eUTxO Fundamentals: Building Cardano Smart Contracts

Starting from zero

AFL v3.0 - 2024

Raul Rosa
aka ElRaulito

INTRODUCTION TO CARDANO SMART CONTRACTS

1. OVERVIEW OF CARDANO BLOCKCHAIN

1.0.1. History

Charles Hoskinson, along with Jeremy Wood, co-founded Cardano, both were part of Ethereum before. In 2015, they established Input Output Hong Kong to create and develop more sustainable blockchain solutions. Utilizing a peer-reviewed approach to blockchain development and introducing a novel consensus mechanism called **Ouroboros**, Cardano was prepared for its Mainnet launch in 2017.

1.0.2. Ouroboros

Ouroboros relies on a **proof of stake** consensus. Rather than requiring nodes to engage in computationally intensive work as in PoW chains, nodes are randomly selected based on the amount of ADA they hold at stake. This approach serves two purposes: it is more energy-efficient and incentivizes nodes to act responsibly as their stake is at risk in case of misbehavior.

1.0.3. Cardano Architecture

The blockchain model comprises several components. Users interact with the current ledger state by creating transactions, which are then submitted to the mempool until they are included in a block. Blocks are mined by stake pools, which are rewarded for their efforts and share these rewards with their delegators. Increased decentralization is achieved with more stake pool operators.

1.0.4. Improvements from Other Blockchains

Cardano offers several advantages over other chains:

- ***Determinism***: This feature enables transaction chaining.
- **Predictable Fees**: There is no risk of pending transactions due to fee increases.
- **Sustainability**: Unlike PoW, which is power-intensive, Cardano's approach is more sustainable.
- **Native Tokens**: Tokens are stored on the ledger, allowing smart contracts to interact with them. Users have control over tokens in their wallets, which cannot be frozen by external parties.

Scaling is currently the most significant challenge for the ecosystem. The ability to handle high volumes of users without being limited by block or transaction size is crucial for increasing adoption.

1.0.5. *Determinism* and predictable fees to make a better user experience

Why ***Determinism*** is important in smart contract programming? Cardano inherits determinism from Bitcoin, once all the fields of a transaction are decided you will always get the same transaction Hash. But more importantly, if you can get the hash of a transaction before actually submitting, you can even create a following transaction, relying on the first one. This is usually called transaction chaining, I can create a chain of transactions that are not submitted, and this can allow me to speed up the user flow. Ok, let's try to simplify even more this concept.

- Alice, Bob and Raul are in a Bar, each has 100 ADA
- Alice sends to Raul 50 ADA but the blockchain right now is super clogged
- However Raul is already able to build a transaction to send 120 ADA to Bob because even if the blockchain is clogged, the hash of the transaction is already decided and won't change at all
- Now even Bob can send 220 ADA to someone else, even before the transactions are confirmed, due to Cardano determinism he can use transactions that are not confirmed yet

Everything seems amazing, perfect. Where is the issue? What happens if for some reason Alice had her transaction with a deadline of 1 hour? In that case, Alice's transaction could never become valid, therefore every other transaction depending on that will never make it. This means that everything goes to the beginning, Alice, Bob and Raul have 100 ADA each. This is a problem if each of them paid for goods in the real world and now they get their money back.

Why do predictable fees matter and what's their role in *determinism*?

On Bitcoin when you set inputs (what you spend), outputs (who gets the money) and fees you can get the transaction hash. This happens also on Cardano, however on Bitcoin, there is a fee market, therefore the fees you set may not be enough to cover the cost of having the transaction in the next blocks. So on Bitcoin fees may need to change, there is an RBF feature that allows you to speed up a transaction increasing the fee cost. But this also leads to a change in the transaction hash, therefore we can't always build a chain of transactions on Bitcoin because one transaction could change, making all the following invalid.

On Cardano there is no fee market, in this way, once you pay enough to cover the processing costs, that transaction will be in the following blocks. Fees are not dynamic and can't change.

Even if it sounds cool, this leads to a problem, if there is no way of speeding up my transaction or making it possible to get priority over others, how can a protocol that needs instant settlement work? This is an open question that lately has been discussed as a tier fee market on Cardano.

1.1. IMPORTANCE AND APPLICATIONS OF SMART CONTRACTS

Smart contracts are a concept born alongside Ethereum, enabling the execution of code and interactions without a third party. Once initiated, the terms of the contract are set by the parties involved, and no one can stop or interfere thereafter.

However, history teaches us that some protocols have included backdoors within their smart contracts, leading to fund theft or enabling bad actors to access users' funds.

Let's start with the basics.

1.1.1. What is a Smart Contract?

A smart contract is a decentralized software accessible to users on the blockchain, typically through a website interface. Users interacting with the contract can perform operations (financial, trading, storage) without requiring permission from a third party.

The essential components of a smart contract are:

- **Parties**: Who can interact with the contract? Is it open to everyone, specific users, or owners of particular assets?
- **Actions**: What operations can users perform with the contract? These could include depositing funds, creating NFTs, storing data, reading data, withdrawing funds, and more.
- **Rules**: Define the actions each party can take under specific conditions.
- **Data Fields**: What data is involved in interactions with the contract, and how can each step of the interaction be tracked?

1.1.2. Applications

In a typical decentralized exchange (DEX) application, the parties are liquidity providers and traders. Liquidity providers can deposit and withdraw liquidity, while traders can only perform swaps. Rules dictate that liquidity providers must hold LP tokens in their wallets, while traders must have sufficient funds to cover transactions. Data fields stored in the contract typically include LP tokens, fees for liquidity providers, and token data.

In a marketplace scenario, the parties are sellers and buyers. Sellers can sell assets, while buyers can buy them. Rules stipulate that sellers must possess the assets they intend to sell, and buyers must have sufficient funds to purchase assets and pay sellers. Data stored includes the seller's address, payment amounts, royalties (if applicable), and platform fees.

On Cardano, specific actions might include ***Cancel Listing*** and ***Buy***. ***Selling/List*** is more of a smart contract interaction than an action.

> A smart contract action involves a transaction where the smart contract is invoked in the inputs. If the smart contract is present only in the outputs, it's considered a smart contract interaction.

There can be many more smart contract applications, imagination is the limit, and some applications may work better on Cardano due to UTXO architecture or on EVM chains due to the account model. Let's consider the case of a dex, it can be either ***orderbook*** or

AMM. The orderbook works perfectly in a UTXO blockchain because every order can be a single UTXO. While on EVM chains orderbook dexes struggle because they are limited by the memory of the smart contract. On the opposite side, building an AMM on Cardano requires a lot of emulation, since the pool is a single UTXO, more parties can't spend it at the same time, that's why we found as solutions the batchers. Batchers match the orders with the single UTXO liquidity pool. This concept is not efficient however AMM are more user-friendly usually and that's why currently Cardano is the one leading in liquidity volumes.

1.1.3. Smart Contract audits

Once a smart contract is developed and ready to launch an audit should be done, this is to ensure that the code is safe and no issues may arise once people start interacting with it. Audits play a critical role in the deployment of smart contracts, serving as a crucial safeguard against potential vulnerabilities and ensuring the integrity of the code. Smart contracts, being immutable, leave little room for error once deployed, making thorough scrutiny prior to launch imperative. Audits help identify security flaws, logic errors, and vulnerabilities that may compromise the contract's functionality or jeopardize users' funds. By subjecting the code to rigorous review by experienced professionals, audits instill confidence among users and investors, fostering trust in the decentralized ecosystem. Moreover, audits contribute to the overall maturation of the blockchain space, driving standards for secure coding practices and enhancing the reliability of smart contract applications. Ultimately, investing in audits upfront mitigates the risk of costly exploits or breaches down the line, safeguarding both the project's reputation and the interests of its stakeholders.

But audits have a cost and sometimes early projects may not afford that much.

Opensource can be a way in order to launch a project asking for external reviews coming from the community, usually bug bounty programs are run in order to incentivize users to collaborate in the task of finding risks in the smart contract code.

1.1.4. Smart Contract risks

On Cardano there are some risks regarding smart contracts that we'll study better in the following chapters, but here are a few of them as a preview:

- Double satisfaction attack: User may spend two inputs that require similar conditions
- Dust attack: Users may add spam tokens in a smart contract making it impossible to retrieve funds from it
- Spam Contract: A second smart contract can run together with the attacked one, making it possible to unlock funds from the first
- Datum attack: A datum of the contract may be corrupted making unspendable the funds inside the contract
- backdoor: All the funds of the contract may be retrieved by someone who coded a backdoor

1.1.5. The cost of deploying a contract

Deploying a contract onto the blockchain carries no direct cost, allowing widespread accessibility. However, it's essential to consider additional expenses, especially if utilizing reference scripts like ADA, which may necessitate funds for storage on the blockchain. Moreover, while users typically prefer interacting with contracts via frontends, developing and maintaining such interfaces entail both frontend and backend costs. It's imperative to conduct thorough economic assessments before project launch, ensuring expenses don't surpass revenues. Abruptly discontinuing services without prior notice could result in users losing their funds, highlighting the importance of transparent communication in managing smart contract projects.

1.2. ADVANTAGES OF CARDANO FOR SMART CONTRACT DEVELOPMENT

Two years ago, if you asked me about the advantages of writing smart contracts on Cardano, I would have struggled to answer. However, now I can easily list several:

- **Composability**: The ability to create a transaction involving multiple contracts and perform actions with each of them.
- **User-Friendly**: No longer requiring Haskell, languages like Aiken, Opshin, and more offer a user-friendly experience.
- **Liquid Staking**: Thanks to Cardano staking, smart contracts can delegate ADA or keep funds staked with liquidity providers.
- **UTxO Skills**: While much of the focus has been on Ethereum Virtual Machine (EVM) smart contracts, the UTXO model is ideal for solutions like ZK rollups, as it's easier to implement compared to the account model.

If you're still interested in becoming a Cardano smart contract wizard after this introduction, we can continue in the next chapter, where we'll install the components needed to **build on Cardano**.

SETTING UP THE DEVEL-OPMENT ENVIRONMENT

2. INSTALLING AND CONFIGURING CARDANO DEVELOPMENT TOOLS

2.1. INSTALLING AND CONFIGURING CARDANO DEVELOPMENT TOOLS

The purpose of this book is to gather all the information for developing Cardano that currently is scattered around. What are we going to use for our project?

- **A hot Wallet**: We are going to use a wallet to test our contracts, this wallet will be used to receive tADA. We'll never store our main ADA holdings in this wallet: Wallets recommended for testnet are Nami on desktop and Vespr for mobile.
- **A indexer account**: Indexers are the ones that will provide us the APIs in order to interact with the chain, we won't need to run a node for testing, let's use services and projects already there like **Maestro**, let's set up an account and get the API key.
- **Lucid library**: Lucid is not maintained anymore as a function and has been replaced by COMING SOON, however for testing and understanding the flow of Cardano transactions it can be really useful.
- **tADA**: How are we going to test without having testnet ADA? let's not mess up real ADA
- **IDE**: Personally I use Visual Studio Code as IDE, but any other editor is ok since we are going to
- **Cardano Node:** This is NOT mandatory at all, as homework, we could try to set up a Cardano node and interact with the chain using cardano-cli (command line), however, this is something we can do in our free time, there are other hobbies out there better than this, swimming, dancing or reading a book.

*A new library is being built and once live it's going to be added in this book.

2.2. HOT WALLETS ON CARDANO

When it comes to the wallet choice on Cardano the question we should ask ourselves is: Desktop or Mobile?

Test the wallet you like most and pick the one that gives you more user-friendly vibes for your use, a developer may require a very detailed wallet, however, a basic user may need just some very simple buttons without details.

2.3. SETTING UP AND CONNECTING TO CARDANO TESTNET

So let's install a wallet and config for testnet

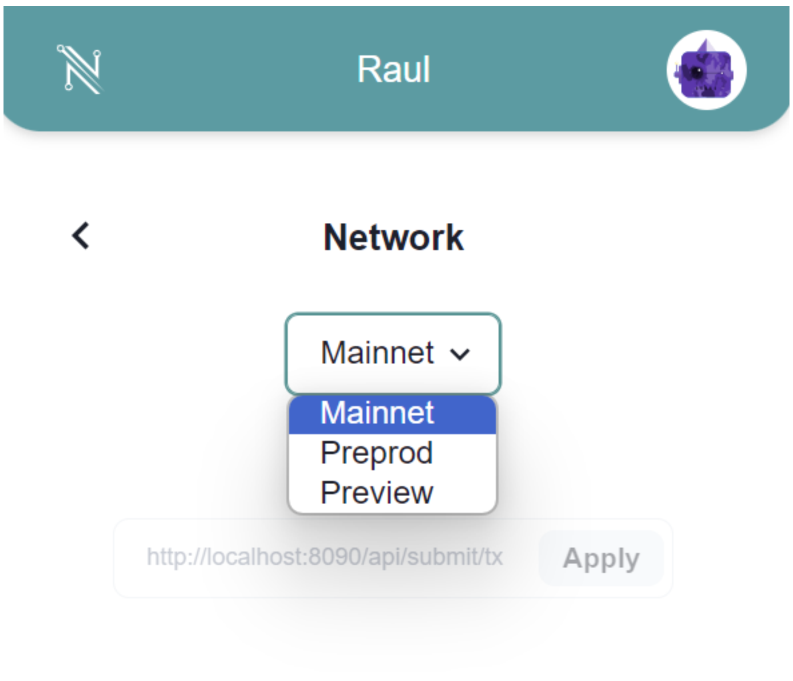
In this example, we'll install the Nami wallet that we can find **here**

Once we install the wallet we'll get 24 seed phrase words

Current Cardano wallets available, updated in Q2 24

Wallets	Desktop	Mobile	Website
Nami	X		https://www.namiwallet.io/
Eternl	X	X	https://eternl.io/
Begin		X	https://begin.is/
Vespr		X	https://vespr.xyz/
Lace	X		https://www.lace.io/
NuFi	X		https://nu.fi/
Yoroi	X	X	https://yoroi-wallet.com/
Flint	X	X	https://flint-wallet.com/
Gero	X		https://www.gerowallet.io/
Typhon	X		https://typhonwallet.io/

> Never share the seedphrase or store it on a cloud, use paper or different ways to store it, software can keep track of your seedphrase and you could lose the funds.



Let's set Nami to **testnet preview** and we'll finally get our wallet in testnet

2.3.1. Preview and Preprod testnets

- **Mainnet**: This is the live network where real transactions occur using actual ADA. It's the primary arena where users engage with Cardano wallets, exchanges, and decentralized applications (dApps).
- **Preprod**: Acting as a staging ground for major upgrades and releases, Preprod is a testing environment where developers validate changes before deploying them to the mainnet. Utilizing test ADA acquired from the faucet, developers simulate real-world scenarios, ensuring everything functions as intended before the changes go live. Preprod typically mirrors mainnet's structure, forking nearly simultaneously to ensure alignment.
- **Preview**: Serving as a testing environment to showcase upcoming features and functionality, Preview allows developers and users to explore and provide feedback on new developments before they reach the wider community. Like Preprod, test ADA from the faucet facilitates testing. Notably, Preview precedes mainnet hard forks by a minimum of four weeks, offering ample time for thorough evaluation and refinement based on community input.

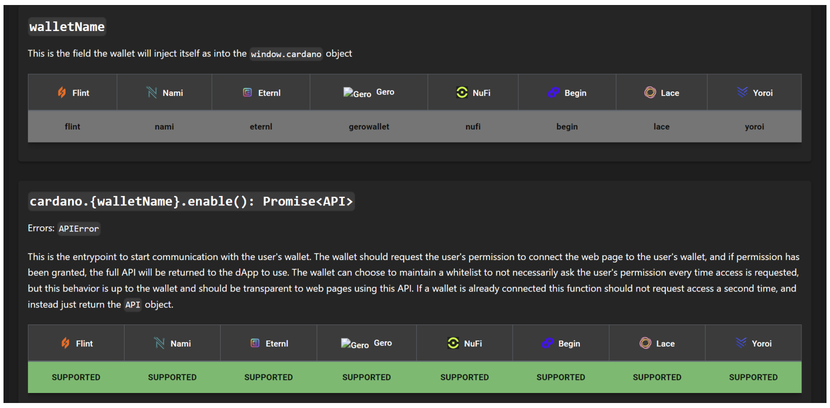
2.3.2. Get tADA

In order to receive tADA we can use the official faucet from Cardano at the following **link**

The process doesn't involve any payment and at the end of your testing, ideally, you should return tADA back so other devs can work with it.

2.3.3. CIP30

To connect our wallet with any webpage we'll use CIP30 reference, we can find the list of methods to connect and invoke the functions of the wallets at this **page**



walletName

This is the field the wallet will inject itself as into the `window.cardano` object

Flint	Nami	Eternl	Gero	NuFi	Begin	Lace	Yoroi
flint	nami	eternl	gerowallet	nufi	begin	lace	yoroi

`cardano.{walletName}.enable(): Promise<API>`

Errors: `APIError`

This is the entrypoint 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. The wallet can choose to maintain a whitelist to not necessarily ask the user's permission every time access is requested, but this behavior is up to the wallet and should be transparent to web pages using this API. If a wallet is already connected this function should not request access a second time, and instead just return the `API` object.

Flint	Nami	Eternl	Gero	NuFi	Begin	Lace	Yoroi
SUPPORTED	SUPPORTED	SUPPORTED	SUPPORTED	SUPPORTED	SUPPORTED	SUPPORTED	SUPPORTED

The steps to interact with a wallet following CIP30 are:

- **cardano.walletName.enable()**: we get an API object as Promise, this will create a popup message to allow the wallet to connect to the current website
- **api.getBalance()**: using the API object we got before, we get the total amount of Lovelace in the wallet (1 ADA = 1000,000 Lovelace)
- **api.signTx**: Signing a tx that was built with Lucid or any other library we sign and interact with the blockchain

> EXERCISE 1: Create a webpage with 2 buttons, 1 to enable the wallet connection and, a second button to view the amount of ADA in the wallet.

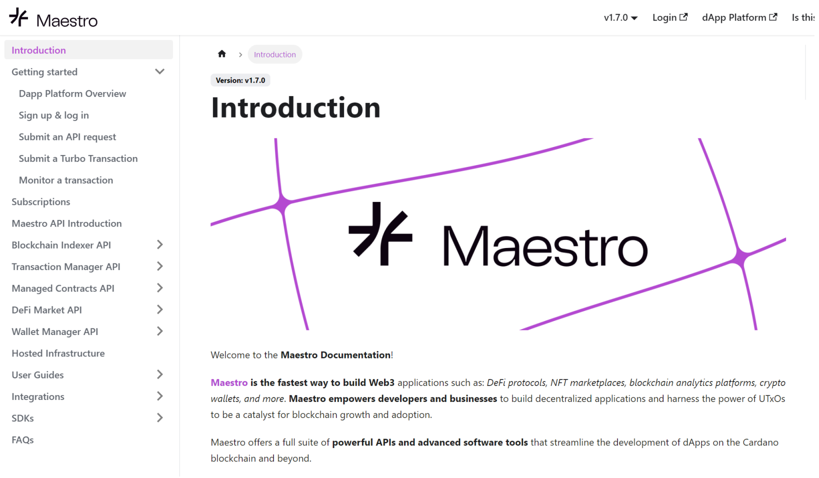
2.4. INTERACTING WITH CARDANO NODE AND WALLET APIS

CIP30 is not enough, what if we want to get the information regarding a specific NFT in our wallet? How to get the list of tokens inside the wallet and get information regarding their circulation supply?

We need an indexer. We could set one on our own or use a service, in this book we'll use **Maestro** as a service provider so the first thing to do is:

2.4.1. Create a Maestro account

Head over **`Maestro login`** page and create an account, here we'll be able to get the API keys to interact with Cardano.



Maestro v1.7.0 Login dApp Platform Is thi

Introduction
Getting started
Dapp Platform Overview
Sign up & log in
Submit an API request
Submit a Turbo Transaction
Monitor a transaction
Subscriptions
Maestro API Introduction
Blockchain Indexer API
Transaction Manager API
Managed Contracts API
DeFi Market API
Wallet Manager API
Hosted Infrastructure
User Guides
Integrations
SDKs
FAQs

Introduction

Version: v1.7.0

Introduction

Maestro

Welcome to the **Maestro Documentation**!

Maestro **is the fastest way to build Web3** applications such as: *DeFi protocols, NFT marketplaces, blockchain analytics platforms, crypto wallets, and more.* **Maestro empowers developers and businesses** to build decentralized applications and harness the power of UTxOs to be a catalyst for blockchain growth and adoption.

Maestro offers a full suite of **powerful APIs and advanced software tools** that streamline the development of dApps on the Cardano blockchain and beyond.

Maestro is going to be our key to getting all the possible APIs in order to interact with Cardano, here are the possible things we can do with these APIs:

- Get the history of an address with this **`API`**
- Get all the assets of a specific policy
- Get the address holding a specific ada handle
- Get the history of holders for a specific NFT
- and much more

Now that we have a way to interact with a wallet and APIs to query the Cardano blockchain we are ready to put our hands on the real coding part.

2.4.2. Indexer alternatives

If you would like to explore additional API providers you should consider the following:

- Blockfrost: The very first API provider for Cardano **`link`**
- Kupo: A tool that requires a node in order to host your own11 indexer **`Github`**
- Db-sync: Additional indexer that requires a Cardano Node, this is the very first one that was created **`Github`**

Now Let's code smart contracts.

> EXERCISE 2: Head over to http://cnftlab.party/ and connect your testnet wallet, mint a collection of NFTs and then use Maestro to get the information of each NFT of your collection.

