

# Cardano Smart Contracts



*with Helios*

Lawrence Ley, B.Sc



This book would not have been possible without Christian Schmitz and the open source project called Helios.

# Preface

With the invention of Bitcoin in 2008, Satoshi's [whitepaper](#) titled "Bitcoin: A Peer-to-Peer Electronic Cash System" ushered in an era of trustless decentralized digital currencies. [Ethereum](#) which launched in 2015, introduced the concepts of programmability into decentralized ledger technology. This allowed application developers to use a blockchain to store data and programs in a decentralized and permissionless way. Cardano launched in 2017 and leverages Bitcoin's secure UTXO architecture and created a new deterministic, scalable and secure programmable blockchain. Cardano can be considered a 3rd generation blockchain which builds upon both Bitcoin and Ethereum and aspires to be the world's financial operating system.

# Table of Contents

<b>Preface</b>	<b>3</b>
<b>Table of Contents</b>	<b>4</b>
<b>Introduction</b>	<b>7</b>
<b>Demeter Run</b>	<b>8</b>
Setup	9
Query Tip	10
Workspaces	10
<b>Addresses</b>	<b>11</b>
Base Address	12
Enterprise Address	14
Transactions	15
Datums	17
Redeemers	18
Metadata	18
Cardano Blockchain Architecture	19
UTXO Model	19
eUTXO Model	19
Plutus Scripts	20
Signing Transactions	21
Cardano-cli	22
Bash Shell Scripts	23
Test Drive	24
<b>Transfer Ada</b>	<b>26</b>
Wallet	27
Testnet Faucet	28
Expose Port	29
Running Next.js Application	30
Test Drive	31
Under The Hood	34
<b>Minting</b>	<b>36</b>
NFT	37
Minting Smart Contract	38
Compiling The Script	40
Test Drive	45
Collateral	46

Submit Tx	47
Cexplorer	49
Multisig NFT	50
Code Changes	51
New Functions	54
Test Drive	55
<b>Validators</b>	<b>60</b>
Vesting	61
Next.js Setup	65
Next.js Startup	66
Smart Contract Viewer	67
Lock Ada	68
Vesting Key Minting Policy	69
Claim Funds or Cancel Vesting	70
Test Drive	73
Donation Traceability (Lock Ada)	76
System Components	77
Smart Contract Code	79
Pay With Ada	81
Locking Ada At The Smart Contract	81
Compile & Deploy	82
Next.js Setup	84
Shopify Settings	85
Test Drive	85
Donation Traceability (Unlock Ada)	90
Unlocking Ada At The Smart Contract	91
Bash Shell Scripts	92
Reference Scripts	94
Spend Script	95
Test Drive	100
Before tx-spend.sh is executed	100
After tx-spend.sh is executed	100
<b>Blockchain Monitoring</b>	<b>102</b>
Polling	103
Cron	103
Valid UTXOs	104
Monitoring	105
Events	106
Oura	106
Event Flow	107

WebHook	108
Smart Contract & Next.js Setup	110
Oura Setup	111
Test Drive	112
<b>Troubleshooting</b>	<b>115</b>
Trace()	115
Show()	117
Print()	118
Common Sources Of Errors	119
Cardano-cli	120
Helios Tx Builder	121
Network Parameters	122
Baseline Comparison	123
CBOR.me	123
<b>Production</b>	<b>126</b>
Legal Notice	127
Audit	127
Test, Test, Test	129
Smart Contract Lifecycle	130
<b>Sources and Acknowledgements</b>	<b>131</b>
Open Source	132
References	133
<b>Appendix</b>	<b>134</b>
Address Key Derivation	135
Installing Next.js	137
Startup Next.js	138
IPFS (NFT Images)	140
Shopify Ada Payments	141
Shopify Configuration	142
<b>About Helios</b>	<b>144</b>
<b>About The Author</b>	<b>145</b>

# Introduction

Developing smart contracts for Cardano required learning Haskell PlutusTx which is not easy to learn. An unfamiliar language, cpu and memory execution constraints and time spent on setup creates friction for new developers.

Fortunately, there are some new alternative plutus smart contract languages that eliminate this friction and allow developers to get started quickly. Greater focus can be spent on application and business logic and avoid reduced velocity caused by an uncommon language syntax, time consuming nix-shell/cabal configurations and compilation.

This book will focus on Helios which is a strongly typed, functional programming language whose compiler is written in javascript. One of the main advantages with Helios, is that it can be easily compiled on almost any target architecture that supports javascript. Additionally, there is an approximate 50% performance overhead improvement using Helios vs Haskell PlutusTx V2. Helios is also a transaction builder that can be used in off-chain code to build, sign and submit transactions.



# Demeter Run

Demeter Run is an excellent platform for setting up a Cardano environment quickly. One of the most important aspects of a Cardano development environment is accessing the Cardano node. This would typically involve downloading the [Cardano Node](#) from source or a docker image and then setting it up with the correct configuration. When you are dealing with production, the cardano node needs to run a separate server with 16GB memory, 100GB+ SSD disk, 2 cpu cores and can take some time to sync even if using a Cardano node db [snapshot](#).

Thankfully, this is where Demeter Run comes in. It will create a workspace for you pre-configured and ready to go!

## Setup

1. Go to <https://demeter.run/> and create an account
2. Select a cluster (US or Europe)
3. Select Discover Plan (free)
4. Select Network (Preview/Preprod/Mainnet)
5. Enter Project name and select Create Project
6. After project has been built select Open
7. Select Setup Dev Workspace
8. Select Clone an existing github repository
  - a. <https://github.com/lley154/helios-examples.git>
9. Select your coding stack as Typescript
10. Select workspace as small
11. Select network to connect to as Preprod
12. Scroll down and Select Create Workspace
13. After the workspace has been completed, select OPEN VSCODE top right on the screen
14. A popup window will appear and select the checkbox to trust the authors of the parent folder workspace. Then choose the button "Yes, I trust the authors"
15. Customize Web VS code as needed
16. Select the hamburger menu (top left) -> Terminal -> New Terminal

Now you have access to your workspace.

## Query Tip

Try to run the following command.

```
$ cardano-cli query tip --testnet-magic 1
{
  "block": 526708,
  "epoch": 45,
  "era": "Babbage",
  "hash": "258c7fc2d52fce305720f9d8b3fb4376100f0914a63d77efb73c5098833a299f",
  "slot": 18213433,
  "syncProgress": "100.00"
}
```

The syncProgress is 100% so this means that the cardano node is fully synchronized with the blockchain for the Preprod testnet.

## Workspaces

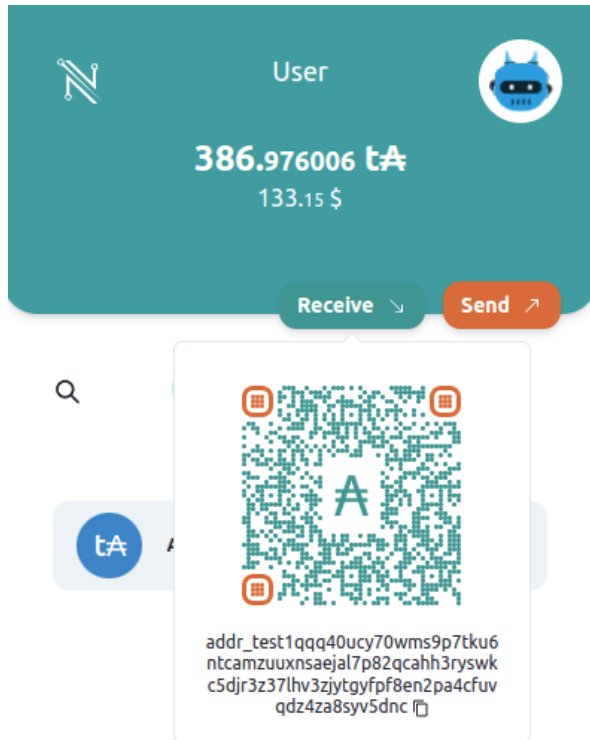
A workspace is temporary, so make sure you commit your work and push it to your online GitHub repository when you are done.

# Addresses

A Cardano Address is at the heart of the Cardano blockchain. Every asset on Cardano needs to reside at an address. There are two types of address that you will typically interact with, a base address or an enterprise address.

## Base Address

A base address is the address you see when receiving Ada from a browser wallet (eg. [Nami](#), [Eternl](#), [Flint](#), etc.)



ⓘ Preprod

In this case, we have the following address:

Addr\_test1qqq40ucy70wms9p7tku6ntcamzuuxnsaeja17p82qcahh3ryswkc5djr3z37lhv3zjygyf8en2pa4cfuvqdz4za8syv5dnc

We can inspect the address with the following command:

```
$ echo
addr_test1qqq40ucy70wms9p7tku6ntcamzuuxnsaeja17p82qcahh3ryswkc5djr3z371hv3zjyt
gyf8pf8en2pa4cfuvqdz4za8syv5dnc | ./utils/cardano-address address inspect
{
  "address_style": "Shelley",
  "address_type": 0,
  "network_tag": 0,
  "spending_key_hash":
    "0157f304f3ddb8143e5db9a9af1dd8b9c34e1dccbbff04ea063b7bc4",
  "spending_key_hash_bech32":
    "addr_vkh1q9t1xp8nmkugp0jahx5678wch8p5u8wvh01sf6sx8daug8cyak6",
  "stake_key_hash":
    "6483ad8a364388a3efdd911488b4112149f33507b5c278c03455174f",
  "stake_key_hash_bech32":
    "stake_vkh1vjp6mz3kgwy28m7ajy2g3dq3y9ylxdg8khp83sp525t57tzwkxk",
  "stake_reference": "by value"
}
```

This is a test address because the `address_type` is 0, but we also see the "addr\_test" prefix which indicates that this is a testnet (eg. preprod) address. If an `address_type` is 1, then it is a production address and will have "addr" as the prefix.

An address is derived from both a `spending_key_hash` and a `stake_key_hash`. The `spending_key_hash` is more commonly known as the public key hash or PKH. The `stake_key_hash` is used when you stake your Ada from a wallet to a [stake pool](#) operator.

## Enterprise Address

Another type of address commonly used is called an enterprise address.

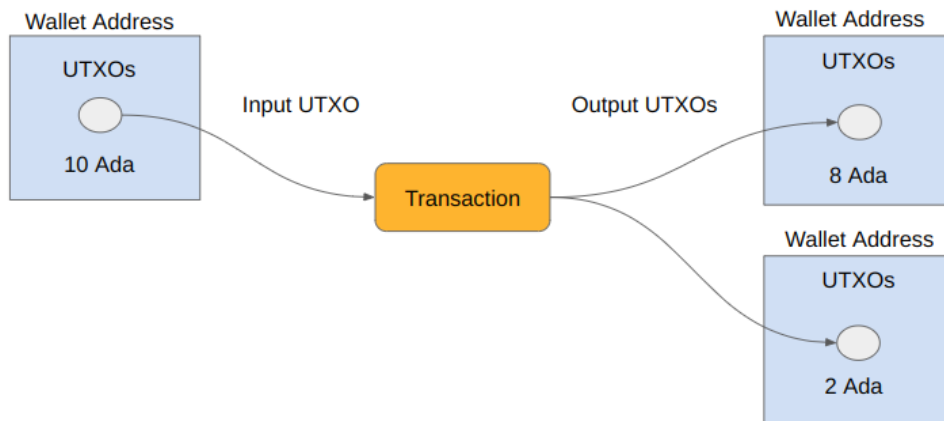
```
$ echo addr_test1vq5s7k4kwqz4rrfe8mm9jz9tpm7c5u93yfwwsaw708yxs5sm70qjg |  
./utils/cardano-address address inspect  
{  
  "address_style": "Shelley",  
  "address_type": 6,  
  "network_tag": 0,  
  "spending_key_hash":  
"290f5ab67005518d393ef65908ab0efd8a70b1225ce875de79c86852",  
  "spending_key_hash_bech32":  
"addr_vkh19y844dnsq4gc6wf77evs32cwlk98pvfztn58thneep59yhwpvvm",  
  "stake_reference": "none"  
}
```

Enterprise addresses do not have a stake\_key\_hash only a spending\_key\_hash.

For more info on how to install `cardano-address` and derive addresses from a passphrase, please see the appendix section on Address Key Derivation.

## Transactions

Transactions are used to transfer assets from one address to another. Every transaction contains one or more inputs and one or more outputs. Here is a simplified diagram showing the transfer of Ada using a Cardano transaction.



A UTXO is an unspent transaction output. A Transaction consumes one or more UTXOs and then produces one or more UTXOs where each UTXO is locked at an address. When trying to determine the amount of assets at an address, we are actually counting the asset amounts in each UTXO locked at an address.



Here is an example of 2 UTXOs at an address \*

```
$ cardano-cli query utxo --address addr_test1vzu...dxn7 --cardano-mode
--testnet-magic 1
TxHash TxIx Amount
-----
158...925 0 3551912116 lovelace + TxOutDatumNone
b4e...0d2 0 5000000 lovelace + TxOutDatumNone
```

\* Please note that the address and TxHash have been abbreviated so they can fit properly on this page.

#### Observations:

- In this example, there are 2 UTXOs
- Transaction hash 158...925 with index 0 has 3,551,912,116 lovelace
- Transaction hash b4e...0d2 with index 0 has 5,000,000 lovelace
- There is 1,000,000 lovelace in 1 Ada
- The amount of Ada locked at the address is

$$\begin{array}{r} 3,551.912116 \\ + 5.000000 \\ \hline 3,556.912116 \end{array}$$

## Datums

In the previous example you may have noticed `TxOutDatumNone` at the end of the UTXO output. This field is used to store persistent information in the UTXO and is called the datum. A datum is created when building and submitting a transaction. `TxOutDatumNone` indicates that there is no datum information associated with this UTXO. Here is an example where an inline datum is present with a UTXO.

```
cardano-cli$ cardano-cli query utxo --address addr_test1wrm...hm3f
--cardano-mode --testnet-magic 1
TxHash TxIx Amount
-----
b07...91da 0 88270000 lovelace + TxOutDatumInline ... (ScriptDataConstructor 0
[ScriptDataNumber 87770000,ScriptDataBytes "4666620215361",ScriptDataBytes
"0.34180"])
b87...107e 1 20000000 lovelace + TxOutDatumNone
```

`TxOutDatumInline` indicates that there is an inline datum present. A datum can also be the hash value of the actual datum. Using a datum hash was the only way datums could be stored on-chain prior to the Vasil hard fork in Fall of 2022. Unless the datum size is large, inline datums are typically the preferred approach.

## Redeemers

Redeemers are also included in a smart contract transaction and may be used to execute different conditions depending on the value of the redeemer. Additionally, Redeemers can be used to pass data to a smart contract at runtime during transaction construction.

## Metadata

A transaction can also generate metadata that is stored on-chain. This can provide human readable information about a transaction and can also be leveraged by applications. Either technique (Datums and/or Metadata) both store persistent data on-chain. The decision on what technique to use depends on how the data will be accessed. If the data needs to be validated inside a smart contract, then use a datum. If you need to provide transaction details to an off-chain application, token minting or human verification, then use metadata.

# Cardano Blockchain Architecture

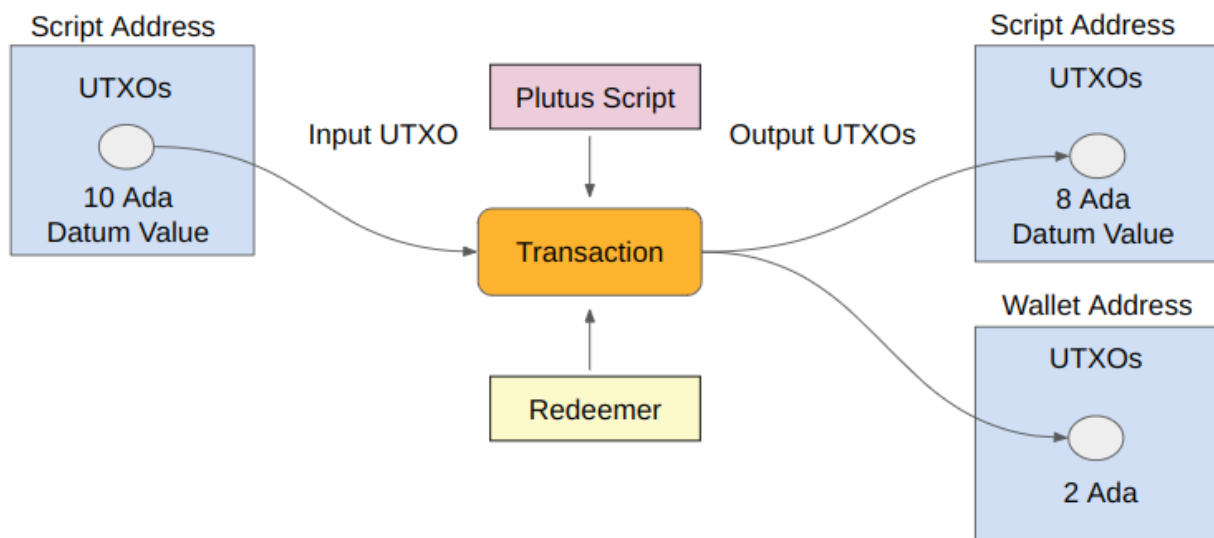
Cardano uses an *extended* UTXO blockchain architecture. What this means is that in addition to using UTXOs for assets, there is the capability to add a datum value to UTXOs as well.

## UTXO Model

In a UTXO model (eg. Bitcoin), the owner of a private key(s) for an address is the only one that can spend a UTXO locked at that address.

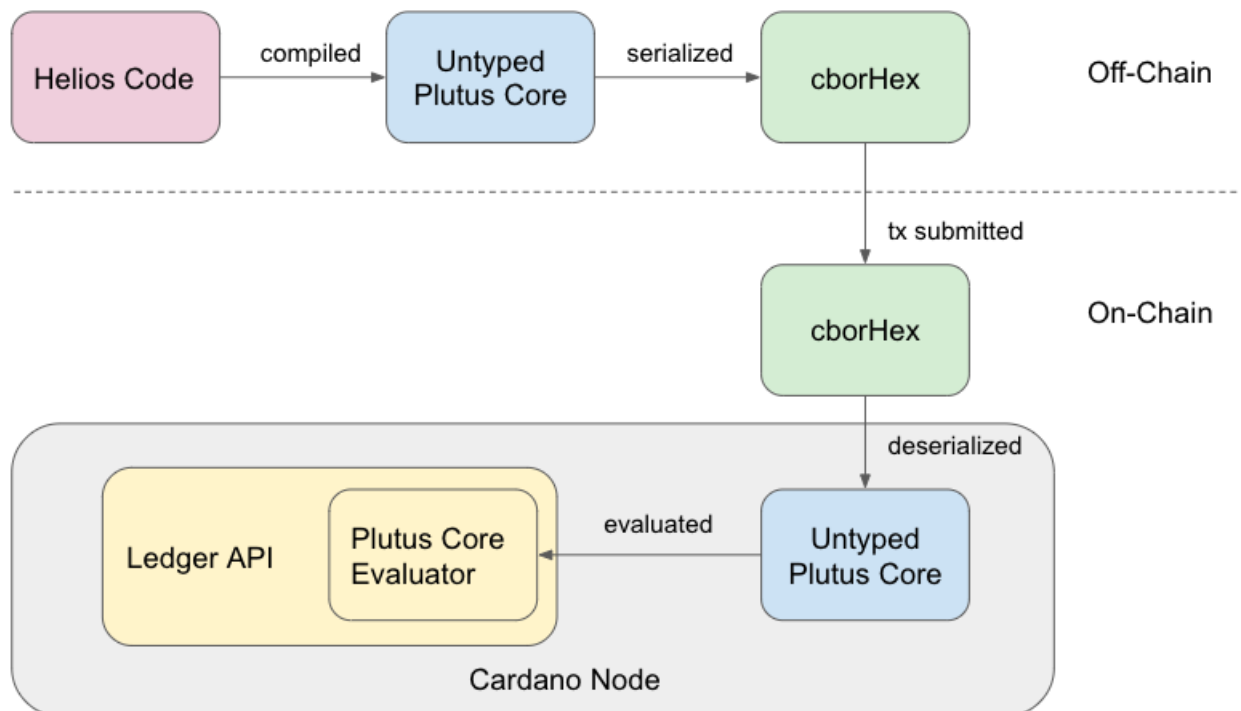
## eUTXO Model

In a eUTXO model (eg. Cardano), a [Turing-complete](#) script can spend a UTXO locked at a script address. Below is a diagram showing a simplified smart contract transaction where 10 Ada is locked at a Plutus Script address.



## Plutus Scripts

Helios code is compiled to Untyped Plutus Core which is the plutus script format needed by the Plutus Core Evaluator. When you compile and submit a transaction with Helios, the smart contract is first evaluated by a local Helios virtual machine. This leverages the [deterministic](#) nature of Cardano where you find out **before** you submit the transaction if it will succeed or fail and the transaction fee. Below is a simplified diagram showing the process of getting a plutus script to the Cardano node and evaluating it as part of a transaction.



## Signing Transactions

In order for a transaction to be executed successfully on the Cardano blockchain, it must contain valid signatures that demonstrate that the owners of the UTXO(s) being spent have agreed to this. This agreement is done by signing the hash of the transaction body with the owner's private key. The signatures are then bundled into a group of transactions known as a transaction witness set. Once the signatures for each UTXO being spent has been included, the transaction can be sent to the blockchain for verification.

## Cardano-cli

Cardano-cli is a widely used command line tool for querying and executing transactions on the Cardano blockchain. It is helpful to be familiar with this tool because by using it, you will have a better understanding on how transactions are constructed, signed and submitted. Cardano-cli is also used in backend servers and batch processing.

