

eUTxO Fundamentals: Building Cardano Smart Contracts

Starting from zero

AFL v3.0 - 2024

Raul Rosa
aka ElRaulito

I		INTRODUCTION TO CARDANO SMART CONTRACTS	4
	1	OVERVIEW OF CARDANO BLOCKCHAIN	5
	1.1	Importance and Applications of Smart Contracts	7
	1.2	Advantages of Cardano for Smart Contract Development	9
II		SETTING UP THE DEVELOPMENT ENVIRONMENT	10
	2	INSTALLING AND CONFIGURING CARDANO DEVELOPMENT TOOLS	11
	2.1	Installing and Configuring Cardano Development Tools	11
	2.2	Hot wallets on Cardano	11
	2.3	Setting Up and Connecting to Cardano Testnet	11
	2.4	Interacting with Cardano node and Wallet APIs	14
	2.5	Setting Up a Cardano node on Contabo Cloud VPS	15
III		EXPLORING THE EUTXO MODEL	19
	3	UNDERSTANDING THE EUTXO MODEL AND ITS COMPONENTS	20
	3.1	Understanding the eUTxO Model and Its Components	20
	3.2	eUTXO Model	20
	3.3	Writing Transactions and Validating Inputs and Outputs	21
	3.4	Cardano Native Scripts	25
	3.5	Using Lucid Library for Cardano Native Scripts	29
	3.6	Interacting with Smart Contracts Using Lucid	30
	3.7	Datum, Redeemers, and Script Context	33
IV		CARDANO SMART CONTRACT LANGUAGES	35
	4	CARDANO SMART CONTRACT LANGUAGES	36
	4.1	Introduction to Plutus Language	36
	4.2	Exploring Helios Language	40
	4.3	Aiken Language: Features and Syntax	43
	4.4	OpShin Language: Concepts and Usage	46
	4.5	Plu-ts: Understanding the basics	47

V EXERCISE SOLUTIONS		51
5	SOLUTIONS	52
5.1	Solutions	52
GLOSSARY		54



INTRODUCTION TO CAR- DANO 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.

1.1.1. 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.

1.1.1.1. 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?

1.1.1.2. 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. Bachers 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 DEVELOPMENT 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 Vesper 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 run the code on the browser.
- **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

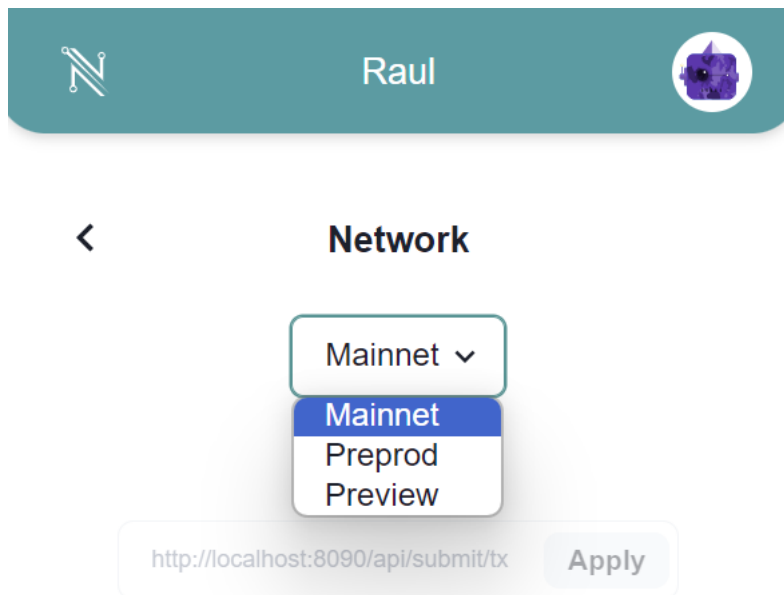
In this example, we'll install the Nami wallet that we can find [here](#)

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.

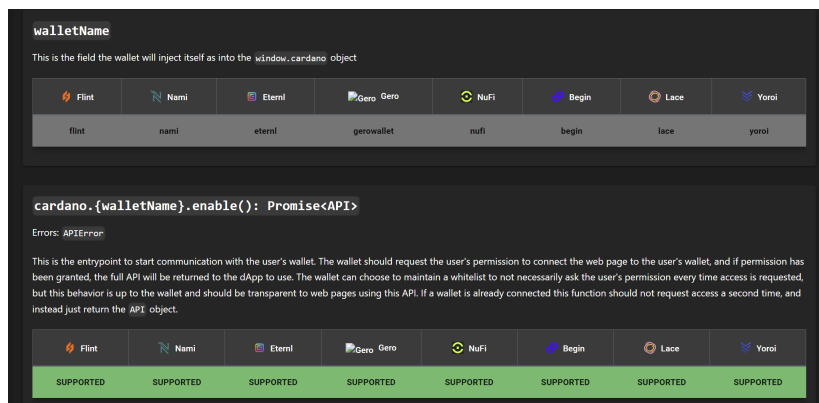
2.3.2. Get tADA

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](#)



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.