2.5. SETTING UP A CARDANO NODE ON CONTABO CLOUD VPS

2.5.1. Step 1: Provisioning a Contabo Cloud VPS

1. Sign up for a Contabo Cloud VPS plan, such as the ”Cloud VPS L”.
2. Once you have access to your VPS, log in via SSH.

2.5.2. Step 2: Downloading and Extracting Cardano node Software

1. Navigate to the official Cardano GitHub release page: **Cardano Node Releases**.
2. Download the Cardano Node software for macOS by running the following command:

```
    wget https://github.com/input-output-hk/cardano-node/releases/download/8.1.2/ca
node-8.1.2-macos.tar.gz
```

3. After the download completes, create a directory named ”node” and extract the downloaded files into it:

```
mkdir node
tar xvzf cardano-node-8.1.2-macos.tar.gz -C node
```

2.5.3. Step 3: Setting Up Node Configuration

1. Create the necessary directories:

```
cd node
mkdir mainnet
cd ..
mkdir sockets
```

2. Create a systemd service file for the Cardano Node:

```
sudo nano /etc/systemd/system/cardano-node.service
```

   Paste the following content in the file:

```
sudo nano /etc/systemd/system/cardano-node.service
Now we should copy and paste the following lines:
[Unit]
Description=Cardano Pool
After=multi-user.target
[Service]
Type=simple
ExecStart=/home/ubuntu/nodev30/cardano-node run --config
/home/ubuntu/nodev30/config/mainnet-config.json --topology
/home/ubuntu/nodev30/config/mainnet-topology.json --database-path
/home/ubuntu/nodev30/mainnet/db/ --socket-path  /home/ubuntu/nodev30/sockets/node.
host-addr 0.0.0.0 --port 3001

KillSignal = SIGINT
RestartKillSignal = SIGINT
```

```
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=cardano
LimitNOFILE=32768

Restart=on-failure
RestartSec=15s
WorkingDirectory=~
User=USERNAMEVPS

[Install]
WantedBy=multi-user.target
```

Save the file and exit the editor.

2.5.4. Step 4: Installing Required Dependencies and Syncing the Blockchain

1. Update package list and install necessary tools:

   ```
   sudo apt update && sudo apt install liblz4-tool jq curl
   ```

2. Fetch the latest blockchain snapshot and sync the blockchain:

   ```
       curl -o - https://downloads.csnapshots.io/snapshots/mainnet/$(curl -s https://c
   db-snapshot.json| jq -r .[].file_name ) | lz4 -c -d - | tar -x -C /root/node/mainn
   ```

   This process may take around 30 minutes.

2.5.5. Step 5: Starting the Cardano node

1. Enable and start the Cardano Node service:

   ```
   sudo systemctl enable cardano-node.service
   sudo systemctl start cardano-node.service
   ```

2. Monitor the node's status:

   ```
   journalctl -u cardano-node.service -f -o cat
   ```

2.5.6. Step 6: Setting Up Cardano Submit API

1. Navigate to the node directory:

   ```
   cd node
   ```

2. Create a configuration file for the transaction submission API:

```
    nano tx-submit-mainnet-config.yaml
```

Paste the content from **Cardano Node GitHub** into this file. Save and exit the editor.

3. Run the Cardano Submit API with the provided configuration:

```
    ./cardano-submit-api --tx-submit-mainnet-config.yaml --socket-path /root/node/
port 8090 --mainnet --host-addr 0.0.0.0
```

2.5.7. Step 7: Accessing Transaction Submission Endpoint

With the Cardano Submit API running, you can now send transactions using your node by accessing the following URL:

```
http://VPSIPADDRESS:8090/api/submit/tx
```

Replace `VPSIPADDRESS` with the IP address of your VPS.

By following these steps, you'll have successfully set up a Cardano node on your Contabo Cloud VPS and be ready to interact with the Cardano blockchain.

EXPLORING THE EUTXO MODEL

3. UNDERSTANDING THE EUTXO MODEL AND ITS COMPONENTS

3.1. UNDERSTANDING THE EUTXO MODEL AND ITS COMPONENTS

Two popular record-keeping models in blockchain networks are the eUTXO (Extended Unspent Transaction Output) Model used by Cardano and the Account/Balance Model employed by Ethereum. This section provides a basic understanding of these models, their differences, and their respective pros and cons.

3.2. EUTXO MODEL

In Cardano, each transaction spends outputs from prior transactions and generates new outputs for future transactions. All unspent transactions are stored in each fully synchronized node, giving rise to the name "eUTXO". A user's wallet tracks unspent transactions associated with the user's addresses, and the wallet balance is the sum of these unspent transactions.

3.2.1. Example

1. Alice gains 12.5 ADA through staking rewards, resulting in one eUTXO of 12.5 ADA.

2. Alice sends 1 ADA to Bob. Alice's wallet uses her eUTXO of 12.5 ADA, sending 1 ADA to Bob and receiving 11.5 ADA as a new eUTXO to a new address owned by Alice.

3. If Bob had an eUTXO of 2 ADA before step 2, his wallet now shows a balance of 3 ADA from two eUTXOs.

3.2.2. Account/Balance Model

The Account/Balance Model maintains the balance of each account as a global state. It checks that an account's balance is sufficient to cover the transaction amount.

3.2.3. Example

1. Alice gains 5 ETH through mining, recorded in the system.

2. Alice sends 1 ETH to Bob, reducing her balance to 4 ETH.

3. Bob's balance increases by 1 ETH, so if he had 2 ETH initially, he now has 3 ETH.

3.2.4. Analogies

- **eUTXO Model**: Similar to using paper bills, where each bill (eUTXO) can be spent once, and change is returned as new eUTXOs.
- **Account/Balance Model**: Similar to a bank's ATM/debit card system, where the bank ensures sufficient balance before approving transactions.

3.2.5. Benefits

3.2.6. eUTXO Model

- **Scalability**: Enables parallel transactions and scalability innovations.
- **Privacy**: Provides higher privacy, especially with new addresses for each transaction.

3.2.7. Account/Balance Model

- **Simplicity**: Easier for developers of complex smart contracts requiring state information.
- **Efficiency**: More efficient as each transaction only validates account balance.

Drawbacks

Account/Balance Model: Susceptible to double-spending attacks, counteracted by an incrementing nonce.

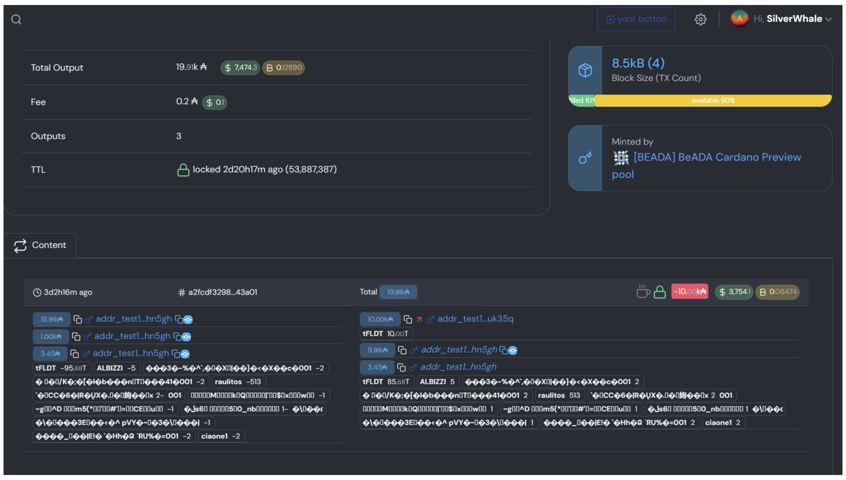
Both models have trade-offs and are chosen based on specific blockchain needs. Some blockchains, like Hyperledger, adopt eUTXO to benefit from Bitcoin's innovations.

3.3. WRITING TRANSACTIONS AND VALIDATING INPUTS AND OUTPUTS

In this section, we'll analyze the components of a Cardano transaction and how it is built using a Cardano node and then using the Lucid library.

Let's start with the following transaction:

`a2fcdf32987ebb729ab8f63b377e360ececbf4805713ff559d2b69bb3c543a01`



```
Total Output   19.91k ₳   $ 7,474.3   B 0.12890
Fee            0.2 ₳   $ 0.1
Outputs        3
TTL            locked 2d20h17m ago (53,887,387)

8.5kB (4)
Block Size (TX Count)
filled 10%   available 90%

Minted by
[BEADA] BeADA Cardano Preview pool

3d2h16m ago   # a2fcdf3298...43a01   Total 19.91k₳   -10.00k₳   $ 3,754.1   B 0.06474

18.91k₳  addr_test1...hn5gh            10.00k₳  addr_test1...uk35q
1.00k₳   addr_test1...hn5gh            tFLDT 10.00T
3.45₳    addr_test1...hn5gh            9.91k₳   addr_test1...hn5gh
tFLDT -95.68T  ALBIZZI -5  ...001 -2   3.45₳    addr_test1...hn5gh
...001 -2  raulitos -513               tFLDT 85.68T  ALBIZZI 5  ...001 2
...x 2- 001  ... -1                    ...001 2  raulitos 513  ...x 2 001
~g...^D ...CE...u -1  ...5O_nb... 1-   ... 1  ~g...^D ...CE...u 1  ...5O_nb... 1
...3E...^ pVY~...3... -1               ...3E...^ pVY~...3... 1  ...=001 2  ciaone1 2
...=001 -2  ciaone1 -2
```

Let's analyze what we can see here:

On the **left**, we have the inputs of the transaction. We can see that we are spending 3 inputs containing tADA and some tokens. The inputs being spent are from the following transactions:

- `5df4a4fb78650b5d8b0e761e0ade2c2ab2289b4feb97f6780b82d78b3d02bf70`
- `f0e29aed793164ce8192aec7ec647b7e7747206d701b0dfd4a810414f7acb96b`
- `5df4a4fb78650b5d8b0e761e0ade2c2ab2289b4feb97f6780b82d78b3d02bf70`

This means that for each transaction, we can obtain the transactions that generated the funds being spent directly from the hash of the inputs. Not just this, it means that we are always able to track if the funds generated by a transaction have already been spent because they will appear as input on another transaction.

But that's not all; for each input, we are able to see from which address they come from, in this case `addr_test1qqw...hn5gh` and also the amount of tADA and tokens that were locked inside those inputs. We'll call this **Value**.

On the **right** side, we can see the outputs of the transaction. Therefore, where is the tADA going?

We have 3 outputs:

- The first output is sending 10,000 tADA and 1M tFLDT to `addr_test1qpku...sjuk35q`.
- The second output is a change to `addr_test1qqwywxe3ag9sf3jjhk8hd...fhn5gh` sending back all the tADA that was left.
- The third output is a change to `addr_test1qqwywxe3ag9sf3jjhk8hd...fhn5gh` sending back all the NFTs and tokens that were leftovers.

Finally, we can see that this transaction was validated during epoch 623, block 2,270,567, and slot 53,865,862. It was confirmed by BEADA stake pool operator, and there was a 0.2 fee paid to the network.

3.3.1. Build a transaction with Cardano node

This part is not mandatory since it requires to run a Cardano node, however it is interesting to see how a transaction is built from scretch

Let's create a folder:

```
mkdir exercise01
```

Now we create the wallet with:

```
cardano-cli address key-gen --verification-key-file payment.vkey --signing-key-
    file payment.skey
```

To see the address, use the following command:

```
cardano-cli address build --payment-verification-key-file payment.vkey --out-
    file payment.addr
```

And then:

```
cat payment.addr
```

And we now see our address!

Checking funds in wallet

Let's run the command:

```
cardano-cli query utxo --address $(cat payment.addr) --mainnet
```

We will see that there are no transactions, so 0 ADA.

Sending funds

Let's send some ADA to the wallet from our main wallet (try with 5 ADA) and run the command again to check the transactions:

```
cardano-cli query utxo --address $(cat payment.addr) --mainnet
```

Now we should see some ADA. For instance, 5,000,000 lovelace means 5 ADA.

Check funds of any address

If you want to check the ADA inside any wallet, the command becomes:

```
cardano-cli query utxo --address ADDRESSTOCHECKHERE --mainnet
```

The result is all the transactions containing ADA and NFTs in the address.

Creating the raw transaction

Let's copy the transaction hash that contains the 5 ADA and the index:

```
cardano-cli transaction build-raw --fee 0 --tx-in HASHOFUNSPENTTRANSACTION#
    INDEX --tx-out ADDRESSRECEIVER+2000000 --tx-out $(cat payment.addr)+0 --
    mainnet --out-file matx.raw
```

Calculate the fee

We must calculate the fee according to the network parameters that we get with the following:

```
cardano-cli query protocol-parameters --mainnet --out-file protocol.json
```

Now:

```
cardano-cli transaction calculate-min-fee --tx-body-file matx.raw --tx-in-count
     1 --tx-out-count 2 --witness-count 1 --mainnet --protocol-params-file
    protocol.json
```

And we get the fee we should pay at least.

Build the final transaction

Now we can finally build the complete transaction with:

```
cardano-cli transaction build-raw --fee FEE_WE_CALCULATED --tx-in
    HASHOFUNSPENTTRANSACTION#INDEX --tx-out ADDRESSRECEIVER+2000000 --tx-out $(
    cat payment.addr)+BALANCE_MINUS_FEES_MINUS_2_ADA --mainnet --out-file matx.
    raw
```

At this point, the transaction is finished. We must sign it with the key to prove we are the owners.

Signing the transaction and submitting it to the blockchain

```
cardano-cli transaction sign --signing-key-file payment.skey --mainnet --tx-
    body-file matx.raw --out-file matx.signed
```

Now using the key in the folder, we approved the transactions, we can send it to the blockchain.

Submitting the transaction

To send it to the blockchain, we can launch the following:

```
cardano-cli transaction submit --tx-file matx.signed
```

And now, after it has been processed, our balance will decrease.

3.3.2. Building a Transaction with Lucid

The *Lucid* library was the very first library to accelerate development on Cardano. The main advantage of this library is its capability to make the entire process faster and easier. The previous example can be simplified as follows:

```
const tx = await lucid.newTx()
  .payToAddress("ADDRESSRECEIVER", {lovelace: 2000000n})
  .complete();
const signedTx = await tx.sign().complete();
const txHash = await signedTx.submit();
```

There is no doubt why *Lucid* has been used by the majority of the **dApps** developed on Cardano.

3.4. CARDANO NATIVE SCRIPTS

Cardano introduced *native scripts* as a foundational feature in the Shelley era, which predated the advent of smart contracts. These scripts provide a way to define complex conditions for spending funds, extending the functionality available in Bitcoin through its script language. While Bitcoin scripts include features like *multisignature* and *timelock*, Cardano's native scripts build upon these concepts with more sophisticated capabilities.

This section explores Cardano's native scripts, focusing on *multisignature scripts* and *time locking*, and comparing them to Bitcoin scripts.

3.4.1. Comparison with Bitcoin Scripts

Bitcoin scripts are a simple stack-based language primarily used for two main features:

- **Multisignature**: Requires multiple signatures to authorize a transaction. For example, a 2-of-3 multisignature scheme requires any two of three possible signatures.
- **Timelock**: Restricts spending of funds until a certain time or block height. Examples include *CheckLockTimeVerify* which locks funds until a specific block height or timestamp.

Cardano extends these features with more advanced scripting capabilities in its native script language.

In the Shelley era and beyond, Cardano introduced a more expressive scripting language that includes *multisignature scripts*. These scripts are used to create addresses that require multiple cryptographic signatures to authorize transactions.

3.4.2. Description

A multisignature script address is one where a transaction must meet specific conditions, such as collecting signatures from multiple keys. The script defines these conditions, and the transaction witness includes both the script and the required signatures.

3.4.3. Multisig Script Language

The multisig script language uses a simple expression tree with four primary constructors:

- **RequireSignature**: `RequireSignature vkeyhash` - Validates that a transaction includes a signature corresponding to the given verification key hash.
- **RequireAllOf**: `RequireAllOf <script> *` - Requires that all included scripts are satisfied.
- **RequireAnyOf**: `RequireAnyOf <script> *` - Requires that at least one of the included scripts is satisfied.
- **RequireMOf**: `RequireMOf <num> <script> *` - Requires that at least M of the included scripts are satisfied.

3.4.4. JSON Script Syntax

Multisignature scripts can be represented in JSON as follows:

```
{
  "type": "sig",
  "keyHash": "e09d36c79dec9bd1b3d9e152247701cd0bb860b5ebfd1de8abb6735a"
}
```

KeyHash is the publicKey of the address allowed to spend from this multisignature, in this case is a simple 1 of 1 multisignature script.

3.4.5. Time Locking Scripts

Cardano introduced time-locking features to the native script language. This extension enables the creation of scripts that restrict transactions based on time conditions.

3.4.6. Description

Time-locking allows conditions like:

- **RequireTimeBefore**: The current slot number must be before a specified slot.
- **RequireTimeAfter**: The current slot number must be after a specified slot.

3.4.7. JSON Script Syntax

Time-locking scripts can be represented in JSON as follows:

```
{
  "type": "all",
  "scripts":
  [
    {
      "type": "after",
      "slot": 1000
    },
    {
      "type": "sig",
      "keyHash": "966e394a544f242081e41d1965137b1bb412ac230d40ed5407821c37"
    }
  ]
}
```

This example shows a script where there are two rules, only the owner of this publicKey can use it: 966e394a544f242081e41d1965137b1bb412ac230d40ed5407821c37 and additionally only after the slot 1000.

3.4.8. Multisignature Script Example

Step 1: Create a Multisignature Script Address

```
{
  "type": "all",
  "scripts":
  [
    {
      "type": "sig",
      "keyHash": "e09d36c79dec9bd1b3d9e152247701cd0bb860b5ebfd1de8abb6735a"
    },
    {
      "type": "sig",
      "keyHash": "a687dcc24e00dd3caafbeb5e68f97ca8ef269cb6fe971345eb951756"
    },
    {
      "type": "sig",
      "keyHash": "0bd1d702b2e6188fe0857a6dc7ffb0675229bab58c86638ffa87ed6d"
    }
  ]
}
```

This type of multisig is a 3 of 3 multisignature, all the 3 signatures must approve the spending of the funds from the address

Step 2: Create the Address

```
cardano-cli address \
  --payment-script-file allMultiSigScript.json \
  --testnet-magic 42 \
  --out-file script.addr
```

Step 3: Construct and Submit a Transaction

```
cardano-cli transaction build-raw \
    --invalid-hereafter 1000 \
    --fee 0 \
    --tx-in <utxoinput> \
    --tx-out "$(cat script.addr) <amount>" \
    --out-file txbody

cardano-cli transaction witness \
  --tx-body-file txbody \
  --signing-key-file <utxoSignKey> \
  --testnet-magic 42 \
  --out-file utxoWitness

cardano-cli transaction assemble \
  --tx-body-file txbody \
  --witness-file utxoWitness \
  --out-file allWitnessesTx
```

Step 4: Submit the Transaction

```
cardano-cli transaction submit \
  --tx-file allWitnessesTx \
  --testnet-magic 42
```

3.4.9. Time Locking Script Example

Example JSON for a Time Locking Script

```
{
  "type": "all",
  "scripts":
  [
    {
      "type": "after",
      "slot": 1000
    },
    {
      "type": "sig",
      "keyHash": "966e394a544f242081e41d1965137b1bb412ac230d40ed5407821c37"
    }
  ]
}
```

Step 1: Create the Address

```
cardano-cli address \
  --payment-script-file timeLockScript.json \
  --testnet-magic 42 \
  --out-file script.addr
```

Step 2: Construct and Submit a Transaction

```
cardano-cli transaction build-raw \
    --invalid-before 1000 \
    --fee 0 \
    --tx-in <txin of script address> \
    --tx-out <yourspecifiedtxout> \
    --out-file spendScriptTxBody

cardano-cli transaction witness \
  --tx-body-file spendScriptTxBody \
  --script-file timeLockScript.json \
  --testnet-magic 42 \
  --out-file scriptWitness

cardano-cli transaction witness \
  --tx-body-file spendScriptTxBody \
  --signing-key-file <paySignKey> \
  --testnet-magic 42 \
  --out-file keyWitness

cardano-cli transaction assemble \
  --tx-body-file spendScriptTxBody \
  --witness-file scriptWitness \
  --witness-file keyWitness \
  --out-file spendTimeLockTx
```

Step 3: Submit the Transaction

```
cardano-cli transaction submit \
  --tx-file spendTimeLockTx \
  --testnet-magic 42
```

3.5. USING LUCID LIBRARY FOR CARDANO NATIVE SCRIPTS

Working with Cardano native scripts can be made significantly easier and faster by using the Lucid library in JavaScript. This library simplifies many operations that would otherwise require complex command-line interactions. Here, we'll demonstrate how to import a Cardano native script and perform transactions using the Lucid library.

3.5.1. Importing and Using Lucid Library

The Lucid library provides a user-friendly interface for interacting with Cardano, allowing developers to perform tasks quickly and efficiently. Here's an example of how to import a Cardano native script and use it for transactions.

Example: Importing a Native Script and Performing a Transaction

First, we need to import the necessary modules from the Lucid library:

```javascript
import { Lucid, Blockfrost, Constr, Utils } from "https://unpkg.com/lucid-
    cardano@0.9.6/web/mod.js";
```

Next, initialize the Lucid library with Blockfrost and specify the network:

```javascript
const lucid = await Lucid.new(
new Blockfrost("https://cardano-mainnet.blockfrost.io/api/v0", "APIKEYHERE"),
"Mainnet",
);
```

Now, we can define a multisignature script in JSON format and convert it to a native script using Lucid:

```
const multisigScript = lucid.utils.nativeScriptFromJson(
{
"type": "all",
scripts: [
{ type: "sig", keyHash: "1
    c471b31ea0b04c652bd8f76b239aea5f57139bdc5a2b28ab1e69175" },
],
}
);
```

With the multisignature script, we can create the corresponding address:

```
const multisigAddress = lucid.utils.validatorToAddress(multisigScript);
```

Finally, we can construct, sign, and submit a transaction:

```
var tx = await lucid
.newTx()
.payToAddress(multisigAddress, { lovelace: 1000000 }) // Example: sending 1 ADA
.attachMintingPolicy(multisigScript)
.addSignerKey("1c471b31ea0b04c652bd8f76b239aea5f57139bdc5a2b28ab1e69175") //
    Signer key
.complete();

const signedTx = await tx.sign().complete();
const txHash = await signedTx.submit();
console.log(txHash);
```

3.5.2. Advantages of Using Lucid Library

Using the Lucid library for managing Cardano native scripts offers several advantages over traditional command-line methods:

- **Ease of Use**: The Lucid library abstracts away much of the complexity involved in script and transaction management, making it easier for developers to interact with the Cardano blockchain.
- **Speed**: Transactions and script operations can be performed more quickly without the need to manually handle and submit command-line operations.
- **Flexibility**: JavaScript provides a more flexible environment for developing and testing blockchain interactions compared to the command-line interface.
- **Integration**: Lucid can be easily integrated into web applications and other JavaScript-based projects, facilitating broader use cases and seamless user experiences.