## Bash Shell Scripts

Using the cardano-cli tool on the command line is awkward so it is more convenient to use it inside a bash shell script or Node.js. Here is a simple bash shell script for executing a simple cardano-cli transaction.

```
#!/usr/bin/env bash
source_addr=addr_test1vq5s7k4kwqz4rrfe8mm9jz9tpm7c5u93yfwwsaw708yxs5sm70qjg
source_utxo=3188f0f28667f753f864b39475acb3d55cf27e146fa6414ac57f5c6c63c705c2#0
destination_addr=addr_test1qqq967dwdp009smfeqtzhve89fyuqjydkvwc9md5atyq2429gnmszjc7hyf
685vp7qxeffjd568s3p234fg5ryhrkvjsn7muqm
user_skey=/config/workspace/repo/.keys/user/key.skey
network="--testnet-magic 1"

# generate param file from cardano-cli tool
cardano-cli query protocol-parameters $network --out-file pparams.json

cardano-cli transaction build --babbage-era --cardano-mode \
    $network \
    --change-address "$source_addr" \
    --tx-in "$source_utxo" \
    --tx-out "$destination_addr+2000000" \
    --protocol-params-file pparams.json \
    --out-file transfer-tx-alonzo.body
echo "tx has been built"

cardano-cli transaction sign \
    --tx-body-file transfer-tx-alonzo.body \
    $network \
    --signing-key-file "$user_skey" \
    --out-file transfer-tx-alonzo.tx
echo "tx has been signed"

echo "Submit the tx with plutus script and wait 5 seconds..."
cardano-cli transaction submit --tx-file transfer-tx-alonzo.tx $network
```



## Test Drive

Here is the **before** snapshot of the source and destination addresses.

### Source Wallet Address

```
$ cardano-cli query utxo --address addr_test1vq5...qjg --cardano-mode
--testnet-magic 1
TxHash TxIx Amount
-----
db34...3ee 0 10000000 lovelace + TxOutDatumNone
```

### Destination Wallet Address

```
$ cardano-cli query utxo --address addr_test1vq7k...pds --cardano-mode
--testnet-magic 1
TxHash TxIx Amount
-----
```

Then we run the script

```
$ ./transfer-tx.sh preprod
Estimated transaction fee: Lovelace 165721
tx has been built
tx has been signed
Submit the tx with plutus script and wait 5 seconds...
Transaction successfully submitted.
```

Here is the snapshot **after** the transaction of source and destination addresses

### Source Wallet Address

```
$ cardano-cli query utxo --address addr_test1vq5s...qjg --cardano-mode
--testnet-magic 1
TxHash TxIx Amount
-----
```

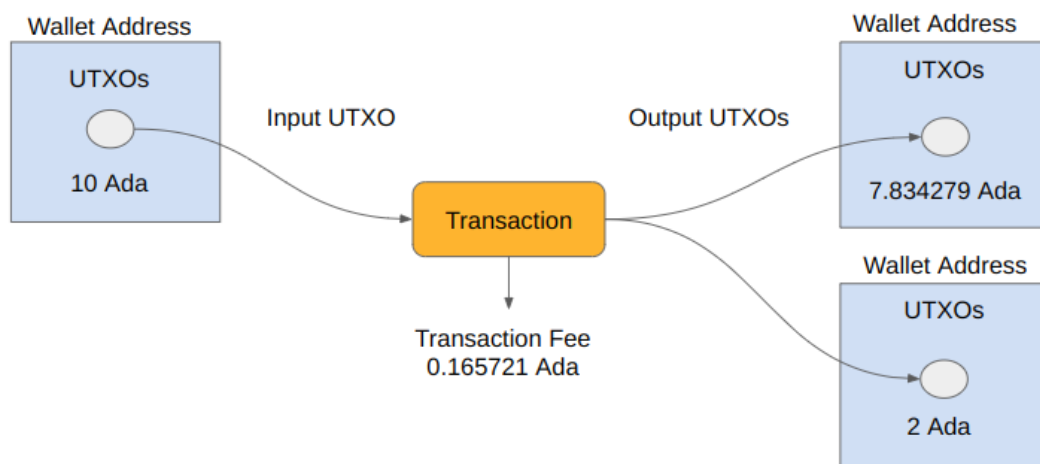
```
996...95c 1 7834279 lovelace + TxOutDatumNone
```

### Destination Wallet Address

```
$ cardano-cli query utxo --address addr_test1vq7...pds --cardano-mode
--testnet-magic 1
TxHash TxIx Amount
-----
```

```
996...95c 0 2000000 lovelace + TxOutDatumNone
```

So, when including the fees, the simple transaction diagram looks like the following.



# Transfer Ada

Helios is both a language that compiles to plutus and an off-chain transaction builder. In this section we will review how to construct a transfer Ada transaction using Helios with a front-end web application.

## Wallet

You will need a Chrome browser wallet extension installed in your browser. For the examples in this book, we will use [Nami wallet](#). After you have installed Nami wallet, please do the following to ensure your wallet is on the correct network.

1. Select the Extensions icon on your browser and select Nami wallet
2. Select the account icon image (top right)
3. Select Settings near the bottom of the popup window
4. Select Network
5. Select Preprod in the drop down menu
6. Go back to the main view and select the account icon image again
7. Now select New Account
8. Create another account name & password
9. You can switch between accounts by selecting the account icon image

## Testnet Faucet

You will need to get some test Ada.

1. Go to the Nami account you want to receive some Ada with and select the Receive button
2. Copy the receiving address to your clipboard
3. Open a new browser tab and go to the [Cardano testnet faucet](#) page
4. In the Environment dropdown, select the Preprod Network
5. In the Address field, past the receiving address you copied in step 2
6. Select Request Funds
7. You may have to wait 10-60 seconds before the funds are in your wallet.

Note: you can only request test Ada once every 24 hours.

## Expose Port

To be able to access a web app with Demeter, you will need to expose the port that the web app binds to. The default port is 3000, so we will expose this port and will be provided with a URL to access the web application.

1. Select your workspace on your Demeter Dashboard
2. Select the Exposed Ports tab
3. Select Expose Port+ button
4. Enter a port name (eg Next.js) and port number (eg 3000)

## Running Next.js Application

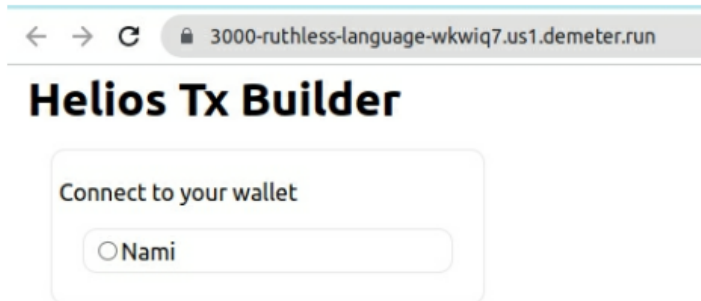
In a VSCode Web Terminal window, execute the following commands to setup and then start the Next.js application

```
$ cd transfer-ada  
$ npm install  
$ npm run dev
```

Use the Exposed Port URL provided above to access the next.js application.

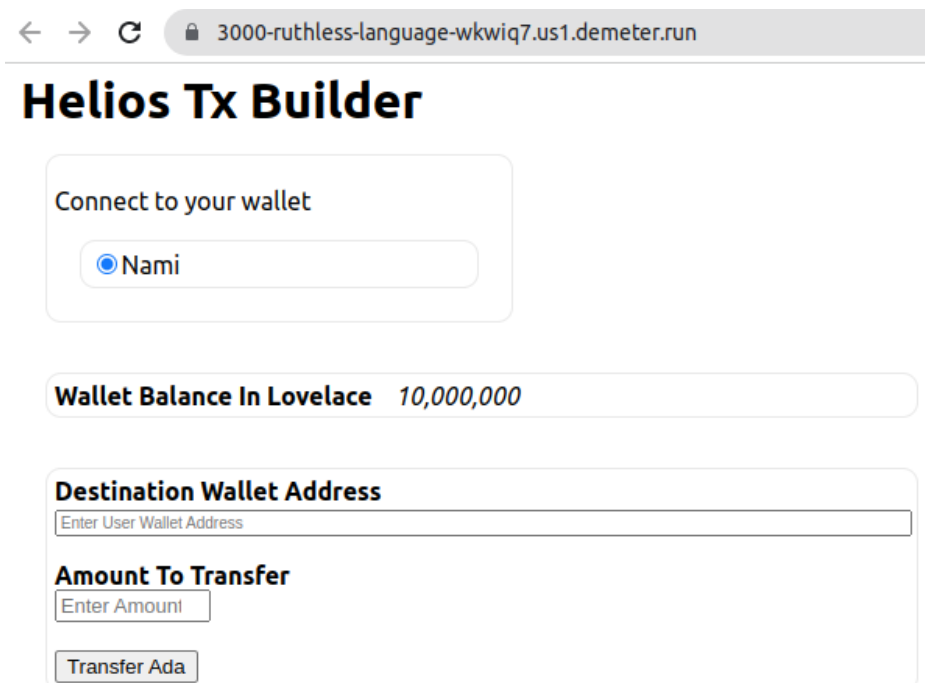
## Test Drive

With the Next.js application now running, select the Nami selector to connect your wallet.



A screenshot of a web browser showing the Helios Tx Builder interface. The browser's address bar displays the URL `3000-ruthless-language-wkwiq7.us1.demeter.run`. The page title is "Helios Tx Builder". Below the title, there is a section titled "Connect to your wallet" containing a single radio button labeled "Nami".

You will now see the following showing the available lovelace balance for the wallet currently selected.



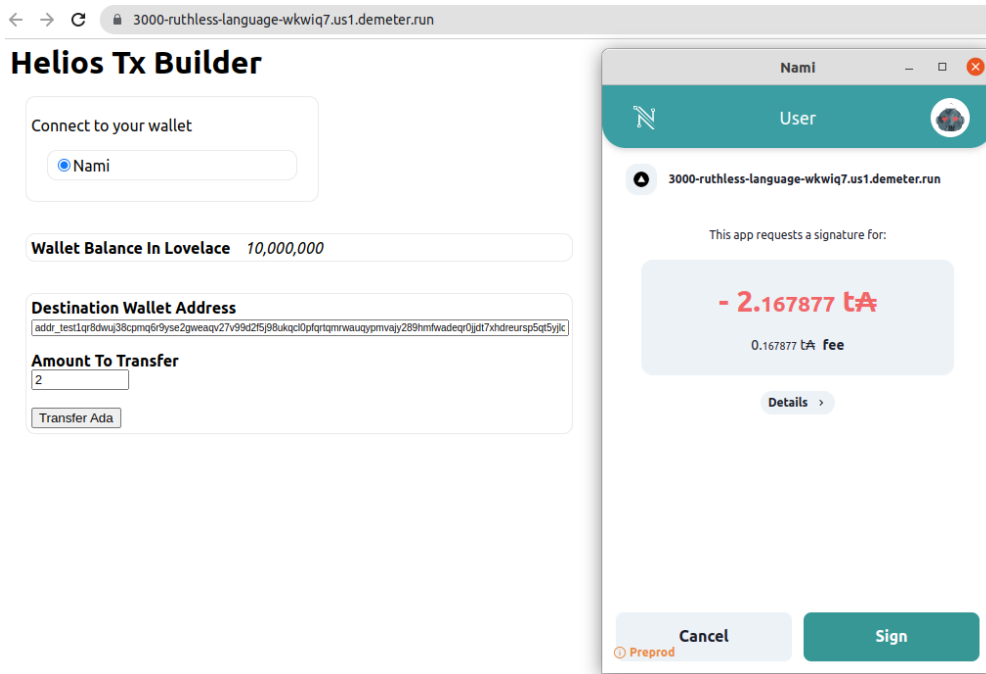
A screenshot of the Helios Tx Builder web application. The browser address bar shows the same URL. The page title is "Helios Tx Builder". Below the title, the "Connect to your wallet" section now shows the "Nami" radio button selected with a blue dot. Below this section, a box displays "Wallet Balance In Lovelace" followed by the value `10,000,000`. Further down, there is a section titled "Destination Wallet Address" with a text input field containing the placeholder "Enter User Wallet Address". Below that is a section titled "Amount To Transfer" with a text input field containing the placeholder "Enter Amount". At the bottom of this section is a button labeled "Transfer Ada".



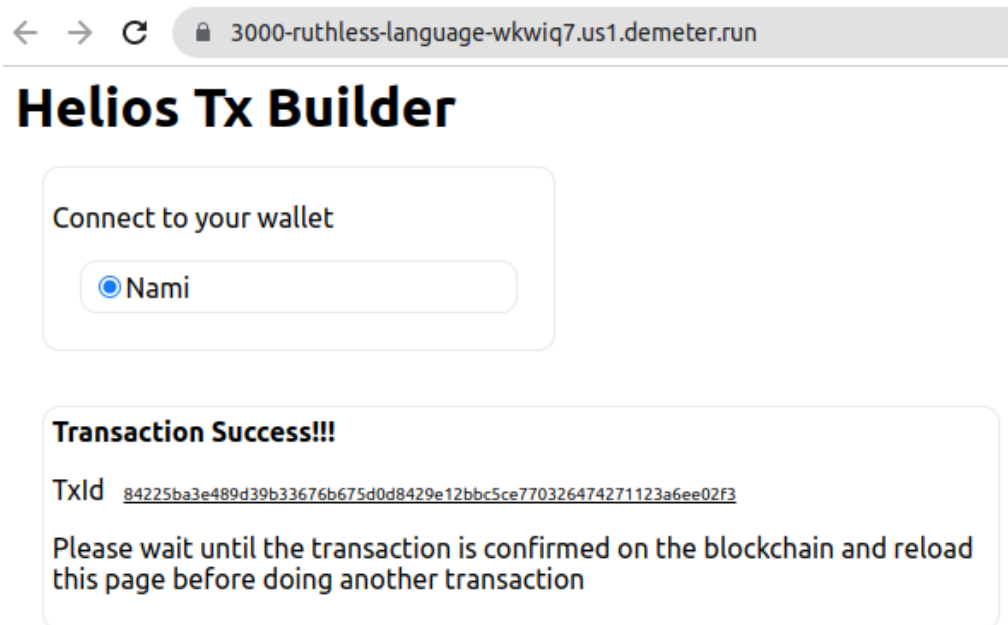
For the other Nami account that does not have any account balance, switch to that account and copy a receiving address. Make sure you **switch back** to the account that has funds available to transfer.

Enter the receiving address and the amount of Ada you want to transfer and select the Transfer Ada button.

The Nami wallet should appear for you to sign the transaction.



After you sign by entering your wallet password, you should see the following success message



You can select the TxId link to confirm the transaction using the blockchain explorer [cexplorer](#) or verify in your Nami wallet.

## Under The Hood

The first step is detecting if the browser wallet exists.

```
const checkIfWalletFound = async () => {  
  
  let walletFound = false;  
  const walletChoice = whichWalletSelected;  
  if (walletChoice === "nami") {  
    walletFound = !!window?.cardano?.nami;  
    return walletFound;  
  }  
}
```

The second step is creating an API object that the Tx builder will be able to use.

```
const enableWallet = async () => {  
  
  try {  
    const walletChoice = whichWalletSelected;  
    if (walletChoice === "nami") {  
      const handle: Cip30Handle  
        = await window.cardano.nami.enable();  
      const walletAPI = new Cip30Wallet(handle);  
      return walletAPI;  
    }  
  } catch (err) {  
    console.log('enableWallet error', err);  
  }  
}
```

Now the Helios Tx can be built using WalletHelper to get key information such as the wallet UTXOs and the change address of the wallet.

```
...
const adaAmountVal = new Value(BigInt((adaQty)*1000000));

// Get wallet UTXOs
const walletHelper = new WalletHelper(walletAPI);
const utxos = await walletHelper.pickUtxos(adaAmountVal);

// Get change address
const changeAddr = await walletHelper.changeAddress;

// Start building the transaction
const tx = new Tx();
tx.addInputs(utxos[0]);

// Add the destination address and the amount of Ada to send
tx.addOutput(new TxOutput(Address.fromBech32(address), adaAmountVal));

const networkParams = new NetworkParams(
  await fetch(networkParamsUrl)
    .then(response => response.json())
)
// Send any change back to the buyer
await tx.finalize(networkParams, changeAddr);

console.log("Verifying signature...");
const signatures = await walletAPI.signTx(tx);
tx.addSignatures(signatures);

console.log("Submitting transaction...");
const txHash = await walletAPI.submitTx(tx);
...
```

# Minting

Minting is the on-chain process of creating new native tokens that are included in UTXO(s) and locked at an address. A native token has the same properties as Ada and is considered a first class citizen on the blockchain. A minting policy script is an efficient mechanism for creating/burning tokens and defines the rules for a minting or burning transaction.

## NFT

A NFT is a non-fungible token that can't be divisible which means that there is only one in existence. So the best way to ensure you are only creating one unique token is to leverage the unique property of a UTXO. If we recall, a UTXO is an unspent transaction output. So once a UTXO is spent, it is no longer an ***unspent*** transaction output. So as long as we include a UTXO in our transaction input and spend it, no other transaction will be able to spend that UTXO.

## Minting Smart Contract

Every Helios smart contract must contain a `func main` where the validation logic occurs. To specify the type of plutus script, we state on the first line `minting` and the name of the program `NFT`.

```
minting nft
```

Next we need to specify the UTXO that will be used as an input and hard code it as a parameter value. Therefore, create a `TxOutputId` object with both the transaction ID of `"#6e1...996"` and a transaction index of 0.

```
const TX_ID: ByteArray = #6e1...996
const txId: TxId = TxId::new(TX_ID)
const outputId: TxOutputId = TxOutputId::new(txId, 0)
```

Since we set a UTXO as a contract parameter, the minting policy hash will be different (unique) based on the value of this parameter.

The `func main` is a required function for all Helios plutus scripts. This is where you can get access to important objects such as the `ScriptContext`. The `ScriptContext` object has information about the current transaction (#1) which is used in the validation logic. We also get the minting policy hash (#2) of this minting policy script and use it to construct an `Asset Class` object (#3) for the token we are minting. Finally, we get the token that was actually minted (#4) as part of this transaction via the `ScriptContext Tx` object.

```
func main(ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx; // #1
    mph: MintingPolicyHash = ctx.get_current_minting_policy_hash(); // #2
    assetclass: AssetClass = AssetClass::new(
        mph,
        "NFT Token Name".encode_utf8()
    ); // #3
    value_minted: Value = tx.minted; // #4
```

Now we are ready to do the actual validation. First we will check that the NFT has the correct minting policy hash, token name, quantity of 1 and has been minted as part of this transaction.

```
(value_minted == Value::new(assetclass, 1)).trace("NFT1: ")
```

Second, we check that the UTXO that we specified as a parameter to the smart contract is actually included as a transaction input.

```
tx.inputs.any((input: TxInput) -> Bool {  
    (input.output_id == outputId).trace("NFT2: ")
```

The `.trace("NFT: ")` function will generate a trace error message with the evaluated value if the validator returns false. This is very helpful to isolate what conditions are true and which one is false.

So, all together, a complete NFT minting smart contract is as follows.

```
minting nft
```

```
const TX_ID: ByteArray = #6e1...996  
const txId: TxId = TxId::new(TX_ID)  
const outputId: TxOutputId = TxOutputId::new(txId, 0)  
func main(ctx: ScriptContext) -> Bool {  
    tx: Tx = ctx.tx;  
    mph: MintingPolicyHash = ctx.get_current_minting_policy_hash();  
    assetclass: AssetClass = AssetClass::new(  
        mph,  
        "NFT Token Name".encode_utf8()  
    );  
    value_minted: Value = tx.minted;  
    // Validator logic starts  
    (value_minted == Value::new(assetclass, 1)).trace("NFT1: ") &&  
    tx.inputs.any((input: TxInput) -> Bool {  
        (input.output_id == outputId).trace("NFT2: ")  
    })  
}
```



## Compiling The Script

We have a number of options to get smart contract code onto the blockchain.

- Store the script inside the Next.js app and compile it in the browser and then submit it as part of a transaction
- Pre-compile the script ahead of time and submit it as part of a transaction
- Pre-compile the script ahead of time and load it on-chain as a plutus reference script. Other transactions can simply use the on-chain reference script and do not need a copy of it.

For this simple contract, we will just include it in the Next.js app we built, compile it on the fly and submit the transaction. The fact that the Helios compiler is written in javascript gives us a lot of flexibility on when and where we compile to plutus code.