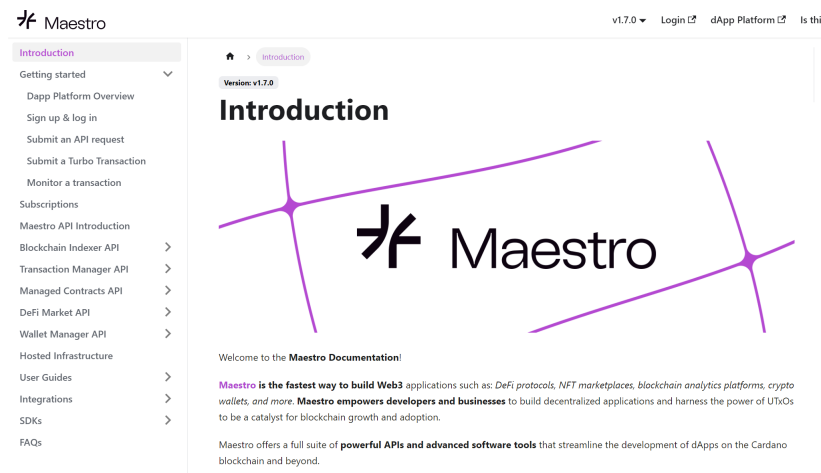
2.4. INTERACTING WITH CARDANO NODE AND WALLET APIS

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 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 \dots
/download/8.1.2/cardano-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 \dots
https://downloads.csnapshots.io/snapshots/mainnet/mainnet-db-snapshot.json | jq
r .[].file_name ) \dots
| lz4 -c -d - | tar -x -C /root/node/mainnet/

```

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 \dots  
/root/node/sockets/socket node.socket --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:

```
a2fcdf32987ebb729ab8f63b377e360ececbbf4805713ff559d2b69bb3c543a01
```



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_test1qpku2n4c42jv32matism...sju`
- The second output is a change to `addr_test1qqwywx3ag9sf3jjhk8hd...fhn5gh` sending back all the tADA that was left.
- The third output is a change to `addr_test1qqwywx3ag9sf3jjhk8hd...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 scratch

Let's create a folder:

```
1 mkdir exercise01
```

Now we create the wallet with:

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

To see the address, use the following command:

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

And then:

```
1 cat payment.addr
```

And we now see our address!

Checking funds in wallet

Let's run the command:

```
1 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:

```
1 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:

```
1 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:

```
1 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:

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

Now:

```
1 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:

```
1 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

```
1 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:

```
1 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:

```
1 const tx = await lucid.newTx()
2   .payToAddress("ADDRESSRECEIVER", {lovelace: 2000000n})
3   .complete();
4 const signedTx = await tx.sign().complete();
5 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:

```

1 {
2   "type": "sig",
3   "keyHash": "e09d36c79dec9bd1b3d9e152247701cd0bb860b5ebfd1de8abb6735a"
4 }
```

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:

```

1 {
2   "type": "all",
3   "scripts":
4   [
5     {
6       "type": "after",
7       "slot": 1000
8     },
9     {
10      "type": "sig",
```

```

11     "keyHash": "966e394a544f242081e41d1965137b1bb412ac230d40ed5407821c37"
12   }
13 ]
14 }

```

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

```

1 {
2   "type": "all",
3   "scripts":
4   [
5     {
6       "type": "sig",
7       "keyHash": "e09d36c79dec9bd1b3d9e152247701cd0bb860b5ebfd1de8abb6735a"
8     },
9     {
10      "type": "sig",
11      "keyHash": "a687dcc24e00dd3caafbeb5e68f97ca8ef269cb6fe971345eb951756"
12    },
13    {
14      "type": "sig",
15      "keyHash": "0bd1d702b2e6188fe0857a6dc7ffb0675229bab58c86638ffa87ed6d"
16    }
17  ]
18 }

```

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

```

1 {
2   "type": "all",
3   "scripts":
4   [
5     {
6       "type": "after",
7       "slot": 1000
8     },
9     {
10      "type": "sig",
11      "keyHash": "966e394a544f242081e41d1965137b1bb412ac230d40ed5407821c37 "
12    }
13  ]
14 }