3.6. INTERACTING WITH SMART CONTRACTS USING LUCID

In this section we'll see how a transaction including smart contracts on Cardano is done, don't worry if you don't get most of it. Focus on looking at the similarities with using a cardano native script as shown before.

Interacting with smart contracts on the Cardano blockchain using the Lucid library is straightforward and similar to working with native scripts, such as multisignature scripts.

However, smart contracts introduce more complex logic, enabling advanced functionalities. In this section, we'll demonstrate how to use a minting smart contract to create tokens on Cardano using Lucid.

3.6.1. Minting Tokens with a Smart Contract

The process of interacting with a smart contract involves defining the contract's logic and using it in a transaction. The example below illustrates how to mint tokens using a smart contract, showcasing how similar the code structure is to working with native scripts while allowing for more sophisticated operations.

Example: Using a Minting Smart Contract

First, we need to import the necessary modules from the Lucid library and initialize it with Blockfrost:

```
import { Lucid, Blockfrost, toHex, fromHex, Data, Constr, fromText } from "
    https://unpkg.com/lucid-cardano@0.10.0/web/mod.js";

const lucid = await Lucid.new(
new Blockfrost("https://cardano-mainnet.blockfrost.io/api/v0", "APIKEYHERE"),
"Mainnet",
);
```

Next, import the Plutus script data and encode it using CBOR:

```
import data from '/plutus.json' assert { type: "json" };
console.log(data);

import * as cbor from "https://deno.land/x/cbor@v1.4.1/index.js";

const mintingPolicy = {
type: "PlutusV2",
script: toHex(cbor.encode(fromHex(data.validators[1].compiledCode)))
};

const storageScript = {
type: "PlutusV2",
script: toHex(cbor.encode(fromHex(data.validators[0].compiledCode)))
};
```

Enable the Nami wallet and select it with Lucid:

```
const api = await window.cardano.nami.enable();
lucid.selectWallet(api);

const { paymentCredential } = lucid.utils.getAddressDetails(
await lucid.wallet.address(),
);
```

Define the policy ID and the storage address for the minting process:

```
const policyId = lucid.utils.mintingPolicyToId(mintingPolicy);
const storageAddress = lucid.utils.validatorToAddress(storageScript);
```

Prepare the UTXO to be used in the transaction and define the redeemer:

```
let utxoSeed = await lucid.utxosByOutRef([{ txHash: "451
    d8129bb9fce9829906fe32bb7c2a93f40493f3ceeb3b003ae7eb7c8a99f52", outputIndex
    : 4 }]);

const redeemer = Data.to(new Constr(0, []));
```

Define the metadata for the new tokens:

```
const metaData = {
decimals: 6,
description: "The official token of FluidTokens, a leading DeFi ecosystem
    fueled by innovation and community backing.",
logo: "https://fluidtokens.com/fldt.png",
name: "FLDT",
ticker: "FLDT",
website: "https://fluidtokens.com/",
};

console.log(metaData);

Object.keys(metaData)
.sort()
.forEach(function(v, i) {
console.log(v, metaData[v]);
});
```

Create the Datum and build the transaction:

```
const Datum = Data.to(new Constr(0, [Data.fromJson(metaData), 1n, 1n]));

const tx = await lucid
.newTx()
.collectFrom(utxoSeed)
.mintAssets({ [${policyId}0014df10${fromText("FLDT")}]: BigInt(100000000 * Math
    .pow(10, 6)), [${policyId}000643b0${fromText("FLDT")}]: 1n }, redeemer)
.payToContract(storageAddress, { inline: Datum }, { [${policyId}000643b0${
    fromText("FLDT")}]: 1n })
.attachMintingPolicy(mintingPolicy)
.complete();
```

Sign and submit the transaction:

```
const signedTx = await tx.sign().complete();
const txHash = await signedTx.submit();
console.log(txHash);
```

3.6.2. Advantages of Using Smart Contracts

Using smart contracts in Cardano allows for more complex and flexible transaction logic compared to native scripts. Here are some of the advantages:

- **Advanced Logic**: Smart contracts enable sophisticated logic that goes beyond simple conditions like multisignature requirements.
- **Automation**: Automate complex workflows and interactions on the blockchain, reducing the need for manual intervention.
- **Flexibility**: Smart contracts can be tailored to a wide variety of use cases, from DeFi applications to NFTs.

- **Efficiency**: With Lucid, interacting with smart contracts is streamlined, making development faster and more efficient.

By leveraging Lucid, developers can easily integrate advanced smart contract functionality into their Cardano applications, enhancing their capabilities and providing richer user experiences.

3.7. DATUM, REDEEMERS, AND SCRIPT CONTEXT

3.7.1. Datum

The datum is data attached to UTxOs. A datum represents the state of a smart contract and is immutable, although the state of the smart contract can change by spending old UTxOs and creating new ones. The 'e' (extended) in eUTxO comes from the datum. Unlike the Bitcoin UTxO model, which lacks datums and thus has limited capabilities, the extended UTxO model (as used by Cardano and Ergo) provides capabilities comparable to an account-based model while maintaining a safer approach to transactions by avoiding global state mutations.

3.7.2. Redeemers

The redeemer is another piece of data provided with the transaction for script execution. The datum and redeemer intervene at two distinct moments: the datum is set when the output is created (similar to attaching a note to a wall), whereas the redeemer is provided only when spending the output (like handing over a form to an employee). Together, they play crucial roles in the functioning of smart contracts.

3.7.3. Script Context

The majority of the logic in smart contracts involves making assertions about certain properties of the `ScriptContext`. The `ScriptContext` contains valuable information, such as:

- When is the transaction occurring?
- What are the inputs of the transaction?
- What are the outputs of the transaction?

All these details are encapsulated in the `ScriptContext` object, which is passed into the contract as the last argument. Understanding and utilizing the `ScriptContext` is essential for validating transactions. The `ScriptContext` can be visualized as an object with the following properties:

```
Transaction {
  inputs: List<Input>,
  reference_inputs: List<Input>,
  outputs: List<Output>,
  fee: Value,
  mint: MintedValue,
  certificates: List<Certificate>,
  withdrawals: Pairs<StakeCredential, Int>,
  validity_range: ValidityRange,
  extra_signatories: List<Hash<Blake2b_224, VerificationKey>>,
  redeemers: Pairs<ScriptPurpose, Redeemer>,
  datums: Dict<Hash<Blake2b_256, Data>, Data>,
  id: TransactionId,
}
```

This structure highlights the importance of reading and understanding the exact details of the inputs, outputs, time, and signatories of the transaction. Simply having the datum and redeemers is not sufficient to validate a transaction; the full context provided by the `ScriptContext` is essential for comprehensive validation and ensuring the security and correctness of the smart contract execution.

CARDANO SMART CONTRAC LANGUAGES

4. CARDANO SMART CONTRACT LANGUAGES

4.1. INTRODUCTION TO PLUTUS LANGUAGE

> Plutus is the native smart contract language for Cardano. It is a Turing-complete language written in Haskell, and Plutus smart contracts are effectively Haskell programs. By using Plutus, you can be confident in the correct execution of your smart contracts. It draws from modern language research to provide a safe, full-stack programming environment based on Haskell, the leading purely functional programming language.

The Plutus Playground has been recently updated, providing a platform for developing smart contract applications using Haskell. At its core, the Plutus Platform enables developers to write smart contracts in Haskell, a high-level, robust programming language. This setup ensures that users can rely on well-established tooling and libraries without needing to learn a new, proprietary language.

4.1.1. Plutus Tx Compiler

The key technology that facilitates this is Plutus Tx, which acts as a compiler from Haskell to Plutus Core, the language executed on the Cardano blockchain. Provided as a GHC (Glasgow Haskell Compiler) plug-in, Plutus Tx compiles Haskell code into executable files for users' computers and Plutus Core for blockchain execution.

4.1.2. Handling Haskell's Complexity

Haskell, known for its complexity and numerous sophisticated extensions, is simplified through the design of GHC. GHC Core, a straightforward representation of Haskell programs, allows the complex surface language to be desugared after initial type checking. This process lets Plutus Tx operate on GHC Core, benefiting from the extensive Haskell language support.

> "Fortunately, the design of GHC, the primary Haskell compiler, makes this possible. GHC has a very simple representation of Haskell programs called GHC Core. After the initial typechecking phase, all of the complex surface language is desugared away into GHC Core, and the rest of the pipeline doesn't need to know about it. This works for us too: we can operate on GHC Core, and get support for the much larger Haskell surface language for free."

While Haskell's type system is complex, Plutus Tx leverages a subset of it, sufficient for blockchain needs. However, not all Haskell features are supported, particularly those irrelevant or difficult to implement on the blockchain. The compiler provides helpful errors when unsupported features are used.

4.1.3. Compilation Pipeline

The compilation process involves multiple stages, utilizing intermediate languages to break down the tasks. The pipeline includes:

1. GHC: Haskell to GHC Core
2. Plutus Tx compiler: GHC Core to Plutus IR
3. Plutus IR compiler: Plutus IR to Typed Plutus Core
4. Type eraser: Typed Plutus Core to Untyped Plutus Core

Plutus IR, closely aligned with GHC Core, reduces the logic within the plug-in and allows for independent testing of subsequent stages.

4.1.4. Integration with GHC

Plutus Tx integrates into the GHC compilation pipeline through GHC plug-ins, which modify the program during compilation. This integration enables the Plutus Tx compiler to produce Plutus Core, embedded back into the Haskell program as an opaque byte string ready for blockchain transactions.

> "Because we are able to modify the program GHC is compiling, we have an obvious place to put the output of the Plutus Tx compiler, back into the main Haskell program! That's the right place for it, because the rest of the Haskell program is responsible for submitting transactions containing Plutus Core scripts. But from the point of view of the rest of the program, Plutus Core is opaque, so we can get away with just providing it as a blob of bytes ready to go into a transaction."

4.1.5. Runtime Considerations

Dynamic elements of smart contracts, such as participant details or transaction amounts, require runtime variability. This is addressed using type classes like `Typeable` and `Lift`, which facilitate turning Haskell types and values into Plutus Core representations.

4.1.6. References

For more detailed information, visit the Plutus Playground and follow the development on `GitHub`.

4.1.7. Setting Up the Environment

To build the Plutus development environment on an Ubuntu 20.04 VM, follow these steps:

4.1.8. Installing Haskell and Components

1. **Create and start a VM:**

- Create a VM with at least 8 GB of RAM and 100 GB of storage, and select Ubuntu 64-bit.
- Install Ubuntu 20.04 on the VM.

2. **Update and upgrade the system:**

```
sudo apt update
sudo apt upgrade -y
```

3. **Install Haskell:**

```
sudo apt-get install haskell-platform
```

4. **Install `ghcup` and required dependencies:**

```
sudo apt install curl
sudo apt install build-essential curl libffi-dev libffi6 libgmp-dev
libgmp10 libncurses-dev libncurses5 libtinfo5 libsodium-dev
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

Restart your terminal.
Check the Haskell version:

```
ghc --version
cabal --version
```

5. **Clone the Plutus repositories:**

```
sudo apt install git
mkdir cardano
cd cardano
git clone https://github.com/input-output-hk/plutus
git clone https://github.com/input-output-hk/plutus-pioneer-program
```

Run `cabal update` and `cabal build` each time you start on a different section of the homework.

4.1.9. Installing `nix-shell` and Plutus Binaries

1. **Install the cache:**

```
sudo mkdir /etc/nix
```

Create a file named `nix.conf` in `/etc/nix/` with the following content:

```
substituters        = https://hydra.iohk.io https://iohk.cachix.org
trusted-public-keys = hydra.iohk.io:f/Ea+s+dFdN+3Y/
G+FDgSq+a5NEWhJGzdjvKNGv0/EQ=iohk.cachix.org-1:
DpRUyj7h7V830dp/i6Nti+NEO2/nhblbov/8MW7Rqoo=
 cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY=
```

2. **Install `nix`:**

```
curl -L https://nixos.org/nix/install | sh
```

After installation, set up the environment variables as instructed. Typically, this involves running:

```
. /home/(user)/.nix-profile/etc/profile.d/nix.sh
```

You may need to add /.nix-profile/bin to your /etc/environment file and restart the computer.

3. **Build Plutus with nix-shell:**

```
cd ~/cardano/plutus
git checkout 3746610e53654a1167aeb4c6294c6096d16b0502
nix build -f default.nix
plutus.haskell.packages.plutus-core.components.library
```

This process may take a while and may produce a warning about dumping a very large path.
The first run of `nix-shell` will also take some time. The first video in the course will guide you through starting the Plutus Playground server and application.

4.1.10. Writing a Simple Smart Contract in Plutus Tx

Here is a simple example of a smart contract in Plutus Tx. This contract verifies that the number passed in the redeemer is 42.

```haskell
{-# LANGUAGE DataKinds            #-}
{-# LANGUAGE ImportQualifiedPost #-}
{-# LANGUAGE NoImplicitPrelude   #-}
{-# LANGUAGE OverloadedStrings   #-}
{-# LANGUAGE TemplateHaskell     #-}

module FortyTwo where

import qualified Plutus.V2.Ledger.Api as PlutusV2
import           PlutusTx             (BuiltinData, compile)
import           PlutusTx.Builtins    as Builtins (mkI)
import           PlutusTx.Prelude     (otherwise, traceError, (==))
import           Prelude              (IO)
import           Utilities            (writeValidatorToFile)

-- This validator succeeds only if the redeemer is 42
--                 Datum         Redeemer     ScriptContext
mk42Validator :: BuiltinData -> BuiltinData -> BuiltinData -> ()
mk42Validator _ r _
    | r == Builtins.mkI 42 = ()
    | otherwise            = traceError "expected 42"
{-# INLINABLE mk42Validator #-}

validator :: PlutusV2.Validator
validator = PlutusV2.mkValidatorScript $$(PlutusTx.compile [||
    mk42Validator ||])

saveVal :: IO ()
saveVal = writeValidatorToFile "./assets/fortytwo.plutus" validator
```

This contract's only function is to verify that the redeemer is 42. It generates a .plutus file, which is used to generate the address and to unlock the UTXOs in the contract.

4.1.11. Additional Resources

For readers seeking more detailed information on how to code with Plutus Tx, the following resources are highly recommended:

- The **Plutus Pioneer Program documentation** offers an extensive guide on using the Plutus Playground and coding Plutus contracts. You can access the documentation at https://plutus-pioneer-program.readthedocs.io/en/latest/pioneer/week1.html#to-the-playground.
- A series of instructional **YouTube videos** is available, providing a practical overview of the Plutus Playground. You can find the playlist at https://www.youtube.com/playlist?list=PLNEK_Ejlx3x3xFHJJKdyfo9eBOIw-OQDd.

This eBook will focus more on the latest languages and developments for Cardano smart contracts, offering a contemporary perspective on the evolving ecosystem.

4.2. EXPLORING HELIOS LANGUAGE

The Helios language is a functional programming language with a syntax that is notably similar to C. It features a straightforward curly braces syntax and is inspired by languages like Go and Rust. Helios aims to balance simplicity and safety in the development of decentralized applications (dApps) on the Cardano blockchain.

4.2.1. Tenets of Helios

- **Readability Over Writeability:** The language emphasizes code readability, ensuring that code is easy to understand and maintain.
- **Easily Auditable:** Helios is designed to be easily auditable, allowing developers and auditors to quickly spot potential malicious code.
- **Opinionated:** The language provides a single, clear way to accomplish tasks, reducing ambiguity and complexity in code development.

4.2.2. Helios as a JavaScript/TypeScript SDK

Helios serves as a JavaScript/TypeScript SDK for the Cardano blockchain, encompassing everything needed to build dApps. This includes a simple smart contract language tailored for the Cardano ecosystem.

4.2.3. Setup of the Helios Library

The Helios library is platform agnostic and supports various methods for integration:

- **Webpage Script Tag:**

```
<script src="https://helios.hyperion-bt.org/<version>/helios.js"
type="module" crossorigin></script>
```

- **Module with CDN URL:** Helios can be imported as a module using the CDN URL. This is compatible with Deno and most modern browsers:

  ```
  import * as helios from "https://helios.hyperion-bt.org/
  <version>/helios.js"
  ```

  or only the necessary parts:

  ```
  import { Program } from "https://helios.hyperion-bt.org/
  <version>/helios.js"
  ```

  Alternatively, use "helios" as a placeholder for the URL and specify the module URL in an importmap (currently only supported by Chrome):

  ```
  // in your JavaScript file
  import * as helios from "helios"
  // in your HTML file
  <script type="importmap">
      {
          "imports": {
              "helios": "https://helios.hyperion-bt.org/<version>/helios.js"
          }
      }
  </script>
  ```

- **npm:** Install the latest version of the Helios library using npm:

  ```
  $ npm i @hyperionbt/helios
  ```

  Or install a specific version:

  ```
  $ npm i @hyperionbt/helios@<version>
  ```

  In your JavaScript/TypeScript file:

  ```
  import { Program } from "@hyperionbt/helios"
  ```

  Note that installing the Helios library globally is not recommended, as the API is subject to frequent changes.

4.2.4. Running Helios Without Installing Anything

If you prefer not to install Helios locally, you can use the Helios Playground. The Playground allows you to write, compile, and download smart contracts directly in your browser. This is a convenient way to experiment with Helios without needing to set up a local development environment. You can access the Helios Playground at the following URL: https://playground.helios.hyperion-bt.org.

4.2.5. Example of a Helios Smart Contract

Here is a simple example of a Helios smart contract:

```
const mainScript = `
spending picoswap

// Note: each input UTxO must contain some lovelace, so the datum price
// will be a bit higher than the nominal price
// Note: public sales are possible when a buyer isn't specified

struct Datum {
    seller: PubKeyHash
    price:  Value
    buyer:  Option[PubKeyHash]
    nonce:  Int // double satisfaction protection

    func seller_signed(self, tx: Tx) -> Bool {
        tx.is_signed_by(self.seller)
    }

    func buyer_signed(self, tx: Tx) -> Bool {
        self.buyer.switch{
            None    => true,
            s: Some => tx.is_signed_by(some)
        }
    }

    func seller_received_money(self, tx: Tx) -> Bool {
        // protect against double satisfaction exploit
        //by datum tagging the output using a nonce
        tx.value_sent_to_datum(self.seller, self.nonce, false) >= self.price
    }
}

func main(datum: Datum, _, ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx;

    // sellers can do whatever they want with the locked UTxOs
    datum.seller_signed(tx) || (
        // buyers can do whatever they want with the locked UTxOs,
        //as long as the sellers receive their end of the deal
        datum.buyer_signed(tx) &&
        datum.seller_received_money(tx)
    )
}`
```

In the above contract, we define the `Datum` structure. The `seller` field is necessary to identify who should receive the payment. The `price` represents the amount that must be verified as correctly sent. The `buyer` could be either defined or undefined: if defined, only the specified buyer can complete the purchase; otherwise, anyone can unlock the UTxO by paying the correct amount. The `nonce` is used to prevent double satisfaction attacks, a concept we will explore further later.

4.2.6. Further Resources

For more information and resources on Helios, you can visit their Discord server at https://discord.gg/VwyYPh65Um or check out their comprehensive guide at https://www.hyperion-bt.org/helios-book/intro.html. These resources offer additional support and detailed explanations to help you get the most out of Helios.

4.3. AIKEN LANGUAGE: FEATURES AND SYNTAX

4.3.1. Introduction to Aiken

Aiken is a modern programming language and toolchain designed specifically for developing smart contracts on the Cardano blockchain. It draws inspiration from various modern languages, such as Gleam, Rust, and Elm, which are renowned for their friendly error messages and an overall excellent developer experience. Aiken focuses on creating on-chain validator scripts, requiring users to write their off-chain code for generating transactions in other languages such as Rust, Haskell, JavaScript, or Python.

4.3.2. Language Features

Aiken is a purely functional language with static typing and type inference. This means that most of the time, the compiler is smart enough to determine the type of something without requiring user annotation. Aiken allows the creation of custom types resembling records and enums but does not include higher-kinded types or type classes, aiming for simplicity. On-chain scripts are typically small in size and scope compared to other kinds of applications being developed today and do not necessitate as many features as general-purpose languages that must tackle far more complex issues.

4.3.3. Comparison with Plutus

Aiken is considered easier to get started with than Plutus, especially for those who are less familiar with functional languages like Haskell. Similar to Plutus, Aiken scripts are compiled down to the untyped Plutus Core (UPLC).

4.3.4. Setting Up Aiken Environment

4.3.5. Installation Instructions

Manually:

```
From a package manager:
npm install -g @aiken-lang/aiken
```

Homebrew:

```
brew install aiken-lang/tap/aiken
```

From Sources (All platforms):

```
cargo install aiken --version 1.0.29-alpha
```

From Nix flakes (Linux & MacOS only):

```
nix build github:aiken-lang/aiken#aiken
```

4.3.6. Language Server

The Aiken command-line comes with a built-in language server. Configure your language client with the following settings (refer to your language client's instructions):

```
command: aiken lsp
root pattern: aiken.toml
filetype: aiken (.ak)
```

The language server supports a variety of capabilities. For more details, refer to the supported capabilities on the main repository.

4.3.7. Auto-completion

The command-line comes with a few auto-completion scripts for bash, zsh, and fish users. The scripts can be obtained using the 'aiken completion <shell>' command. Install completions in their standard/default locations as follows:

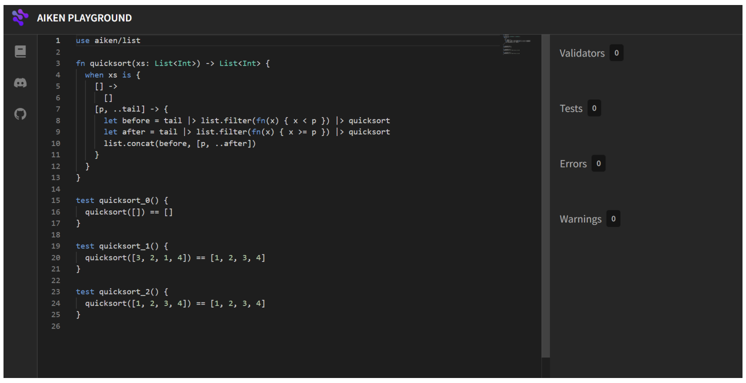
```
aiken completion bash --install

# or, manually

aiken completion bash > /usr/local/share/bash-completion/completions/aiken
source /usr/local/share/bash-completion/completions/aiken
```

4.3.8. Editor Integrations

The following plugins provide syntax highlighting and indentation rules for Aiken:

- **VSCode:** `aiken-lang/vscode-aiken`
- **Vim/Neovim:** `aiken-lang/editor-integration-nvim`
- **Emacs:** `aiken-lang/aiken-mode`

4.3.9. Aiken Playground



```
use aiken/list

fn quicksort(xs: List<Int>) -> List<Int> {
  when xs is {
    [] ->
      []
    [p, ..tail] -> {
      let before = tail |> list.filter(fn(x) { x < p }) |> quicksort
      let after = tail |> list.filter(fn(x) { x >= p }) |> quicksort
      list.concat(before, [p, ..after])
    }
  }
}

test quicksort_0() {
  quicksort([]) == []
}

test quicksort_1() {
  quicksort([3, 2, 1, 4]) == [1, 2, 3, 4]
}

test quicksort_2() {
  quicksort([1, 2, 3, 4]) == [1, 2, 3, 4]
}
```

The Aiken Playground (https://play.aiken-lang.org/) is an online environment where developers can test and experiment with Aiken functions and smart contracts without needing to download and install the software on their local device. Similar to the Helios playground, this tool provides an easy and accessible way to get started with Aiken, allowing users to write, compile, and run Aiken code directly in the browser. It is especially useful for learning and prototyping, providing a convenient platform for exploring Aiken's features and capabilities.

4.3.10. Example of a Smart Contract with Aiken

In Aiken, we need to manually define the import libraries at the very beginning of the contract. Then, similar to Helios, we define the `Redeemer` and `Datum` types. The following smart contract will unlock the UTXO if the signer is the owner and the redeemer is the right user.