Below is the on-chain and off-chain code for the mintNFT function. It includes dynamic compiling of Helios code with runtime variables as contract parameters. Also note that there is a specific Metadata data type that needs to be created so it can be attached to a transaction.

```

const mintNFT = async (params : any) => {

    const address = params[0];
    const name = params[1];
    const description = params[2];
    const img = params[3];
    const minAda: number = 2000000; // minimum Ada for NFT
    const maxTxFee: number = 500000; // max estimated tx fee
    const minChangeAmt: number = 1000000; // min Ada for change
    const minAdaVal = new Value(BigInt(minAda));
    const minUTXOVal = new Value(BigInt(minAda + maxTxFee + minChangeAmt));

    // Get wallet UTXOs
    const walletHelper = new WalletHelper(walletAPI);
    const utxos = await walletHelper.pickUtxos(minUTXOVal);
    // Get change address
    const changeAddr = await walletHelper.changeAddress;
    // Determine the UTXO used for collateral
    const colatUtxo = await walletHelper.pickCollateral();
    // Start building the transaction
    const tx = new Tx();
    // Add the UTXOs as inputs
    tx.addInputs(utxos[0]);

```

```

const mintScript = `minting nft

const TX_ID: ByteArray = #` + utxos[0][0].txId.hex + `
const txId: TxId = TxId::new(TX_ID)
const outputId: TxOutputId =
    TxOutputId::new(txId, ` + utxos[0][0].utxoIdx + `)

func main(ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx;
    mph: MintingPolicyHash = ctx.get_current_minting_policy_hash();

    assetclass: AssetClass = AssetClass::new(
        mph,
        "` + name + `".encode_utf8()
    );
    value_minted: Value = tx.minted;

    // Validator logic starts
    (value_minted == Value::new(assetclass, 1)).trace("NFT1: ") &&
    tx.inputs.any((input: TxInput) -> Bool {
        (input.output_id == outputId).trace("NFT2: ")
    })
}

// Compile the helios minting script
const mintProgram = Program.new(mintScript).compile(optimize);
// Add the script as a witness to the transaction
tx.attachScript(mintProgram);
// Construct the NFT that we will want to send as an output
const nftTokenName = ByteArrayData.fromString(name).toHex();
const tokens: [number[], bigint][] = [[hexToBytes(nftTokenName),
    BigInt(1)]];

```

```

// Create an empty Redeemer because we must always send a Redeemer with
// a plutus script transaction even if we don't actually use it.
const mintRedeemer = new ConstrData(0, []);
// Indicate the minting we want to include as part of this transaction
tx.mintTokens(
  mintProgram.mintingPolicyHash,
  tokens,
  mintRedeemer
)
// Construct the output and include both the minimum Ada as
// well as the minted NFT
tx.addOutput(new TxOutput(
  Address.fromBech32(address),
  new Value(minAdaVal.lovelace, new Assets([[mintProgram.mintingPolicyHash,
                                              tokens]]))
));
// Add the collateral utxo
tx.addCollateral(colatUtxo);
const networkParams = new NetworkParams(
  await fetch(networkParamsUrl)
    .then(response => response.json())
)
// Attached the metadata for the minting transaction
tx.addMetadata(721, {"map":
  [[mintProgram.mintingPolicyHash.hex,
    {"map": [[name, { "map": [{"name", name},
                            ["description", description],
                            ["image", img]
                          ]
                        }
                    ]
                ]
            ]
  ]
});

```

```
// Send any change back to the buyer
await tx.finalize(networkParams, changeAddr);

console.log("Verifying signature...");
const signatures = await walletAPI.signTx(tx);
tx.addSignatures(signatures);

console.log("Submitting transaction...");
const txHash = await walletAPI.submitTx(tx);

console.log("txHash", txHash.hex );
setTx({ txId: txHash.hex });
}
```

## Test Drive

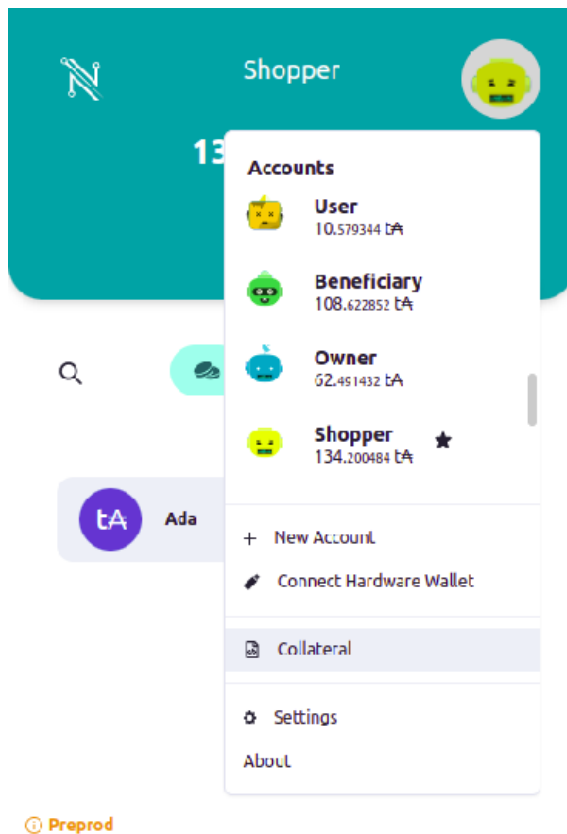
You can now mint your own NFT by starting up the Next.js application and submitting the transaction.

```
$ cd nft/app  
$ npm install  
$ npm run dev
```

Make sure you expose port 3000 for your workspace in Demeter Run, on the Exposed Ports tab. Then use the URL on the Exposed Ports tab to access the Next.js application.

## Collateral

A small amount of Ada (~5 Ada) is used as collateral to help protect the network against spam attacks when executing smart contracts. Open up your Nami wallet and go to the account you want to mint the NFT from. Select the account icon and then select Collateral in the popup window.



## Submit Tx

Select your wallet and then enter in the required information

[←](#) [→](#) [↻](#) [🔒 3000-precious-baseball-lfkchu.us1.demeter.run](#)

# Helios Tx Builder

Connect to your wallet

☒ Nami

**Wallet Balance In Lovelace** 1,455,260,289

**Destination Wallet Address**

addr\_test1qz7gedd4dv6y57dxc0l2awp2uq08tzt5ygyuzrs2qvncr5378p8rj3dgxkjg5knpa3uvtwggqd48t66ad05eexasgq5prv

**NFT Token Name**

Mad Dog

**NFT Description**

Crazy dog that loves treats

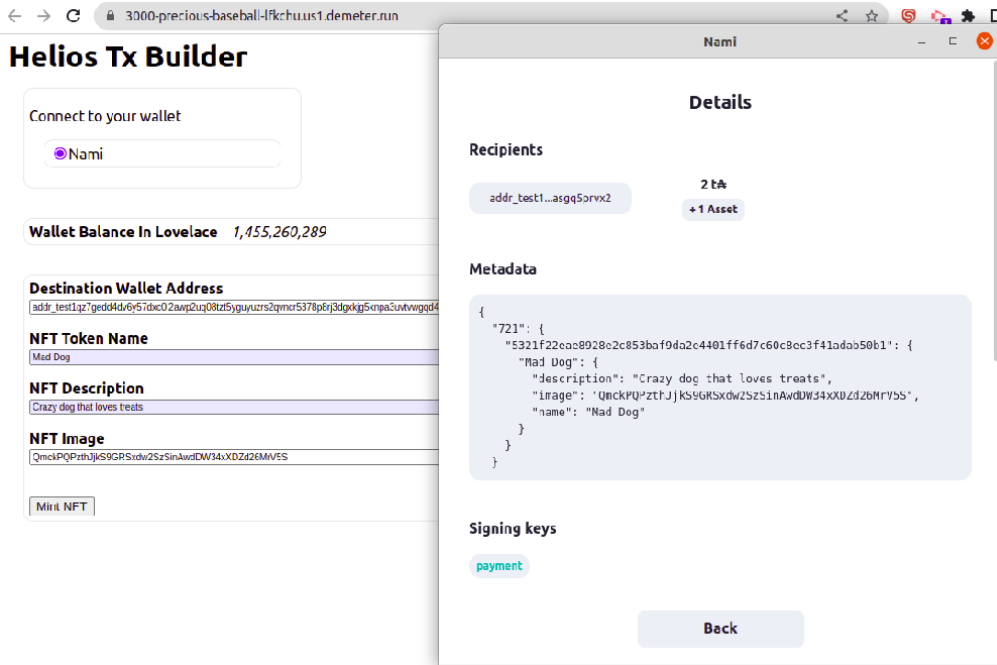
**NFT Image**

QmckPQPzthJjkS9GRSxdw2SzSinAwdDW34xXDZd26MrV5S

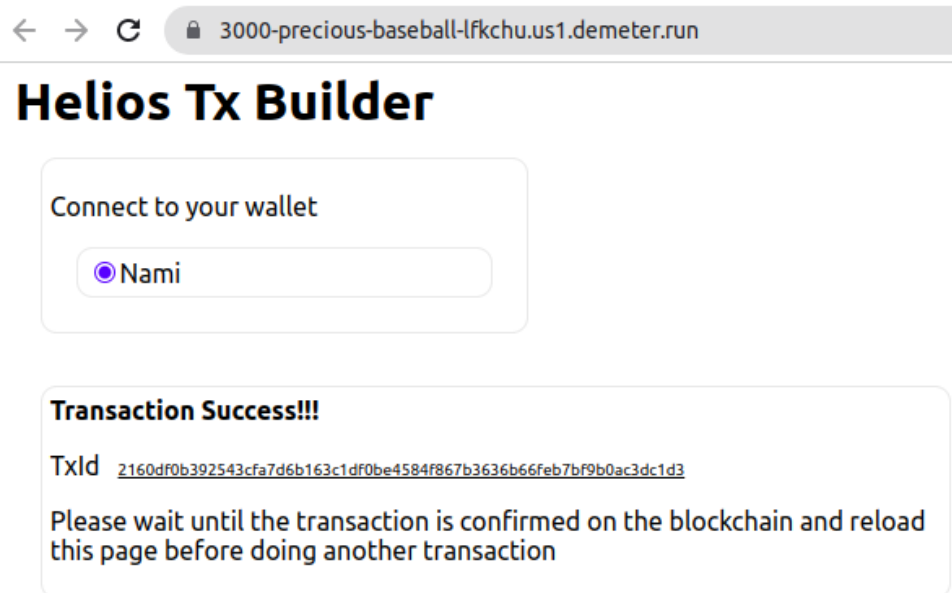
Mint NFT

When you select the Mint NFT button and are asked to sign the transaction, you can go to the Details tab in Nami wallet and see the NFT being minted including the metadata.





After submitting the transaction and waiting 10-60 seconds, you should be able to see the NFT on the blockchain explorer.




<https://preprod.cexplorer.io/tx/2160df0b392543cfa7d6b163c1df0be4584f867b3636b66feb7bf9b0ac3dc1d3>

## Cexplorer

To see the NFT on Cexplorer, do the following steps:

1. Select the TxId link in the Transaction Success box
2. Scroll down and select the Mint Tab
3. Select that Asset (in this case Mad Dog)
4. And then you will see all the properties of this NFT

← → ↻ 🔒 preprod.cexplorer.io/asset/asset1urm04nzzj2qru43p08swg7rzztw9kl7ru73hdy/preview#data

 **Cexplorer.io** preprod

CARDANO EXPLORER

Dashboard

Watchlist

Pools

Assets

Blocks

Transactions

dApps

Metadata

More

EDUCATION

Articles

Videos

Wiki


ANALYTICS

Decentralization

Network


Assets / Policy 5321f2eae8928e2c853baf... / Mad Dog / NFT preview

☆ Mad Dog > NFT preview

Fingerprint:  asset1urm04nzzj2qru43p08swg7rzztw9kl7ru73hdy

Name (onchain)

Mad Dog

Encoded:  4d616420446f67

Onchain data (epoch snapshot)

Supply1

Ownerstake\_test...7n4dr

Preview

TXs

Mints


Metadata

Owners

Embed

Mad Dog

Note: Data are loaded directly from blockchain and decentralised ipfs storage and are NOT stored or hosted on our website.



Metadata

```
{
  "name": "Mad Dog",
  "image": "QmckPQPzthJjks9GRSxdw2SzSsInAwdDW34xxDZd26MrV5S",
  "description": "Crazy dog that loves treats"
}
```

49

## Multisig NFT

The reason why you may want to have multi-signature (multisig) signing during an NFT mint is that you want the buyer to be able to confirm what he is getting before it is actually minted and submitted to the blockchain. By including the public key hashes of both the buyer and seller in the minting policy, we can ensure that only when both signatures have been obtained will we be able to mint the NFT.

Your application will typically need to store the transaction and witness updates. In this example, the Next.js application stores the transaction in memory for simplicity. This could also have been stored in a database, filesystem or even included in a message to the recipients who need to sign the transaction and then pass it along to the next witness.

We will use the NFT minting script from the previous section and simply add 2 signers that must include their signature in the transaction. Since we are able to dynamically compile the plutus script in the browser, we will be able to add the signers to the plutus script as we build it.

## Code Changes

Here is what the updated off-chain and on-chain code and NFT minting script will look like.

```
const mintNFT = async (params : any) => {

    const address = params[0];
    const name = params[1];
    const description = params[2];
    const img = params[3];
    const sellerAddr = params[4];

    const buyerPkh = Address.fromBech32(address).pubKeyHash;
    const sellerPkh = Address.fromBech32(sellerAddr).pubKeyHash;
    const minAdaVal = new Value(BigInt(2000000)); // minimum Ada needed to send an NFT

    // Get wallet UTXOs
    const walletHelper = new WalletHelper(walletAPI);
    const utxos = await walletHelper.pickUtxos(minAdaVal);
    // Get change address
    const changeAddr = await walletHelper.changeAddress;
    // Determine the UTXO used for collateral
    const colatUtxo = await walletHelper.pickCollateral();
    // Start building the transaction
    const tx = new Tx();
    // Add the UTXO as inputs
    tx.addInputs(utxos[0]);

    const mintScript = `minting nft

    const TX_ID: ByteArray = #` + utxos[0][0].txId.hex + `
    const txId: TxId = TxId::new(TX_ID)
    const outputId: TxOutputId = TxOutputId::new(txId, ` + utxos[0][0].utxoIdx + `)
    const BUYER: PubKeyHash = PubKeyHash::new(#` + buyerPkh.hex + `)
    const SELLER: PubKeyHash = PubKeyHash::new(#` + sellerPkh.hex + `)
```

```

func main(ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx;
    mph: MintingPolicyHash = ctx.get_current_minting_policy_hash();
    assetclass: AssetClass = AssetClass::new(
        mph,
        "` + name + `".encode_utf8()
    );
    value_minted: Value = tx.minted;

    // Validator logic starts
    (value_minted == Value::new(assetclass, 1)).trace("NFT1: ") &&
    (tx.inputs.any((input: TxInput) -> Bool {
        (input.output_id == outputId).trace("NFT2: ")
    }) &&
    tx.is_signed_by(BUYER).trace("NFT3: ") &&
    tx.is_signed_by(SELLER).trace("NFT4: ")
    )
}

// Compile the helios minting script
const mintProgram = Program.new(mintScript).compile(optimize);
// Add the script as a witness to the transaction
tx.attachScript(mintProgram);
// Construct the NFT that we will want to send as an output
const nftTokenName = ByteArrayData.fromString(name).toHex();
const tokens: [number[], bigint][] = [[hexToBytes(nftTokenName), BigInt(1)]];

// Create an empty Redeemer because we must always send a Redeemer with
// a plutus script transaction even if we don't actually use it.
const mintRedeemer = new ConstrData(0, []);
// Indicate the minting we want to include as part of this transaction
tx.mintTokens(
    mintProgram.mintingPolicyHash,
    tokens,
    mintRedeemer
)

```

```

// Construct the output and include both the minimum Ada as well as the minted NFT
tx.addOutput(new TxOutput(
    Address.fromBech32(address),
    new Value(minAdaVal.lovelace, new Assets([[mintProgram.mintingPolicyHash,
tokens]]))
));
// Add the collateral utxo
tx.addCollateral(colatUtxo);
// Add buyer and seller required PKHs for the tx
tx.addSigner(buyerPkh);
tx.addSigner(sellerPkh);

const networkParams = new NetworkParams(
    await fetch(networkParamsUrl)
        .then(response => response.json())
)
// Attached the metadata for the minting transaction
tx.addMetadata(721, {"map": [[mintProgram.mintingPolicyHash.hex, {"map": [[name,
    {
        "map": [[{"name", name},
            [{"description", description},
            [{"image", img}
        ]
    ]
    }
]]}
]]}
]);

console.log("tx before final", tx.dump());
// Send any change back to the buyer
await tx.finalize(networkParams, changeAddr);
console.log("tx after final", tx.dump());
// Store the transaction so it can be signed by the buyer and the seller
setTxBodyBuyer(tx);
}

```

## New Functions

Next, we need to add 2 new functions in the off-chain code to handle the signing of the buyer and the signing and submitting of the seller.

```
const buyerSign = async () => {

  console.log("Verifying buyer signature...");
  const signatures = await walletAPI.signTx(txBodyBuyer);
  txBodyBuyer.addSignatures(signatures);

  console.log("buyerSigned", txBodyBuyer);
  setTxBodySeller(txBodyBuyer);

}

const sellerSignSubmit = async () => {

  console.log("Verifying seller signature...");
  const signatures = await walletAPI.signTx(txBodySeller);
  txBodySeller.addSignatures(signatures);

  console.log("Submitting transaction...");
  const txHash = await walletAPI.submitTx(txBodySeller);

  console.log("txHash", txHash.hex);
  setTx({ txId: txHash.hex });

}
```

## Test Drive

The first screen will require the seller address which will be converted into the seller public key hash (PKH). The buyer (destination) address will be converted into the buyer PKH as well.

The screenshot shows a web browser window with the title 'Helios Tx Builder' and a single tab. The address bar shows the URL '3000-material-baseball-zwzft4.us1.demeter.run'. The page content includes a 'Connect to your wallet' section with a radio button selected for 'Nami'. Below this is a 'Wallet Balance In Lovelace' section showing '14,000,000'. The main form area contains fields for 'Buyer Wallet Address', 'NFT Token Name' (filled with 'Mad Dog'), 'NFT Description' (filled with 'Crazy dog that loves treats'), 'NFT Image' (filled with a long alphanumeric string), and 'Seller Wallet Address' (filled with another long alphanumeric string). At the bottom of the form is a 'Mint NFT' button.

Jan 23 09:48

Helios Tx Builder

3000-material-baseball-zwzft4.us1.demeter.run

# Helios Tx Builder

Connect to your wallet

☒ Nami

**Wallet Balance In Lovelace** 14,000,000

**Buyer Wallet Address**

addr\_test1qq5vczw9q0tagsk5v8k5tr35kzd06ur3grdv7afmt3us5mdq56r869vgx0wnpw5cerwfgs5glfml9069dys57z0lqvq7vhg

**NFT Token Name**

Mad Dog

**NFT Description**

Crazy dog that loves treats

**NFT Image**

QmckPQPzthJjkS9GRSxdw2SzSinAwdDW34xXDZd26MrV5S

**Seller Wallet Address**

addr\_test1qrrkvmepf8lzh6mj4yn237f63tm4tzu7gchahvImqnn2jkevly9p983fmyphftw2g6nlplq60pv4207lwwhqf97tj68s58rv7

Mint NFT



When the Mint NFT button is selected, the Buyer Sign button will show. As the transaction is being built, it will be stored as a state variable inside the Next.js application.

