provide to transaction builder. With this array we can give transaction builder all the addresses required to successfully craft a transaction where the inputs and outputs are perfectly balanced.

flaskOffer

This endpoint is called by the front-end after a user clicks the submit escrow button in the Make Offer dialog pop-up on the UI. A form on this dialog is used to set the token being offered and to set the datum unlock_policy, unlock_name and the amount required to redeem this offer from the contract. The flaskOffer function will return a transaction which needs to be signed by the user through the UI. This response, pictured below, will be a simple JSON object containing the transaction in CBOR notion, which the front-end can parse and provide to the wallet through a CIP30 call. A callback on the UI will receive this response and trigger a sign event for the user which will open the user's wallet with this transaction then request the user to review and sign the transaction. If the user signs the transaction, the UI will then send a second API request to the flaskWitnessed endpoint which will submit the signed contract to the blockchain.

```
response = jsonify({"Cbor_to_sign": unsigned_cbor })
```

buildTx

flaskOffer uses the buildTx function which is defined in load_contract.py. Notice we import the datum class type directly from the Opshin contract simplifying datum management:

```
from v3_nescrow_dex import Listing
```

This Function is used to:

- Scale supported token decimals.
- Define the order Datum.
- Find addresses and Coin Selection.
- Calculate the minimum Lovelace needed to accompany a native asset.
- Generate the transaction CBOR.

```
datum = Listing(bytes.fromhex(bidPolicy),
            bytes.fromhex(bidNameHex),
            bidAmountReq,
            bytes.fromhex(pk),
            bytes.fromhex(sk),
            FEE,
            CANCEL_FEE,
            bytes.fromhex(FEEHASH),
            bytes.fromhex(secrets.token_hex(28)))
```

flaskWitnessed

This endpoint is called whenever a transaction has been signed by a user and is ready to be submitted to the chain. When submitting a transaction signed by a CIP30 browser wallet we need to combine the CBOR containing the transaction details with a signature approving the transaction. The signature is referred to as the Witness Set. The front-end POSTS the 'raw' transaction body CBOR and the witnessed CBOR as a JSON object which our off-chain code uses to recreate the full signed transaction in pycardano. If no errors are raised the transaction id is returned to the front-end and provided to the user as a hyperlink to view the transaction through https://cexplorer.io/ . Transaction ids are the hash of the transaction_body in hex format.

```
def flaskWitnessed():
    reqAuth = request.headers.get("Auth")
    if reqAuth == FLASKAUTH:
        try:
            reqIn = request.get_json()
            witnessed_cbor = reqIn['witness']
            tx_raw_cbor = reqIn['tx_body_cbor']
            witness = TransactionWitnessSet.from_cbor(witnessed_cbor)
            tx = Transaction.from_cbor(tx_raw_cbor)
            tx.transaction_witness_set.vkey_witnesses = witness.vkey_witnesses
            tx_id = tx.transaction_body.hash().hex()
            chain_context.submit_tx(tx.to_cbor())
            logger.info(f'Tx Submission Success! {tx_id}')
            #print(tx_id)
            response = jsonify({"txId": tx_id })
        except Exception as e:
            response = jsonify({"txId": 'Error' })
            #print(e)
            logger.error(e)
        return response
    else:
        return 'Error', 403
```

flaskFindOffers

When a user clicks either the In Escrow or See Offers buttons on the front-end this endpoint is called to query the orders sitting at the contract address. An empty array found_offers is initialised and used to store the results of the function. We call a function get_script_utxos() to retrieve the UTxO's from the contract address using a blockfrost API query. We then iterate over the results and parse the key information about the order. If the In Escrow button was pressed, we only need to display the orders owned by the user otherwise we can just append all the orders found in get_script_utxos() into the found_offers array.

```
script_utxos.append({
          'tx_hash': n['tx_hash'],
          'tx_id': n['tx_index'],
          'datum_hash':n['data_hash'] ,
          'amount': n['amount'],
          'unlock_policy': unlock_policy,
          'unlock_name': unlock_name,
          'lockedAssetAmount':lockedAssetAmount,
```

```
            'owner':owner,
            'nft_offer_policy':nft_offer_p,
            'nft_offer_hexname':nft_offer_n,
            'nft_offer_name':nft_offer_n,
            'offer_img':offerImg,
            'unlock_name_utf':unlock_name_utf,
            'lovelace':ada,
            'unlock_amount':unlock_amount,
            'stake_cred':stake_cred,
            'rnd':rnd,
            'unlock_decimals':str(unlock_decimals),
            'locked_decimals': str(locked_decimals)})
```

flaskAcceptOffer

When a user clicks the Accept button on a specific offer card of the UI the front-end sends a POST request to the flaskAcceptOffer endpoint. The endpoint uses a helper function defined in the unlock_contract.py file called buildAcceptTx. The front-end supplies all the order data required to build the transaction. buildAcceptTx imports the Listing type, and the Buy redeemer from the Opshin contract to utilise in the transaction. We recreate the datum for the order and find the UTxO from the contract we are attempting to spend. The Nescrow contract reference script is added to the builder. Notice the reference script is a UTxO recreated from REF_CBOR which is a static variable containing the UTxO housing the script in CBOR notation.

```
from v3_nescrow_dex import Buy

redeemer = Redeemer(Buy())
ref_script_utxo = UTxO.from_cbor(REF_CBOR)
builder.add_script_input(utxo_to_spend, script=ref_script_utxo, redeemer=redeemer)
```