```
use aiken/hash.{Blake2b_224, Hash}
use aiken/list
use aiken/transaction.{ScriptContext}
use aiken/transaction/credential.{VerificationKey}

type Datum {
  owner: Hash<Blake2b_224, VerificationKey>,
}

type Redeemer {
  msg: ByteArray,
}

validator {
  fn hello_world(datum: Datum, redeemer: Redeemer, context: ScriptContext) -> Bool {
    let must_say_hello =
      redeemer.msg == "Hello, World!"

    let must_be_signed =
      list.has(context.transaction.extra_signatories, datum.owner)

    must_say_hello && must_be_signed
  }
}
```

4.4. OPSHIN LANGUAGE: CONCEPTS AND USAGE

Opshin is an implementation of smart contracts for Cardano which are written in a strict subset of valid Python. The general philosophy of this project is to write a compiler that ensures the following:

- If the program compiles, then:
  - It is a valid Python program.
  - The output running it with Python is the same as running it on-chain.

4.4.1. Why Opshin?

- **100% valid Python:** Leverage the existing tool stack for Python, including syntax highlighting, linting, debugging, unit-testing, property-based testing, and verification.
- **Intuitive:** Just like Python.
- **Flexible:** Imperative, functional, the way you want it.
- **Efficient & Secure:** Static type inference ensures strict typing and optimized code.

Opshin is a pythonic language for writing smart contracts on the Cardano blockchain. The goal of Opshin is to reduce the barrier of entry in smart contract development on Cardano. Opshin is a strict subset of Python, meaning anyone who knows Python can get up to speed with Opshin quickly.

4.4.2. Setting Up the Environment

Check out the **OpShin Book** for an introduction to this tool and detailed guidance on writing smart contracts. This section outlines the basic usage of the tool.

4.4.3. Installation

Install Python 3.8, 3.9, 3.10, or 3.11. Then run:

```
python3 -m pip install opshin
```

4.4.4. Example of a Smart Contract

4.4.5. Example Validator - Gift Contract

In this simple example, we'll write a gift contract that will allow a user to create a gift UTXO that can be spent by:

1. The creator cancelling the gift and spending the UTXO.
2. The recipient claiming the gift and spending the UTXO.

```python
# gift.py

# The Opshin prelude contains a lot of useful types and functions
from opshin.prelude import *
```

```python

# Custom Datum
@dataclass()
class GiftDatum(PlutusData):
    # The public key hash of the gift creator.
    # Used for cancelling the gift and refunding the creator (1).
    creator_pubkeyhash: bytes

    # The public key hash of the gift recipient.
    # Used by the recipient for collecting the gift (2).
    recipient_pubkeyhash: bytes

def validator(datum: GiftDatum, redeemer: None, context: ScriptContext) -> None:
    # Check that we are indeed spending a UTxO
    assert isinstance(context.purpose, Spending), "Wrong type of script invocation"

    # Confirm the creator signed the transaction in scenario (1).
    creator_is_cancelling_gift = datum.creator_pubkeyhash in context.tx_info.signatories

    # Confirm the recipient signed the transaction in scenario (2).
    recipient_is_collecting_gift = datum.recipient_pubkeyhash in context.tx_info.signatories

    assert creator_is_cancelling_gift or recipient_is_collecting_gift, "Required signature missing"
```

[redacted]

This might be a bit to take in, especially the logic for checking the signatures. The most important part is to see the parameters and the return type, as well as the assert statements actually controlling the validation. For more details, refer to the **OpShin Book**.

4.5. PLU-TS: UNDERSTANDING THE BASICS

Plu-ts is a library designed for building Cardano dApps in an efficient and developer-friendly way. It is composed of two main parts:

- **plu-ts/onchain:** An eDSL (embedded Domain Specific Language) that leverages TypeScript as the host language, designed to generate efficient Smart Contracts.
- **plu-ts/offchain:** A set of classes and functions that allow reuse of onchain types.

4.5.1. Design Principles

Plu-ts was designed with the following goals in mind, in order of importance:

- **Smart Contract efficiency**
- **Developer experience**
- **Reduced script size**
- **Readability**

For more information, see the **Plu-ts Book**.

4.5.2. Setting Up the Environment

From npm:

```
npm install @harmoniclabs/plu-ts
```

NPM: NPM is the package manager used by NodeJS. You can install Node and NPM from the **NodeJS website**.

From source:

```
git clone https://github.com/HarmonicLabs/plu-ts
cd plu-ts
npm run build
```

The dist Folder: The library is then available in the `dist` folder. You can move the directory where you need it.

4.5.3. Quick Start

First, create a new directory where to build your project:

```
mkdir my-pluts-project
cd my-pluts-project
```

Then initialize your Node project with npm:

```
npm init
```

Install TypeScript and the TypeScript compiler `tsc` if it is not already available globally:

```
npm install --save-dev typescript
```

Finally, install Plu-ts:

```
npm install @harmoniclabs/plu-ts
```

4.5.4. Example of a Smart Contract

4.5.5. The Contract

In this example, we'll write a contract that expects a `MyDatum`, a `MyRedeemer`, and finally a `PScriptContext` to validate a transaction.

```
import { Address, bool, compile, makeValidator, PaymentCredentials, pBool, pfn, Script, ScriptType, V2 } from "@harmoniclabs/plu-ts";
import MyDatum from "./MyDatum";
import MyRedeemer from "./MyRedeemer";

export const contract = pfn([
    MyDatum.type,
    MyRedeemer.type,
    V2.PScriptContext.type
],  bool)
(( datum, redeemer, ctx ) =>
    // always succeeds
    pBool( true )
);

export const untypedValidator = makeValidator( contract );
export const compiledContract = compile( untypedValidator );
export const script = new Script(
    ScriptType.PlutusV2,
    compiledContract
);

export const scriptMainnetAddr = new Address(
    "mainnet",
    new PaymentCredentials(
        "script",
        script.hash
    )
);

export const scriptTestnetAddr = new Address(
    "testnet",
    new PaymentCredentials(
        "script",
        script.hash.clone()
    )
);

export default contract;
```

This contract expects a `MyDatum`, a `MyRedeemer`, and a `PScriptContext` to validate a transaction.

Custom Datum and Redeemer

`MyDatum` and `MyRedeemer` are types defined by us in `src/MyDatum/index.ts` and `src/MyRedeemer/inde` respectively.

```
import { int, pstruct } from "@harmoniclabs/plu-ts";

// modify the Datum as you prefer
const MyDatum = pstruct({
    Num: {
        number: int
    },
    NoDatum: {}
});
```

```

export default MyDatum;
```

```
import { pstruct } from "@harmoniclabs/plu-ts";

// modify the Redeemer as you prefer
const MyRedeemer = pstruct({
    Option1: {},
    Option2: {}
});

export default MyRedeemer;
```

Entry Point

Finally, the contract is used in `src/index.ts`, which is our entry point.

```
import { script } from "./contract";

console.log("validator compiled successfully! \n");
console.log(
    JSON.stringify(
        script.toJson(),
        undefined,
        2
    )
);
```

This index file imports the script from `src/contract.ts` and prints it out in JSON form. In this example, we see something different; instead of writing everything in the same file, we split it into several files to increase readability.

Also, note that the current contract always returns `true` as output, therefore it will always unlock the UTXO.

For more details, refer to the **Plu-ts Book**.

The Plu-ts contract above is divided into the following key files:

- `src/contract.ts` - Contains the main validator function and the script logic.
- `src/MyDatum/index.ts` - Defines the data structure for the Datum.
- `src/MyRedeemer/index.ts` - Defines the data structure for the Redeemer.
- `src/index.ts` - Serves as the entry point, importing and displaying the compiled script.

This organization enhances readability and separates concerns, making it easier for developers to manage and understand the contract's different components.

In the provided example, the contract always returns `true` as the output. This means that the contract will always validate successfully, effectively always unlocking the UTXO. While this makes the contract straightforward, it also means that the contract does not perform any meaningful validation or logic checks.

SMART CONTRACT RUNTIME

5. SMART CONTRACT RUNTIME AND EXECUTION

5.1. SMART CONTRACT RUNTIME AND EXECUTION

This part has been updated to be Chang compatible. The code here is already compatible with DReps, the new Aiken version, and Plutus v3.

5.1.1. Overview of Cardano's Smart Contract Execution Model

Smart contracts on Cardano are simple constructs based on **validator scripts**. These scripts define the logic or rules to be enforced by Cardano nodes when a transaction attempts to move a UTXO locked inside the script's address.

Validator scripts can access the **transaction context** (e.g., who signed it, and which assets are sent to/from where) and the **datum** of the locked UTXO being moved, allowing the creation of complex contracts. For instance:

> With smart contracts, we can add conditions to stake delegation, withdraw Cardano rewards from the protocol, or create minting policies with more dynamic conditions than regular ones.

Validator scripts are executed with three main arguments:

- **Datum:** Attached to the output locked by the script, often carrying state.
- **Redeemer:** Attached to the spending input, typically providing an input to the script. For example, a validator can apply the redeemer to the datum and verify it matches the output UTXO datum.
- **Context:** Contains transaction-level information, used to assert properties like "Bob signed it."

Transaction context properties:

Property	Description
inputs	Outputs to be spent.
reference inputs	Inputs used for reference only, not spent.
outputs	New outputs created by the transaction.
fees	Transaction fees.
minted value	Minted or burned value.
certificates	Digest of certificates in the transaction.
withdrawals	Used to withdraw rewards from the stake pool.
valid range	Time range in which the transaction is valid.
signatories	List of transaction signatures.
redeemers	Data used as input to the script.
id	Transaction identification.

5.1.2. Understanding the Transaction Context Table

Inputs: These represent UTXOs being *spent* in the transaction. In the UTXO model, every transaction produces outputs, which in turn become inputs for future transactions. Understanding this flow is key to interpreting inputs and outputs.

Outputs: These are the new UTXOs created by the transaction, ready to be used as inputs in subsequent transactions.

Fees: The amount of lovelace spent for transaction execution. This value is predictable and depends on the transaction size. Fees can often be optimized.

Minted Value: This indicates any minting or burning of tokens that occurs in the transaction.

Certificates: Information about stake operations. For example:

- Registering a stake key.
- Delegating to a DRep.

Withdrawals: Rewards withdrawn from stake keys. For example, if you use a wallet like Lace, which allows delegation to multiple pools, you may see multiple withdrawals in one transaction.

Valid Range: A time frame during which the transaction is valid. This is useful for ensuring a transaction only executes within specific boundaries.

Signatories: A list of hashes representing who signed the transaction. This is essential for multi-signature transactions.

Redeemers: A list of redeemers used by the contracts executed in the transaction.

ID: The transaction hash, uniquely identifying the transaction.

During runtime, each validator checks its *datum*, *redeemer*, and the *context* (or local transaction state).

Important exercise: Imagine a transaction that:

- Spends **three different UTXOs** from the same contract.
- Withdraws staking rewards from a stake key contract.
- Mints **two different tokens** under separate contract policies.

Questions:

1. How many unique contracts are executed?
2. How many datums are present?
3. How many redeemers?

Answer: After a blank page—*Take your time; don't rush!*

Real answer:

1. 4
2. 3
3. 6

Did you get them? Let's try to understand why. Spending inputs from a contract, even if it's coming from the same contract is treated like a unique execution. Therefore for each input coming from contract A, we will have it's own Datum and it's own redeemer.

Withdraw and minting contract do not have datums, they have redeemers. You can mint or withdraw from the same contract only once, therefore the logic behind these contracts must be more flexible to allow different logic inside the same execution.

It will be easier to understand at the end of this chapter, you need to take home this:

> SPEND contracts are executed once for each input, while minting and withdrawals contracts are executed once for each transaction

5.2. TRANSACTION VERIFICATION AND SCRIPT VALIDATION

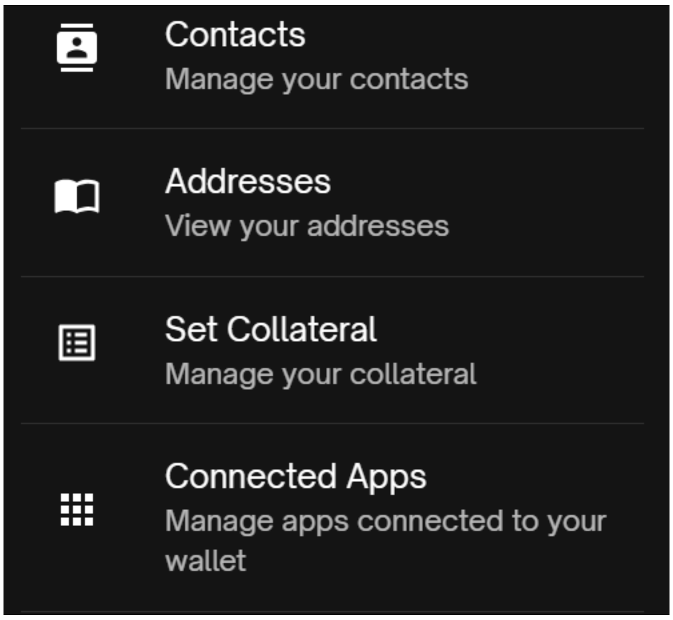
To understand transaction validation on Cardano, it is crucial to recognize that even if a script validates a transaction, the transaction might still fail due to the network's robust two-phase validation mechanism. This section explores this process and the role of collateral in ensuring successful smart contract execution.

Two-Phase Validation Mechanism

Cardano employs a two-phase validation scheme to minimize uncompensated work for nodes, ensuring efficiency and security:

- **Phase 1: Structural Validation**
  - Checks if the transaction is correctly constructed.
  - Ensures that the transaction can pay its processing fee.
  - If Phase 1 fails, the transaction is immediately discarded without running any scripts.
- **Phase 2: Script Execution**
  - Executes the scripts included in the transaction.
  - If a script fails, the transaction fails, and collateral is used to compensate nodes.

The Role of Collateral



Collateral ensures that nodes are compensated for their work if Phase 2 validation fails. It acts as a monetary guarantee, encouraging careful design and testing of smart contracts. Key details about collateral:

- **Collateral Inputs:** The collateral amount is determined by the total balance of UTXOs marked as collateral inputs.
- **Safety for Honest Users:** Collateral is not collected if a transaction succeeds or is invalid at Phase 1.
- **Deterministic Costs:** Cardano's deterministic design allows users to calculate execution costs and collateral requirements in advance, unlike Ethereum where gas costs can vary based on network activity.
- **Vasil Upgrade Improvement:** Developers can specify a change address for script collateral, ensuring only the required amount is taken if a script fails, with the remainder returned to the specified address.

Importance of Collateral

Without collateral, malicious actors could exploit the network by flooding it with invalid transactions at little cost. By requiring collateral, Cardano ensures:

- Transactions calling non-native (Phase 2) smart contracts include sufficient collateral to cover potential failure costs.
- Denial of Service (DoS) attacks become prohibitively expensive.
- Honest users never lose their collateral as long as transactions are valid and successful.

Technical Details

Phase 2 scripts on Cardano can perform arbitrary computations and require a budget of execution units (ExUnits) to quantify resource usage. This budget is included in the transaction fee calculation. Collateral provides additional safeguards:

- **Multi-Signature Scripts:** Introduced in the Shelley era, Phase 1 scripts follow deterministic ledger rules, enabling straightforward cost assessment.
- **ExUnit Budgeting:** Phase 2 scripts require a resource budget for metrics like memory usage and execution steps, ensuring fair cost allocation.

The Cardano testnet provides a safe environment with free test ADA, enabling developers to rigorously test smart contracts before deploying them on the mainnet. This ensures that transactions and scripts function correctly under real-world conditions.

5.3. DEBUGGING AND TROUBLESHOOTING SMART CONTRACTS

Debugging smart contracts can be challenging, but interacting directly with the blockchain is not always necessary. Thanks to tools like Aiken and Gastronomy, developers can efficiently test and debug their smart contracts in a controlled environment.

Debugging with Aiken

Aiken offers first-class support for unit tests and property-based tests, allowing developers to write and execute tests directly in Aiken without deploying to the blockchain. The toolkit (‘aiken check‘) parses tests, runs them, and displays detailed reports.

UNIT TESTS

Unit tests in Aiken are written using the ‘test‘ keyword. A test is a named function that takes no arguments and returns a boolean. It is valid (i.e., it passes) if it returns ‘True‘.

```
test foo() {
  1 + 1 == 2
}
```

Unit tests can call functions and use constants, and they execute on the same virtual machine used for on-chain contracts. This ensures that tests mirror the production environment.

EXAMPLE UNIT TESTS

```
lib/example.ak
fn add_one(n: Int) -> Int {
  n + 1
}

test add_one_1() {
  add_one(0) == 1
}
```

```
test add_one_2() {
  add_one(-42) == -41
}
```

Running ‘aiken check‘ generates a report grouping tests by module and displaying the memory and CPU execution units needed for each test. This report can also be used as a benchmark to compare execution costs of different approaches.

Tests in Aiken can be as complex as necessary, without the execution limits imposed on on-chain scripts.

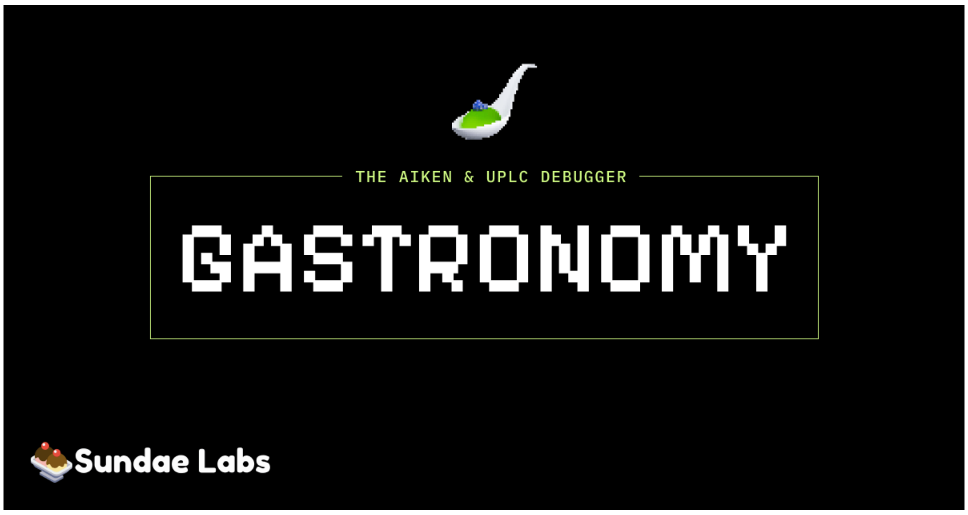
TRACE AND DEBUGGING

Aiken supports debugging via CBOR diagnostic traces. Developers can use ‘trace‘ to print diagnostic data (e.g., integers or byte arrays) in the event of a contract failure. If the contract succeeds, no output is shown.