Jan 23 09:49

Helios Tx Builder x +

← → ↻ 3000-material-baseball-zwzft4.us1.demeter.run

## Helios Tx Builder

Connect to your wallet

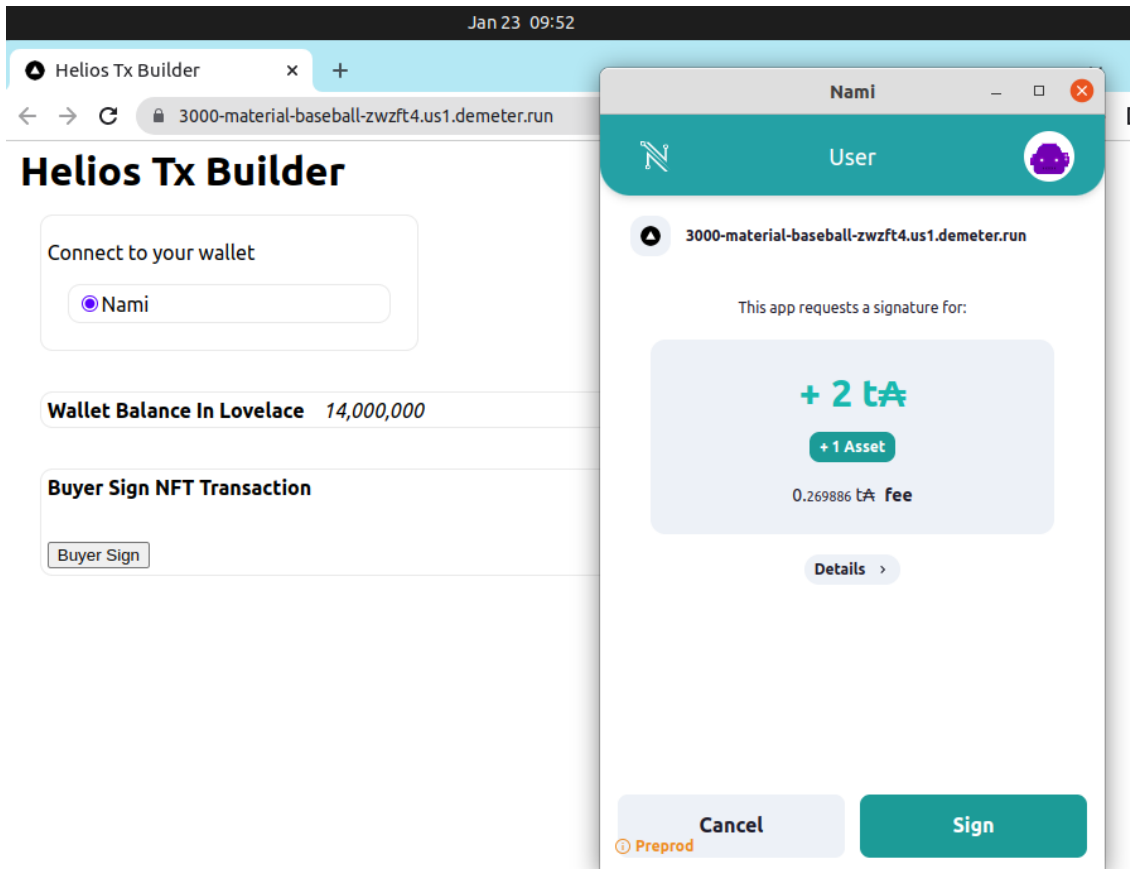
☒ Nami

Wallet Balance In Lovelace 14,000,000

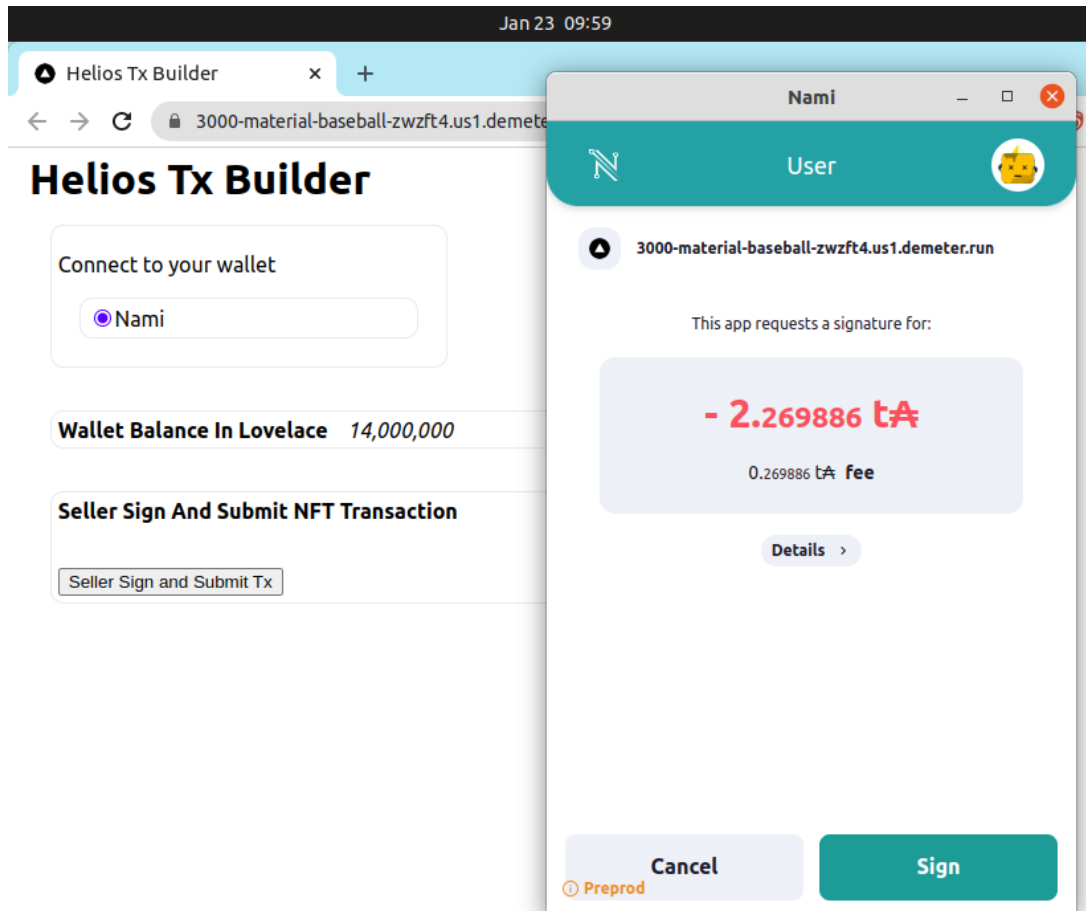
Buyer Sign NFT Transaction

Buyer Sign

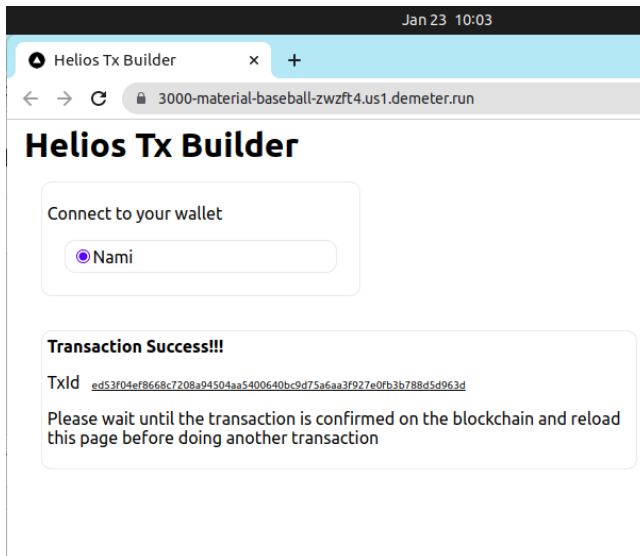
Now, we will switch the account in Nami wallet to the buyer account which is the same account that we used to provide the buyer address. Once that is done you will see the following request to sign the transaction.



If you selected an account that is not the correct buyer account, you will not be able to sign the transaction because the PKH will not match what is in the minting contract. Once you sign with the correct buyer wallet, you will now see the button for the seller to sign. Switch to the seller wallet account and select the sign and submit tx button.



Once you sign the transaction as the seller, you should now see that the transaction was submitted and the NFT has been minted.



The transaction is confirmed in cexplorer.io.

Google Chrome Jan 23 10:06

Helios Tx Builder Transaction ed53f04ef8668c7208a94504aa5400640bc9d75a6aa3f927e0fb3b788d5d963d

preprod.cexplorer.io/tx/ed53f04ef8668c7208a94504aa5400640bc9d75a6aa3f927e0fb3b788d5d963d

Cexplorer.io preprod

Transaction Hash: ed53f04ef8668c7208a94504aa5400640bc9d75a6aa3f927e0fb3b788d5d963d

Date	2m38s ago
Epoch	47
Block	551076
Slot	18,802,989 (epoch slot 140,589)
Total Output	9.73 ₳ (\$ 3.0) (0.000016)
Fee	0.27 ₳ (\$ 0)
Outputs	2

9/8 Assurance

1.91kB (0.83kB) Transaction Size

1.08kB Script Size

2.34kB (2) Block Size (TX Count)

Minted by pool1z063u\_jwOe7

Content Contracts (1) Collateral (1) Mint (1) Metadata (1) Script (1)

2m38s ago # ed53f04ef8...d963d Total 9.73 ₳

10 ₳ addr\_test1...8rvz7

773 ₳ addr\_test1...8rvz7

2 ₳ addr\_test1...vhgn8

Mad Dog 1

# Validators

A validator is used to confirm if a transaction is able to spend UTXOs locked at the validator script address. A Redeemer, Datum and UTXOs are all required as part of a transaction when executing a validator smart contract script.

## Vesting

A vesting smart contract locks up assets for a specific period of time and then allows the beneficiary to access them once a deadline has passed. In this example from the Helios documentation, there is also the ability to cancel the vested tokens if the deadline has not already passed.

We will introduce the Datum and Redeemer which are key components of a smart contract transaction. The Datum is used to store persistent data on-chain. The Redeemer is used to tell the smart contract what type of transaction is occurring. In this case, the transaction is either a Cancel or Claim and processes the validation logic accordingly.

The Datum is a struct and looks like the following where Time is in POSIX second format.

```
struct Datum {  
    creator: PubKeyHash  
    beneficiary: PubKeyHash  
    deadline: Time  
}
```

The Redeemer is an enum datatype and is as follows.

```
enum Redeemer {  
    Cancel  
    Claim  
}
```

The complete validator Helios code is as follows. The Datum is passed into the main function along with the Redeemer and Script Context. We obtain the current time by pulling out the time range of this transaction using `tx.time_range.start`; We will see in the off-chain code later in this section on how to set the time range for this transaction.

spending vesting

```
struct Datum {
    creator: PubKeyHash
    beneficiary: PubKeyHash
    deadline: Time
}

enum Redeemer {
    Cancel
    Claim
}

func main(datum: Datum, redeemer: Redeemer, context: ScriptContext) -> Bool {
    tx: Tx = context.tx;
    now: Time = tx.time_range.start;
    redeemer.switch {
        Cancel => {
            // Check if deadline hasn't passed
            (now < datum.deadline).trace("VS1: ") &&

            // Check that the owner signed the transaction
            tx.is_signed_by(datum.creator).trace("VS2: ")
        },
        Claim => {
            // Check if the deadline has passed.
            (now > datum.deadline).trace("VS3: ") &&

            // Check that the beneficiary signed the transaction.
            tx.is_signed_by(datum.beneficiary).trace("VS4: ")
        }
    }
}
```

When dealing with more complex smart contracts, it is easier to work on it directly as a separate file where you can compile it beforehand. Additionally, there is a VS Code extension for Helios that you can use as well.

I would also recommend using deno to compile the smart contract to get the generated plutus code.

To setup deno, do the following using the VS Code Web terminal window.

```
$ cd ~/workspace/repo
```

Install deno testing a simple welcome typescript program.

```
$ npx deno-bin run https://deno.land/std/examples/welcome.ts
```

Next, go to the vesting directory and execute the following deno command to compile the Helios code.

```
$ cd vesting
$ npx deno-bin run --allow-read --allow-write ./src/deploy-vesting.js
```

The compiler will give you errors if there are any syntax or type related errors. So you can fix those now and not later when you are running your code in the Next.js application. You can also see the resulting files located in the deploy directory

```
$ ls -l deploy/
total 12
-rw-rw-r-- 1 lawrence lawrence 63 Jan 19 10:58 vesting.addr
-rw-rw-r-- 1 lawrence lawrence 56 Jan 19 10:58 vesting.hash
-rw-rw-r-- 1 lawrence lawrence 788 Jan 19 10:58 vesting.plutus
```



Although we will not need these generated files for this example, there may be times when you need the generated plutus script, validator hash and/or validator address beforehand.

Once we confirm that the Helios source code compiles without errors, we will copy them over to the contracts directory.

```
$ cp src/vesting.hl contracts/
```

## Next.js Setup

Now we need to set some environment variables that will be needed for this example. It is always best practice to control environment specific variables outside of the application code.

1. Using VS Code Web, select File -> Open
2. In the popup window enter the following path and filename /config/.bashrc
3. Authorize VS Code Web access if requested
4. Add the following lines to the bottom of the .bashrc file

```
export NEXT_PUBLIC_BLOCKFROST_API_KEY="get-your-blockfrost-api-key"
export NEXT_PUBLIC_BLOCKFROST_API="https://cardano-preprod.blockfrost.io/api/v0"
export NEXT_PUBLIC_NETWORK_PARAMS_URL="https://d1t0d7c2nekuk0.cloudfront.net/preprod.json"
```
5. Now, in the Terminal window type the following command to load these environment variables into your shell. These environment variables will automatically load next time you log in as well.
6. `$ source ~/.bashrc`

Note: For step 4, you will need to set up and get your own [Blockfrost](#) account and api key.

## Next.js Startup

Now we are ready to initialize and start up the Next.js application

```
$ npm install
```

```
$ npm run dev
```

Make sure you have exposed port 3000 on the Demeter Run workspace Exposed Port tab. Select the export port URL link to launch the application. Once you select your wallet, you should see.

← → ↻ 🔒 3000-troubled-sympathy-5oh0q3.us1.demeter.run

## Helios Tx Builder

Connect to your wallet

☒ Nami

View Smart Contract:

[vesting.ht](#)


Wallet Balance In Lovelace

511,331,294

**Beneficiary Wallet Address**

**Amount Of Ada To Lock**

**Vesting Expiry Date**



**Claim Funds**

**Cancel Vesting**

## Smart Contract Viewer

We can verify the smart contract within the application by selecting the View Smart Contract link. This links to an API that reads the same file on the filesystem that is used during Helios Tx transaction builder construction.

```
← → ↻ 🔒 3000-troubled-sympathy-5oh0q3.us1.demeter.run/api/vesting

spending vesting

struct Datum {
  creator: PubKeyHash
  beneficiary: PubKeyHash
  deadline: Time
}

enum Redeemer {
  Cancel
  Claim
}

func main(datum: Datum, redeemer: Redeemer, context: ScriptContext) -> Bool {
  tx: Tx = context.tx;
  now: Time = tx.time_range.start;

  redeemer.switch {
    Cancel => {
      // Check if deadline hasn't passed
      (now < datum.deadline).trace("VS1: ") &&

      // Check that the owner signed the transaction
      tx.is_signed_by(datum.creator).trace("VS2: ")
    },
    Claim => {
      // Check if deadline has passed.
      (now > datum.deadline).trace("VS3: ") &&

      // Check that the beneficiary signed the transaction.
      tx.is_signed_by(datum.beneficiary).trace("VS4: ")
    }
  }
}
```

## Lock Ada

Now, let's look at the off-chain Helios TX builder code. We will focus on some of the key new areas of code that have been added for this example. The lockAda function creates a datum and populates it with form values and the PKH of the owner wallet.

```
const lockAda = async (params : any) => {
  const benAddr = params[0] as string;
  const adaQty = params[1] as number;
  const dueDate = params[2] as string;
  const deadline = new Date(dueDate + "T00:00");
  const benPkh = Address.fromBech32(benAddr).pubKeyHash;
  const adaAmountVal = new Value(BigInt((adaQty)*1000000));

  ...
  // Construct the datum
  const datum = new ListData([new ByteArrayData(ownerPkh.bytes),
                                new ByteArrayData(benPkh.bytes),
                                new IntData(BigInt(deadline.getTime()))]);

  const inlineDatum = Datum.inline(datum);

  ...
}
```

The last change is to create a Tx Output to include both the Ada and the vesting key token.

```
...
// Construct the NFT that we will want to send as an output
const nftTokenName = ByteArrayData.fromString("Vesting Key").toHex();
const tokens: [number[], bigint][] = [[hexToBytes(nftTokenName), BigInt(1)]];

...
const lockedVal = new Value(adaAmountVal.lovelace,
  new Assets([[mintProgram.mintingPolicyHash, tokens]]));

// Add the destination address and the amount of Ada to lock & datum
tx.addOutput(new TxOutput(valAddr, lockedVal, inlineDatum));

...
```

## Vesting Key Minting Policy

We also include an inline minting policy to create the vesting key token. This leverages the same technique that we used to mint a unique NFT and will be used to help locate the locked UTXO.

```
...
const mintScript = `minting nft

const TX_ID: ByteArray = #` + utxos[0][0].txId.hex + `
const txId: TxId = TxId::new(TX_ID)
const outputId: TxOutputId =
    TxOutputId::new(txId, ` + utxos[0][0].utxoIdx + `)

func main(ctx: ScriptContext) -> Bool {
    tx: Tx = ctx.tx;
    mph: MintingPolicyHash = ctx.get_current_minting_policy_hash();

    assetclass: AssetClass = AssetClass::new(
        mph,
        "Vesting Key".encode_utf8()
    );
    value_minted: Value = tx.minted;

    // Validator logic starts
    (value_minted == Value::new(assetclass, 1)).trace("NFT1: ") &&
    tx.inputs.any((input: TxInput) -> Bool {
        (input.output_id == outputId).trace("NFT2: ")
    })
} `
...

```

## Claim Funds or Cancel Vesting

To claim or cancel from the vesting contract, the user must supply the vesting key token. We use this key to find the correct UTXO locked at the smart contract script address. The `getKeyUtxo` function does the lookup and returns a Helio UTXO Object.

```
// Get the utxo with the vesting key token at the script address
const getKeyUtxo = async (scriptAddress : string,
                           keyMPH : string, keyName : string ) => {

const blockfrostUrl : string = blockfrostAPI + "/addresses/" +
                               scriptAddress + "/utxos/" + keyMPH + keyName;

let resp = await fetch(blockfrostUrl, {
  method: "GET",
  headers: {
    accept: "application/json",
    project_id: apiKey,
  },
});

if (resp?.status > 299) {
  throw console.error("vesting key token not found", resp);
}

const payload = await resp.json();

if (payload.length == 0) {
  throw console.error("vesting key token not found");
}

const lovelaceAmount = payload[0].amount[0].quantity;
const mph = MintingPolicyHash.fromHex(keyMPH);
const tokenName = hexToBytes(keyName);

const value = new Value(BigInt(lovelaceAmount),
                        new Assets([[mph,
                                     [[tokenName, BigInt(1)]]]]
                        ));
};
```

```
return new UTxO(  
    TxId.fromHex(payload[0].tx_hash),  
    BigInt(payload[0].output_index),  
    new TxOutput(  
        Address.fromBech32(scriptAddress),  
        value,  
        Datum.inline(ListData.fromCbor(hexToBytes(payload[0].inline_datum)))  
    )  
);  
}
```



Both claimFunds or cancelVesting functions are very similar. The code is almost identical to create a redeemer, get the UTXOs locked at the script address and confirm if the deadline has been reached.

```
...
// Create the Claim redeemer to spend the UTXO locked
// at the script address
const valRedeemer = new ConstrData(1, []);

// Get the UTXO that has the vesting key token in it
const valUtxo = await getKeyUtxo(valAddr.toBech32(), keyMPH,
    ByteArrayData.fromString("Vesting Key").toHex());

// Must include a redeemer to spend a UTXO at a script address
tx.addInput(valUtxo, valRedeemer);

// Send the value of the of the valUTXO to the recipient
tx.addOutput(new TxOutput(claimAddress, valUtxo.value));

// Specify when this transaction is valid from. This is needed so
// time is included in the transaction which will be use by the
// script. Add two hours for time to live and offset the current time
// by 5 mins.
const currentTime = new Date().getTime();
const earlierTime = new Date(currentTime - 5 * 60 * 1000);
const laterTime = new Date(currentTime + 2 * 60 * 60 * 1000);

tx.validFrom(earlierTime);
tx.validTo(laterTime);
...
```

## Test Drive

When the owner submits the transaction to lock up their funds at the smart contract, they are shown the vesting key which is used to unlock the funds afterwards.

[←](#) [→](#) [↻](#) [🔒 3000-troubled-sympathy-5oh0q3.us1.demeter.run](#)

---

## Helios Tx Builder

Connect to your wallet

☒ Nami

View Smart Contract: [vesting.hl](#)

**Transaction Success!!!**

TxId [36ae88746977376c8f12d34a96a195d7d0d06929b4533e3eaa134b8259bf89c5](#)

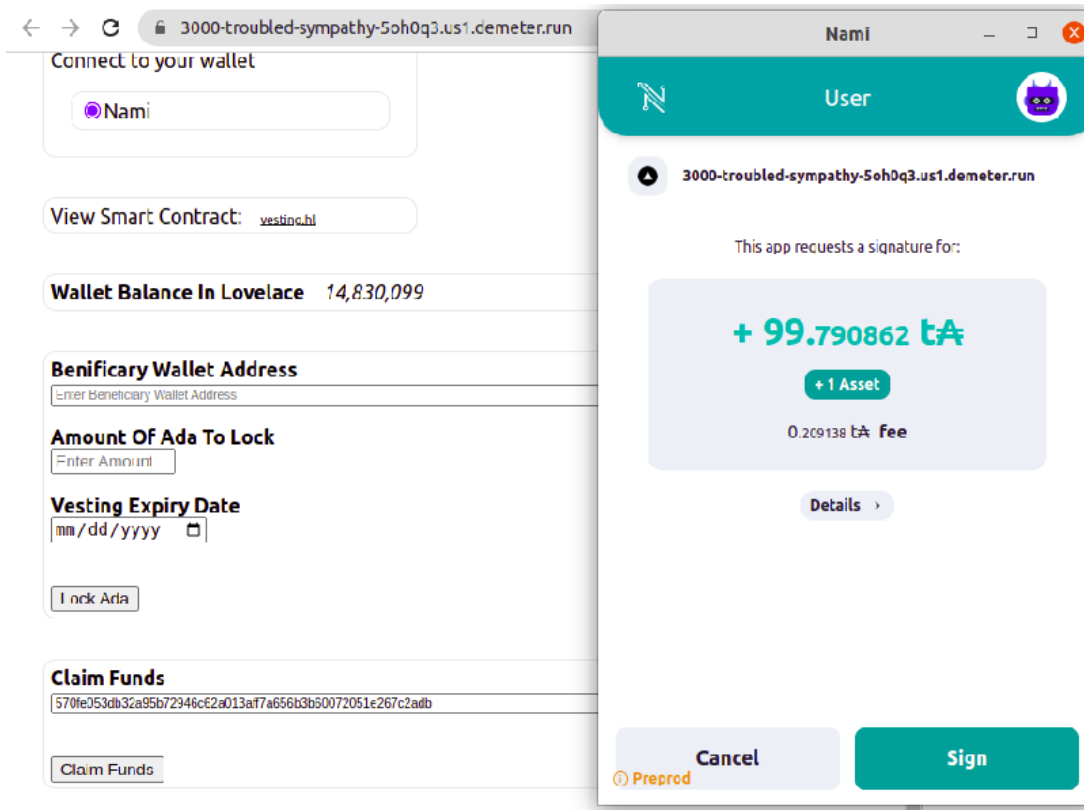
Please wait until the transaction is confirmed on the blockchain and reload this page before doing another transaction

Please copy and save your vesting key

**570fe053db32a95b72946c62a013aff7a656b3b60072051e267c2adb**

You will need this key to unlock your funds

After entering the vesting key and selecting Claim Funds, the beneficiary needs to then sign and submit the transaction. As can be seen in this example, the beneficiary will be receiving 99.790862 tAda



On preprod explorer, both the Ada locked and the vesting key token has been transferred to the beneficiary. The smart contract allowed this transaction to succeed because the deadline was set in the past and the beneficiary wallet used had the correct PKH.