If a multi address wallet is detected, we call the multi_address_unlock function defined in multi_address.py to identify all the addresses pycardano's transaction builder needs to successfully balance the transaction. We then add the outputs required to satisfy the validator script (contract) and finally build an unsigned transaction to return to the front-end for signing by the user.

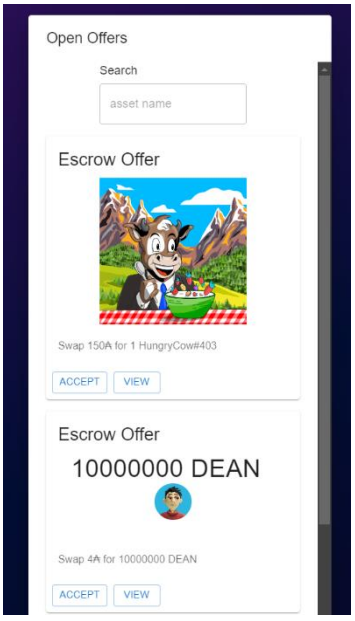

Figure 13 See Offers Dialog Cards

flaskCancelOrder

A user can peruse outstanding orders they have on the platform using the 'In Escrow' button on the UI. If a user clicks the CANCEL button the front-end will initiate a call to the flaskCancelOrder endpoint.

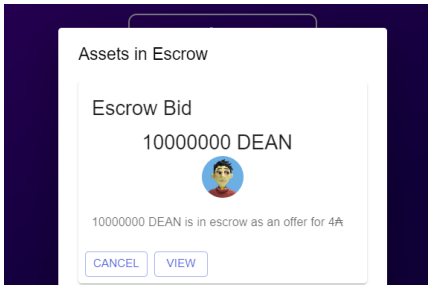

Figure 14 Cancel Order UI image.

The front-end uses a POST request with a payload identifying the order to be cancelled, and ancillary wallet information used to create the cancel transaction. The main function managing this transaction buildCancelTx is called to create the unsigned transaction CBOR, this function is defined in the cancel_contract.py file.

buildCancelTx imports the Unlist redeemer from the Opshin smart contract file and uses the data from the payload to create the order datum. The function uses the offer_utxo information passed from the front-end and iterates over the UTxO's at the contract script address to identify the correct order in the contract. If the datum hex for the UTxO at the contract matches our recreated datum, the offer_utxo transaction hash and id matches the front-end tx_hash and tx_id – we have correctly identified the UTxO to unlist.

```
for utxo in script_utxos:
            if utxo.output.datum is not None:
                if utxo.output.datum.cbor.hex() == datum.to_cbor() and
utxo.input.transaction_id.payload.hex() == offerUtxo['tx_hash'] and
utxo.input.index == offerUtxo['tx_id'] :
                    print('___ Target -- Acquired ___')
                    utxo_to_spend = utxo
                    print(f'UTXO toSPEND {utxo_to_spend}')
```

We define the Unlist redeemer and add the script_inputs to the builder along with the ref_script_utxo holding the contract. If a multi address wallet is detected, we call the multi_address_cancel function to perform coin selection and generate a list of addresses for the wallet to allow a balanced transaction to be created by transaction builder. We generate the unsigned transaction CBOR by calling builder.build_and_sign(), leaving the witness set empty so the front-end can issue the unsigned transaction to a CIP30 wallet for signing. The off-chain API then replies to the front-end post request with the unsigned CBOR transaction to initiate transaction signing.

flaskProcessUtxos

This endpoint is used to streamline processing the contents of a user's wallet. When a user first connects to the application, we query the UTxO's in a wallet and then send a POST request to the flaskProcessUtxos endpoint with a list of the UTxO's in CBOR representation. This function will output a set of UTxO's containing only ADA, the set of UTxO's containing native assets, and a set of UTxO's containing *unique* native assets. We iterate over the list of UTxO's and convert them into pycardano UTxO types by using the UTxO.from_cbor() function. getDecimals is called to determine the decimal definitions for any native assets. This function is defined in the token_registry.py file.

```
def getDecimals(policy,assetNameHex):
    decimals = 0
    for c in CURATED_TOKENS:
        decimals = 0
        found = False
        if policy == c['policy'] and c['hexname'] == assetNameHex:
            decimals = c['decimals']
            found = True
            break
```