```
// An Int becomes a CBOR int
trace cbor.diagnostic(42)

// A ByteArray becomes a CBOR bytestring
trace cbor.diagnostic("foo")
```

This feature provides valuable insights during debugging by simulating breakpoints.

Advanced Debugging with Gastronomy



Gastronomy, developed by Sundae Labs, is a powerful UPLC debugger designed to aid in diagnosing failed scripts based on error codes. It allows developers to step through script execution with ease.

FEATURES

- Stores the state of the machine at every execution step.
- Allows stepping forward and backward to analyze script behavior.
- Provides a user-friendly interface for debugging.

QUICK START

CLI Tool:

```
gastronomy-cli run test_data/fibonacci.uplc 03
N - Advance to the next step
P - Rewind to the previous step
Q - Quit
```

GUI Tool: Simply run ‘gastronomy‘ to launch the graphical interface.

CONFIGURATION

Gastronomy can be configured using environment variables or a ‘.gastronomyrc.toml‘ file in the home directory. Example configuration:

Setting	Environment Variable	Description
blockfrost.key	BLOCKFROST_KEY	The API key to use when querying Block

By leveraging Aiken and Gastronomy, developers can thoroughly test and debug their smart contracts, ensuring reliability and efficiency before deployment.

5.4. TX OPTIMIZATION TECHNIQUES FOR EFFICIENT EXECUTION

In recent years, several effective techniques have been discovered to optimize smart contracts on Cardano. Many of these can be found in the repository at **Aiken Design Patterns**. Here, we will discuss two game-changing techniques: the ”Withdraw 0 Trick” and ”Parametric Scripts.”

Withdraw 0 Trick

The ”Withdraw 0 Trick” is particularly useful for validators that need to handle multiple inputs efficiently. By splitting logic into two distinct parts—a minimal spending logic and an arbitrary withdrawal logic—scripts can be made significantly more efficient.

HOW IT WORKS

Withdraw scripts are executed only once per transaction, whereas spend validators are executed once for every input originating from the same smart contract. For example, if purchasing multiple NFTs from a marketplace, the transaction's execution logic grows proportionally to the number of NFTs.

Using the "Withdraw 0 Trick," this logic is split:

- The spend validator checks that a "withdraw 0" operation is executed.
- The withdrawal logic is handled independently in the "withdraw 0" contract, decoupling it from the number of NFTs.

This approach reduces complexity, ensures scalability, and maintains elegance in design.

IMPLEMENTATION DETAILS

The module offers two key functions for spending endpoints:

- **spend** – Traverses both the withdrawals and redeemers fields, validating against both the redeemer and the withdrawal quantity.
- **spend_minimal** – Traverses only the withdrawals, ideal when no validation is needed on the staking script's redeemer or withdrawal quantity.

Additionally, the **withdraw** function unwraps the staking credential and provides the underlying hash, facilitating minimal execution logic.

Parametric Scripts

Parametric scripts are another crucial optimization technique, significantly enhancing execution efficiency by predefining key parameters within the script itself.

WHY USE PARAMETRIC SCRIPTS?

Certain contracts require specific parameters, such as a spend contract that must identify an NFT with a matching policy script hash. Without predefined parameters, the contract would consume excessive CPU resources to compute its own script hash. By using parametric scripts, the contract already "knows" the required values, saving computation time and resources.

PRACTICAL EXAMPLE

Consider a minting script that restricts token destinations to instances of a specific spending script parameterized by user wallets. Each wallet results in a different script address, making verification challenging. Using parametric scripts, the minting script can:

- Validate instances as the result of applying specific parameters to a parameterized script.
- Ensure robust asset flow control.

ON-CHAIN VALIDATION REQUIREMENTS

To validate parameterized scripts on-chain, the following restrictions apply:

- Script parameters must have constant lengths (achieved via hashing).
- Redeemers must supply resolved values of parameters for each transaction.
- Dependent scripts must include CBOR bytes of instances before and after parameter application.
- Logic wrapping ensures a single occurrence of each parameter.

IMPLEMENTATION STEPS

Define parameterized scripts and generate instances with dummy data to obtain required prefix and postfix values. Use these values in your target script. For detailed examples, refer to `validators/apply-params-example.ak`.

These optimization techniques provide a solid foundation for efficient, scalable smart contract execution on Cardano.

WRITE YOUR FIRST SMART CONTRACT

6. GETTING STARTED WITH CARDANO SMART CONTRACT DEVELOPMENT

6.1. KEY CONCEPTS AND TERMINOLOGY

In this chapter, we will get hands-on experience by writing a smart contract in Aiken and the corresponding off-chain code, typically implemented in the frontend using Mesh. Finally, we will submit a transaction and interact with it.

At the end of the chapter, you will also find exercises to test your understanding and skills.

6.1.1. Key Concepts

Here is a list of key concepts and terminology that will help you:

- **Collateral**: A specific UTxO required to interact with smart contracts. It ensures that nodes are compensated in case the contract validation fails.
- **Script Hash**: A unique hash that identifies the smart contract, allowing transactions to reference it securely.
- **Locked UTxO**: A transaction output that holds funds, NFTs, or tokens locked within a contract. These can only be spent if the contract validation logic passes.
- **CIP69 Contract**: A contract where the datum is optional. It behaves like a user, allowing it to receive funds. However, every UTxO associated with this contract must execute the validation logic to be spendable.
- **Multipurpose Script**: A versatile script that can perform multiple functions, such as locking funds, minting NFTs, and more.

6.2. WRITING YOUR FIRST SMART CONTRACT

Let's write and execute a smart contract on Cardano in 10 minutes. Yes, you read that right.

You can find code supporting this tutorial on Aiken's main repository.

6.2.1. Covered in this tutorial

- Writing a basic Aiken validator;
- Writing & running tests with Aiken;
- Troubleshooting smart contracts.

When encountering an unfamiliar syntax or concept, do not hesitate to refer to the language-tour for details and extra examples.

6.2.2. Pre-requisites

We'll use Aiken to write the script, so make sure the command-line tool is installed already. Otherwise, refer to the installation instructions.

6.2.3. Scaffolding

First, let's create a new Aiken project:

```
aiken new aiken-lang/hello-world
cd hello-world
```

This command scaffolds an Aiken project. In particular, it creates a `lib` and `validators` folder in which you can put Aiken source files.

```
./hello-world

   README.md
   aiken.toml
   lib
   validators
```

6.2.4. Using the standard library

We'll use the standard library for writing our validator. Fortunately, `aiken new` did automatically add the standard library to our `aiken.toml` for us. It should look roughly like that:

```
aiken.toml
name = "aiken-lang/hello-world"
version = "0.0.0"
license  = "Apache-2.0"
description = "Aiken contracts for project 'aiken-lang/hello-world'"

[repository]
user = 'aiken-lang'
project = 'hello-world'
platform = 'github'

[[dependencies]]
name = "aiken-lang/stdlib"
version = "v2"
source = "github"
```

Now, running `aiken check`, we should see dependencies being downloaded. That shouldn't take long.

```
  aiken check
    Compiling aiken-lang/hello-world 1.0.0 (examples/hello-world/)
    Resolving aiken-lang/hello-world
      Fetched 1 package in 0.01s from cache
    Compiling aiken-lang/stdlib v2 (/Users/aiken/Documents/aiken-lang/hello-
world/build/packages/aiken-lang-stdlib)
      Summary 0 errors, 0 warnings
```

6.2.5. Our First Validator

Let's write our first validator as `validators/hello_world.ak`:

```
validators/hello_world.ak
use aiken/collection/list
use aiken/crypto.{VerificationKeyHash}
use cardano/transaction.{OutputReference, Transaction}

pub type Datum {
  owner: VerificationKeyHash,
}

pub type Redeemer {
  msg: ByteArray,
}

validator hello_world {
  spend(
    datum: Option<Datum>,
    redeemer: Redeemer,
    _own_ref: OutputReference,
    self: Transaction,
  ) {
    expect Some(Datum { owner }) = datum
    let must_say_hello = redeemer.msg == "Hello, World!"
    let must_be_signed = list.has(self.extra_signatories, owner)
    must_say_hello && must_be_signed
  }
}
```

Our first validator is rudimentary, yet there's already a lot to say about it.

It looks for a verification key hash (`owner`) in the datum and a message (`msg`) in the redeemer. Remember that, in the eUTxO model, the datum is set when locking funds in the contract and can be seen as configuration. Here, we'll indicate the owner of the contract and require a signature from them to unlock funds—very much like it already works on a typical non-script address.

Moreover, because there's no ”Hello, World!” without a proper ”Hello, World!”, our little contract also demands this very message, as a UTF-8-encoded byte array, to be passed as redeemer (i.e. when spending from the contract).

It's now time to build our first contract!

```
aiken build
```

This command generates a CIP-0057 Plutus blueprint as `plutus.json` at the root of your project. This blueprint describes your on-chain contract and its binary interface. In particular, it contains the generated on-chain code that will be executed by the ledger and a hash of your validator(s) that can be used to construct addresses.

This format is framework-agnostic and is meant to facilitate interoperability between tools. The blueprint is fully integrated into Aiken, which can automatically generate it based on your type definitions and comments.

6.2.6. Let's see the validator in action!

- Interact with a validator on the Preview network;
- Using Mesh through Blockfrost;
- Getting test funds from the Cardano Faucet;
- Using web explorers such as CardanoScan.

6.2.7. Pre-requisites

We assume that you have followed the "Hello, World!"'s First steps and thus, have Aiken installed and ready-to-use. We will also use Mesh, so make sure you have your dev environment ready for some JavaScript!.

You can install Mesh and set up the project as follows:

```
npm init -y
npm install @meshsdk/core tsx
```

6.2.8. Getting Funds

For this tutorial, we will use the validator we built in First steps. Yet, before moving on, we'll need some funds, and a public/private key pair to hold them. We can generate a private key and an address using MeshWallet.

```
generate-credentials.ts
import { MeshWallet } from '@meshsdk/core';
import fs from 'node:fs';

const secret_key = MeshWallet.brew(true) as string;

fs.writeFileSync('me.sk', secret_key);

const wallet = new MeshWallet({
  networkId: 0,
  key: {
    type: 'root',
    bech32: secret_key,
  },
});

fs.writeFileSync('me.addr', wallet.getUnusedAddresses()[0]);
```

You can run the instructions above via:

```
npx tsx generate-credentials.ts
```

Now, we can head to the Cardano faucet to get some funds on the preview network to our newly created address (inside `me.addr`).

Make sure to select ”Preview Testnet” as the network.

Using CardanoScan, we can watch for the faucet sending some ADA our way. This should be pretty fast (a couple of seconds).

6.3. DEPLOYING AND INTERACTING WITH SMART CONTRACTS

No need to deploy like in ethereum, as soon as we send a transaction in the contract we are ready to go.

Now that we have some funds, we can lock them in our newly created contract. We'll use Blockfrost Provider to construct and submit our transaction through Blockfrost.

This is only one example of a possible setup using tools we love. For more tools, make sure to check out the Cardano Developer Portal!

6.3.1. Setup

First, we set up Mesh with Blockfrost as a provider. This will allow us to let Mesh handle transaction building for us, which includes managing changes. It also gives us a direct way to submit the transaction later on.

```
common.ts
import fs from "node:fs";
import {
  BlockfrostProvider,
  MeshTxBuilder,
  MeshWallet,
  serializePlutusScript,
  UTxO,
} from "@meshsdk/core";
import { applyParamsToScript } from "@meshsdk/core-csl";

const blockchainProvider = new BlockfrostProvider(process.env.BLOCKFROST_PROJECT_ID)

// wallet for signing transactions
```

```
export const wallet = new MeshWallet({
  networkId: 0,
  fetcher: blockchainProvider,
  submitter: blockchainProvider,
  key: {
    type: "root",
    bech32: fs.readFileSync("me.sk").toString(),
  },
});
```

Note that the highlighted line above looks for an environment variable named `BLOCKFROST_PROJECT_ID` which its value must be set to your Blockfrost project id. You can define a new environment variable in your terminal by running (in the same session you're also executing the script!):

```
export BLOCKFROST_PROJECT_ID=preview...
```

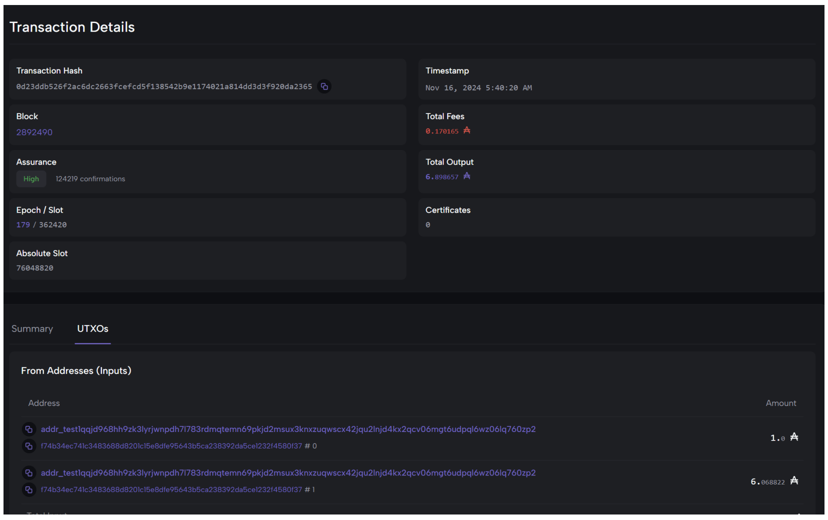
Replace `preview...` with your actual project id.

6.3.2. Locking Funds into the Contract

Now that we can read our validator, we can make our first transaction to lock funds into the contract. The datum must match the representation expected by the validator (and as specified in the blueprint), so this is a constructor with a single field that is a byte array.

```
lock.ts
import { Asset, deserializeAddress, mConStr0 } from "@meshsdk/core";
import { getScript, getTxBuilder, wallet } from "./common";

async function main() {
  // these are the assets we want to lock into the contract
  const lockAmount = 10_000_000;

  // get a valid address for the contract
  const validatorAddr = deserializeAddress(
    "addr_test1qxygsw9r9rt5q0jx8m6lmcjlsrn2ck7c7tpekl9wzyj8nvc70wwdlcs78s7ntljtm6xqf
  );

  // lock the funds!
  const { tx, requiredSigners } = await getTxBuilder().build({
    assets: [new Asset("lovelace", lockAmount)],
    recipients: [
      {
        address: validatorAddr,
        datum: { owner: wallet.getUnusedAddresses()[0] },
      },
    ],
  });
  await wallet.signAndSubmit(tx, requiredSigners);
}
```

```
main();
```


Transaction Details

Transaction Hash
0d23ddb526f2ac6dc2663fcefcd5f138542b9e1174021a814dd3d3f920da2365

Timestamp
Nov 16, 2024 5:40:20 AM

Block
2892490

Total Fees
0.170165 ₳

Assurance
High 124219 confirmations

Total Output
6.898657 ₳

Epoch / Slot
179 / 362420

Certificates
0

Absolute Slot
76048820

Summary UTXOs

From Addresses (Inputs)

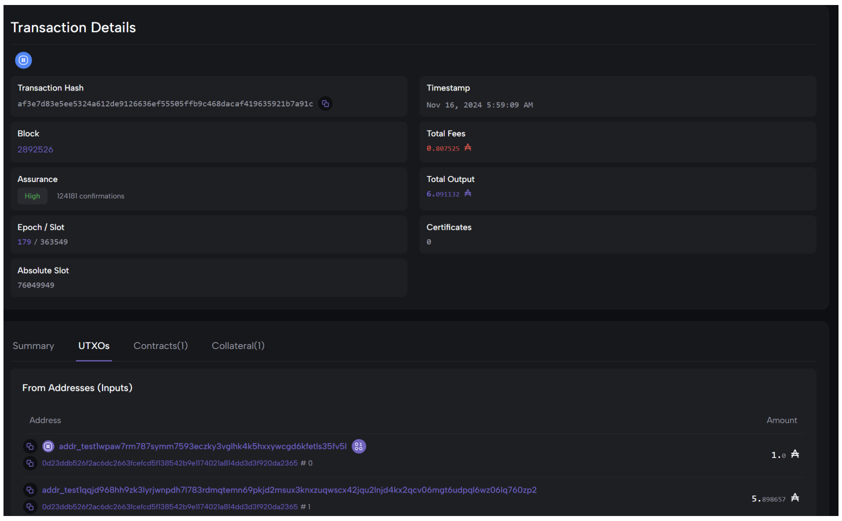
Address	Amount
addr_test1qqjd968hh9zk3lyrjwnpdh7l783rdmqtemn69pkjd2msux3knxzuqwscx42jqu2lnjd4kx2qcv06mgt6udpql6wz06lq760zp2 f74b34ec741c3483688d8201c15e8dfe95643b5ca238392da5ce1232f4580f37 # 0	1.0 ₳
addr_test1qqjd968hh9zk3lyrjwnpdh7l783rdmqtemn69pkjd2msux3knxzuqwscx42jqu2lnjd4kx2qcv06mgt6udpql6wz06lq760zp2 f74b34ec741c3483688d8201c15e8dfe95643b5ca238392da5ce1232f4580f37 # 1	6.068822 ₳

Once again, you may check the transaction on CardanoScan.

6.3.3. Spending from the Contract

With our funds now locked in the contract, we can now spend them. First, we'll need a transaction that spends the funds, passing in the correct redeemer with the value `Hello, World!`.

```
spend.ts
import { getTxBuilder, wallet } from "./common";

async function main() {
  const tx = await getTxBuilder().build({
    assets: [{ asset: "lovelace", amount: 10_000_000 }],
    recipients: [
      {
        address: wallet.getUnusedAddresses()[0],
        redeemer: { msg: "Hello, World!" },
      },
    ],
  });
  await wallet.signAndSubmit(tx);
}

main();
```



Transaction Details

Transaction Hash
af3e7d83e5ee5324a612de9126636ef55505ffb9c468dacaf419635921b7a91c

Timestamp
Nov 16, 2024 5:59:09 AM

Block
2892526

Total Fees
0.807525 ₳

Assurance
High 124181 confirmations

Total Output
6.091132 ₳

Epoch / Slot
179 / 363549

Certificates
0

Absolute Slot
76049949

Summary | UTXOs | Contracts(1) | Collateral(1)

From Addresses (Inputs)

Address	Amount
addr_test1wpaw7rm787symm7593eczky3vglhk4k5hxxywcgd6kfetls35fv5l 0d23ddb526f2ac6dc2663fcefcd5f138542b9e1174021a814dd3d3f920da2365 # 0	1.0 ₳
addr_test1qqjd968hh9zk3lyrjwnpdh7l783rdmqtemn69pkjd2msux3knxzuqwscx42jqu2lnjd4kx2qcv06mgt6udpql6wz06lq760zp2 0d23ddb526f2ac6dc2663fcefcd5f138542b9e1174021a814dd3d3f920da2365 # 1	5.898657 ₳

Once the transaction is complete, go ahead and look for it on CardanoScan again to confirm the transaction has been processed.

You can look for it using the transaction hash: *af3e7d83e5ee5324a612de9126636ef55505ffb9c468dacaf419635921b7a91c*

6.4. BEST PRACTICES FOR SECURE AND ROBUST SMART CONTRACT

Auditing should be mandatory before any contract goes live. While open-source code is beneficial for transparency and trust, it is not sufficient on its own. Audits help identify potential vulnerabilities and ensure that the contract behaves as expected under various conditions.

A well-conducted audit verifies that the code meets security standards and mitigates risks, including potential exploits or misuse. Moreover, it's important to have a robust testing framework in place to ensure the contract functions correctly in both expected and edge cases.

Following these best practices ensures the safety of assets and enhances the reliability of smart contracts in decentralized applications.

Here is a list of checks that you should perform on your contract before going live:

- **Dust attack:** The contract may be used by an attacker who adds additional tokens, making the UTXO locked forever.
- **Double spending:** Ensure that the contract logic is enforced so that it is only executed for one input at a time, preventing multiple inputs from being withdrawn from the contract.
- **Stake key:** Verify that funds are sent to the correct contract under the right stake key. This ensures that the true owner receives staking rewards.
- **Mint-burn:** Confirm that the mint or burn functions are validating the correct policy, and not just using a random asset name.

6.5. EXERCISES TO TEST YOUR SMART CONTRACT SKILLS

EXERCISE 3: Make all tutorials available in the following links:
- https://aiken-lang.org/example--vesting
- https://aiken-lang.org/example--gift-card

EXERCISE 4: Create a minting policy that allows the following:
- Mint the token `always` to everyone, at any time.
- Mint the token `onetime` only to the project owner, and only once forever.
- Mint the token `fenix` only if both `always` and `onetime` have been burned.
- Any other token name is forbidden.

EXERCISE 5: Create a smart wallet account where users can receive payments. The datum is optional, as it can function like a regular wallet. The rules are as follows:
- If the datum is empty, the owner can spend the tokens.
- If the datum is not empty, it indicates the time when the user can withdraw it, as a POSIX timestamp.
- The owner can be a user, a smart contract, or a multisig, allowing multisignatures and smart contracts to use it as well.

EXERCISE 6: Implement a contract where a user can lock 1 NFT and generate an amount of N tokens as a result. The conditions are as follows:
- The only way to unlock the NFT is to burn a number M of tokens in the transaction.
- The unit of the NFT, the number N of tokens generated, and the number M of tokens to burn are parameters of the contract.

BUILDING A MARKETPLAC SMART CONTRACT

7. BUILDING A MARKETPLACE

7.1. DEFINING REQUIREMENTS FOR A MARKETPLACE CONTRACT

In this section, we define the essential requirements for our marketplace contract. The contract should include the following checks:

- **Owner's Actions:** An owner can cancel or create a listing in the contract.
- **Buyer Limitation:** A buyer can purchase only one NFT at a time to avoid double-spending, a potential issue to be explained later.
- **NFT Transfer:** Ensure that the NFT is correctly transferred to the buyer upon purchase.
- **Seller Payment:** The contract must ensure that the seller receives the correct payment.
- **Marketplace Payment:** The marketplace fee must be paid correctly.
- **Royalties:** If royalties are present, they must be paid to the designated address.

The contract will store the following information for each NFT listing:

- `nSeller :: PubKeyHash the address to pay`
- `nPrice :: Integer the amount`
- `nCurrency :: CurrencySymbol the policy of the token`
- `nToken :: TokenName the asset name of the token`
- `nRoyalty :: PubKeyHash the address of the royalty recipient`
- `nRoyaltyPercent :: Integer the percentage of royalty`

7.2. IMPLEMENTING MARKETPLACE CONTRACTS

To implement marketplace contracts, we will begin by analyzing the contract implementation in **PlutusTX**. We can refer to the [JPG Store contract repository](https://github.com/jpg-store/current-jpg-store-contracts/blob/main/src/Market/Onchain.hs).

In the repository, we can see how the marketplace contract is implemented in `Onchain.hs`, which defines the main logic and transaction validation.

7.2.1. PlutusTX Code for Marketplace Contract

We provide the following code implementation for the market contract in Plutus:

```
{-# LANGUAGE DataKinds                   #-}
{-# LANGUAGE NoImplicitPrelude           #-}
{-# LANGUAGE OverloadedStrings           #-}
{-# LANGUAGE TemplateHaskell             #-}
{-# LANGUAGE TypeApplications            #-}
{-# LANGUAGE TypeFamilies                #-}
{-# LANGUAGE DerivingStrategies          #-}
{-# LANGUAGE NumericUnderscores          #-}

module Market.Onchain
    ( apiBuyScript
    , buyScriptAsShortBs
```

```haskell
    , typedBuyValidator
    , Sale
    , buyValidator
    , nftDatum
    ) where

import qualified Data.ByteString.Lazy     as LB
import qualified Data.ByteString.Short    as SBS
import           Codec.Serialise          ( serialise )

import           Cardano.Api.Shelley      (PlutusScript (..),
   PlutusScriptV1)
import qualified PlutusTx
import PlutusTx.Prelude
import PlutusTx.Ratio
import Ledger
    ( TokenName,
      PubKeyHash(..),
      CurrencySymbol,
      DatumHash,
      Datum(..),
      txOutDatum,
      txSignedBy,
      ScriptContext(scriptContextTxInfo),
      TxInfo,
      TxInInfo(..),
      txInfoInputs,
      txOutDatumHash,
      Validator,
      TxOut,
      txInfoSignatories,
      unValidatorScript, valuePaidTo )
import qualified Ledger.Typed.Scripts      as Scripts
import qualified Plutus.V1.Ledger.Scripts as Plutus
import           Ledger.Value              as Value ( valueOf )
import qualified Plutus.V1.Ledger.Ada as Ada (fromValue, Ada (
   getLovelace))

import           Market.Types               (NFTSale(..), SaleAction
   (..))