→ [preprod.cexplorer.io/tx/f527d6db27bbcdb9ac8bcf4d4b4f37f5764d20ed278355e2fb8af9ac8c193dd9](https://preprod.cexplorer.io/tx/f527d6db27bbcdb9ac8bcf4d4b4f37f5764d20ed278355e2fb8af9ac8c193dd9)

**Cexplorer.io** preprod

CARDANO EXPLORER

- Dashboard
- Watchlist
- Pools
- Assets

Content Contracts (1) Collateral (1)

1m54s ago # f527d6db27...93dd9 Total 109.62A

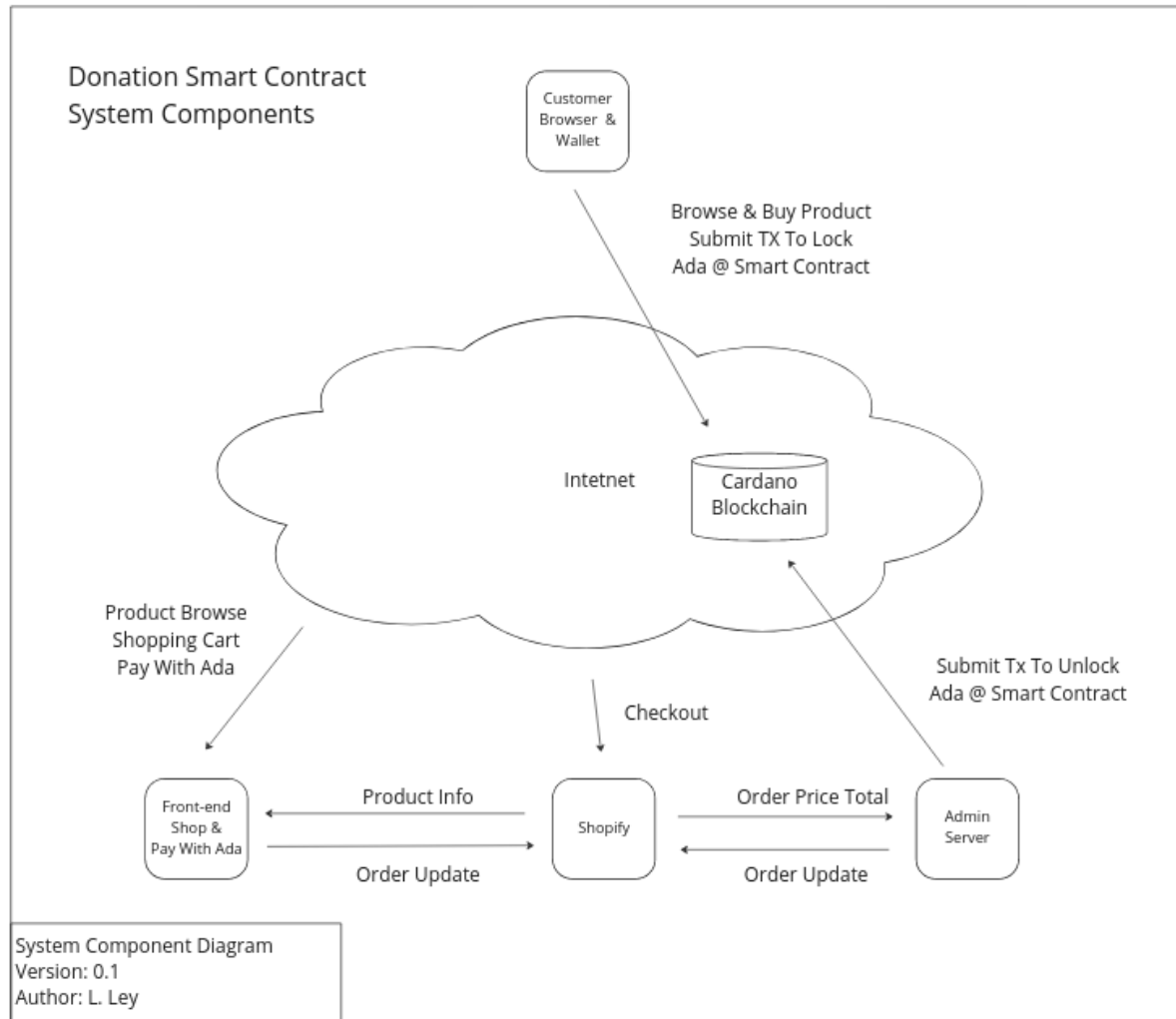
100A	addr_test1...qekxy	100A	addr_test1...zcsuy
Vesting Key -1		Vesting Key 1	
9.83A	addr_test1...zcsuy	9.62A	addr_test1...zcsuy

## Donation Traceability (Lock Ada)

In this final example we will review a donation traceability smart contract. It provides transparency and auditability on the percentage of an eCommerce order that is being donated.

## System Components

The following diagrams show the systems involved interacting with the smart contract.



The main systems involved in this architecture are:

- Customer Browser
- Next.js Front-end Application
- Shopify
- Admin Server for batch processing
- Cardano Blockchain

When users purchase a product with Ada, the order amount will be locked into a smart contract. The smart contract has very specific rules on the percentage allocation and what wallet addresses the Ada can go to. This allows for traceability of the donation so everyone can see distribution of Ada to the merchant and charity accordingly.

## Smart Contract Code

Here is the code for the smart contract for donation verification. Notice that the wallet addresses and the donation split allocation is hard coded as contract parameters. This means that they cannot change once the smart contract is compiled and loaded on-chain. We also want to lock down the script so only the admin can run it for added security, but this is an optional design choice.

spending vesting

```
struct Datum {
    orderAmount: Int
    orderId: ByteArray
    adaUsdPrce: ByteArray
}

enum Redeemer {
    Spend
    Refund
}

// Define the pkh of the merchant
const MERCHANT_PKH: ByteArray = #3d6...38c
const merchantPkh: PubKeyHash = PubKeyHash::new(MERCHANT_PKH)

// Define the pkh of the Donation
const DONOR_PKH: ByteArray = #b2b...7a9f
const donorPkh: PubKeyHash = PubKeyHash::new(DONOR_PKH)

// Define the pkh of the Refund
const REFUND_PKH: ByteArray = #a0a...9a9
const refundPkh: PubKeyHash = PubKeyHash::new(REFUND_PKH)

// Define the pkh of the Admin
const ADMIN_PKH: ByteArray = #b9a...7682
const adminPkh: PubKeyHash = PubKeyHash::new(ADMIN_PKH)

const SPLIT: Int = 90 // Define the split merchant to donor
const minAda = 1000000 // Define minimum Ada for a donation
const version = 2 // Increment version number for the contract if needed
```



```

func getDonationAmt (orderAmt: Int) -> Int {
    donationAmt: Int = (orderAmt * (100 - SPLIT)) / 100;
    if (donationAmt < minAda) {
        minAda
    } else {
        donationAmt
    }
}

func main(datum: Datum, redeemer: Redeemer, context: ScriptContext) -> Bool {
    tx: Tx = context.tx;
    redeemer.switch {
        Spend => {
            orderAmt: Int = datum.orderAmount;
            donationAmt: Int = getDonationAmt(orderAmt);
            merchantAmt: Int = orderAmt - donationAmt;
            donationAmtVal: Value = Value::lovelace(donationAmt);
            merchantAmtVal: Value = Value::lovelace(merchantAmt);
            merchOutTxS : []TxOutput = tx.outputs_sent_to(merchantPkh);
            donorOutTxS : []TxOutput = tx.outputs_sent_to(donorPkh);
            tx.is_signed_by(adminPkh).trace("DN1: ") &&
            (merchOutTxS.head.value == merchantAmtVal).trace("DN2: ") &&
            (donorOutTxS.head.value == donationAmtVal).trace("DN3: ")
        },
        Refund => {
            orderAmt: Int = datum.orderAmount;
            returnAmtVal: Value = Value::lovelace(orderAmt);
            refundOutTxS : []TxOutput = tx.outputs_sent_to(refundPkh);
            tx.is_signed_by(adminPkh).trace("DN4: ") &&
            (refundOutTxS.head.value == returnAmtVal).trace("DN5: ")
        }
    }
}

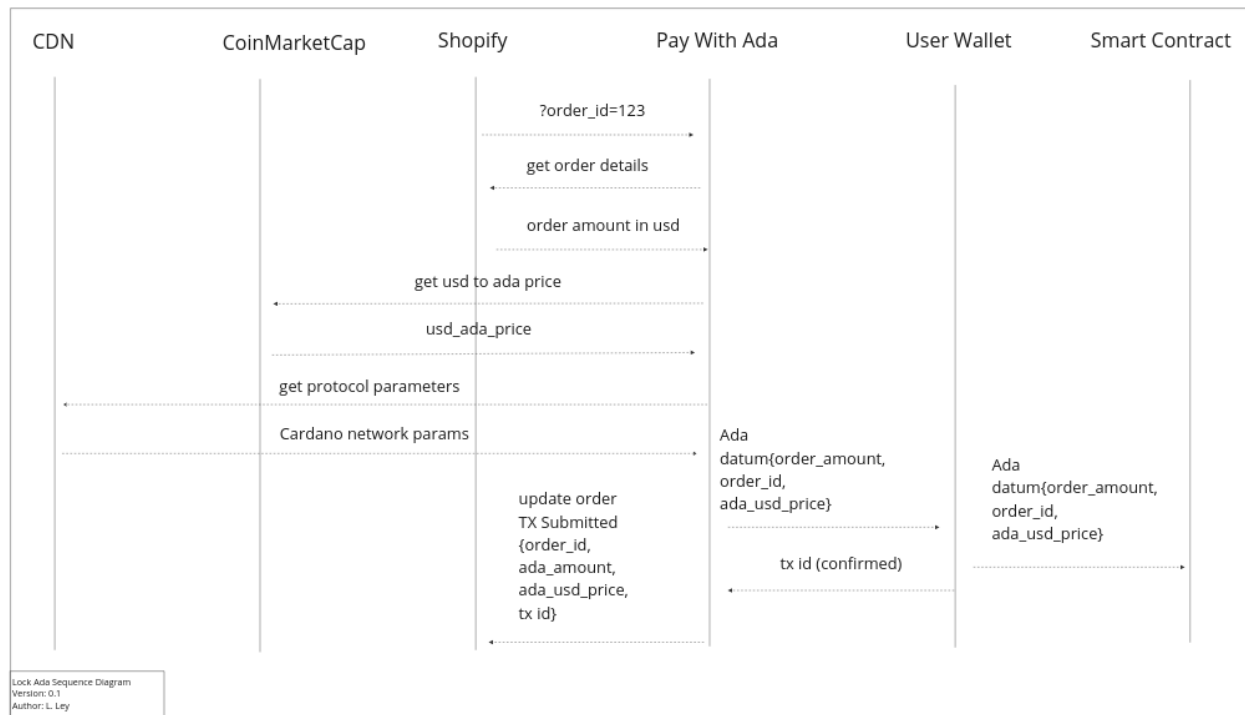
```

We will use hard coded contract parameters for the merchant, donor, refund and admin PKH. This will allow us to validate that the correct percentage of the order amount is actually getting locked at the merchant, donor or refund wallets. We also want to lock down the script so only the admin can run it for added security, but this is an optional design choice.

## Pay With Ada

The following sequence diagram shows the logic for the pay with Ada component. This is needed so a customer can pay with Ada when completing their eCommerce order.

### Locking Ada At The Smart Contract



The Pay With Ada component will get an order id from Shopify via a query string parameter in the browser URL. The Next.js app will then do a Shopify Admin API lookup to get the amount for the order. Once it has the amount of the order, it will do another lookup to get the ADA/USD conversion so we know how much Ada is required. Finally, when the user selects the Pay With Ada Button, a spend transaction is constructed with a datum for the user to sign and submit.

## Compile & Deploy

Open the Terminal window in VS Code Web.

```
$ cd donation
```

Update the `src/donation.h1` smart contract with any changes in the parameters such as the merchant, donor, refund and admin PKH. After the changes are done, compile the Helios code in the donation directory.

If deno is not installed, run the following

```
$ npx deno-bin run https://deno.land/std/examples/welcome.ts
```

Now run the following command to compile the Helios smart contract to a plutus script.

```
$ npx deno-bin run --allow-read --allow-write ./src/deploy-donation.js
```

Then copy the generated file(s) to the scripts data directory and Next.js contracts directory respectively.

```
$ cp deploy/* scripts/cardano-cli/preprod/data  
$ cp src/donation.h1 contracts
```

Next we need to add/update the following environment variables by editing the .bashrc file. You can access this via VS Code Web -> Open File and paste /config/.bashrc in the popup window.

Note, you may need to go to [Shopify](#) and [CoinMarketCap](#) and create accounts so you can get the required API keys.

```
export NEXT_PUBLIC_NETWORK="preprod"
export NEXT_PUBLIC_BLOCKFROST_API_KEY="blockfrost api key"
export NEXT_PUBLIC_BLOCKFROST_API="https://cardano-preprod.blockfrost.io/api/v0"
export NEXT_PUBLIC_NETWORK_PARAMS_URL="https://d1t0d7c2nekuk0.cloudfront.net/preprod.json"
export NEXT_PUBLIC_MIN_ADA=2000000
export NEXT_PUBLIC_SHOP="https://shopify-store-url"
export NEXT_PUBLIC_ACCESS_TOKEN="shopify access token"
export NEXT_PUBLIC_COIN_API_KEY="coin market cap api key"
export NEXT_PUBLIC_SERVICE_FEE=500000
export NEXT_PUBLIC_ORDER_API_KEY="create a unique base64 hash key"
```

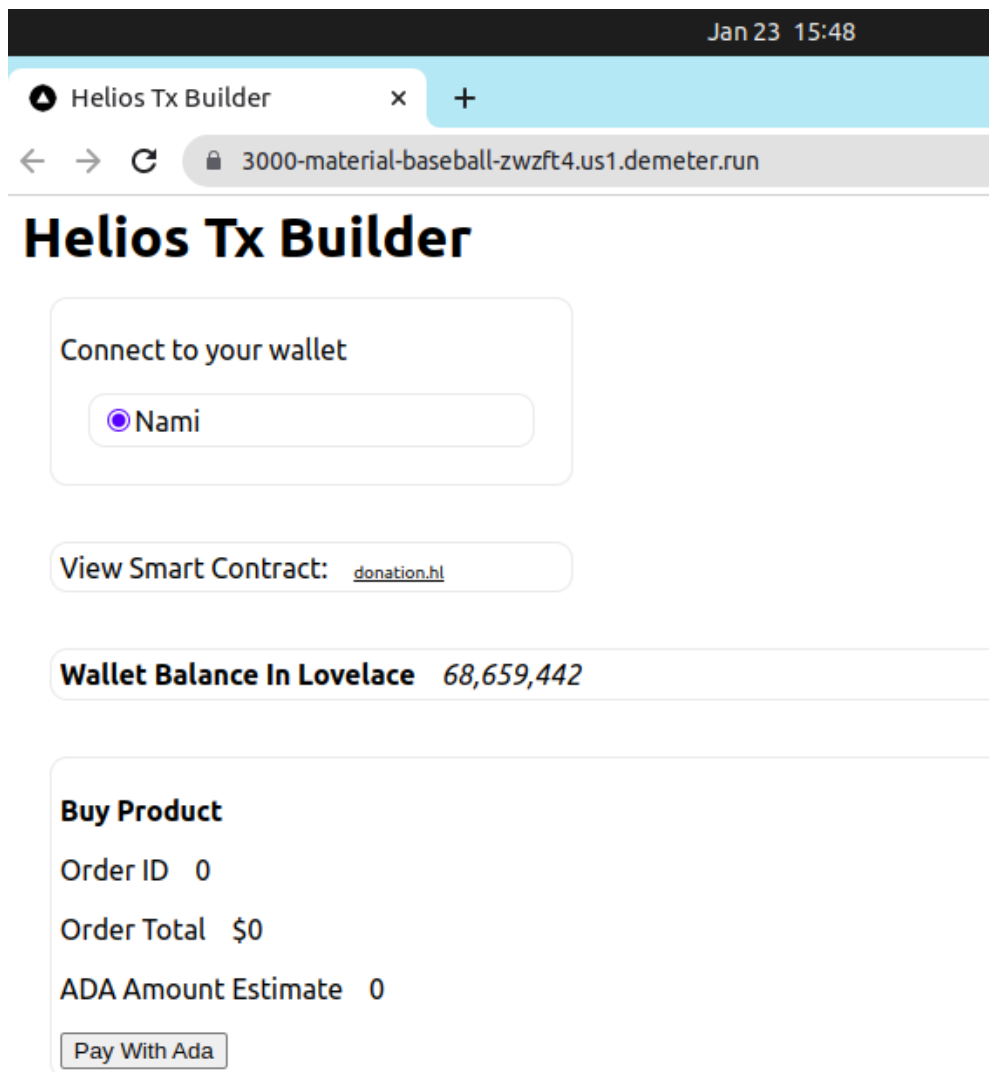
Load in the environment variables to your current shell. These will get loaded automatically every time you log in if they exist in your .bashrc file

## Next.js Setup

Now install the npm modules and start Next.js. Remember to expose port 3000 in the Demeter Run workspace Exposed Ports tab.

```
$ source ~/.bashrc
$ npm install
$ npm run dev
```

You should see the following which is correct when no order id is passed via the query string. Copy the Next.js app web URL which you will need in the next step.



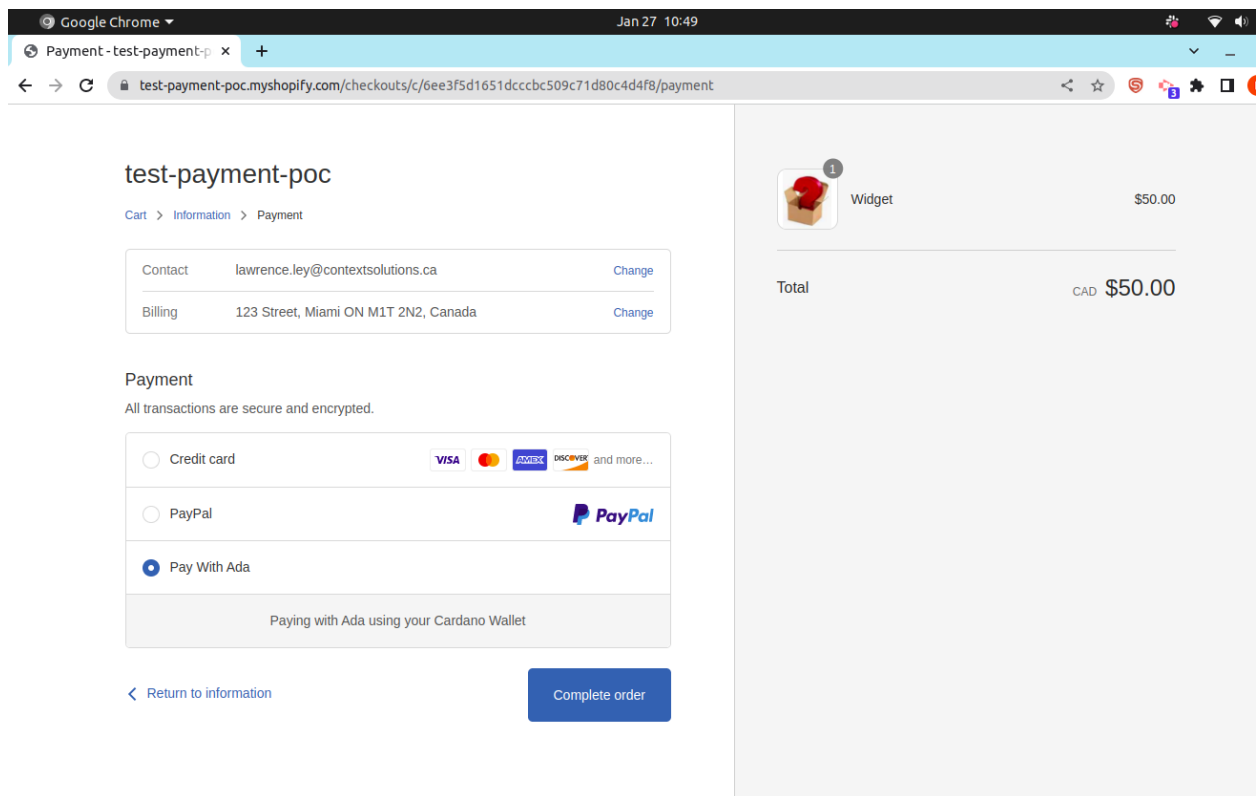
## Shopify Settings

We will need to update the URL that is used to pay for Ada. Please see Shopify Ada Payments in the Appendix if you need help configuring a Shopify store for Ada payments.

1. Go to your shopify store admin section
2. Select Settings
3. Select Checkout
4. Scroll to the bottom and update the variable in the Additional Scripts dialog box with the URL you copied in the last step. For example:  
`var url1Str = "https://3000-material-baseball-zwzft4.us1.demeter.run/";`
5. Select Save and close the Settings Window

## Test Drive

Now go to your shopify store and proceed to checkout with a product and Select the Pay With Ada as a payment option.



Select Complete order and the user will be taken to the order status page.

Google Chrome Jan 27 10:50

Thank you for your purchase x

test-payment-poc.myshopify.com/checkouts/c/51399cc5f7a9dba3ce62879b4e741b2c/thank\_you

### test-payment-poc

5247401754903  
Thank you!

☐ Email me with news and offers

**Your order is confirmed**  
Please select to the Pay Now With Ada link below to pay using your Cardano Wallet

**Pay To Complete Your Order**


[Pay Now In Ada](#)

**Customer information**

<b>Contact information</b> lawrence.ley@contextolutions.ca	<b>Payment method</b> Pay With Ada - \$50.00
<b>Billing address</b> Test 123 Street Miami ON M1T 2N2 Canada	

Need help? [Contact us](#)

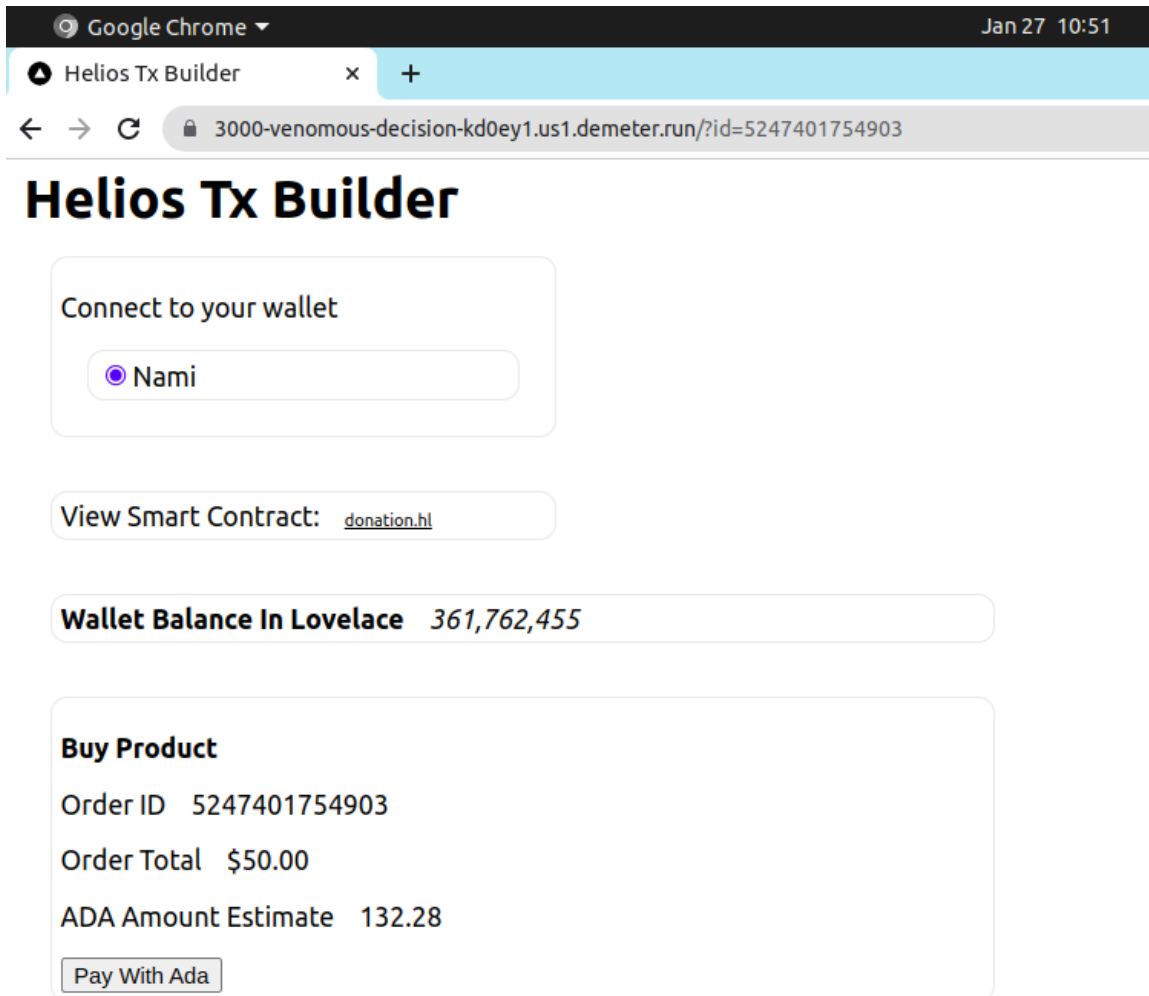
[Continue shopping](#)

 **Widget** \$50.00

---

**Total** CAD **\$50.00**

Select the Pay Now in Ada link and you will be directed to the Next.js web application. The URL will contain a query string which allows the application to retrieve the order dollar amount and then a ADA/USD conversion for the amount of Ada due.