Decimals

Decimals are another pain-point within Cardano development. Some token decimals are defined in the metadata of the token minting transaction while others are defined in the token registry, with some defined in both. There is not to my knowledge one definitive place for all decimal definitions. Nescrow only officially supports tokens which are present on the UI in our drop-down token selection menu. When tokens are supported by Nescrow they are added to the CURATED_TOKENS dictionary, and to a curated tokens object on the front-end. This Nescrow getDecimals function would benefit from an upgrade or from a Cardano Service that can provide up-to-date Decimal definitions for all tokens in one place.

https://github.com/cardano-foundation/cardano-token-registry

dApp Website

The Nescrow marketplace front-end is written in typescript with the Next.JS React framework (13.2.4). "React and Next.js allow you to create hybrid web applications where parts of your code can be rendered on the server or the client" - https://nextjs.org/docs/app/building-your-application/rendering . To simplify execution the website was designed as a single page application with dialog style popups. The code can be roughly categorized into 3 main parts: variable/type definitions, functions and handlers, HTML.

Imports

At the very beginning of the of the index.tsx file we import additional libraries to increase functionality and ease development. The project uses minimal styling and relies mainly on the Open-Source MUI library. Material UI is an open-source React component library that implements Google's Material Design https://mui.com/material-ui/getting-started/. Other notable imports are the react component and the CBOR library. The CBOR library is used to decode CBOR from the payloads of CIP30 wallet API requests.

Variables

The next section at the beginning of the index.tsx file is where we define our custom types and most of the variables that require initialisation are declared. There are 4 main type declarations which are critical for iterating over lists and for managing consistent communication with the off-chain code: mywallet, nfts, offutxo, and tokenreg. The offutxo type is a critical type definition which contains all the information required about an offer UTxO – the hash, owner, the offered policy id, and the policy id of the token which can unlock the offer etc.

```
type offutxo = {
      datum_hash: string,
      owner: string,
      tx_hash: string,
      tx_id: string ,
      unlock_name: string,
      lockedAssetAmount: string,
      unlock_policy:string,
      nft_offer_policy:string,
      nft_offer_hexname:string,
      nft_offer_name:string,
      offer_img:string,
      unlock_name_utf:string,
      lovelace:string,
      unlock_amount: string,
      stake_cred:string,
      rnd: string,
      unlock_decimals: string,
      locked_decimals:string
      }
```

The tokenReg variable and type contain the tokens which have decimal support on Nescrow. This list is also defined on the off-chain code. This allows the front-end to display, and the users to enter, tokens with scaled decimals as per the token registry.

Functions and Handlers

searchFx

Used to filter out the list of offers displayed on the see offers dialog pop-up. The function declares an array called filtList of the offutxo type. We iterate over the list of offeredUtxos and match the characters written in the searchBox with the offer_name or the unlock_name, for every match in the list of UTxO's we push that UTxO to into the filtList. We then set the state of the FilterOfferUtxos variable with are filtered UTxO offers by calling setFilterOfferUtxos. The onChange event handler of the searchBox TextField is bound to the searchValidate variable. This variable uses the setSearchBox state to store the characters written into the searchBox and calls the searchFx function passing in the character(s) entered into the searchBox. This function replaced the deprecated slideFilter Function.

```
<TextField
      id="searchBox" variant="outlined" type="text"
      placeholder='asset name' value={searchBox}
      onChange={e => searchValidate(e)}>
</TextField>
```

getIPFS

This function is used to retrieve IPFS image links so the front-end can display NFT or token images. We use blockfrost to request information about the asset passed into the function – the API should return a response which contains a field called onchain_metada.image. We slice out characters of the IPFS (Inter Planetary File System) address and then append the IPFS address onto a base Cloudflare html address 'https://cloudflare-ipfs.com/ipfs/'.

aboutDHandler

A simple function used to set the state of the AboutDialog to true. This function is called by the onClick property of the About Button. When the AboutDialog is set to true it opens the About Dialog box. The About dialog popup contains information about the application and common trouble-shooting suggestions.

makeOfferDHandler

Another simple function used to set state for the MakeOfferDialog. This function is called by the onClick property of the Make Offer button. When the MakeOfferDialog is set to true the Make Offer Dialog is opened to display the form used to create offer orders on Nescrow.

getWalletsAsync

When a user first connects to Nescrow we interrogate the browser context to find the wallet extensions installed on their browser. The getWalletsAsync constant contains the function to find the wallets available in the browser context. This function uses an array called wal of type *mywallet* to store the available wallet ids, names, apiVersion, and icon. We then update the state of the Wallets variable with the wal array by calling setWallets. The list of wallets is displayed for selection when the Connect button is pressed.