```

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:

```

1 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:

```

1 const lucid = await Lucid.new(
2   new Blockfrost("https://cardano-mainnet.blockfrost.io/api/v0", "APIKEYHERE"),
3   "Mainnet",
4 );

```

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

```
1 const multisigScript = lucid.utils.nativeScriptFromJson(
2 {
3   "type": "all",
4   scripts: [
5     { type: "sig", keyHash: "1
      c471b31ea0b04c652bd8f76b239aea5f57139bdc5a2b28ab1e69175" },
6   ],
7 }
8 );
```

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

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

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

```
1 var tx = await lucid
2   .newTx()
3   .payToAddress(multisigAddress, { lovelace: 1000000 }) // Example: sending 1 ADA
4   .attachMintingPolicy(multisigScript)
5   .addSignerKey("1c471b31ea0b04c652bd8f76b239aea5f57139bdc5a2b28ab1e69175") //
      Signer key
6   .complete();
7
8 const signedTx = await tx.sign().complete();
9 const txHash = await signedTx.submit();
10 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:

```
1 import { Lucid, Blockfrost, toHex, fromHex, Data, Constr, fromText } from "
   https://unpkg.com/lucid-cardano@0.10.0/web/mod.js";
2
3 const lucid = await Lucid.new(
4   new Blockfrost("https://cardano-mainnet.blockfrost.io/api/v0", "APIKEYHERE"),
5   "Mainnet",
6 );
```

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

```
1 import data from '/plutus.json' assert { type: "json" };
2 console.log(data);
3
4 import * as cbor from "https://deno.land/x/cbor@v1.4.1/index.js";
5
6 const mintingPolicy = {
7   type: "PlutusV2",
8   script: toHex(cbor.encode(fromHex(data.validators[1].compiledCode)))
9 };
10
11 const storageScript = {
12   type: "PlutusV2",
13   script: toHex(cbor.encode(fromHex(data.validators[0].compiledCode)))
14 };
```

Enable the Nami wallet and select it with Lucid:

```
1 const api = await window.cardano.nami.enable();
2 lucid.selectWallet(api);
3
4 const { paymentCredential } = lucid.utils.getAddressDetails(
5   await lucid.wallet.address(),
6 );
```

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

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

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

```

1 let utxoSeed = await lucid.utxosByOutRef([ { txHash: "451
    d8129bb9fce9829906fe32bb7c2a93f40493f3ceeb3b003ae7eb7c8a99f52", outputIndex
    : 4 } ]);
2
3 const redeemer = Data.to(new Constr(0, []));

```

Define the metadata for the new tokens:

```

1 const metaData = {
2   decimals: 6,
3   description: "The official token of FluidTokens, a leading DeFi ecosystem
    fueled by innovation and community backing.",
4   logo: "https://fluidtokens.com/fldt.png",
5   name: "FLDT",
6   ticker: "FLDT",
7   website: "https://fluidtokens.com/",
8 };
9
10 console.log(metaData);
11
12 Object.keys(metaData)
13   .sort()
14   .forEach(function(v, i) {
15     console.log(v, metaData[v]);
16   });

```

Create the Datum and build the transaction:

```

1 const Datum = Data.to(new Constr(0, [Data.fromJson(metaData), 1n, 1n]));
2
3 const tx = await lucid
4   .newTx()
5   .collectFrom(utxoSeed)
6   .mintAssets({ [{policyId}0014df10${fromText("FLDT")}]: BigInt(100000000 * Math
    .pow(10, 6)), [{policyId}000643b0${fromText("FLDT")}]: 1n }, redeemer)
7   .payToContract(storageAddress, { inline: Datum }, { [{policyId}000643b0${
    fromText("FLDT")}]: 1n })
8   .attachMintingPolicy(mintingPolicy)
9   .complete();

```

Sign and submit the transaction:

```

1 const signedTx = await tx.sign().complete();
2 const txHash = await signedTx.submit();
3 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:


```

1 Transaction {
2   inputs: List<Input>,
3   reference_inputs: List<Input>,
4   outputs: List<Output>,
5   fee: Value,
6   mint: MintedValue,
7   certificates: List<Certificate>,
8   withdrawals: Pairs<StakeCredential, Int>,
9   validity_range: ValidityRange,

```

```
10     extra_signatories: List<Hash<Blake2b_224, VerificationKey>>,
11     redeemers: Pairs<ScriptPurpose, Redeemer>,
12     datums: Dict<Hash<Blake2b_256, Data>, Data>,
13     id: TransactionId,
14 }
```

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 CONTRACT 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/E
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.