The screenshot shows a Google Chrome browser window with the title 'Helios Tx Builder'. The address bar displays the URL '3000-venomous-decision-kd0ey1.us1.demeter.run/?id=5247401754903'. The main heading is 'Helios Tx Builder'. Below it, there is a section 'Connect to your wallet' with a radio button selected for 'Nami'. A 'View Smart Contract:' link points to 'donation.hl'. A 'Wallet Balance In Lovelace' is shown as '361,762,455'. The 'Buy Product' section lists 'Order ID 5247401754903', 'Order Total \$50.00', and 'ADA Amount Estimate 132.28', with a 'Pay With Ada' button at the bottom.

Google Chrome Jan 27 10:51

Helios Tx Builder

3000-venomous-decision-kd0ey1.us1.demeter.run/?id=5247401754903

## Helios Tx Builder

Connect to your wallet

☒ Nami

View Smart Contract: [donation.hl](#)

**Wallet Balance In Lovelace** 361,762,455

**Buy Product**

Order ID 5247401754903

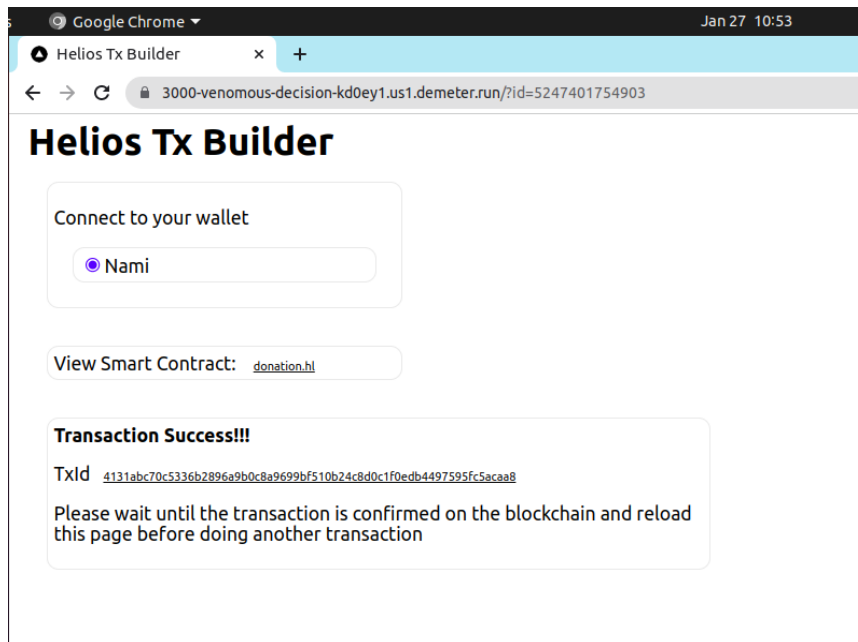
Order Total \$50.00

ADA Amount Estimate 132.28

Pay With Ada



Select the Pay With Ada button and then sign and submit the transaction using your wallet. You have now paid for the order and locked the Ada into the donation smart contract address. You may also notice that no collateral was needed. We are simply locking the Ada (with a datum) at an address and not executing a smart contract yet. This design choice helps reduce friction during the shopping experience for users that have never used a Cardano dapp before.



The transaction of locking the Ada and a datum can be seen on cexplorer.io..

Google Chrome Jan 27 10:54

Helios Tx Builder Transaction 4131abc70c5

preprod.cexplorer.io/tx/4131abc70c5336b2896a9b0c8a9699bf510b24c8d0c1f0edb4497595fc5acaa8

Cexplorer.io

Transaction detail

Hash: 4131abc70c5336b2896a9b0c8a9699bf510b24c8d0c1f0edb4497595fc5acaa8

Date	41s ago
Epoch	48
Block	564,494
Slot	19,151,618 (epoch slot 57,218)
Total Output	361.59 ₳ <span>\$ 1977</span> <span>0.00598</span>
Fee	0.17 ₳ <span>\$ 01</span>
Outputs	2

1/8 Assurance

0.29kB (0.29kB)  
Transaction Size  
this tx 99%

0.29kB (1)  
Block Size (TX Count)  
available 100%

Minted by  
pooltz063u\_jw0e7

Content

41s ago # 4131abc70c5acaa8 Total 361.59 ₳ \$ 1977 0.00598

361.76 ₳ \$ 1977 0.00598 addr\_test1\_pmr54

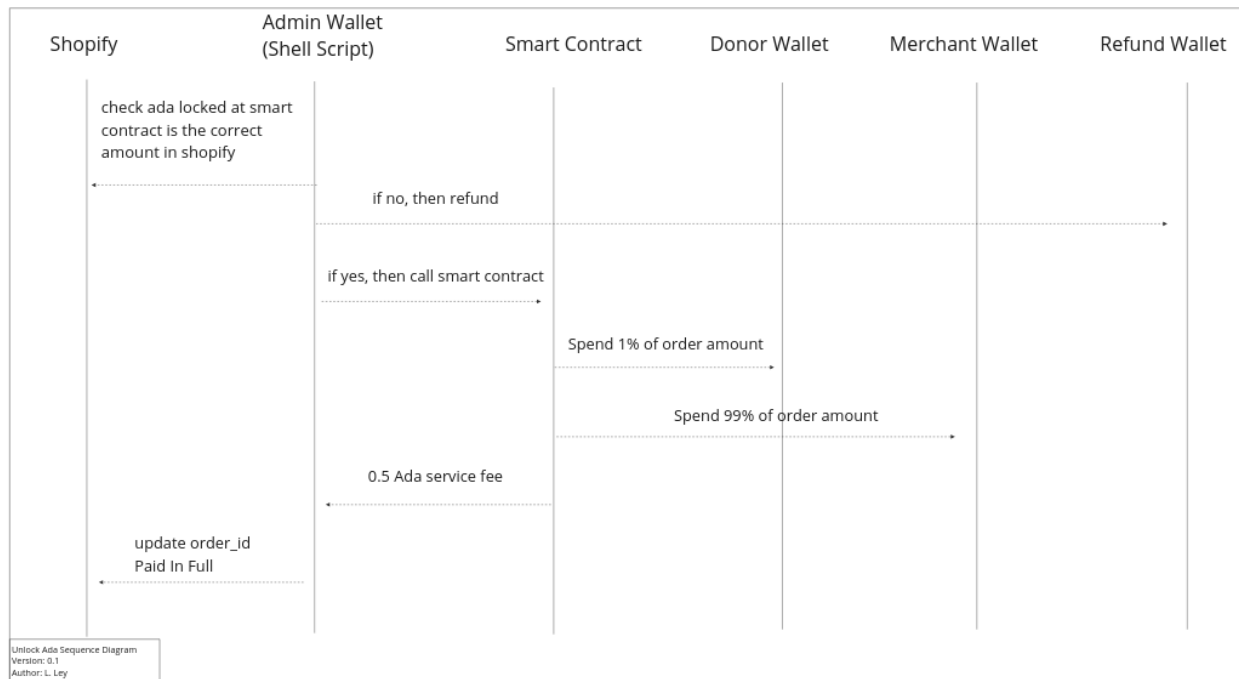
229.31 ₳ \$ 1224 0.00219 addr\_test1\_pmr54

132.28 ₳ \$ 653 0.00379 addr\_test1\_ckk0k

## Donation Traceability (Unlock Ada)

In the previous section, we locked Ada to the donation traceability smart contract while purchasing an eCommerce order. This section will walk through how we unlock that Ada and send it to the Merchant and Charity accordingly.

## Unlocking Ada At The Smart Contract



The unlocking of Ada involves executing a bash shell script by an administrator. If we recall, spending of Ada locked at the donation smart contract will only succeed if:

1. The transaction is signed by the admin PKH
2. The output goes to the merchant and donor addresses accordingly
3. The amount of Ada sent to the merchant and donor matches the donation split defined in the smart contract.

There are some additional checks during processing to confirm that the order amount in the datum matches the order amount in Shopify. This is needed because anyone can lock Ada with a datum value at the smart contract script address. We need additional off-chain logic that can confirm the locked UTXO is valid. Finally, if the transaction is successful, we update Shopify that the order has been paid in full.

## Bash Shell Scripts

Using the Terminal window in VS Code Web, cd donation/scripts directory to see the bash shell scripts and the files needed to run them.

Bash shell scripts

```
./cardano-cli/init-tx.sh  
./cardano-cli/refund-tx.sh  
./cardano-cli/spend-tx.sh
```

Environment specific settings

```
./cardano-cli/preprod/global-export-variables.sh
```

Environment specific data files

```
./cardano-cli/preprod/data/donation.plutus  
./cardano-cli/preprod/data/donation.addr  
./cardano-cli/preprod/data/redeemer-spend.json  
./cardano-cli/preprod/data/black-list-utxo.txt  
./cardano-cli/preprod/data/donation.hash  
./cardano-cli/preprod/data/redeemer-refund.json  
./cardano-cli/preprod/data/donation-spend-metadata.json
```

We will need to load the smart contract onto the blockchain as a reference script by running the init-tx.sh shell script. But before we can do that we must ensure there are 2 UTXO at the admin wallet address with 5 Ada of collateral and over 25 Ada to pay for uploading the script and transaction fees.

Something like the following would work.

```
$ cardano-cli query utxo --address addr_test1vzu6...dxn7 --cardano-mode
--testnet-magic 1
TxHash TxIx Amount
```

```
-----
9f0...d82 1 79759567 lovelace + TxOutDatumNone
e06...8d6 0 5000000 lovelace + TxOutDatumNone
```

Update the scripts/cardano-cli/preprod/global-export-variables.sh for the environment you are targeting (preprod). Most of the values should work out of the box for initial setup, but the addresses and keys need to change to addresses and keys you control.

```
# Define export variables
export BASE=/config/workspace/repo/donation
export WORK=$BASE/work
export TESTNET_MAGIC=1
export ADMIN_VKEY=/config/workspace/repo/.keys/admin/key.vkey
export ADMIN_SKEY=/config/workspace/repo/.keys/admin/key.skey
export ADMIN_PKH=/config/workspace/repo/.keys/admin/key.pkh
export MIN_ADA_OUTPUT_TX=2000000
export MIN_ADA_OUTPUT_TX_REF=25000000
export COLLATERAL_ADA=5000000
export
MERCHANT_ADDR=addr_test1vq7k90717e59t52skm8e0ezsnmmc7h4xy30kg2klwc5n8rqug2pds
export
DONOR_ADDR=addr_test1vzetpfww4aaunft0ucvcrxugj8nt4lhltsktya0rx0uh48cqghjfg
export
REFUND_ADDR=addr_test1vzs24vjh8salzqt5pahgvr34ewfwagxaxr5pz7eswugzn2gmw4f5w
export
VAL_REF_SCRIPT=b8a3af2835b5f2789b1f9e1d53c953a26db9a7dfab0e204e8629d85772c9fc3
0#1
export SPLIT=90
export MIN_ADA_DONATION=1000000
```

## Reference Scripts

Reference scripts are read only UTXOs that can be included as inputs into a transaction, but cannot be spent. This way, a reference script UTXO can be reused by more than one transaction either in parallel or sequentially.

Upload the plutus script onto the block chain by running the `init-tx.sh` shell script.

```
$ cd scripts/cardano-cli
$ ./init-tx.sh preprod
```

After the transaction has been successfully submitted, you should be able to see the reference UTXO with 25 Ada locked at the script address. You will also see the UTXO with the Ada locked from the order in the previous section. It has an inline datum with an order total amount, order Id and ADA/USD price conversion.

```
$ cardano-cli query utxo --address addr_test1wr...kk0k --cardano-mode
--testnet-magic 1
```

Amount		TxHash	TxIx
413...aa8	0	132280000 lovelace + TxOutDatumInline ReferenceTxInsScriptsInlineDatumsInBabbageEra (ScriptDataList [ScriptDataNumber 131780000,ScriptDataBytes "5247401754903",ScriptDataBytes "0.37941"])	
d6c...2ce	0	25000000 lovelace + TxOutDatumNone	

Next we need to update the `preprod/global-environment-variables.sh` file with the reference script UTXO TxHash and TxIdx.

```
export VAL_REF_SCRIPT=d6c...2ce3#0
```

## Spend Script

Now we can run the spend-tx.sh shell script.

```
./spend-tx.sh preprod
```

And this is what the spend-tx.sh bash shell script looks like.

```
#!/usr/bin/env bash
set -e
set -o pipefail

# enabled debug flag for bash shell
set -x

# check if command line argument is empty or not present
if [ -z $1 ];
then
    echo "process-tx.sh: Invalid script arguments"
    echo "Usage: process-tx.sh [devnet|preview|preprod|mainnet]"
    exit 1
fi
ENV=$1

# Pull in global export variables
MY_DIR=$(dirname $(readlink -f $0))
source $MY_DIR/$ENV/global-export-variables.sh
if [ "$ENV" == "mainnet" ];
then
    network="--mainnet"
else
    network="--testnet-magic $TESTNET_MAGIC"
fi
mkdir -p $WORK
mkdir -p $WORK-backup
rm -f $WORK/*
rm -f $WORK-backup/*

# generate values from cardano-cli tool
cardano-cli query protocol-parameters $network --out-file $WORK/pparms.json

# load in local variable values
validator_script="$BASE/scripts/cardano-cli/$ENV/data/donation.plutus"
validator_script_addr=$(cardano-cli address build --payment-script-file
"$validator_script" $network)
redeemer_file_path="$BASE/scripts/cardano-cli/$ENV/data/redeemer-spend.json"
admin_pkh=$(cat $ADMIN_PKH)
```



```
#####
# Spend the donation UTXO
#####
# Step 1: Get UTXOs from admin
#####
# There needs to be at least 2 utxos that can be consumed.
# One for spending of the token and one uxto for collateral
#####

admin_utxo_addr=$(cardano-cli address build $network
--payment-verification-key-file "$ADMIN_VKEY")
cardano-cli query utxo --address "$admin_utxo_addr" --cardano-mode $network
--out-file $WORK/admin-utxo.json

cat $WORK/admin-utxo.json | jq -r 'to_entries[] | select(.value.value.lovelace
> '$COLLATERAL_ADA' ) | .key' > $WORK/admin-utxo-valid.json

readarray admin_utxo_valid_array < $WORK/admin-utxo-valid.json

admin_utxo_in=$(echo $admin_utxo_valid_array | tr -d '\n')

cat $WORK/admin-utxo.json | jq -r 'to_entries[] | select(.value.value.lovelace
== '$COLLATERAL_ADA' ) | .key' > $WORK/admin-utxo-collateral-valid.json

readarray admin_utxo_valid_array < $WORK/admin-utxo-collateral-valid.json

admin_utxo_collateral_in=$(echo $admin_utxo_valid_array | tr -d '\n')

readarray black_list_utxo_array <
$BASE/scripts/cardano-cli/$ENV/data/black-list-utxo.txt

#####
# Step 2: Get the donation smart contract utxos
#####
cardano-cli query utxo --address $validator_script_addr $network --out-file
$WORK/validator-utxo.json

cat $WORK/validator-utxo.json | jq -r 'to_entries[] |
select(.value.inlineDatum | length > 0) | .key' > $WORK/order_utxo_in.txt
readarray order_utxo_in_array < $WORK/order_utxo_in.txt

order_array_length="${#order_utxo_in_array[@]}"

order_utxo_in=""
# Find a utxo that is not in the blacklist
for (( c=0; c<$order_array_length; c++ ))
do
    if printf '%s' "${black_list_utxo_array[@]}" | grep -q -x
"${order_utxo_in_array[$c]}";
```

```

    then
        echo "UTXO on blacklist: ${order_utxo_in_array[$c]}"
    else
        order_utxo_in=$(echo ${order_utxo_in_array[$c]} | tr -d '\n')
        break
    fi
Done

# Check if there are any utxos at the validator that we can
# use, if not, then exit
if [ -z $order_utxo_in ];
then
    exit 0
Fi

# Get the correct datum UTXO
order_datum_in=$(jq -r 'to_entries[]
| select(.key == "'$order_utxo_in'")
| .value.inlineDatum ' $WORK/validator-utxo.json)

echo -n "$order_datum_in" > $WORK/datum-in.json

# Get the order details from the datum
order_ada=$(jq -r '.list[0].int' $WORK/datum-in.json)

order_id_encoded=$(jq -r '.list[1].bytes' $WORK/datum-in.json)
echo -n "$order_id_encoded" > $WORK/order_id.encoded
order_id=$(python3 hexdump.py -r $WORK/order_id.encoded)

ada_usd_price_encoded=$(jq -r '.list[2].bytes' $WORK/datum-in.json)
echo -n "$ada_usd_price_encoded" > $WORK/ada_usd_price.encoded
ada_usd_price=$(python3 hexdump.py -r $WORK/ada_usd_price.encoded)

merchant_split=$SPLIT
donor_split=$((100 - $SPLIT))
donor_ada_amount=$((($order_ada * $donor_split / 100))

if (($donor_ada_amount < $MIN_ADA_DONATION ));
Then
    donor_ada=$MIN_ADA_DONATION
else
    donor_ada=$donor_ada_amount
Fi

merchant_ada=$((($order_ada - $donor_ada))

now=$(date '+%Y/%m/%d-%H:%M:%S')

# verify that the amount paid of the order is the same

```

```

# as the order amount in shopify
shopify_order_amount=$(curl -H "X-Shopify-Access-Token:
$NEXT_PUBLIC_ACCESS_TOKEN"
"$NEXT_PUBLIC_SHOP/admin/api/2022-10/orders/"$order_id".json" | jq -r
'.order.total_price')

shopify_order_ada=$(python3 -c "print(round(($shopify_order_amount /
$ada_usd_price), 3))")

shopify_order_lovelace=$(python3 -c "print($shopify_order_ada * 1000000)")
shopify_order_ada_truncated=${shopify_order_lovelace%.*}

difference=$(( $order_ada - $shopify_order_ada_truncated))
difference_abs=$(echo ${difference#-})

if (( $difference_abs > 10000 ));
then
    echo "Order amount mismatch between order amount in datum vs order amount
    in shopify for $order_id"
    exit -1
fi

metadata="{
  \"1\" : {
    \"order_detail\" : {
      \"date\" : \"$now\",
      \"donation_ada_amount\" : \"$donor_ada\",
      \"donation_split\" : \"$donor_split%\",
      \"order_id\" : \"$order_id\",
      \"order_ada_amount\" : \"$order_ada\",
      \"ada_usd_price\" : \"$ada_usd_price\",
      \"version\" : \"0.1\"
    }
  }
}"

echo $metadata >
$BASE/scripts/cardano-cli/$ENV/data/donation-spend-metadata.json
metadata_file_path="$BASE/scripts/cardano-cli/$ENV/data/donation-spend-metadat
a.json"

#####
# Step 3: Build and submit the transaction
#####
cardano-cli transaction build \
--babbage-era \
--cardano-mode \
$network \
--change-address "$admin_utxo_addr" \

```

```

--tx-in-collateral "$admin_utxo_collateral_in" \
--tx-in "$admin_utxo_in" \
--tx-in "$order_utxo_in" \
--spending-tx-in-reference "$VAL_REF_SCRIPT" \
--spending-plutus-script-v2 \
--spending-reference-tx-in-inline-datum-present \
--spending-reference-tx-in-redeemer-file "$redeemer_file_path" \
--tx-out "$MERCHANT_ADDR+$merchant_ada" \
--tx-out "$DONOR_ADDR+$donor_ada" \
--required-signer-hash "$admin_pkh" \
--protocol-params-file "$WORK/pparms.json" \
--metadata-json-file "$metadata_file_path" \
--out-file $WORK/spend-tx-alonzo.body

echo "tx has been built"

cardano-cli transaction sign \
--tx-body-file $WORK/spend-tx-alonzo.body \
$network \
--signing-key-file "${ADMIN_SKEY}" \
--out-file $WORK/spend-tx-alonzo.tx

echo "tx has been signed"

echo "Submit the tx with plutus script and wait 5 seconds..."
cardano-cli transaction submit --tx-file $WORK/spend-tx-alonzo.tx $network

# Update shopify that the order is paid in full
curl -s -S -d '{"order":{"id":"'${order_id}',"tags":["PAID IN FULL"]}}' \
-X PUT "${NEXT_PUBLIC_SHOP}admin/api/2022-10/orders/${order_id}.json" \
-H "X-Shopify-Access-Token: $NEXT_PUBLIC_ACCESS_TOKEN" \
-H "Content-Type: application/json" > /dev/null

```

## Test Drive

### Before tx-spend.sh is executed

```
$ cardano-cli query utxo --address
addr_test1wr3m0vrrcccxygtmexw7ur7vujjz5gqg4fjn0k2sjsjn04sdckk0k --cardano-mode --testnet-magic 1
      TxHash                                TxIx      Amount
-----
4131abc70c5336b2896a9b0c8a9699bf510b24c8d0c1f0edb4497595fc5acaa8      0      132280000 lovelace
+ TxOutDatumInline ReferenceTxInsScriptsInlineDatumsInBabbageEra (ScriptDataList
[ScriptDataNumber 131780000,ScriptDataBytes "5247401754903",ScriptDataBytes "0.37941"])
d6c4a585ffbbdcc371a3472e829133643c9dfa4f1b61a4ccb8e75fd4480c22ce      0      25000000 lovelace +
TxOutDatumNone
```

### After tx-spend.sh is executed

```
$ cardano-cli query utxo --address
addr_test1wr3m0vrrcccxygtmexw7ur7vujjz5gqg4fjn0k2sjsjn04sdckk0k --cardano-mode --testnet-magic 1
      TxHash                                TxIx      Amount
-----
d6c4a585ffbbdcc371a3472e829133643c9dfa4f1b61a4ccb8e75fd4480c22ce      0      25000000 lovelace +
TxOutDatumNone
```

We can see the successful transaction on explorer.io

The screenshot shows the Cardano Explorer interface in Google Chrome. The browser tabs include 'Helios Tx Builder' and 'Transaction c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074'. The address bar shows the URL 'preprod.cexplorer.io/tx/c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074'. The page title is 'Transaction detail'. The transaction hash is 'c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074'. The transaction details table shows: Date (6m20s ago), Epoch (48), Block (504,597), Slot (19,153,936), Total Output (634.07 ADA, \$241.5, B 0.001048), Fee (0.23 ADA, \$0.0), and Outputs (3). The right sidebar shows: 22 Assurance, 0.71kB (0.71kB) Transaction Size (this tx 99%), 0.71kB (1) Block Size (TX Count) (available 99%), and Minted by [BLADE] BLADE Pool - bladepool.com. The bottom section shows the transaction content and metadata, including a table of outputs with addresses and amounts.