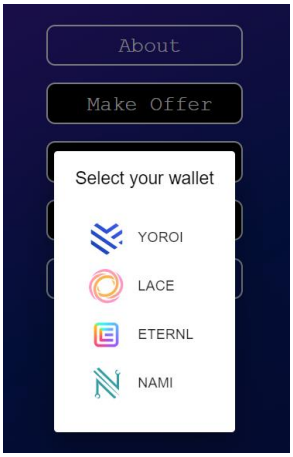

Figure 15 Browser Wallet List Display

onConnectWallet

This is a variable containing a asynchronous function that requires a input called wallet of type mywallet. After successfully enabling the wallet a host of CIP30 API calls are made and stored in state variables. “This is the entry point to start communication with the user's wallet. The wallet should request the user's permission to connect the web page to the user's wallet, and if permission has been granted, the full API will be returned to the dApp to use” https://cips.cardano.org/cip/CIP-30/ . This function gathers the critical information from the wallet including the but not limited to the network, change address, balance, UTxO’s and collateral.

getNetwork

An asynchronous function used to find the browser wallet network. The enabled wallet API is used for the getNetworkId() API endpoint. We set the state of Networkid and then also use the value of the network to set static variables for querying the blockfrost indexer API.

getBalance

Another CIP30 wallet API call to find the total balance of the wallet. Here we employ the CBOR library we imported at the start of the project as the API returns CBOR encoded data. We decode the CBOR then convert the balance from lovelaces to ADA and set the state of the Balance variable by calling setBalance(balance). After successfully connecting a wallet from the select wallet list the balance is displayed inside the Connect button on the homepage next to the icon for the selected wallet.

Figure 16 Balance and Wallet Icon

getCollateral

A CIP30 API call requiring the amount argument to define the minimum Lovelace a set of UTxO’s must contain. We set the state of CollateralHex to the first available UTXO of the payload array if there are any UTxO’s greater than 5ADA returned. “This shall return a list of one or more UTXOs (unspent transaction outputs) controlled by the wallet that are required to reach **AT LEAST** the combined ADA value target specified in amount “ https://cips.cardano.org/cip/CIP-30/ .

getUtxos

A critical function called when a wallet first connects to the dApp to find the assets owned by the user. This is another CIP30 API call which returns an array of CBOR encoded UTxO’s. The function uses the payload from the API request to query the offchain code by calling the sendUtxos function.

This function sends a request to the offchain code flaskProcessUtxos to process the assets in the wallet. Refer to the offchain code flaskProcessUtxos for further details. The offchain code returns sorted, human readable objects which we the use to set the state of 3 variables: AssetUtxos,UtxoObj and NftObj.

getChangeAddress

This function uses the CIP30 getUsedAddresses endpoint. We set the state of UsedAddresses with the response, but also set the first address in the payload to the RawAddress and we set the state of the Pk variable to the contents of address 0 from character 2 to 58 (0 based).

signRawIn

We declare a variable constant signdedTx which waits on the promise from the CIP30 signTx API endpoint. This endpoint will pop-up a CIP30 wallet onto the user's screen and prompt them to sign the transaction that has been crafted for their order action. The function requires an argument which contains the CBOR encoded unsigned transaction we want to sign. We use signRawIn only during the flaskOffer function when trying to submit an offer order.

signRawOut

We use signRawOut during the flaskAcceptOffer and the FlaskCancelOrder functions. The function is identical to signRawIn but called in different places.

offerValidate

This function is used to validate or restrict the acceptable inputs for multiple components in the MakeOfferDialog to prevent malicious inputs or inputs which can cause errors. We use regular expressions (regex) to perform various tests on multiple inputs including:

- bidAmountRequested
- offervalue
- AssetAmount

See https://en.wikipedia.org/wiki/Regular_expression for more information on regex syntax.

If the tests are passed, we set the state of the variables, otherwise we set the state of the InvalidOfferAlert to true. Simple tests including ensuring the value is not negative or 0. A '0' can be present in the input if it is not the first character in the input field. This function is tied to the onChange event of the HTML components for the above variables.

```
<TextField id="bidAmountReq" variant="outlined" type="text" label="Amount"
placeholder="Enter Amount..." value={bidAmountRequested}
            onChange={e => offerValidate(e)} />
```

assetNameValidate

A validation function which prevents the user from offering and requesting the same NFT. JSX is used to conditionally display a MUI alert on the MakeOfferDialog if the validation is failed.

```
{invalidOfferAlert && <Alert severity="info">Cannot offer and Request the Same
NFT</Alert>  }
```

This function is called by the onChange property of the MUI bidNameTextField HTML component on the MakeOfferDialog HTML object.