{-# INLINABLE nftDatum #-}
nftDatum :: TxOut -> (DatumHash -> Maybe Datum) -> Maybe NFTSale
nftDatum o f = do
    dh <- txOutDatum o
    Datum d <- f dh
    PlutusTx.fromBuiltinData d

{-# INLINABLE ensureOnlyOneScriptInput #-}
ensureOnlyOneScriptInput :: ScriptContext -> Bool
ensureOnlyOneScriptInput ctx =
  let
    isScriptInput :: TxInInfo -> Bool
    isScriptInput i = case (txOutDatumHash . txInInfoResolved) i of
      Nothing -> False
      Just _ -> True
  in if length (filter isScriptInput $ txInfoInputs (
    scriptContextTxInfo ctx)) <= 1
       then True
       else False

```

```
{-# INLINABLE mkBuyValidator #-}
mkBuyValidator :: PubKeyHash -> NFTSale -> SaleAction -> ScriptContext
    -> Bool
mkBuyValidator pkh nfts r ctx =
    case r of
        Buy    -> traceIfFalse "NFT not sent to buyer" checkNFTOut &&
                  traceIfFalse "Seller not paid" checkSellerOut &&
                  traceIfFalse "Fee not paid" checkMarketplaceFee &&
                  traceIfFalse "Royalities not paid" checkRoyaltyFee &&
                  traceIfFalse "More than one script input"
    onlyOneScriptInput
        Close -> traceIfFalse "No rights to perform this action"
    checkCloser
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    tn :: TokenName
    tn = nToken nfts

    cs :: CurrencySymbol
    cs = nCurrency nfts

    seller :: PubKeyHash
    seller = nSeller nfts

    sig :: PubKeyHash
    sig = case txInfoSignatories info of
            [pubKeyHash] -> pubKeyHash
            _ -> error ()

    price :: Integer
    price = nPrice nfts

    checkNFTOut :: Bool
    checkNFTOut = valueOf (valuePaidTo info sig) cs tn == 1

    marketplacePercent :: Integer
    marketplacePercent = 20

    marketplaceFee :: Ratio Integer
    marketplaceFee = max (1_000_000 % 1) (marketplacePercent % 1000 *
    fromInteger price)

    checkMarketplaceFee :: Bool
    checkMarketplaceFee
      = fromInteger (Ada.getLovelace (Ada.fromValue (valuePaidTo info
    pkh)))
      >= marketplaceFee

    royaltyFee :: Ratio Integer
    royaltyFee = max (1_000_000 % 1) (nRoyaltyPercent nfts % 1000 *
    fromInteger price)

    checkRoyaltyFee :: Bool
    checkRoyaltyFee = if nRoyaltyPercent nfts > 0
      then fromInteger (Ada.getLovelace (Ada.fromValue (valuePaidTo
    info $ nRoyalty nfts))) >= royaltyFee
      else True

    checkSellerOut :: Bool
```

```haskell
    checkSellerOut
      =  fromInteger (Ada.getLovelace (Ada.fromValue (valuePaidTo info
   seller)))
      >= ((fromInteger price - marketplaceFee) - royaltyFee)

    checkCloser :: Bool
    checkCloser = txSignedBy info seller

    onlyOneScriptInput :: Bool
    onlyOneScriptInput = ensureOnlyOneScriptInput ctx


data Sale
instance Scripts.ValidatorTypes Sale where
    type instance DatumType Sale    = NFTSale
    type instance RedeemerType Sale = SaleAction


typedBuyValidator :: PubKeyHash -> Scripts.TypedValidator Sale
typedBuyValidator pkh = Scripts.mkTypedValidator @Sale
    ($$(PlutusTx.compile [|| mkBuyValidator ||]) `PlutusTx.applyCode`
   PlutusTx.liftCode pkh)
    $$(PlutusTx.compile [|| wrap ||])
  where
    wrap = Scripts.wrapValidator @NFTSale @SaleAction


buyValidator :: PubKeyHash -> Validator
buyValidator = Scripts.validatorScript . typedBuyValidator

buyScript :: PubKeyHash -> Plutus.Script
buyScript = Ledger.unValidatorScript . buyValidator

buyScriptAsShortBs :: PubKeyHash -> SBS.ShortByteString
buyScriptAsShortBs = SBS.toShort . LB.toStrict . serialise . buyScript

apiBuyScript :: PubKeyHash -> PlutusScript PlutusScriptV1
apiBuyScript = PlutusScriptSerialised . buyScriptAsShortBs
```

7.2.2. Types.hs Code for Marketplace Contract

Here is the code for `types.hs`, which defines the datum structures used in the marketplace contract:

```haskell
{-# LANGUAGE DeriveAnyClass        #-}
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE DeriveGeneric         #-}
{-# LANGUAGE ScopedTypeVariables   #-}
{-# LANGUAGE TemplateHaskell       #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeOperators         #-}

module Market.Types
    ( NFTSale (..)
    , SaleAction (..)
    , SaleSchema
    , StartParams (..)
    , BuyParams (..)
    )
    where

import           Data.Aeson                   (ToJSON, FromJSON)
import           GHC.Generics                 (Generic)
import           Prelude                      (Show (..))
import qualified Prelude                      as Pr

import           Schema                       (ToSchema)
import qualified PlutusTx
import           PlutusTx.Prelude             as Plutus ( Eq(..), (&&),
   Integer )
import           Ledger                       ( TokenName, CurrencySymbol,
    PubKeyHash )
import           Plutus.Contract              ( Endpoint, type (.\/) )

-- This is the datum type, carrying the previous validator params
data NFTSale = NFTSale
    { nSeller          :: !PubKeyHash
    , nPrice           :: !Integer
    , nCurrency        :: !CurrencySymbol
    , nToken           :: !TokenName
    , nRoyalty :: !PubKeyHash
    , nRoyaltyPercent :: !Integer
    } deriving (Pr.Eq, Pr.Ord, Show, Generic, ToJSON, FromJSON,
   ToSchema)

instance Eq NFTSale where
    {-# INLINABLE (==) #-}
    a == b = (nSeller    a == nSeller    b) &&
             (nPrice     a == nPrice     b) &&
             (nCurrency  a == nCurrency  b) &&
             (nToken     a == nToken     b) &&
             (nRoyalty a == nRoyalty b) &&
             (nRoyaltyPercent a == nRoyaltyPercent b)

PlutusTx.makeIsDataIndexed ''NFTSale [('NFTSale, 0)]
PlutusTx.makeLift ''NFTSale


data SaleAction = Buy | Close
    deriving Show

PlutusTx.makeIsDataIndexed ''SaleAction [('Buy, 0), ('Close, 1)]
PlutusTx.makeLift ''SaleAction


-- We define two different params for the two endpoints start and buy
   with the minimal info needed.
-- Therefore the user doesn't have to provide more that what's needed
   to execute the said action.
{- For StartParams we ommit the seller
    because we automatically input the address of the wallet running
   the startSale enpoint

   For BuyParams we ommit seller and price
    because we can read that in datum which can be obtained with just
   cs and tn of the sold token -}

```

```
data BuyParams = BuyParams
    { bCs :: CurrencySymbol
    , bTn :: TokenName
    } deriving (Pr.Eq, Pr.Ord, Show, Generic, ToJSON, FromJSON,
   ToSchema)


data StartParams = StartParams
    { sPrice :: Integer
    , sCs    :: CurrencySymbol
    , sTn    :: TokenName
    , sRoyaltyAddress :: !PubKeyHash
    , sRoyaltyPercent :: !Integer
    } deriving (Pr.Eq, Pr.Ord, Show, Generic, ToJSON, FromJSON,
   ToSchema)


type SaleSchema = Endpoint "close" BuyParams .\/ Endpoint "buy"
   BuyParams .\/ Endpoint "start" StartParams
```

[redacted]

The process is straightforward: when a seller wishes to list an NFT for sale, they send the NFT to the marketplace contract address and attach the correct datum. This allows the seller to retain the ability to cancel the listing later if they change their mind.

A buyer can purchase the NFT by fulfilling the following conditions:

- Paying the correct amount to the seller.
- Paying the applicable fees to the marketplace.
- Paying royalties, if any are set.
- Purchasing only one NFT at a time.

You don't need to fully grasp every detail of the contract at this moment. This book is here to guide you through the learning process, not to overwhelm you with technicalities. Let's simplify the contract and break it down into more accessible terms with other programming languages.

7.3. MARKETPLACE IN HELIOS

To implement the marketplace contract in Helios, we can follow a similar structure to the Plutus contract but using the Helios language. In this subsection, we'll define the contract's key components, including the datum, actions, and the main program.

DEFINING THE CONTRACT NAME

First, we define the name of the contract as follows:

```
spending marketplace
```

[redacted]

DEFINING THE DATUM

Next, we define the `Datum`, which holds the key data for each NFT sale listing. The datum will contain the seller's address, the price of the NFT, the currency, token name, the royalty recipient, and the royalty percentage:

```
    struct Datum{
        nSeller: PubKeyHash
        nPrice: Int
        nCurrency:AssetClass
        nRoyalty: PubKeyHash
        nRoyaltyPercent: Int
    }
```

[redacted]

DEFINING THE REDEEMER

We define the possible actions (or `Redeemer`) that users can perform. The two actions in our marketplace contract are `Cancel` and `Buy`, which control the logic for canceling a listing or purchasing the NFT:

```
enum Redeemer{
    Cancel
    Buy
}
```

[redacted]

MAIN PROGRAM LOGIC

Finally, we define the main program for the marketplace contract. The `main` function takes the `Datum`, the `Redeemer`, and the script context as inputs. The main logic will validate the transaction based on the action chosen by the redeemer. Here's the initial version of the contract:

```
func main(datum: Datum, redeemer: Redeemer, ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx;

    redeemer.switch{
        Cancel => {
            true
        },
        Buy => {
            false
        }
    }
}
```

[redacted]

At this point, we have defined the basic structure of the marketplace contract in Helios. The next step will be to replace `true` and `false` with the actual logic for checking that the correct payments are made, the buyer gets the NFT, and the appropriate parties (seller, marketplace, and royalty recipient) are paid. This basic flow sets the foundation for a fully functional smart contract in Helios.

In this section, we will add the first validation checks to our marketplace contract: one for the `Cancel` action and multiple checks for the `Buy` action.

CANCEL FLOW

The only way a listing can be canceled is if the transaction is signed by the owner of the NFT. We define this rule in the `Cancel` case, where the transaction must be signed by the seller's address, as indicated by the `nSeller` field in the `Datum`:

```
Cancel => {
    tx.is_signed_by(datum.nSeller)
}
```

[redacted]

This ensures that only the owner of the NFT (the seller) can cancel the listing.

BUY FLOW

The `Buy` action has several checks to ensure that the transaction is valid:

- The buyer must send the correct amount of money to the seller, as specified in the `nPrice` field.
- A marketplace fee of 2% is collected and sent to the designated marketplace fee address.
- If royalties are set, the appropriate royalty percentage is sent to the royalty recipient address.
- The contract checks that only one input with a datum is present to prevent multiple NFTs from being purchased in a single transaction.

Here's the complete Helios code with the checks for both `Cancel` and `Buy` actions:

```
spending marketplace

struct Datum{
    nSeller: PubKeyHash
    nPrice: Int
    nCurrency: AssetClass
    nRoyalty: PubKeyHash
    nRoyaltyPercent: Int
}

enum Redeemer{
    Cancel
    Buy
}

```

```
func main(datum: Datum, redeemer: Redeemer, ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx;
    marketplaceFeeAddress: PubKeyHash = PubKeyHash::new(#87beb2237d63ff03bc842950a88959d69abbe29e8adea8176b32fd69);
    datums: Int = tx.inputs.map((x: TxInput) -> Int { x.output.datum.switch{None => 0, else => 1} }).fold((sum: Int, x: Int) -> Int { sum + x }, 0);

    redeemer.switch {
        Cancel => {
            tx.is_signed_by(datum.nSeller)
        },
        Buy => {
            tx.value_sent_to(datum.nSeller).contains(Value::new(datum.nCurrency, datum.nPrice)) &&
            tx.value_sent_to(marketplaceFeeAddress).contains(Value::new(datum.nCurrency, datum.nPrice * 20 / 1000)) &&
            tx.value_sent_to(datum.nRoyalty).contains(Value::new(datum.nCurrency, datum.nPrice * datum.nRoyaltyPercent / 1000)) &&
            datums == 1
        }
    }
}
```

EXPLANATION OF THE CODE

In the code above:

- `Cancel`: The transaction must be signed by the seller (the owner of the NFT), ensuring that only the owner can cancel the listing.
- `Buy`:
  - The buyer must pay the correct price to the seller.
  - The marketplace receives a 2% fee, calculated as `datum.nPrice * 20 / 1000`.
  - If royalties are set, the specified royalty percentage is paid to the royalty address.
  - The `datums == 1` check ensures that only one NFT can be purchased at a time by counting how many inputs contain a datum.

These checks ensure that the transaction is valid for both canceling and buying actions. The marketplace fee and royalties are handled, and the buyer is restricted to purchasing only one NFT per transaction.

You can find the code at [link](https://www.hyperion-bt.org/helios-playground/?share=a0e1ed7521aba9
and export it to use it.

7.4. OPSHIN MARKETPLACE

In this subsection, we will explore how to implement a marketplace contract using the `OpShin` framework. OpShin is a Python-based framework for developing smart contracts, and it provides a more user-friendly development experience compared to `PlutusTx`. You can find an example of this contract on their

DATUM DEFINITION

In OpShin, the datum for the marketplace contract is defined as follows. The `Listing` datum includes the price of the listing, the vendor (the owner of the listed item), and the owner who can withdraw the listing:

```python
@dataclass
class Listing(PlutusData):
    CONSTR_ID = 0
    # Price of the listing in lovelace
    price: int
    # the owner of the listed object
    vendor: Address
    # whoever is allowed to withdraw the listing
    owner: PubKeyHash
```

[redacted]

ACTIONS DEFINITION

OpShin defines the possible actions that users can perform in the marketplace using Python dataclasses. The actions include `Buy`, which allows the buyer to purchase the listed item, and `Unlist`, which enables the owner to remove the listing:

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

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

ListingAction = Union[Buy, Unlist]
```

[redacted]

SMART CONTRACT IMPLEMENTATION

Now we can define the smart contract logic in OpShin, which includes several auxiliary functions to check payments, prevent double spending, and verify that the correct parties are signing the transaction.

Here are the auxiliary functions used in the contract:

- **check_paid**: This function ensures that the correct amount of lovelace has been paid to the vendor.
- **check_single_utxo_spent**: This function prevents double spending by ensuring that only one UTxO is spent from the contract address.

- **check_owner_signed**: This function checks that the owner of the listing has signed the transaction when unlisting.

The contract is implemented as follows:

```python
def check_paid(txouts: List[TxOut], addr: Address, price: int) -> None:
    """Check that the correct amount has been paid to the vendor (or more)"""
    res = False
    for txo in txouts:
        if txo.value.get(b"", {b"": 0}).get(b"", 0) >= price and txo.address == addr:
            res = True
    assert res, "Did not send required amount of lovelace to vendor"

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
    for txi in txins:
        if txi.resolved.address == addr:
            count += 1
    assert count == 1, f"Only 1 contract utxo allowed but found {count}"


def check_owner_signed(signatories: List[PubKeyHash], owner: PubKeyHash) -> None:
    assert (
        owner in signatories
    ), f"Owner did not sign transaction, requires {owner.hex()} but got {[s.hex() for s in signatories]}"

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

    check_single_utxo_spent(tx_info.inputs, own_addr)
    # It is recommended to explicitly check all options with isinstance for user input
    if isinstance(redeemer, Buy):
        check_paid(tx_info.outputs, datum.vendor, datum.price)
    elif isinstance(redeemer, Unlist):
        check_owner_signed(tx_info.signatories, datum.owner)
    else:
        assert False, "Wrong redeemer"
```

[redacted]

EXPLANATION OF THE CODE

In the smart contract:

- `check_paid`: Verifies that the correct amount of lovelace has been paid to the vendor.
- `check_single_utxo_spent`: Ensures that only one UTxO is spent, preventing double spending.
- `check_owner_signed`: Ensures that the owner has signed the transaction for the unlisting action.

The contract logic starts by checking that the transaction purpose is `Spending`, resolving the spent UTxO, and ensuring that only one contract UTxO is unlocked. Depending on the redeemer type, it either checks that the payment has been made correctly (for the `Buy` action) or that the owner has signed the transaction (for the `Unlist` action).

CONCLUSION

This example demonstrates how OpShin provides a more user-friendly approach to building smart contracts using Python. By leveraging Python's simplicity and the `PlutusData` class, we can easily define the datum, actions, and contract logic. This approach is ideal for developers who prefer working with Python and want to avoid the complexities of `PlutusTx`.

7.5. PLU-TS MARKETPLACE

In this subsection, we will explore how to implement a marketplace contract using `Plu-ts`, a TypeScript-based framework for writing Cardano smart contracts. With `Plu-ts`, we can write Cardano smart contracts in a more familiar programming language like TypeScript, which can make the development process easier for web developers.

You can find the `Plu-ts` framework and more examples in the official [Plu-ts GitHub repository](https://github.com/harmoniclabs/plu-ts).

DATUM DEFINITION

In `Plu-ts`, we define the datum structure for the marketplace contract as follows. The `MarketDatum` includes the seller's public key hash and the price (in lovelace) of the listed item:

```
import { PPubKeyHash, int, pstruct } from "@harmoniclabs/plu-ts";

const MarketDatum = pstruct({
    MarketDatum: {
        seller: PPubKeyHash.type,
        lovelace: int // price in lovelace
    }
});

export default MarketDatum;
```

[redacted]

This structure defines the key data that will be used in the contract: the seller's public key hash and the price of the item in lovelace.

CONTRACT DEFINITION

Now, we define the marketplace contract using the `pfn` function in `Plu-ts`. This function allows us to write the contract logic. In the contract, we will check the transaction purpose and the seller's signature, ensuring that the correct parties are involved.

Here's the main contract definition:

```
/* imports */

import { PPubKeyHash, int, pstruct } from "@harmoniclabs/plu-ts";

const MarketDatum = pstruct({
    MarketDatum: {
        seller: PPubKeyHash.type,
        lovelace: int // price in lovelace
    }
});

const plovelaces = phoist(
    pfn([ PValue.type ], int)
    ( value => value.head.snd.head.snd )
);

export default MarketDatum;

/* imports */

export const contract = pfn([ PScriptContext.type ], unit)(
    ( {redeemer, tx, purpose} ) => {

        const maybeDatum = plet(
            pmatch(purpose)
                .onSpending(({ datum }) => datum)
                ._(_ => perror(PMaybe(data).type))
        );

        const datum = plet( punsafeConvertType( maybeDatum.unwrap, MarketDatum.type ) );

        const seller = plet( datum.seller );

        // use `peq` instead of `eq` to pass the function as argument or save it as variable
        const isSeller = plet( datum.seller.peq );

        const signedBySeller = tx.signatories.some( isSeller );

        const sellerGetsPaid = plet(
          tx.outputs.some( out =>
            isSeller.$( out.address.credential.hash )
            .and( plovelaces.$( out.value ).eq( datum.lovelace ) )
          )
        );
        return passert.$(
            (ptraceIfFalse.$(pdelay(pStr("Error in signedBySeller"))).$(signedBySeller))
            .or( ptraceIfFalse.$(pdelay(pStr("Seller not paid"))).$( sellerGetsPaid ) )
        );
    }
);

/* other code */
```

[redacted]

EXPLANATION OF THE CODE

In this contract:

- `maybeDatum`: This variable attempts to extract the datum from the transaction based on the script's purpose (in this case, spending).
- `datum`: The datum is then converted to the correct `MarketDatum` type.
- `signedBySeller`: This checks whether the transaction has been signed by the seller (using the public key hash in the datum).
- `passert`: This is used to assert conditions within the contract, ensuring that the seller has signed the transaction and that the seller has been paid the correct amount.
- `ptraceIfFalse`: If any assertion fails, a trace message is printed to indicate what went wrong (e.g., "Error in signedBySeller" or "Seller not paid").

The contract logic checks that:

- The transaction has been signed by the seller.
- The seller has been paid the correct amount of lovelace as indicated in the `MarketDatum`.