Google Chrome Jan 27 11:38

Helios Tx Builder Transaction c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074

preprod.cexplorer.io/tx/c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074

Cexplorer.io

Transaction detail

Hash: c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074

Date	6m20s ago
Epoch	48
Block	504,597
Slot	19,153,936 (epoch slot 59,536)
Total Output	634.07 ADA <span>\$241.5</span> <span>B 0.001048</span>
Fee	0.23 ADA <span>\$0.0</span>
Outputs	3

22 Assurance

0.71kB (0.71kB) Transaction Size  
this tx 99%

0.71kB (1) Block Size (TX Count)  
available 99%

Minted by [BLADE] BLADE Pool - bladepool.com

Content Metadata (1)

6m20s ago # c3b2f3721139c7d38885f012bab6cae0cff96a405c14a228fe382f6b7b21f074 Total 634.07 ADA \$241.5 B 0.001048

502.02 ADA <span>\$192.8</span> <span>B 0.000728</span> <span>addr_test1_gdvn7</span>	502.29 ADA <span>\$192.8</span> <span>B 0.000728</span> <span>addr_test1_gdvn7</span>
132.28 ADA <span>\$50.3</span> <span>B 0.000728</span> <span>addr_test1_ckk0k</span>	118.6 ADA <span>\$45.2</span> <span>B 0.000728</span> <span>addr_test1_g2pds</span>
	13.18 ADA <span>\$5.0</span> <span>B 0.000728</span> <span>addr_test1_gjfg</span>

# Blockchain Monitoring

In this chapter we will focus on how to monitor the blockchain for changes that our application needs to react to. The most basic type of monitoring is polling which we will describe first. Then we will look at event based monitoring which is more advanced and has more capabilities for error handling and event filtering.

## Polling

Referring to the previous donation example, we need to monitor when Ada is locked at the smart contract. When this occurs, a script will execute a transaction to spend the UTXOs accordingly.



## Cron

One approach is to simply set up a cron job to kick off the `spend-tx.sh` bash script at regular intervals. This script could also be written with Node.js, but in this case we will re-use the bash shell script as is.

A simple cron job to run the script every minute is as follows.

```
* * * * * (cd /absolute-path-to-script-directory/; ./spend-tx.sh preprod >>
/absolute-path-to-log-directory/preprod.out)
```

## Valid UTXOs

When the bash shell script runs, it pulls all the UTXOs currently locked at the smart contract script address. For a valid UTXO entry, the bash script builds a transaction, signs and submits it. Because anyone can lock anything to the script address, a mechanism was created to black list certain UTXOs if required. A UTXO that can't be spent could be the result of an incorrect or missing datum value.

## Monitoring

Below is a simple monitoring script that if the count of the UTXOs at the script address is beyond 1, then manual investigation is required. If a UTXO can't be spent, then it can be added to the blacklist so the script will skip that particular UTXO. When there is more than 1 UTXO locked to the script address, a message will be sent to a Telegram group for a call to action.

Here is the bash shell monitoring script

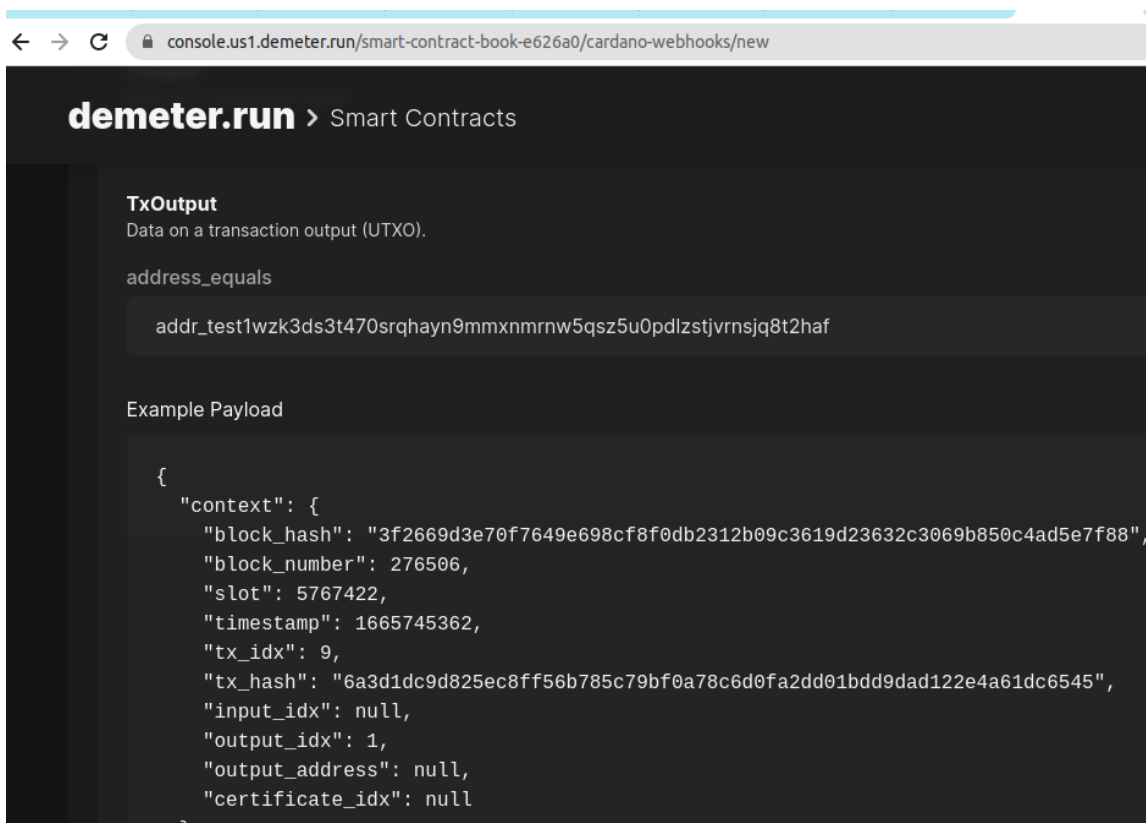
```
#!/usr/bin/bash
count=$(/usr/local/bin/cardano-cli query utxo --address
addr_test1wzk3ds3t470srqhayn9mmxnmrnw5qsz5u0pdlzstjvrnsjq8t2haf --cardano-mode
--testnet-magic 1 | wc -l)
if (( $count > 3 ));
then
  /usr/bin/curl -X POST \
  -H 'Content-Type: application/json' \
  -d '{"chat_id": "XXXXXXXXXX", "text": "Prod utxo count: '$count'",
  "disable_notification": true}' \
  https://api.telegram.org/botYYYYYY:ZZZZZZZZZ\_xtc/sendMessage
fi
```

## Events

Event based monitoring involves monitoring the blockchain activity typically in real-time. For example, a transaction locking Ada at a smart contract address could generate an event and the application responds with some type of action. If needed, an event could be triggered only after a number of [block confirmations](#) has occurred. Additionally, an event could be triggered if a blockchain [rollback](#) has occurred and the application needs to handle this accordingly.

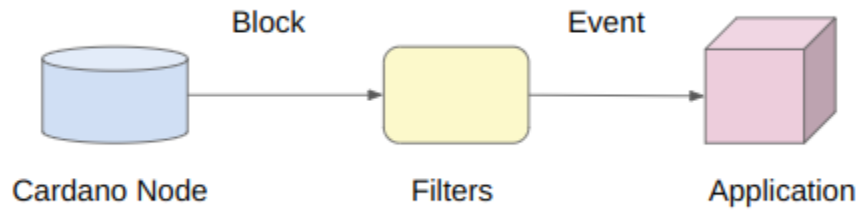
## Oura

There are now service providers that you can subscribe to who have all of the infrastructure and connectivity to the Cardano blockchain set up and ready to go. All you need to do is select the blockchain event you want to define an action for. In our case we will use Oura and select a TxOutput event and watch the donation smart contract script address. The action will then call our webhook with a transaction hash and index and our Next.js application will process it.



## Event Flow

The following high level diagram illustrates the way events are created and handed. Filters are set up on incoming blocks from the Cardano Node. If there is a filter match, then an event is triggered and an application webhook is called.



For more information about Oura can be found at <https://txpipe.github.io/oura/>

## WebHook

Select an event we want to monitor and the URL of the webhook that we want Oura to call when such an event occurs. The webhook is created by setting up an API in our Next.js application. This API will execute the `spend-tx.sh` script with a specific transaction hash and index.

Here is what the `events.ts` code looks like for the new API created to handle an incoming webhook from Oura.

```
import path from 'path';
import type { NextApiRequest, NextApiResponse } from 'next'
import initMiddleware from '../lib/init-middleware'
import validateMiddleware from '../lib/validate-middleware'
import { check, validationResult } from 'express-validator'

const validateBody = initMiddleware(
  validateMiddleware([
    check('context.tx_idx').isInt({ min: 0, max: 255}),
    check('context.tx_hash').isBase64(),
  ], validationResult)
)

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse ) {
  const network = process.env.NEXT_PUBLIC_NETWORK as string;
  const eventAPIKey =
    process.env.NEXT_PUBLIC_EVENT_API_KEY as string;
  if (req.method === 'POST') {
    // Check for basic auth header
    if (!req.headers.authorization ||
      req.headers.authorization.indexOf('Basic ') === -1) {
      throw { status: 401,
        message: 'Missing Authorization Header' };
    }
    // Verify auth credentials
    const apiKey = req.headers.authorization.split(' ')[1];
    if (eventAPIKey !== apiKey) {
      throw { status: 401,
        message: 'Invalid Authentication Credentials' };
    }
  }
}
```

```

// Sanitize body inputs
await validateBody(req, res)
const errors = validationResult(req)
if (!errors.isEmpty()) {
    return res.status(422).json({ errors: errors.array() })
}
const txId = req.body.context.tx_hash + ` ` +
              req.body.context.tx_idx;

const shell = require('shelljs');
const scriptsDirectory = path.join(process.cwd(),
                                     'scripts/cardano-cli');
const cmd = `(cd ` + scriptsDirectory + `; ./spend-tx.sh ` +
              network + ` ` + txId + `)`;

if (shell.exec(cmd).code !== 0) {
    res.setHeader('Tx-Status', 'Tx Failed');
    // Return 200 status because we don't want Oura to retry
    res.status(200).json(`Tx Failed: ` + txId);
} else {
    res.setHeader('Tx-Status', 'Tx Submitted');
    res.status(200).json(`Tx Submitted: ` + txId);
}
}
else {
    res.status(400);
    res.send(`Invalid API Request`);
}
}

```

The `spend-tx.sh` script is changed to accept a transaction hash (`tx_hash`) and index (`tx_idx`) via the command line arguments.

## Smart Contract & Next.js Setup

Make sure that everything is set up correctly.

1. Increment the donation.hl version number (if required)
2. `$ npx deno-bin run --allow-read --allow-write ./src/deploy-donation.js`
3. `$ cp deploy/* scripts/cardano-cli/preprod/data/`
4. `$ cp src/donation.hl contracts/`
5. `$ cd scripts/cardano-cli/`
6. `$ ./init-tx.sh preprod`
7. Query the new script address and make sure the reference script is there.

```
$ cardano-cli query utxo --address addr_test1wpg...42k --cardano-mode --testnet-magic 1
```

		TxHash		TxIx	Amount
-----					
78d...9ac	0	25000000	lovelace + TxOutDatumNone		

8. Update the global-export-variables.sh file with scripts reference UTXO from the previous step
9. Ensure the correct env variables are set (see Donation section)
10. Start the Next.js application

```
$ npm install
$ npm run dev
```
11. Create a webhook for the new smart contract address in Oura

## Oura Setup

The following steps are required to set up a Demeter Run webhook using the Oura hosted solution.

1. Go to Demeter Run and log in
2. Open or create a project
3. Select the Features tab
4. Select Cardano Webhooks
5. Select Create Webhook button
6. Enter the URL of your webhook (<https://some-url/api/events>)
7. Select Headers -> Add
  - a. Authorization
  - b. Basic YOUR-API-KEY
8. Toggle TxOutput Event on



# Test Drive

An order is submitted on-chain and locked at the smart contract address.

The screenshot shows the Cexplorer.io interface in Google Chrome. The browser address bar displays the URL: `preprod.cexplorer.io/tx/704b195961b0c4c72cb496971d81e57f0344e3edb1030d81d6239d2d7e0654b9`. The page title is "Transaction 704b195961b0c4c72cb496971d81e57f0344e3edb1030d81d6239d2d7e0654b9".

The left sidebar contains navigation links for CARDANO EXPLORER (Dashboard, Watchlist, Pools, Assets, Blocks, Transactions, dApps, Metadata, More), EDUCATION (Articles, Videos, Wiki), and ANALYTICS (Decentralization, Network).

The main content area is titled "Transaction detail" and shows the transaction hash: `704b195961b0c4c72cb496971d81e57f0344e3edb1030d81d6239d2d7e0654b9`. The transaction details are as follows:

Field	Value
Date	4m20s ago
Epoch	48
Block	564,953
Slot	19,162,514 (epoch slot 68,114)
Total Output	203.43 ADA (\$ 77.4) B 0.00336
Fee	0.17 ADA (\$ 0.1)
Outputs	2

On the right side, there are three summary cards:

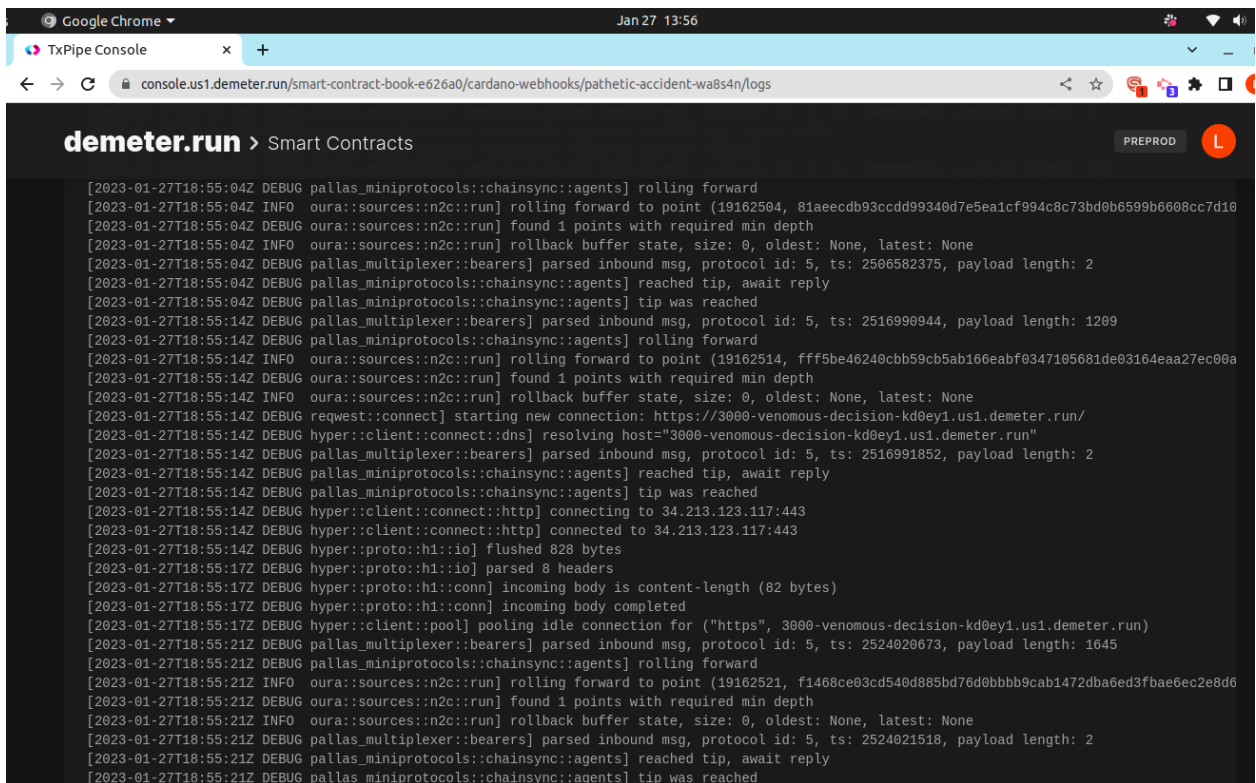
- 16 Assurance** (Green checkmark icon)
- 0.29kB (0.29kB) Transaction Size** (Blue icon, bar chart showing 99% completion)
- 0.29kB (1) Block Size (TX Count)** (Blue icon, bar chart showing 100% completion)

Below these cards, it says "Minted by [BGR] BURGER STAKE POOL" with a blue icon.

The bottom section, titled "Content", shows the transaction's output details:

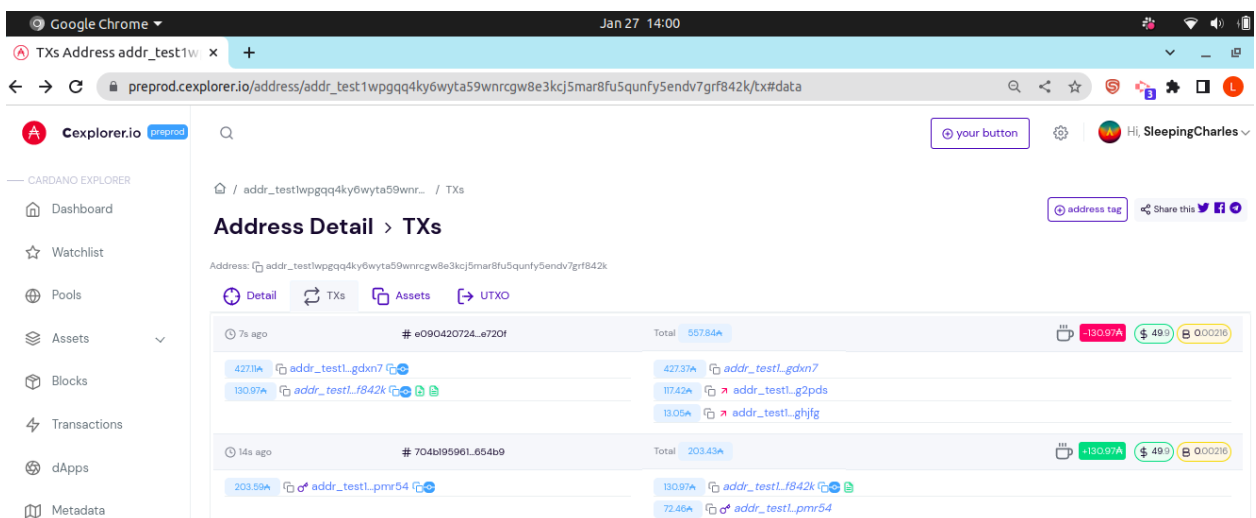
Address	Amount	Label
<code>addr_test1_pmr54</code>	130.97 ADA	<code>addr_test1_f842k</code>
<code>addr_test1_pmr54</code>	72.46 ADA	

A transaction event at the smart contract script address is detected and an action is triggered in Oura.



The screenshot shows a web browser window with the URL `console.us1.demeter.run/smart-contract-book-e626a0/cardano-webhooks/pathetic-accident-wa8s4n/logs`. The page title is "demeter.run > Smart Contracts". The log displays a series of debug and info messages from the Oura smart contract, including rolling forward, parsing inbound messages, and reaching tips. The log entries are timestamped and include details about the contract's state and the network connection.

The webhook API is called and successfully submits the transaction.