handleOfferNFT

This function is tied to the onChange event of the Form Control used to house a selector drop-down menu which allows a user to select an asset in their wallet to offer in an order on the MakeOfferDialog. The dropdown menu displays the list of assets in the nftobj, which is the state from the results of the sendUtxos function which processes wallet contents. After a user selects an asset, we iterate through the nftobj map and when the event target value == an assetname in the nftobj list we set the state of the OfferNftObj, OfferNftHexName, OfferNftPolicy. This sets the state data in preparation for building the offer transaction.

handleTokenSelector

This function is tied to the onChange event of the Assets drop down selector in the Form Control on the Make Offer Dialog . This selector is used in the Request Asset or NFT section the Make Offer form. When a user clicks on a token the event is triggered and this sets the bidPolicy and the BidName state to be used for creating an offer transaction.

flaskOffer

This function is called when the 'Submit Escrow' button on the MakeOfferDialog screen is pressed. flaskOffer is an asynchronous function which uses a HTTP POST request to the offchain flaskOffer endpoint. The function takes several arguments which are necessary for the formulation of the offer transaction. We use JSON.stringify to encode this data in the buy_req variable as a JSON string for easy parsing by the python offchain code. The response to the POST request contains the transaction CBOR which we store in the Cbor_to_sign variable. After checking the POST request has not returned an error, we then call the signRawIn function with the unsigned CBOR transaction which will open the user's wallet and prompt them to sign the transaction. If the user signs the transaction, we JSON.stringify the signed_cbor response as well as the original Cbor_to_sign object and use them in another HTTP POST request to the flaskWitnessed endpoint where the transaction will be submitted. flaskWitnessed will return a transaction status containing the transaction id or an error state which an if statement will catch and use to set the state of setAlertState to true.

flaskFindScriptUtxos

An asynchronous function that requires multiple arguments to feed into the flaskFindOffers offchain code endpoint. The purpose of this function is to search through the marketplace smart contract address and find all the offers locked into the contract. This function is called by two other functions – seeOffersDHandler and seeEscrowDHandler. We are able to use the same function to find the script UTxO's for the See Offers Dialog and the In Escrow dialog. We differentiate between the two requests in the offchain code using the 'escrow' key we set in the POST request.

```
let find_req = JSON.stringify({'rawaddress': hashin,'NetworkId': netid,
'utxoList': utxos, 'nftList':nftobj, 'escrow': escrow})
```

acceptOfferHandler

This function is bound to the onClick event handler of the Accept button on the See Offers dialog. The onClick event is provided with the transaction hash of the offer to be accepted through the data-msg property of the button which we can use to perform additional checks over the users wallet contents to ensure they can actually satisfy the requirements of the swap. When a user clicks the Accept button this function will for a for each loop to iterate over the users assets and check they have the correct number of assets required to satisfy the order. If these checks are passed we then call the flaskAcceptOffer endpoint on the offchain code with all of the data necessary to formulate the transaction.

```
flaskAcceptOffer(
    offer,
    rawaddress,
    networkid,
    utxoobj,
    usedAddresses,
    assetUtxos,
    CollateralHex
);
```

cancelOrderHandler

This function is tied to the onClick property of the cancel button in the In Escrow dialog. The data-mssg property of the button contains the transaction hash identifying the order. We iterate over the map of escrowUtxos and check if the dataset.mssg == the transaction hash. When we find the correct match, we call the flaskCancelOrder function with the arguments required to cancel the right transaction.

flaskCancelOrder

This function is called by the cancelOrderHandler and issues a HTTP POST request to the offchain flaskCancelOrder endpoint. We use JSON.stringify to encode the order data as a JSON string for easy decoding by the offchain code. By using the cancel order dataset.mssg transaction hash attached through the cancel button and cancelOrderHandler we correctly identify the order and the data needed for the cancel transaction. Successful execution of the offchain function will result in a reply to the post request containing the unsigned transaction CBOR. We close the Escrow dialog window and open the SubmitDialog window by setting the state of the respective variables to false and true. A CIP30 wallet API call is used using the signRawOut function to request the user to sign the transaction. After successfully signing the witnessed 'signed_cbor' variable and transaction body 'cbor_obj.Cbor_to_sign' is sent back to the offchain flaskWitnessed endpoint for submission to the blockchain. If the transaction is submitted successfully the offchain code will return the transaction id which will be displayed for the user.

sendUtxos

After a wallet connects to Nescrow the sendUtxos function is called to decode the contents of the wallet into manageable tranches. The function takes an argument which contains the list of CBOR encoded UTxO's. This list is sent to the offchain code flaskProcessUtxos endpoint. sendUtxos itself is called within the getUtxos function which is documented above.