ADDITIONAL FEATURES

This contract provides a simple structure for handling the sale of items, ensuring that only the seller can sign the transaction and that they are paid correctly. You can easily extend this contract to include additional features, such as marketplace fees, royalty payments, or checks for multiple buyers.

CONCLUSION

The `Plu-ts` framework provides a way to write Cardano smart contracts using TypeScript, making it easier for web developers familiar with haskell/TypeScript to build blockchain applications. With simple constructs like `pfn`, `pstruct`, and `passert`, `Plu-ts` provides a clean and readable way to define and enforce contract logic on the Cardano blockchain.

This contract has been written in plutus v3, how do we know? The Datum is optional and we see that the contracts try to read it before running.

7.6. AIKEN MARKETPLACE

In this subsection, we will explore how to implement a marketplace contract using Aiken, a smart contract language built on top of the Cardano blockchain. This contract is the latest version from JPG Store and is available in their [GitHub repository](https://github.com/jpg-store/contracts-v3/blob/main/validators/ask.ak).

Aiken offers a clean and expressive syntax for building Cardano smart contracts. Below, we will break down the key parts of this marketplace contract, focusing on the Datum, Redeemer, and the Validator.

DATUM DEFINITION

In Aiken, the datum for the marketplace listing contains two primary fields: 1. `payouts`: A list of addresses that need to be paid, such as the seller and royalties. 2. `owner`: The owner's verification key hash, which determines who can update or cancel the listing.

Here is the `Datum` type definition:

```
type Datum {
  payouts: List<Payout>,  -- List of payouts (royalty, seller, etc.)
  owner: VerificationKeyHash,  -- Owner of the listing
}
```

[redacted]

Additionally, the `Payout` type represents a payment that needs to be made, with an `address` and the `amount_lovelace` that needs to be paid:

```
pub type Payout {
  address: Address,
  amount_lovelace: Int,
}
```

[redacted]

This setup means that the price of the listed NFT is always in ADA (lovelace), and the contract enforces payments to the specified addresses.

REDEEMER DEFINITION

The contract allows two types of actions through the `Redeemer`: 1. `Buy`: The buyer can purchase the listed item. The `payout_outputs_offset` specifies the starting point of the outputs containing the payouts. 2. `WithdrawOrUpdate`: The owner can update or cancel the listing. This action requires the owner's signature.

Here is the `Redeemer` definition:

```
type Redeemer {
  Buy { payout_outputs_offset: Int }  -- Buy action with payout offset
  WithdrawOrUpdate  -- Cancel or update the listing
}
```

[redacted]

CONTRACT FLOW

The flow of the contract is as follows:

- **Buy**: When a user buys the NFT, the contract checks the payouts to the correct addresses and calculates the marketplace fee if necessary. If any of the authorized users have signed the transaction, a discount may be applied.
- **WithdrawOrUpdate**: If the owner wants to cancel or update the listing, the contract checks whether the transaction is signed by the owner of the NFT.

Here's the core logic of the contract:

```
validator {
  fn spend(datum: Datum, redeemer: Redeemer, ctx: ScriptContext) -> Bool {
    let ScriptContext { transaction, purpose } = ctx
    let Transaction { outputs, extra_signatories, .. } = transaction

    when redeemer is {
      Buy { payout_outputs_offset } -> {
        expect Spend(out_ref) = purpose

        let datum_tag = out_ref
          |> serialise_data
          |> blake2b_256
          |> InlineDatum

        let Datum { payouts, .. } = datum
        let payout_outputs = find_payout_outputs(outputs, payout_outputs_offset)

        let can_have_discount = constants.authorizers()
          |> list.any(fn(authorizer) { list.has(extra_signatories, authorizer) })

        if can_have_discount {
          check_payouts(payout_outputs, payouts, datum_tag) > 0
        } else {
          expect [marketplace_output, ..rest_outputs] = payout_outputs

          let payouts_sum = check_payouts(rest_outputs, payouts, NoDatum)
          let marketplace_fee = payouts_sum * 50 / 49 / 50

          check_marketplace_payout(marketplace_output, marketplace_fee, datum_tag)
        }
      }

      WithdrawOrUpdate ->
        list.has(extra_signatories, datum.owner)
    }
  }
}
```

[redacted]

EXPLANATION OF THE CODE

In the contract:

- **Buy Action**:
  - The contract expects a `Spend` purpose, indicating that the transaction is a purchase.
  - The contract checks the payouts by using `find_payout_outputs` to locate the relevant outputs and then verifying that the correct amounts are paid.
  - If authorized users have signed the transaction, a discount can be applied to the marketplace fee.
  - If no discount is applied, the marketplace fee is calculated and verified.
- **WithdrawOrUpdate Action**:
  - The contract verifies that the transaction is signed by the owner of the listing, allowing them to update or cancel the listing.

DISCOUNT HANDLING

One interesting feature of this contract is the ability to apply a discount to the marketplace fee. This is done by checking if any of the authorizers (presumably JPG Store's trusted signers) have signed the transaction. If so, the contract assumes that JPG Store has correctly calculated the fee off-chain and applies the discount:

```
let can_have_discount = constants.authorizers()
  |> list.any(fn(authorizer) { list.has(extra_signatories, authorizer)
    })
```

[redacted]

CONCLUSION

This Aiken marketplace contract is a powerful and flexible solution for managing NFT listings on the Cardano blockchain. By using Aiken's expressive syntax, the contract efficiently handles buy and update actions while ensuring proper payout distribution. The ability to allow discounts based on JPG Store's off-chain calculations is also a unique feature that makes this contract more adaptable to real-world usage.

With Aiken, Cardano developers can write clean, readable smart contracts that interact seamlessly with Cardano's blockchain, while also offering advanced features like dynamic fee handling and marketplace control.

7.7. TESTING AND DEPLOYING MARKETPLACE CONTRACTS

Now that we've defined and explained the marketplace contract using the Aiken language, it's time to interact with it. This process involves obtaining the contract address and interacting with the contract, which can be done with various tools and libraries.

7.7.1. Getting the Marketplace Address

To begin interacting with the marketplace contract, we first need to obtain the address of the deployed contract. This is crucial for sending transactions that interact with the contract, such as listing or delisting NFTs. The address can be obtained by using the Plutus bytecode, which is stored in the `plutus.json` file.

In the case of the marketplace contract provided earlier, the contract's compiled code is stored in the `plutus.json` file, which contains all the necessary information to interact with the contract. You can find an example of this file from the JPG Store repository on GitHub [here](https://github.com/jpg-store/contracts-v3/blob/main/plutus.json). This file contains the contract's bytecode and other details required to interact with it.

The `plutus.json` file looks like this:

```
{
  "validators": [
    {
      "name": "ask_validator",
      "compiledCode": "<compiled contract bytecode>",
      "isMultiAsset": false
    }
  ]
}
```

From this file, we can extract the compiled bytecode to deploy the contract on the Cardano blockchain.

7.7.2. Using MeshJS to Build the Frontend

To interact with the Cardano blockchain and deploy the marketplace contract, we will use MeshJS. MeshJS is a powerful library designed for interacting with Cardano-based smart contracts. It allows us to build a frontend interface to interact with the marketplace contract, enabling features like listing NFTs, delisting them, and managing transactions.

MeshJS provides an easy-to-use framework to connect to Cardano's blockchain, deploy smart contracts, and manage transactions. By integrating MeshJS into our frontend, we can perform tasks such as:

- Connecting to the Cardano blockchain.
- Sending transactions to deploy or interact with the marketplace contract.
- Managing wallet connections and user interactions.
- Signing and submitting transactions to the blockchain.

In the following sections, we'll demonstrate how to use MeshJS to list and delist NFTs in the marketplace, as well as perform a purchase.

Listing an NFT

The first part of the code is used by the seller to list an NFT for sale and set the price. The seller locks the NFT and specifies the amount of ADA (lovelace) that the item is priced at. This is achieved by sending the assets (NFT and the specified ADA price) to the contract address.

Here's how the listing works:

```
import { Asset, deserializeAddress, mConStr0 } from "@meshsdk/core";
import { getScript, getTxBuilder, wallet } from "./common";

async function main() {
  // These are the assets we want to lock into the contract
  const assets: Asset[] = [
    {
      unit: "lovelace",
      quantity: "2000000", // minADA attached
    },
    {
      unit: "NFT_UNIT",
      quantity: "1", // nft to list
    },
  ];

  // Get utxo and wallet address
  const utxos = await wallet.getUtxos();
  const walletAddress = (await wallet.getUsedAddresses())[0];

  const { scriptAddr } = getScript();

  // Hash of the public key of the wallet, to be used in the datum
  const signerHash = deserializeAddress(walletAddress).pubKeyHash;

  // Build transaction with MeshTxBuilder
  const txBuilder = getTxBuilder();
  await txBuilder
    .txOut(scriptAddr, assets) // Send assets to the script address
    .txOutInlineDatumValue(mConStr0([[],signerHash])) //let's start by setting only the owner and no payout list
    .changeAddress(walletAddress) // Send change back to the wallet address
    .selectUtxosFrom(utxos)
    .complete();
  const unsignedTx = txBuilder.txHex;

  const signedTx = await wallet.signTx(unsignedTx);
  const txHash = await wallet.submitTx(signedTx);
  console.log(`NFT locked into the contract at Tx ID: ${txHash}`);
}

main();
```

[redacted]

In this example, the seller is locking an NFT and setting the price. The datum contains the seller's public key hash, and the transaction ensures the correct amount of ADA is transferred to the contract.

Delisting or Purchasing an NFT

The second part is used for delisting or purchasing an NFT. When a buyer wants to purchase the NFT or the seller wants to remove it from the marketplace, they must interact with the contract to either complete the purchase or delist the item.

Here's how the delisting and purchase flow works:

```
import {
  deserializeAddress,
  mConStr0,
  stringToHex
} from "@meshsdk/core";
import { getScript, getTxBuilder, getUtxoByTxHash, wallet } from "./
    common";

export function getScript() {
  const scriptCbor = applyParamsToScript(
    blueprint.validators[0].compiledCode,
    []
  );

  const scriptAddr = serializePlutusScript(
    { code: scriptCbor, version: "V3" },
  ).address;

  return { scriptCbor, scriptAddr };
}

async function main() {
  // Get utxo, collateral and address from wallet
  const utxos = await wallet.getUtxos();
  const walletAddress = (await wallet.getUsedAddresses())[0];
  const collateral = (await wallet.getCollateral())[0];

  const { scriptCbor } = getScript();

  // Hash of the public key of the wallet, to be used in the datum
  const signerHash = deserializeAddress(walletAddress).pubKeyHash;
  const scriptUtxo = await getUtxoByTxHash(txHashFromDesposit);

  // Build transaction with MeshTxBuilder
  const txBuilder = getTxBuilder();
  await txBuilder
    .spendingPlutusScript("V2") // We used Plutus V3
    .txIn( // Spend the utxo from the script address
      scriptUtxo.input.txHash,
      scriptUtxo.input.outputIndex,
      scriptUtxo.output.amount,
      scriptUtxo.output.address
    )
    .txInScript(scriptCbor)
    .txInRedeemerValue(mConStr1([]))
    .txInInlineDatumPresent()
    .requiredSignerHash(signerHash)
    .changeAddress(walletAddress)
    .txInCollateral(
      collateral.input.txHash,
      collateral.input.outputIndex,
      collateral.output.amount,
      collateral.output.address
    )
    .selectUtxosFrom(utxos)
    .complete();
  const unsignedTx = txBuilder.txHex;

  const signedTx = await wallet.signTx(unsignedTx);
  const txHash = await wallet.submitTx(signedTx);
  console.log(`1 tADA unlocked from the contract at Tx ID: ${txHash}`);
}

main();
```

[redacted]

Purchase code

```
  import {
    deserializeAddress,
    mConStr0,
    stringToHex
  } from "@meshsdk/core";
  import { getScript, getTxBuilder, getUtxoByTxHash, wallet } from "./
   common";

  async function main() {
    // Get utxo, collateral and address from wallet
    const utxos = await wallet.getUtxos();
    const walletAddress = (await wallet.getUsedAddresses())[0];
    const collateral = (await wallet.getCollateral())[0];

    const { scriptCbor } = getScript();

    // Hash of the public key of the wallet, to be used in the datum
    const signerHash = deserializeAddress(walletAddress).pubKeyHash;
    const scriptUtxo = await getUtxoByTxHash(txHashFromDesposit);

    // Build transaction with MeshTxBuilder
    const txBuilder = getTxBuilder();
    await txBuilder
      .spendingPlutusScript("V2") // We used Plutus V3
      .txIn( // Spend the utxo from the script address
        scriptUtxo.input.txHash,
        scriptUtxo.input.outputIndex,
        scriptUtxo.output.amount,
        scriptUtxo.output.address
      )
      .txInScript(scriptCbor)
      .txInRedeemerValue(mConStr0([0]))
      .txInInlineDatumPresent()
      .requiredSignerHash(signerHash)
      .changeAddress(walletAddress)
      .txInCollateral(
        collateral.input.txHash,
        collateral.input.outputIndex,
        collateral.output.amount,
        collateral.output.address
      )
      .selectUtxosFrom(utxos)
      .txOut(marketplaceaddress, {lovelace:fee})
      .txOut(sellerAddress, {lovelace:fee})
      .complete();
    const unsignedTx = txBuilder.txHex;

    const signedTx = await wallet.signTx(unsignedTx);
    const txHash = await wallet.submitTx(signedTx);
    console.log(`1 tADA unlocked from the contract at Tx ID: ${txHash}`);
  }

  main();

```

[redacted]

In this example: 1. Purchasing the NFT: If the buyer sends the correct amount of ADA to the marketplace contract, the contract checks that the buyer is paying the right amount, and the NFT is transferred to the buyer. 2. Delisting the NFT: The seller can delist their NFT by interacting with the contract, provided they are the one who signed the transaction.

Conclusion

Using MeshJS, we can easily build a frontend to interact with the marketplace contract deployed on Cardano. The listing functionality locks the NFT and sets a price, while the delisting or purchase functionalities allow users to either remove the NFT from the marketplace or complete the purchase.

These examples demonstrate how to manage NFTs on the Cardano blockchain using MeshJS, making it possible for developers to create simple, intuitive interfaces for interacting with smart contracts.

7.8. ENHANCING MARKETPLACE CONTRACTS

In this section, we explore potential enhancements for the marketplace contracts. These could include adding new features like auction support, bidding systems, or enhancing security by implementing advanced validation techniques. We will also discuss how we can extend the contract to support multi-asset marketplaces and integrate additional services.

7.8.1. Improving Marketplace Contract Features

Many marketplaces have been developed with basic functionalities, but several improvements could enhance their capability and security. Some key features that could be added or improved include:

- **Listing an NFT for Different Currencies**: Allowing NFTs to be listed for various currencies such as ADA or tokens, providing more flexibility in transaction options.
- **Listing an NFT for a Specific Buyer**: Introducing functionality where an NFT can only be purchased by a specific person, adding more control over the marketplace.
- **Listing a Bundle of NFTs**: Although bundles of NFTs can already be listed, this feature could be enhanced further. Understanding why and how this works in the current ecosystem is crucial for improving the process.

One example of a successful enhancement is the Aiken version, which allows multiple NFTs to be purchased in a single transaction. This is beneficial for security, as it mitigates risks related to transaction errors.

However, in the `plu-ts` smart contract, there is a potential vulnerability in the absence of checks for single inputs coming from the contract. This lack of validation could present an attack vector for double-spending issues. To address this, additional validation mechanisms could be implemented to ensure that each input is validated and that transactions are secure.

Now, it's time to reflect on these improvements and consider how we can achieve the desired results in our marketplace contracts.

> EXERCISE 7: Write a function to delist an NFT from a marketplace based on the following conditions:
> - The redeemer to delist is represented by the following redeemer value:
>
>   redeemerCancel = `Data.to( new Constr(0,[]) )`
>
> - The datum is hashed, not inline.
> - The contract bytecode for the delisting function is available at: `https://github.com/elRaulito/cnft-delist/blob/main/contract.json`.

CONCLUSION AND NEXT STEPS

8. CONCLUSIONS

8.1. RECAP OF KEY CONCEPTS AND TAKEAWAYS

Cardano's UTXO architecture allows for both native scripts and smart contracts, offering a unique approach to blockchain development. Native scripts enable features like multisig and timelock rules, which are essential for secure spending and minting of assets. Smart contracts, on the other hand, provide more flexibility, enabling developers to implement a wide range of decentralized applications (dApps).

It is also important to note that, while Haskell was once considered the primary language for programming on Cardano, this is no longer the case. Today, developers have access to more than four different programming languages that allow them to interact with and develop on Cardano's UTXO model. These languages make it easier for new developers to start working on Cardano, expanding the ecosystem.

8.2. RESOURCES FOR FURTHER LEARNING AND EXPLORATION

Here are some useful resources to continue your journey in Cardano development:

Resource	**Link**
Aiken Page	https://aiken-lang.org/
Aiken Docs	https://aiken-lang.github.io/stdlib/
Aiken Discord	https://discord.com/invite/Vc3x8N9nz2
Aiken GitHub	https://github.com/aiken-lang/aiken
Helios Page	https://www.helios-lang.io/
Helios Discord	https://discord.com/invite/XTwPrvB25q
Helios GitHub	https://github.com/orgs/HeliosLang/repositories
Pluts Docs	https://pluts.harmoniclabs.tech/
Pluts GitHub	https://github.com/HarmonicLabs/plu-ts
Anastasia Labs GitHub	https://github.com/Anastasia-Labs
Anastasia Labs Discord	https://discord.gg/VDXbPkcmjT
Mesh Page	https://meshjs.dev/
Mesh Discord	https://discord.com/invite/WvnCNqmAxy
Opshin Page	https://opshin.dev/
Opshin Book	https://book.opshin.dev/

Useful Resources for Cardano Smart Contract Development

8.3. CONTRIBUTING TO THE CARDANO SMART CONTRACT ECOSYSTEM

The best way to contribute to the Cardano smart contract ecosystem is by getting involved with the community on GitHub and Discord. These platforms are where the majority of development happens, and interacting with other developers is a great way to learn and contribute. Don't be afraid to ask questions—those who are not learning are often the ones who don't ask questions. By being curious and engaged, you can contribute to the growth of the ecosystem and help build a vibrant, decentralized future.

Dear builder, if you reach this part let me tell you that I admire your passion and dedication. Please make sure to reach out to me on twitter, discord or telegram.

This book is only the starting block of your journey as Cardano Developer

EXERCISE SOLUTIONS

9. SOLUTIONS

9.1. SOLUTIONS

9.1.1. Exercise 1

You can find the solution file for easy copy and paste: **Solution 1 File (formatted code)**

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Cardano Wallet Balance</title>
  </head>
  <body>
    <div id="balance"></div>

    <script type="module">
        import { Maestro, Lucid } from "https://deno.land/x/lucid/mod.ts";

        const lucid = await Lucid.new(
            new Maestro({
                network: "Preview",  // For MAINNET: "Mainnet".
                apiKey: "<Your-API-Key>",  // Get yours by visiting https://docs.gomaestro.org/docs/Getting-started/Sign-up-login.
                turboSubmit: false  // Read about paid turbo transaction submission feature at https://docs.gomaestro.org/docs/Dapp%20Platform/Turbo%20Transaction.
            }),
            "Preview",  // For MAINNET: "Mainnet".
        );

        // Enable Cardano wallet
        const api = await window.cardano.eternl.enable();
        lucid.selectWallet(api);

        // Get wallet balance
        const balance = await lucid.wallet.getBalance();

        // Display balance on the page
        document.getElementById('balance').innerText = `Wallet Balance: ${balance}`;
    </script>
  </body>
</html>
```

9.1.2. Exercise 2

You can find the solution file for easy copy and paste: **Solution 2 File (formatted code)**

```python
import requests
import json

url = "https://preview.gomaestro-api.org/v1/policy/YOURPOLICYID/
    addresses"
api_key = "<Your-API-Key>"
headers = {
    'Accept': 'application/json',
    'api-key': api_key
}

def get_addresses(url, headers):
    addresses = []
    next_cursor = None

    while True:
        params = {'cursor': next_cursor} if next_cursor else {}
        response = requests.get(url, headers=headers, params=params)

        if response.status_code == 200:
            data = response.json()
            addresses.extend(data.get('data', []))
            next_cursor = data.get('next_cursor')

            if not next_cursor:
                break
        else:
            print("Error:", response.status_code)
            break

    return addresses

addresses = get_addresses(url, headers)

# Write the JSON response to a file
with open('addresses.json', 'w') as file:
    json.dump(addresses, file, indent=4)

print("Addresses written to addresses.json")
```

9.1.3. Exercise 4

You can find the solution file for easy copy and paste: **Solution 4 File (formatted code)**

```
use aiken/collection/dict
use aiken/collection/list
use aiken/crypto.{VerificationKeyHash}
use aiken/primitive/bytearray.{to_string}
use aiken/primitive/string.{to_bytearray}
use cardano/address.{Address, Script}
use cardano/assets.{PolicyId, from_asset, from_asset_list, zero}
use cardano/transaction.{
  Input, NoDatum, Output, OutputReference, Transaction, placeholder,
}
use mocktail/virgin_key_hash.{mock_policy_id, mock_pub_key_hash}
use mocktail/virgin_output_reference.{mock_utxo_ref}

```