```
1 {-# LANGUAGE DataKinds           #-}
2 {-# LANGUAGE ImportQualifiedPost  #-}
3 {-# LANGUAGE NoImplicitPrelude    #-}
4 {-# LANGUAGE OverloadedStrings    #-}
5 {-# LANGUAGE TemplateHaskell      #-}
6
7 module FortyTwo where
8
9 import qualified Plutus.V2.Ledger.Api as PlutusV2
10 import      PlutusTx                  (BuiltinData, compile)
11 import      PlutusTx.Builtins         as Builtins (mkI)
12 import      PlutusTx.Prelude         (otherwise, traceError, (==))
13 import      Prelude                   (IO)
14 import      Utilities                 (writeValidatorToFile)
15
16 -- This validator succeeds only if the redeemer is 42
17 --                               Datum      Redeemer      ScriptContext
18 mk42Validator :: BuiltinData -> BuiltinData -> BuiltinData -> ()
19 mk42Validator _ r _
20   | r == Builtins.mkI 42 = ()
21   | otherwise            = traceError "expected 42"
22 {-# INLINABLE mk42Validator #-}
23
24 validator :: PlutusV2.Validator
25 validator = PlutusV2.mkValidatorScript $(PlutusTx.compile [| |
26   mk42Validator |])
27
28 saveVal :: IO ()
29 saveVal = writeValidatorToFile "./assets/fortytwo.plutus" validator
```



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

- **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 bi
// 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 u
    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 se
    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

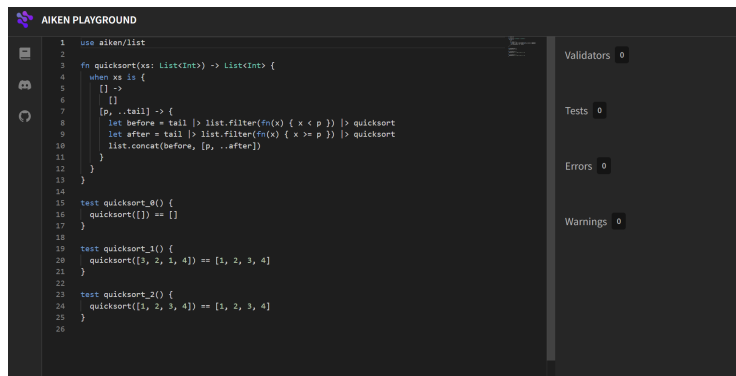
```
aiken completion bash > /usr/local/share/bash-completion/completions/aiken
source /usr/local/share/bash-completion/completions/aiken
```

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



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.

```
1 use aiken/hash.{Blake2b_224, Hash}
2 use aiken/list
3 use aiken/transaction.{ScriptContext}
4 use aiken/transaction/credential.{VerificationKey}
5
6 type Datum {
7   owner: Hash<Blake2b_224, VerificationKey>,
8 }
9
10 type Redeemer {
11   msg: ByteArray,
12 }
13
14 validator {
15   fn hello_world(datum: Datum, redeemer: Redeemer, context:
16     ScriptContext) -> Bool {
17     let must_say_hello =
18       redeemer.msg == "Hello, World!"
19
20     let must_be_signed =
21       list.has(context.transaction.extra_signatories, datum.owner)
22
23     must_say_hello && must_be_signed
24   }
25 }
```

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.

```
1 # gift.py
2
3 # The Opshin prelude contains a lot of useful types and functions
4 from opshin.prelude import *
```

```

5
6 # Custom Datum
7 @dataclass()
8 class GiftDatum(PlutusData):
9     # The public key hash of the gift creator.
10    # Used for cancelling the gift and refunding the creator (1).
11    creator_pubkeyhash: bytes
12
13    # The public key hash of the gift recipient.
14    # Used by the recipient for collecting the gift (2).
15    recipient_pubkeyhash: bytes
16
17 def validator(datum: GiftDatum, redeemer: None, context: ScriptContext)
18     -> None:
19     # Check that we are indeed spending a UTxO
20     assert isinstance(context.purpose, Spending), "Wrong type of script
21     invocation"
22
23     # Confirm the creator signed the transaction in scenario (1).
24     creator_is_cancelling_gift = datum.creator_pubkeyhash in context.
25     tx_info.signatories
26
27     # Confirm the recipient signed the transaction in scenario (2).
28     recipient_is_collecting_gift = datum.recipient_pubkeyhash in
29     context.tx_info.signatories
30
31     assert creator_is_cancelling_gift or recipient_is_collecting_gift,
32     "Required signature missing"

```

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.