flaskAcceptOffer

After a user clicks on an Accept button in one of the offer cards on the See Offers dialog screen, flaskAcceptOffer is called through the acceptOfferHandler function. The offer being accepted is passed into the function through the handler as well as the user wallet data required by the backend flaskAcceptOffer end point to create the transaction.

```
let find_req = JSON.stringify({
      'offerUtxo': offerUtxo,
      'rawaddress': hashin,
      'NetworkId': netid,
      'utxoList': utxos,
      'usedAddresses': usedAddresses,
```

```
      'assetUtxos': assetUtxos,
      'collateralHex': collateralHex}) ;
const accept_res = await fetch(
  FLASK_URL + '/flaskAcceptOffer',
  { method: "POST",
    headers: requestHeaders,
    body: find_req
  }
  )
const cbor_to_sign: string = await accept_res.json();
```

The offchain code returns an unsigned CBOR encoded transaction which the frontend encodes as JSON and provides to the signRawOut function to prompts a user to sign the transaction. After signing the signed transaction CBOR and the original transaction are JSON encoded and sent to the flaskWitnessed endpoint for submission to the blockchain. If no errors are encountered, we display a link to the transaction hash on cexplorer.io for a user to view the successful transaction.

HTML

Nescrow is designed as a single page application with minimal custom CSS to simplify the design and remove the need for complex routing. The interface would benefit from an experienced design engineer creating a more interactive and aesthetic user experience. A lot of the design is directly imported using Material UI (MUI) components https://mui.com/ which reduces development time. The page is broken up between the main landing page which contains 5 buttons and MUI Dialog pop up screens. All dialog screen popups utilize the MUI Dialog component. There is minimal CSS (Home.module.css) which defines a few containers, custom button styling with some basic CSS transitions, and a custom background. The background is a simple linear gradient providing a modern feel.

The HTML code written in the index.tsx file is not direct HTML and relies on JSX syntax. *JSX* is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file https://react.dev/learn/writing-markup-with-jsx . This gives you the ability to add some logic or conditional statements to the HTML code.

```
background-image: linear-gradient(to bottom right, #8b20d7 ,#051c6e, #18033a);
```

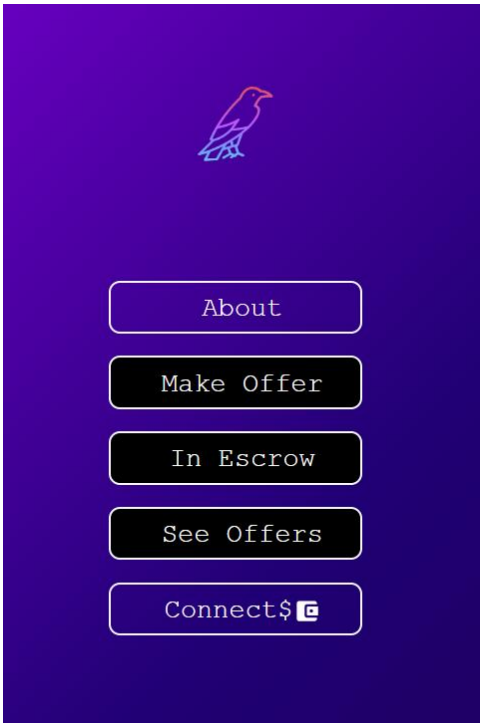

Figure 17 Nescrow Home Page

Landing Div

The landing div imports a custom CSS style from Home.module.css which sets the viewing height and the background image. Within the landing div another sub div class 'cmain' is used to align items, justify content, and set flex direction. We then use a MUI Image tag to show the Nescrow logo. The next sub-div uses the r1 class which styles the 5 home screen buttons. The buttons are tied to the Handler functions aboutDHandler, makeOfferDHandler, seeEscrowDHandler, and seeOffersDHandler respectively. These buttons are also disabled depending upon the state of walletdelay – after a wallet has connected and we have successfully executed the getUtxos function, the state of WalletDelay is set to false, allowing the buttons to be pressed.

```
{(walletconnected == false) && <>     Connect&#x24;
      <AccountBalanceWalletRoundedIcon/>
</> }
{(walletconnected && currentWallet != undefined) && <>
      {balance}A  <Avatar src={currentWallet.icon} /> </>
 }
```

When a user first connects the lowest button will display the 'Connect' text and a wallet Icon if the walletconnected state is false. If the walletconnected state is true and the currentWallet is not undefined we can display the wallet balance and the current wallet icon in this button.

Wallet Connector Dialog

We use the MUI Dialog component to set a simple popup which is opened and closed using the showConnectWallet state. A HTML list is created from the wallets populated by the getWalletsAsync function called when a user first connects to the webpage. Simple JSX is used to iterate over the

wallets map state to create the list with the respective wallet icons and names. If the wallets state is undefined or there are no browser wallets founds the window will show 'No Wallets found'.

Balance Dialog

A deprecated dialog which was used to prompt a user if no collateral UTxO was found.