```
pub type Action {
  Mint
  Burn
}

validator custom_minting(
  utxo_ref: OutputReference,
  owner: Option<VerificationKeyHash>,
) {
  mint(rdmr: Action, policy_id: PolicyId, tx: Transaction) {
    let Transaction { inputs, mint, .. } = tx

    let assets =
      mint
        |> assets.tokens(policy_id)
        |> dict.to_pairs()

    when rdmr is {
      Mint -> {
        let is_token_name_valid = validate_token_name(assets)
        let is_mint_valid = validate_mint(assets, tx, inputs, utxo_ref,
     owner)

        is_token_name_valid? && is_mint_valid?
      }
      Burn -> {
        let is_burn_valid = validate_burn(assets)

        is_burn_valid?
      }
    }
  }

  else(_) {
    fail
  }
}

fn validate_token_name(tokens) {
  let token_names =
    [@"always", @"onetime", @"fenix"]

  if list.all(
    tokens,
    fn(Pair(name, _amount)) {
      list.any(token_names, fn(n) { n == to_string(name) })
    },
  ) {
    trace @"All token names correct!..."
    True
  } else {
    trace @"Ooops.... wrong token name found"
    False
  }
}

fn validate_mint(
  tokens: Pairs<ByteArray, Int>,
  tx: Transaction,
  inputs: List<Input>,
  utxo_ref: OutputReference,
  owner: Option<VerificationKeyHash>,
) {
  let fenix_exists =
    if list.any(
      tokens,
      fn(Pair(name, _amount)) { to_bytearray(to_string(name)) == "fenix
" },
    ) {
      True
    } else {
      False
    }

  if list.all(
    tokens,
    fn(Pair(name, amount)) {
      when to_bytearray(to_string(name)) is {
        "always" ->
          if fenix_exists {
            amount < 0
          } else {
            amount >= 1
          }
        "onetime" ->
          if fenix_exists {
            amount < 0
          } else {
            expect Some(_input) =
              list.find(
                inputs,
                fn(input) { input.output_reference == utxo_ref },
              )
            expect Some(owner_vk) = owner
            let is_signed = list.has(tx.extra_signatories, owner_vk)
            is_signed && amount == 1
          }
        "fenix" -> and {
            amount >= 1,
            list.any(
              tokens,
              fn(Pair(name, amount)) {
                to_bytearray(to_string(name)) == "always" && amount < 0
              },
            ),
            list.any(
              tokens,
              fn(Pair(name, amount)) {
                to_bytearray(to_string(name)) == "onetime" && amount <
0
              },
            ),
          }
        _ -> False
      }
    },
  ) {
    trace @"Tokens for minting validated successfully"
    True
  } else {
    trace @"Ooops. Validation of tokens for minting failed"
    False
  }
}

fn validate_burn(tokens: Pairs<ByteArray, Int>) {
  if list.all(tokens, fn(Pair(_name, amount)) { amount < 0 }) {
    trace @"Tokens for spending/burning validated successfully"
    True
  } else {
    trace @"Ooops. Validation of tokens for spending/burning failed"
    False
  }
}

fn get_tx(mint, sign: List<VerificationKeyHash>) {
  let tx_input =
    Input {
      output_reference: mock_utxo_ref(0, 0),
      output: Output {
        address: Address {
          payment_credential: Script(mock_policy_id(0)),
          stake_credential: None,
        },
        value: zero,
        datum: NoDatum,
        reference_script: None,
      },
    }
  let tx_inputs =
    [tx_input]

  Transaction {
    ..placeholder,
    mint: mint,
    inputs: tx_inputs,
    extra_signatories: sign,
  }
}

// mock new test
test burn_always_and_onetime() {
  let utxo = mock_utxo_ref(0, 0)

  let test_asset_name1 = "always"
  let test_asset_name2 = "onetime"
  let mint =
    from_asset_list(
      [
        Pair(
          mock_policy_id(0),
          [Pair(test_asset_name1, -3), Pair(test_asset_name2, -1)],
        ),
      ],
    )

  let tx = get_tx(mint, [])

  custom_minting.mint(utxo, None, Burn, mock_policy_id(0), tx)
}

test burn_always() {
  let utxo = mock_utxo_ref(0, 0)
  let mint = from_asset_list([Pair(mock_policy_id(0), [Pair("always",
    -3)])])

  let tx = get_tx(mint, [])

  custom_minting.mint(utxo, None, Burn, mock_policy_id(0), tx)
}

test burn_onetime() {
  let utxo = mock_utxo_ref(0, 0)

  let mint = from_asset_list([Pair(mock_policy_id(0), [Pair("onetime",
    -1)])])

  let tx = get_tx(mint, [])

  custom_minting.mint(utxo, None, Burn, mock_policy_id(0), tx)
}

test burn_fenix() {
  let utxo = mock_utxo_ref(0, 0)

  let mint = from_asset_list([Pair(mock_policy_id(0), [Pair("fenix",
    -1)])])

  let tx = get_tx(mint, [])

  custom_minting.mint(utxo, None, Burn, mock_policy_id(0), tx)
}

test mint_fenix() {
  let utxo = mock_utxo_ref(0, 0)

  let test_asset_name1 = "always"
  let test_asset_name2 = "onetime"
  let test_asset_name3 = "fenix"
  let mint =
    from_asset_list(
      [
        Pair(
          mock_policy_id(0),
          [
            Pair(test_asset_name1, -3),
            Pair(test_asset_name3, 1),
            Pair(test_asset_name2, -1),
          ],
        ),
      ],
    )

  let tx = get_tx(mint, [])

  // minting fenix token burns always and onetime
  custom_minting.mint(utxo, None, Mint, mock_policy_id(0), tx)
}

test mint_always() {
  let utxo = mock_utxo_ref(0, 0)

  let mint = from_asset_list([Pair(mock_policy_id(0), [Pair("always",
    3)])])

  let tx = get_tx(mint, [])

  custom_minting.mint(utxo, None, Mint, mock_policy_id(0), tx)
}

test mint_onetime() {
  let utxo = mock_utxo_ref(0, 0)

  let mint = from_asset_list([Pair(mock_policy_id(0), [Pair("onetime",
    1)])])

  let tx = get_tx(mint, [mock_pub_key_hash(0)])

  custom_minting.mint(
    utxo,
    Some(mock_pub_key_hash(0)),
    Mint,
    mock_policy_id(0),
    tx,
  )
}

test fail_mint_onetime() fail {
  // Testing onetime minting with different input utxos
  // Error parameter is a wrong output reference in tx2

  let utxo = mock_utxo_ref(0, 0)

  let mint = from_asset_list([Pair(mock_policy_id(0), [Pair("onetime",
    1)])])

  let tx = get_tx(mint, [mock_pub_key_hash(0)])

  let tx_input =
    Input {
      output_reference: mock_utxo_ref(1, 0),
      output: Output {
        address: Address {
          payment_credential: Script(mock_policy_id(0)),
          stake_credential: None,
        },
        value: zero,
        datum: NoDatum,
        reference_script: None,
      },
    }
  let tx_inputs =
    [tx_input]

  let tx2 =
    Transaction {
      ..placeholder,
      mint: mint,
      inputs: tx_inputs,
      extra_signatories: [mock_pub_key_hash(0)],
    }

  custom_minting.mint(
    utxo,
    Some(mock_pub_key_hash(0)),
    Mint,
    mock_policy_id(0),
    tx,
  ) && custom_minting.mint(
    utxo,
    Some(mock_pub_key_hash(0)),
    Mint,
    mock_policy_id(0),
    tx2,
  )
}

test mint_single_always() {
  let utxo = mock_utxo_ref(0, 0)

  let mint = from_asset(mock_policy_id(0), "always", 3)

  let tx = get_tx(mint, [])

  custom_minting.mint(utxo, None, Mint, mock_policy_id(0), tx)
}
```

9.1.4. Exercise 5

You can find the solution file for easy copy and paste: **Solution 5 File (formatted code)**

```
use aiken/collection/list
use aiken/crypto.{ScriptHash, VerificationKeyHash}
use aiken/interval.{Finite}
use cardano/address.{Script}
use cardano/transaction.{Input, Transaction, ValidityRange, placeholder
  }
use mocktail/virgin_key_hash.{mock_pub_key_hash}

pub type User {
  SingleSigner(VerificationKeyHash)
  MultiSigner(ScriptHash)
}

pub type SWDatum {
  lock_duration: Int,
}

validator smart_wallet(user: User) {
  spend(datum: Option<SWDatum>, _r: Data, _o: Data, self: Transaction)
    {
    let Transaction { inputs, .. } = self

    and {
      is_signed(self, user, inputs),
      when datum is {
        Some(SWDatum { lock_duration }) ->
          is_time_reached(self.validity_range, lock_duration)
        _ -> True
      },
    }
  }

  else(_) {
    fail
  }
}

fn is_signed(tx: Transaction, user: User, inputs: List<Input>) {
  when user is {
    SingleSigner(signer_vk) -> list.has(tx.extra_signatories, signer_vk
    )

    MultiSigner(script_hash) -> {
      let script_cred = Script(script_hash)
      list.any(
        inputs,
        fn(input) {
          let address = input.output.address
          address.payment_credential == script_cred
        },
      )
    }
  }
}

fn is_time_reached(range: ValidityRange, lock_time: Int) {
  when range.lower_bound.bound_type is {
    Finite(time_now) -> lock_time <= time_now
    _ -> False
  }
}

// Tests

fn get_tx(sign: List<VerificationKeyHash>) {
  Transaction { ..placeholder, extra_signatories: sign }
}

test test_single_signer() {
  let tx = get_tx([mock_pub_key_hash(0)])

  smart_wallet.spend(SingleSigner(mock_pub_key_hash(0)), None, "", "",
    tx)
}
```

9.1.5. Exercise 6

You can find the solution file for easy copy and paste: **Solution 6 File (formatted code)**

```
use aiken/collection/dict
use aiken/collection/list
use aiken/primitive/bytearray.{to_string}
use aiken/primitive/string.{from_bytearray}
use cardano/address.{Address, Script}
use cardano/assets.{
  AssetName, PolicyId, from_asset, from_asset_list, quantity_of,
}
```

```
use cardano/transaction.{
  Input, NoDatum, Output, OutputReference, Transaction, placeholder,
}
use mocktail/virgin_key_hash.{mock_policy_id}
use mocktail/virgin_output_reference.{mock_utxo_ref}

pub type Action {
  Mint
  Burn
}

validator fract_nft(
  nft_policy_id: PolicyId,
  nft_asset_name: AssetName,
  mint_tokens_quantity: Int,
  burn_tokens_quantity: Int,
  uxto_onetime: OutputReference,
) {
  mint(redeemer: Action, policy_id: PolicyId, self: Transaction) {
    let Transaction { inputs, outputs, mint, .. } = self

    let assets = mint |> assets.tokens(policy_id) |> dict.to_pairs()

    expect [Pair(fract_asset_name, _)] = assets
    let valid_fract_asset_name =
      string.concat(left: @"fract-", right: from_bytearray(
   nft_asset_name))

    expect valid_fract_asset_name == to_string(fract_asset_name)

    when redeemer is {
      Mint -> {
        expect Some(_input) =
          list.find(
            inputs,
            fn(input) { input.output_reference == uxto_onetime },
          )
        validate_mint(
          assets,
          inputs,
          outputs,
          policy_id,
          nft_policy_id,
          nft_asset_name,
          mint_tokens_quantity,
        )
      }
      Burn ->
        validate_burn(
          assets,
          policy_id,
          inputs,
          nft_policy_id,
          nft_asset_name,
          burn_tokens_quantity,
        )
    }
  }

  spend(_d: Option<Data>, _r: Data, utxo: OutputReference, self:
   Transaction) {
    let Transaction { inputs, mint, .. } = self

    expect Some(own_input) =
      list.find(inputs, fn(input) { input.output_reference == utxo })

    expect Script(policy_id) = own_input.output.address.payment_credential

    expect [Pair(_, quantity)] =
      mint |> assets.tokens(policy_id) |> dict.to_pairs

    (quantity <= -burn_tokens_quantity)?
  }

  else(_) {
    fail
  }
}

fn validate_mint(
  assets: Pairs<ByteArray, Int>,
  inputs: List<Input>,
  outputs: List<Output>,
  policy_id: PolicyId,
  nft_policy_id: PolicyId,
  nft_asset_name: AssetName,
  mint_tokens_quantity: Int,
) {
  let is_nft_in_inputs =
    inputs
      |> list.any(
          fn(input) {
            quantity_of(input.output.value, nft_policy_id, nft_asset_name) == 1
          },
        )

  let is_nft_in_outputs =
    outputs
      |> list.any(
          fn(output) {
            and {
              output.address.payment_credential == Script(policy_id),
              quantity_of(output.value, nft_policy_id, nft_asset_name) == 1,
            }
          },
        )

  let is_mint_quantity_valid = and {
      list.length(assets) == 1,
      assets
        |> list.all(fn(Pair(_, quantity)) { quantity == mint_tokens_quantity }),
    }

  and {
    is_nft_in_inputs?,
    is_nft_in_outputs?,
    is_mint_quantity_valid?,
  }
```

```
}

fn validate_burn(
  assets: Pairs<ByteArray, Int>,
  policy_id: PolicyId,
  inputs: List<Input>,
  nft_policy_id: PolicyId,
  nft_asset_name: AssetName,
  burn_tokens_quantity: Int,
) {
  let is_burn_tokens_valid =
    assets
      |> list.any(fn(Pair(_, quantity)) { quantity <= -
   burn_tokens_quantity })

  let is_input_valid =
    inputs
      |> list.any(
          fn(input) {
            and {
              quantity_of(input.output.value, nft_policy_id,
   nft_asset_name) == 1,
              input.output.address.payment_credential == Script(
   policy_id),
            }
          },
        )

  is_burn_tokens_valid? && is_input_valid?
}

// Tests

test test_mint() {
  let the_nft_policy_id = mock_policy_id(0)
  let the_nft_asset_name = "my_nft"
  let the_nft = from_asset(the_nft_policy_id, the_nft_asset_name, 1)

  let tx_input =
    Input {
      output_reference: mock_utxo_ref(0, 0),
      output: Output {
        address: Address {
          // input from some place
          payment_credential: Script(mock_policy_id(5)),
          stake_credential: None,
        },
        value: the_nft,
        datum: NoDatum,
        reference_script: None,
      },
    }

  let tx_output =
    Output {
      address: Address {
        // Output going into the validator
        payment_credential: Script(mock_policy_id(1)),
        stake_credential: None,
      },
      value: the_nft,
      datum: NoDatum,
      reference_script: None,
    }

  let mint =
    from_asset_list([Pair(mock_policy_id(1), [Pair("fract-my_nft", 100)
   ])])

  let tx =
    Transaction {
      ..placeholder,
      mint: mint,
      inputs: [tx_input],
      outputs: [tx_output],
    }

  fract_nft.mint(
    the_nft_policy_id,
    the_nft_asset_name,
    100,
    50,
    mock_utxo_ref(0, 0),
    Mint,
    mock_policy_id(1),
    tx,
  )
}

test test_burn() {
  let the_nft_policy_id = mock_policy_id(0)
  let the_nft_asset_name = "my_nft"
  let the_nft = from_asset(the_nft_policy_id, the_nft_asset_name, 1)
  let validator_hash = mock_policy_id(1)

  let tx_input =
    Input {
      // output reference changed
      output_reference: mock_utxo_ref(0, 1),
      output: Output {
        address: Address {
          // input from validator
          payment_credential: Script(validator_hash),
          stake_credential: None,
        },
        value: the_nft,
        datum: NoDatum,
        reference_script: None,
      },
    }

  let mint =
    from_asset_list([Pair(mock_policy_id(1), [Pair("fract-my_nft", -50)
   ])])

  let tx = Transaction { ..placeholder, mint: mint, inputs: [tx_input]
   }

  fract_nft.mint(
    the_nft_policy_id,
    the_nft_asset_name,
    100,
    50,
    mock_utxo_ref(0, 0),
    Burn,
    validator_hash,
    tx,
  )
}

test test_spend() {
  let the_nft_policy_id = mock_policy_id(0)
  let the_nft_asset_name = "my_nft"
  let the_nft = from_asset(the_nft_policy_id, the_nft_asset_name, 1)
  let validator_hash = mock_policy_id(1)

  let own_oref = mock_utxo_ref(0, 1)

  let tx_input =
    Input {
      // output reference changed
      output_reference: own_oref,
      output: Output {
        address: Address {
          // input from validator
          payment_credential: Script(validator_hash),
          stake_credential: None,
        },
        value: the_nft,
        datum: NoDatum,
        reference_script: None,
      },
    }

  let mint = from_asset_list([Pair(mock_policy_id(1), [Pair("fract1",
    -50)])])

  let tx = Transaction { ..placeholder, mint: mint, inputs: [tx_input]
    }

  fract_nft.spend(
    the_nft_policy_id,
    the_nft_asset_name,
    100,
    50,
    mock_utxo_ref(0, 0),
    None,
    "",
    own_oref,
    tx,
  )
}

test test_mint_with_spend() {
  let the_nft_policy_id = mock_policy_id(0)
  let the_nft_asset_name = "my_nft"
  let the_nft = from_asset(the_nft_policy_id, the_nft_asset_name, 1)
  let validator_hash = mock_policy_id(1)

  let own_oref = mock_utxo_ref(0, 1)

  let tx_input =
    Input {
      // output reference changed
      output_reference: own_oref,
      output: Output {
        address: Address {
          // input from validator
          payment_credential: Script(validator_hash),
          stake_credential: None,
        },
        value: the_nft,
        datum: NoDatum,
        reference_script: None,
      },
    }

  let mint =
    from_asset_list([Pair(mock_policy_id(1), [Pair("fract-my_nft", -50)
   ])])

  let tx = Transaction { ..placeholder, mint: mint, inputs: [tx_input]
   }

  and {
    fract_nft.mint(
      the_nft_policy_id,
      the_nft_asset_name,
      100,
      50,
      mock_utxo_ref(0, 0),
      Burn,
      validator_hash,
      tx,
    ),
    fract_nft.spend(
      the_nft_policy_id,
      the_nft_asset_name,
      100,
      50,
      mock_utxo_ref(0, 0),
      None,
      "",
      own_oref,
      tx,
    ),
  }
}
```

9.1.6. Exercise 7

You can find the solution file for easy copy and paste: **Solution 7 File (formatted code)**

```
import { Lucid , Blockfrost,toUnit,Data,Constr,  fromHex,
    toHex,sha256  } from "https://unpkg.com/lucid-cardano@0.10.0/web/
   mod.js"
  const lucid = await Lucid.new(
    new Blockfrost("https://cardano-mainnet.blockfrost.io/api/v0", "API
     KEY"),
    "Mainnet",
  );

  import cnftScript from './contract.json' assert {type: 'json'};

   //Here replace with the wallet name according to cip30
   var wallet="eternl"

   let api = await window.cardano[wallet].enable();
   lucid.selectWallet(api);
   window.owner=await lucid.wallet.address()
   let{ paymentCredential,stakeCredential } = lucid.utils.getAddressDetails(
   await lucid.wallet.address(),
   );

  //this is the most tricky part, go to https://cbor.me/ and paste this datum, replace the fields with your listing data and get the new datum to replace this one
  let datumCancel="D8668200844 1FB4133D866820082181814D866820180"

  //example
  //102([0, [h'SELLER_SPENDING_KEY_HASH', h'ROYALTY_SPENDING_KEY_HASH', 102([0, [//LISTING_PRICE, //ROYALTY_PERCENTAGE_IN_MILLESIMALS]]), 102([1, []])]])
  //102([0, [h'FB2CD544A148D0BBC70C9863E3448224EE95C3DB699F864F8D6305E2', h'3342CA8C073A11B7664BD105123353E79C01116CC465915133FDCF75', 102([0, [249999000000, 20]]), 102([1, []])]])



    var redeemerCancel=Data.to(
      new Constr(0,[])
    )


    //THIS IS THE TXHASH OF THE LISTING TO BE CANCELLED REPLACE IT
    const utxoHash="dbd761d11fa7dde62c4899b0168c6618789ad68d34560e9fd7a40e5d498d3044"
    const index=0

    var utxo=await lucid.utxosByOutRef([{ txHash: utxoHash, outputIndex: index }])
    utxo[0].datum=datumCancel

    const tx = await lucid
    .newTx()
    .collectFrom(utxo,redeemerCancel)
    .attachSpendingValidator(cnftScript)
    .addSignerKey(paymentCredential.hash)
    .complete();

    const signedTx = await tx.sign().complete();
    const txHash = await signedTx.submit();
```

GLOSSARY

AMM Automatic market maker dexes involve a liquidity pool, the pool has two pair tokens, usually ADA and the Cardano native token, users can sell or buy tokens from this pool and the price is adjusted according to the market need . 9

block A block in Cardano is a record of transactions and other information produced by a slot leader during a slot. Blocks are added to the blockchain sequentially. Each block contains a header with metadata, such as the previous block hash, and a body that includes the transaction data and other relevant information. Blocks are produced by slot leaders, which are chosen through the Ouroboros consensus protocol, Cardano's proof-of-stake mechanism. Blocks are essential for maintaining the integrity and continuity of the blockchain, as they confirm and validate transactions. . 23

CIP Cardano Improvement Proposals that if approved can change the current ledger or chain parameters, usually are also standards to develop in a similar way between projects . 15

Composability Also referred to as transaction in transaction, it's the ability to interact with multiple parties in the same transaction, this is not possible in the account model, however, this also raises the concurrency issue when two parties or transactions want to spend the same utxo. 10

Determinism Transaction and blockchain behavior are predictable, given a sort of input and outputs, once the fee is decided the transaction hash will always be the same. 6

epoch An epoch in Cardano is a fixed period during which a set of blocks is produced. The duration of an epoch is predefined and consistent. As of the current Cardano implementation, an epoch lasts for 5 days. At the end of each epoch, rewards are calculated and distributed, and a new epoch begins. Epochs help structure the blockchain into manageable time periods, enabling efficient consensus and reward mechanisms. . 23

inputs Inputs in a UTXO model transaction specify which unspent outputs are being consumed, so which funds coming from previous transactions are being spent. . 23

Liquid Staking Cardano staking is referred to as Liquid, no locking mechanism is needed to get the staking rewards. This becomes useful because users can move their ADA around inside smart contracts while keeping the delegation rewards . 10

orderbook In this configuration each order placed by users is a single entry with a price and amount of token willing to sell (ADA or native tokens), swaps happen matching the orders . 8

slot A slot is a smaller time unit within an epoch. An epoch is divided into a large number of slots. Each slot represents a potential opportunity to produce a block. In the current implementation of Cardano, there are 432,000 slots in an epoch, with each slot lasting 1 second. However, not every slot will necessarily have a block produced, as block production depends on the consensus protocol and slot leader election. . 23