```

1 import { Address, bool, compile, makeValidator, PaymentCredentials,
  pBool, pfn, Script, ScriptType, V2 } from "@harmoniclabs/plu-ts";
2 import MyDatum from "./MyDatum";
3 import MyRedeemer from "./MyRedeemer";
4
5 export const contract = pfn([
6   MyDatum.type,
7   MyRedeemer.type,
8   V2.PScriptContext.type
9 ], bool)
10 (( datum, redeemer, ctx ) =>
11   // always succeeds
12   pBool( true )
13 );
14
15 export const untypedValidator = makeValidator( contract );
16 export const compiledContract = compile( untypedValidator );
17 export const script = new Script(
18   ScriptType.PlutusV2,
19   compiledContract
20 );
21
22 export const scriptMainnetAddr = new Address(
23   "mainnet",
24   new PaymentCredentials(
25     "script",
26     script.hash
27   )
28 );
29
30 export const scriptTestnetAddr = new Address(
31   "testnet",
32   new PaymentCredentials(
33     "script",
34     script.hash.clone()
35   )
36 );
37
38 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/index.ts` respectively.

```

1 import { int, pstruct } from "@harmoniclabs/plu-ts";
2
3 // modify the Datum as you prefer
4 const MyDatum = pstruct({
5   Num: {
6     number: int
7   },
8   NoDatum: {}
9 });

```

```

10
11 export default MyDatum;

1 import { pstruct } from "@harmoniclabs/plu-ts";
2
3 // modify the Redeemer as you prefer
4 const MyRedeemer = pstruct({
5     Option1: {},
6     Option2: {}
7 });
8
9 export default MyRedeemer;

```

Entry Point

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

```

1 import { script } from "./contract";
2
3 console.log("validator compiled successfully! \n");
4 console.log(
5     JSON.stringify(
6         script.toJson(),
7         undefined,
8         2
9     )
10 );

```

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.



EXERCISE SOLUTIONS



5. SOLUTIONS

5.1. SOLUTIONS

5.1.1. Exercise 1

```

1 <!DOCTYPE html>
2 <html lang="en" dir="ltr">
3   <head>
4     <meta charset="utf-8">
5     <title>Cardano Wallet Balance</title>
6   </head>
7   <body>
8     <div id="balance"></div>
9
10    <script type="module">
11      import { Maestro, Lucid } from "https://deno.land/x/lucid/mod.
12      ts";
13
14      const lucid = await Lucid.new(
15        new Maestro({
16          network: "Preview", // For MAINNET: "Mainnet".
17          apiKey: "<Your-API-Key>", // Get yours by visiting
18          https://docs.gomaestro.org/docs/Getting-started/Sign-up-login.
19          turboSubmit: false // Read about paid turbo
20          transaction submission feature at https://docs.gomaestro.org/docs/
21          Dapp%20Platform/Turbo%20Transaction.
22        }),
23        "Preview", // For MAINNET: "Mainnet".
24      );
25
26      // Enable Cardano wallet
27      const api = await window.cardano.eternl.enable();
28      lucid.selectWallet(api);
29
30      // Get wallet balance
31      const balance = await lucid.wallet.getBalance();
32
33      // Display balance on the page
34      document.getElementById('balance').innerText = `Wallet Balance:
35      ${balance}`;
36    </script>
37  </body>
38 </html>

```

5.1.2. Exercise 2

```

1 import requests
2 import json
3
4 url = "https://preview.gomaestro-api.org/v1/policy/YOURPOLICYID/
5     addresses"
6 api_key = "<Your-API-Key>"
7 headers = {

```

```

7      'Accept': 'application/json',
8      'api-key': api_key
9  }
10
11  def get_addresses(url, headers):
12      addresses = []
13      next_cursor = None
14
15      while True:
16          params = {'cursor': next_cursor} if next_cursor else {}
17          response = requests.get(url, headers=headers, params=params)
18
19          if response.status_code == 200:
20              data = response.json()
21              addresses.extend(data.get('data', []))
22              next_cursor = data.get('next_cursor')
23
24              if not next_cursor:
25                  break
26          else:
27              print("Error:", response.status_code)
28              break
29
30      return addresses
31
32  addresses = get_addresses(url, headers)
33
34  # Write the JSON response to a file
35  with open('addresses.json', 'w') as file:
36      json.dump(addresses, file, indent=4)
37
38  print("Addresses written to addresses.json")

```

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 . 8
- 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. . 22
- 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 . 14
- 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. 9
- 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. 5
- 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. . 22
- inputs** Inputs in a UTXO model transaction specify which unspent outputs are being consumed, so which funds coming from previous transactions are being spent. . 22
- 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 . 9
- 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 . 7
- 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. . 22