Make Offer Dialog

After a user has successfully connected their wallet and clicks the Make Offer button the Make Offer Dialog screen will open. The state variable makeofferdialog is used for opening and closing the dialog. We use multiple MUI form control elements to create the inputs which are used to specify the offer being locked into the contract.

Offer Asset

First the user uses the offerswitch toggle button to select between offering a native asset or ADA. If the switch is in the Native Asset position a user has an AssetAmount Textfield input which utilises the onChange property to call the offerValidate function to sanitise the inputs. A dropdown selector is populated with the contents of the assets in the user's wallet for easily selecting an Asset or NFT for offering.

```
<Select id="nftselector"
        labelId="test-select-label"
        value={offerNftName}
        onChange={handleOfferNFT}>
        {nftobj?.map(function(nft,list) {
             return <MenuItem
                         sx={{ fontSize: 16}} key={list}
                         value={nft.assetname}>{nft.assetname}
                    </MenuItem>})}
</Select>
```

If the toggle switch is in the $ADA position the input Textfield offerValue is used with the value also being validated by offerValidate(e). We use JSX to automatically display the policy id and hex encoded asset name if the offerswitch is in the Native asset position and the OfferNftHexName is not blank.

Request Asset

Another toggle switch is used to conditionally display an input TextField for requesting $ADA or input fields for requesting native tokens. When requesting tokens 4 input fields are presented. Because Nescrow allows the trading of any asset for any asset any policy id and asset name can be requested. The first input field is a quick selection drop-down menu with the verified tokens from our curated list of tokens defined by our token registry. Tokens available in this drop-down menu have proper decimal support. An on-change event is tied to the selection of tokens in this drop-down menu which automatically sets the state variables BidPolicy and BidName for the user. If a user is manually setting the Policy(id="bidPolicyTextField"), and the asset Name (id="bidNameTextField"), these Textfields will be validated by the policyValidate and assetNameValidate functions respectively. The final input for requesting assets is the amount Textfield(id="bidAmountReq") whose input is validated through calling the offerValidate function onChange.

IPFS Box

A MUI Box is used to display NFT or native asset images. If the bidimg state is populated with an address, the ipfsReqState state is true and ipfsError state false a MUI card is displayed with a HTML <img> tag pointing to the bidimg which contains a cloud fare URL pointing to the IPFS hosted image.

Alerts

Three JSX conditional MUI Alert bars have been configured. One alert checks a user has not offered and requested the same NFT for sale (thereby forever locking the NFT in the contract) while the next alerts users if the IPFS image cannot be loaded and the final alert is to notify a user if they attempt to offer more ADA than the balance in their wallet.

Submission

A MUI submit button 'Submit Escrow' is used to submit the offer. This button is disabled under a wide variety of circumstances to try and prevent invalid orders from even being processed by the offchain code. We cannot allow orders that do not offer a NFT or Offer any ADA for instance, if the offerswitch is in the Asset position an asset amount must be provided. See below for full list of conditions.

```
<Button variant="contained"  disabled={(
      (offerNftName == '' && offervalue == '') ||
      (!offerswitch && offerAssetAmount =='') ||
      (offerswitch && bidSwitch ) ||
      (bidSwitch == false && (bidName == '' || bidPolicy == '' )) ||
      bidAmountRequested == '' || Number(offervalue) > balance ||
      invalidOfferAlert )
  } onClick={() =>
      flaskOffer(rawaddress,networkid,utxoobj,offerNftObj,offervalue,bidPolic
      y,bidName,bidAmountRequested,offerAssetAmount,usedAddresses,nftobj,asse
      tUtxos)}>
            Submit Escrow
 </Button>
```

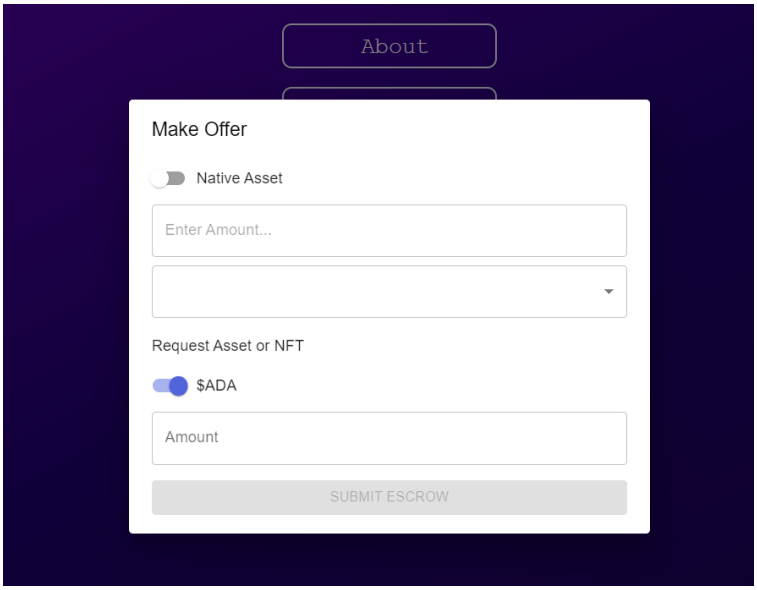

Figure 18 Make Offer Dialog

About Dialog

The about dialog is used to convey critical application information to the user. The dialog is tied to a state variable which is initialized to true in the variable declaration to ensure the screen loads immediately when the page loads. We provide background information pertaining to the dApp, wallet debugging guidelines, notes on the tokens with decimal support, and licencing links.

Network Dialog

The Network Dialog will pop up to alert a user if their wallet is not set on the Cardano Mainnet network. We set the network dialog to true within the getNetwork function if the wallet is not on Mainnet.

Submit Dialog

The submit dialog is opened after a transaction has been submitted so we can display the status of the order and provide a link to the transaction on cexplorer.io. A MUI alert will popup if the alertstate variable is set to true during the processing of the transaction. If the txId state variable doesn't contain 'Error' and the alertstate is false we display – Success with a celebration emoji (🥳). If the txId == Error we display Transaction Failed and a sad face emoji to the user.

See Offers Dialog

The See Offers dialog screen is activated by the 'See Offers' button on the Nescrow Homepage and allows users to accept offers locked by the contract. After a user connects their wallet, they can click the See offers button. Once clicked the seeOffersDHandler function will trigger a query from the front-end to the flaskFindScriptUtxos off chain endpoint to find all the offers locked into the Nescrow smart contract. A TextField searchBox which calls the searchValidate function is used to filter through the offers to limit the offers displayed. We use the state of the FilterOfferUtxos array to iterate over the list of offers at the contract and display MUI Cards. For every item in the array, we create a Card with the asset offered, the image of the asset if available, the details of the swap, an accept button and a view button with a link to view the order on https://cexplorer.io . If the tokens offered or requested exist within the token registry, we augment the display of the token to reflect the decimals registered in the token registry. The Accept button onClick property triggers the acceptOfferHandler function passing in the data-mssg property containing the transaction hash linked to the MUI card to accept the correct order.

```
Swap {Number(datum.unlock_amount) / (10 ** Number(datum.unlock_decimals))}
{datum.unlock_name_utf} for {Number(datum.lockedAssetAmount) / (10 **
Number(datum.locked_decimals))} {datum.nft_offer_name}
```

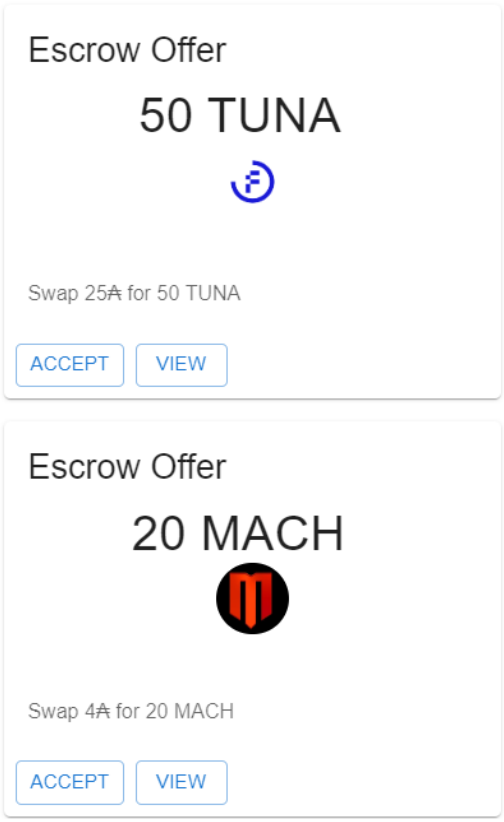

Figure 19 See Offers Dialog Cards

In Escrow Dialog

The in-escrow dialogue is used to show any orders locked into the contract by the current user and allow them to be cancelled. Technically it works very similarly to the See Offers dialog but instead of iterating over FilterOfferUtxos we iterate over the state of escrowUtxos. Both Dialogs rely on the triggering of flaskFindScriptUtxos to find orders locked within the contract. The Cards displayed contain a cancel and a view button. The cancel button onClick property calls the cancelOrderHandler and passes in the datum.tx_hash (transaction hash) of the order in the contract being cancelled, which the cancel order function can use to correctly identify the users order to cancel based on the card button they clicked.

```
<Button size="small"  data-mssg={datum.tx_hash} onClick={cancelOrderHandler}
variant="outlined">Cancel</Button>
```

Loading Dialog

The loading dialog contains a MUI 'CircularProgress' spinner to communicate when the site is loading information. During multiple functions we use the state of the loadingdialog variable to open the loading dialog screen during API calls which can take some time to complete.