The screenshot shows the Cexplorer.io interface. The top navigation bar includes the Cexplorer.io logo, a search bar, and a "your button" button. The main content area displays the "Address Detail > TXs" for the address `addr_test1wpqgq4ky6wytas9wnrcgw8e3k3j5mar8fu5qunfy5endv7grf842k/bx#data`. The address is highlighted in blue. The interface shows a list of transactions, including a transaction with ID `4271a` and another with ID `13097a`. The transactions are listed with their respective IDs, addresses, and transaction hashes. The interface also includes a sidebar with navigation links for Dashboard, Watchlist, Pools, Assets, Blocks, Transactions, dApps, and Metadata.

The metadata for the transaction has also been recorded on-chain.

Google Chrome Jan 27 14:01

Metadata Transaction e0... x +

preprod.cexplorer.io/tx/e0904207245ed7b3c1e8ee7d23b28bdb9667e6635ca096309b0a8378663e720f/metadata#data

Cexplorer.io preprod

your button

Hi SleepingCharles

CARDANO EXPLORER

- Dashboard
- Watchlist
- Pools
- Assets
- Blocks
- Transactions
- dApps
- Metadata
- More

EDUCATION

- Articles
- Videos
- Wiki

ANALYTICS

- Decentralization
- Network

Date	6m13s ago
Epoch	48
Block	564,954
Slot	19,162,521 (epoch slot 68,121)
Total Output	557.84 ₳ <span>\$ 212.3</span> <span>0.00922</span>
Fee	0.23 ₳ <span>\$ 01</span>
Outputs	3

19 Assurance

0.71kB (0.71kB)  
Transaction Size  
this tx 92%

0.71kB (1)  
Block Size (TX Count)  
available 99%

Minted by  
[BGR] BURGER STAKE POOL

Content Metadata (1)

Label	Data
1	<pre>{   "order_detail": {     "date": "2023/01/27-10:55:15",     "version": "0.1",     "order_id": "5247520448598",     "ada_used_price": "0.30324",     "donation_split": "10%",     "order_ada_amount": "130470860",     "donation_ada_amount": "130470860"   } }</pre>

# Troubleshooting

The following section describes some tips and tricks to help debug issues that may occur when developing Cardano smart contracts.

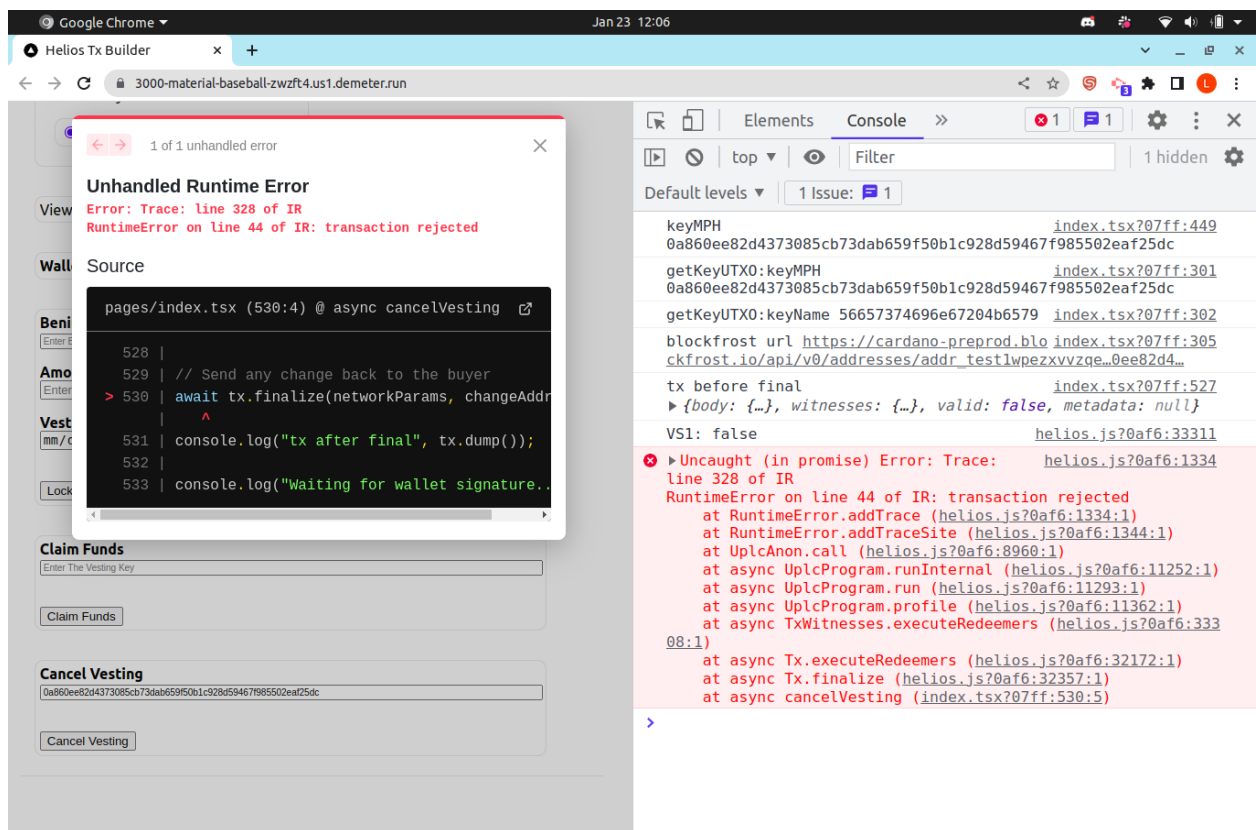
## Trace()

A trace is added to the Helios code by using `.trace()` for an evaluation. The `.trace()` function is actually a wrapper for the `.print()` function and is used for convenience.

For example:

```
// Check if deadline hasn't passed
(now < datum.deadline).trace("VS1: ")
```

The trace can then be seen in the console logs where the plutus script fails.



## Show()

To add more information when debugging, you can add `.show()` to basic types which can then be included in the `.trace()` or `.print()` statements. For example:

```
func main(datum: Datum, redeemer: Redeemer, context: ScriptContext) -> Bool {
    tx: Tx = context.tx;
    now: Time = tx.time_range.start;
    redeemer.switch {
    Cancel => {
        // Check if deadline hasn't passed
        (now < datum.deadline).trace("now: " + now.show() + " VS1: ")
    }
}
```

## Print()

You can also use the `print()` function to print out variables in the execution stack which will show in the browser console when the smart contract transaction fails:

```
func main(datum: Datum, redeemer: Redeemer, context: ScriptContext) -> Bool {  
    tx: Tx = context.tx;  
    now: Time = tx.time_range.start;  
    print(now.show());  
    redeemer.switch {  
    Cancel => {  
        // Check if deadline hasn't passed  
        (now < datum.deadline).trace("VS1: ")  
    }  
}
```

## Common Sources Of Errors

Here is a list of common sources of errors and how to address them.

Issue	Fix
Plutus script exceeds cpu/mem units	Set optimize=true when calling <code>Program.new(script).compile(optimize)</code>  Avoid unnecessary logic or transactions
Invalid reference UTXO	Make sure you update the global-export-variables file has the correct reference UTXO
Wrong script address	Make sure that the helio.js used in src directory and the helios npm node module are the same version.
Collateral not set	Make sure the wallet has collateral set which is needed to execute smart contracts
Insufficient funds	Make sure the wallet has sufficient funds for the transaction and that you are on the correct network.
Wallet account not set correctly	For Nami, make sure that when you are selecting between accounts that the current account select is the active one. You may also need to refresh the browser page as well.
Bash shell environment variables not set	Make sure you run <code>source ~/.bashrc</code> to load the environment variables into your current shell.



## Cardano-cli

To measure the amount of cpu and memory units the plutus script will use, for cardano-cli you just need to replace the output line in the bash shell scripts with the following.

```
--calculate-plutus-script-cost "$BASE/scripts/$ENV/data/mint-tx.costs"
```

So the end of the script will look like:

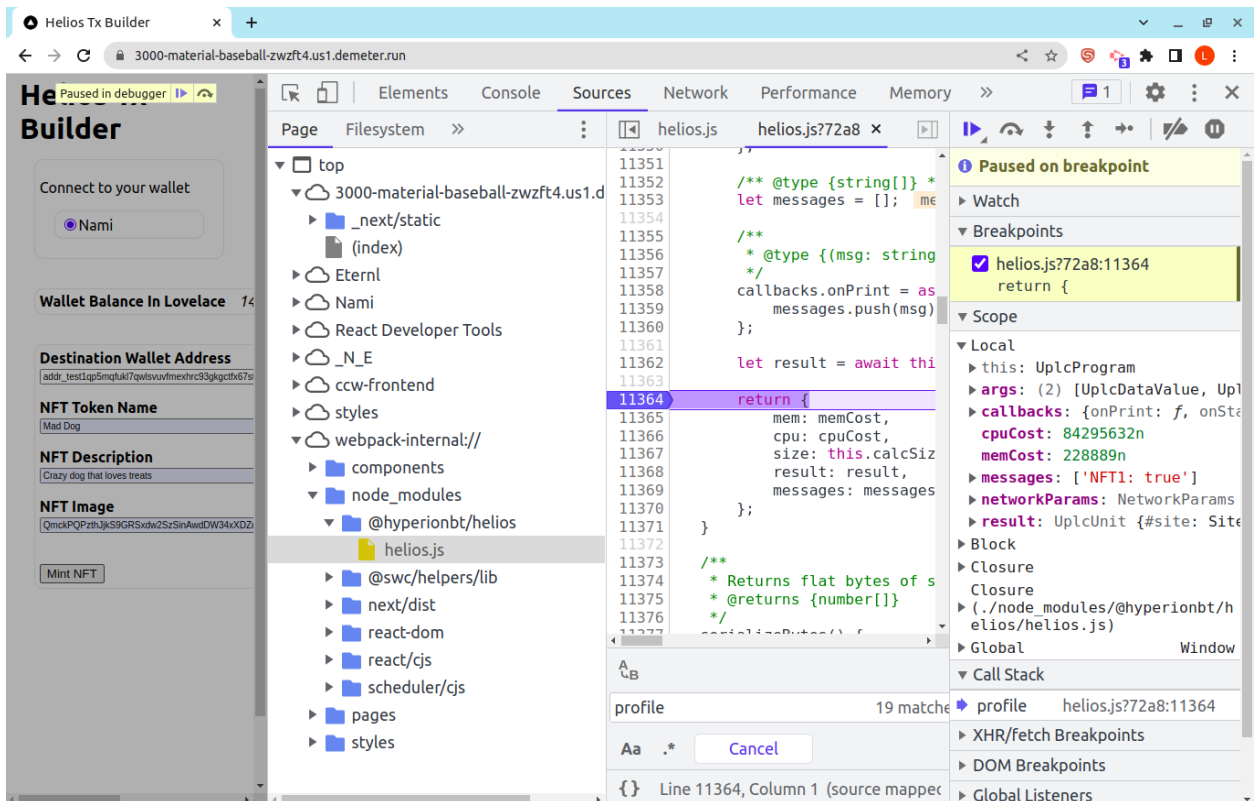
```
...
# Step 3: Build and submit the transaction
cardano-cli transaction build \
--babbage-era \
--cardano-mode \
$network \
--change-address "$admin_utxo_addr" \
--tx-in-collateral "$admin_utxo_collateral_in" \
--tx-in "$admin_utxo_in" \
--tx-in "$sevent_utxo" \
--spending-tx-in-reference "$VAL_REF_SCRIPT" \
--spending-plutus-script-v2 \
--spending-reference-tx-in-inline-datum-present \
--spending-reference-tx-in-redeemer-file "$redeemer_file_path" \
--tx-out "$MERCHANT_ADDR+$merchant_ada" \
--tx-out "$DONOR_ADDR+$donor_ada" \
--required-signer-hash "$admin_pkh" \
--protocol-params-file "$WORK/pparms.json" \
--metadata-json-file "$metadata_file_path" \
--calculate-plutus-script-cost "$BASE/scripts/$ENV/data/spend-tx.costs"
```

Which produces results in the spend-tx.costs file.

```
$ more spend-tx.costs
[
  {
    "executionUnits": {
      "memory": 417629,
      "steps": 153838242
    },
    "lovelaceCost": 35189,
    "scriptHash": "4734d1f1927b26089874c6b3b369c860d450189f6c1844352623ede6"
  }
]
```

# Helios Tx Builder

1. Start up the Next.js application and fill in input fields
2. Right click to inspect the page
3. Select sources -> Webpack internals -> @hyberionbt/helios
4. In the helio.js file search for the word profile
5. At the line with return for the profile function, set a breakpoint
6. Then select Mint in this case
7. The cpuCost and memCost are displayed in Scope -> Local as cpuCost and memCost variables.



## Network Parameters

To find out the current [cost model parameters](#) and the transaction max script size, cpu and mem units on the network, issue the following command:

```
cardano-cli query protocol-parameters --testnet-magic 1
```

Then look for the following values in the output.

```
...  
"maxTxExecutionUnits": {  
  "memory": 14000000,  
  "steps": 10000000000  
},  
"maxTxSize": 16384,  
...
```

## Baseline Comparison

Here are the details of a comparison between a plutus validator script written in Haskell PlutusTx V2 and then again in Helios.

Metric	Haskell Plutus V2	Helios Plutus V2	Reduction
executionUnits Memory	1,899,564	805,465	~42.4 %
executionUnits Steps	528,441,393	297,796,866	~56.4 %
lovelaceCost	147,706	67,947	~46.0 %
Script size	8,125 KB	4,106 KB	~50.5 %

Helios Plutus V2 baseline testing log:

<https://github.com/lley154/littercoin/blob/preprod-5.0/testing/baseline.log>

Haskell Plutus V2 baseline testing log:

<https://github.com/lley154/littercoin/blob/baseline/testing/baseline.log>

## CBOR.me

For advanced troubleshooting, you may need to inspect the transaction on-chain. It is stored in CBOR format and it can be decoded using the following web tool

<https://cbor.me/>

Copy the cborHex and paste it into cbor.me and make sure you select the “as text” option. For example:

```
$ cat work/spend-tx-alonzo.tx
{
  "type": "Witnessed Tx BabbageEra",
  "description": "Ledger Cddl Format",
  "cborHex": "84aa00828...63302e31"
}
```



## CBOR

Diagnostic ☒ as text ☐ utf8 ☐ emb cbor ☐ cborseq

enter hex below or  No file chosen

Diagnostic notation: [1, [2, 3]]

84aa00828258201e75fabdd062ac9eb670a5bfb5adcc7cac0e8ce2789f3d907d769c8d69f65eb02825820bc730168d07eec6f11385242046978c02fbf1b94fbeb0ec408ac0da6d2a309c53000d81825820e06d00315b692ac604cbd2105019dc7d524d3e500d6a0b8c0258ce38699978d60012818258206af1c7ea85bd3222be2e31e5fcd664318b0897b2ab5b68a4db667508aa21a5f8000183a200581d603d62bffdff66855d150b6cf97e4509ef78f5ea6245f642adf7629338c011a071f5188a200581d60b2b0a5ceaf7bc9a56fe619819b8891e6bafef5c2cb275e333f97a9f011a00ca9748a200581d60b9abcf6867519e28042048aa11207214a52e6d5d3288b752d1c27682011a1074f50510a200581d60b9abcf6867519e28042048aa11207214a52e6d5d3288b752d1c27682011a0046fd0f111a00054e31021a000389760e81581cb9abcf6867519e28042048aa11207214a52e6d5d3288b752d1c276820b5820c1d4ececde03173d11592082528d907393fa67815f59d291712adaab8a471353075820b0f31849f135a4b646a0356561d93bb42a167f8618dc56682debc5ae46c4750a200818258207afdf742e52fc46d9a6f2183204c3d5ca3533450b2674f61adff33172d6d9ad65840fdc063509755eed8b17d206d921e0460ee82e4ad35c0166a26383dcfb1d7e04806adbb151081c554c26b9c7f5383e71886d94e1bab1e1ec69dcf72eea8e4c040581840001d87980821a00065f5d1a092b62a2f5d90103a100a101a16c6f726465725f64657461696ca76d6164615f7573645f707269636567302e3337363630646461746573323032332f30312f32342d31333a34353a343873646f6e6174696f6e5f73706c697463313025706f726465725f6164615f616d6f756e746963333237373030306e646f6e6174696f6e5f73706c697463313025706f726465725f6164615f616d6f756e746963333237373030306e6f726465725f69646d353234353436333133343438376776657273696f6e63302e31

Then select the very small arrow in green pointing to the left and you will get the transaction details on the left side.

CBOR playground

← → ↺ cbor.me/?bytes=

CBOR playground. See [RFC 8949](#) for the CBOR specification, and [cbor.io](#) for more background information.

# CBOR

[Diagnostic](#)  ☐ plain hex

 728 Bytes ☒ as text ☐ utf8 ☐ emb cbor ☐ cborseq

enter hex below or  No file chosen

```
[{0:
  [h'1E75FABDD062AC9EB670A5BFB5ADCC7CAC0E8CE2789F3D907D769C8D69F6
  5EB', 2],
  [h'BC730168D07EEC6F11385242046978C02F8F1B94FBE0EC408AC0DA6D2A309C
  53', 0]], 13:
  [h'E06D00315B692AC604CBD2105019DC7D524D3E500D6A0B8C0258CE3869997
  8D6', 0]], 18:
  [[h'6AF1C7EA85BD3222BE2E31E5FCD664318B0897B2AB5B68A4DB667508AA21A
  5F8', 0]], 1: [{0:
    h'603D62BDFDF66855D150B6CF97E4509EF78F5EA6245F642ADF7629338C', 1:
    119493000}, {0:
    h'60B2B0A5CEAF7BC9A56FE619819B8891E6BAFEFF5C2CB275E333F97A9F', 1:
    13277000}, {0:
    h'60B9ABCF6867519E28042048AA11207214A52E6D5D3288B752D1C27682', 1:
    276100357}], 16: {0:
    h'60B9ABCF6867519E28042048AA11207214A52E6D5D3288B752D1C27682', 1:
    4652303}, 17: 347697, 2: 231798, 14:
    [h'B9ABCF6867519E28042048AA11207214A52E6D5D3288B752D1C27682'],
    11:
    h'C1D4ECCDE03173D11592082528D907393FA67815F59D291712ADAAB8A47135
    3', 7:
    h'B0F31849F135A4B646A0356561D93BB42A167F8618DC56682DEBCD5AE46C475
    0'}, {0:
    [h'7AFDF742E52FC46D9A6F2183204C3D5CA3533450B2674F61ADFF33172D6D9
    AD6',
    h'FDC0635097555EED8B17D206D921E0460EE82E4AD35C0166A26383DCF81D7E0
    4806ADB8151081C554C26B9C7F5383E71886D94E1BAB1E1EC69DCF72EEA8E4C04
    ']], 5: [[0, 1, 121([)], [417629, 153838242]]], true, 259({0:
    {1: {"order_detail": {"ada_usd_price": "0.37660", "date":
    "2023/01/24-13:45:48", "donation_ada_amount": "13277000",
```

```
84 AA # array(4)
00 # map(10)
82 # unsigned(0)
82 # array(2)
58 20 # bytes(32)
1E75FABDD062AC9EB670A5BFB5ADCC7CAC0E8CE2789F3D907D769C8D69F65EB
#
"u001Eu\xFA\xBB\xDD\u0006*\xC9\xEBg\n[\xFBZ\xDC\xC7\xCA\xC0\xE8
\xCE'\x89\xF3\xD9\xa\xD7i\xC8e\xEB"
02 # unsigned(2)
82 # array(2)
58 20 # bytes(32)
BC730168D07EEC6F11385242046978C02F8F1B94FBE0EC408AC0DA6D2A309C53
#
"\xBcS\u0001h\xD0-\xECo\u00118RB\u0004ix\C0/\xBF\xe\x94\xFB\xE0\
xEC@\x8A\xC0\xDAm*0\x9CS"
00 # unsigned(0)
0D # unsigned(13)
81 # array(1)
82 # array(2)
58 20 # bytes(32)
E06D00315B692AC604CBD2105019DC7D524D3E500D6A0B8C0258CE38699978D6
#
"\xE0m\u0000[i*\xC6\u0004\xCB\xD2\u0010P\u0019\xDC}RM>P\rj\\v\x8
C\u0002X\xCE8i\x99\xD6"
00 # unsigned(0)
```

Since we are currently in the [Babbage era](#), the CDDL spec is located [here](#).

# Production

This final section will review some of the considerations that should be taken into account when deploying a smart contract into production.

# Legal Notice

MIT License

Copyright (c) 2023 Context Solutions Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



## Audit

Hiring an [independent auditor](#) to review smart contract development for production is strongly recommended for any projects involving customer funds. Please also be aware that Helios is a very new language and currently does not have the same level of audit and production verification as Haskell PlutusTx. That said, as more developers and project teams take advantage of the benefits of using Helios, this risk will diminish over time.

## Test, Test, Test

Once a smart contract has been deployed, it cannot be rolled back or undone. This is the point of the blockchain to be immutable so developers must be aware of this fact when building blockchain applications.

To mitigate the risk of anything unexpected on mainnet, testing must be a key part of the development lifecycle. Helios has a unit testing framework called [Fuzzy Test](#) that can be used in addition to integration, functional and user acceptance testing.

All of the code examples have completed integration and functional testing with documented results located in the GitHub project [testing folder](#). Remember, it is important to test both the positive and negative scenarios.

## Smart Contract Lifecycle

It is important to understand the smart contract lifecycle prior to deploying to production.

- What does an upgrade path look like and can it be done?
- Can the front-end application point to a new smart contract or does it involve coordination with end users?
- How will you remediate issues identified with a smart contract after a mainnet launch?
- How will you retire a smart contract?
- Monitoring [Cardano CIPs](#) for any changes to the plutus core evaluator that may impact smart contract functionality.

## Sources and Acknowledgements

## Open Source

The complete source code for all examples in this book are open source and located at <https://github.com/lley154/helios-examples>. There may be minor differences between the code samples in this book and the repository and this is mainly due to formatting to fit the page layout and for readability.

Without open source projects, this book and countless projects in our ecosystem would not have been possible. Please consider donating to one of your favorite open source projects so they can continue to build great things!

## References

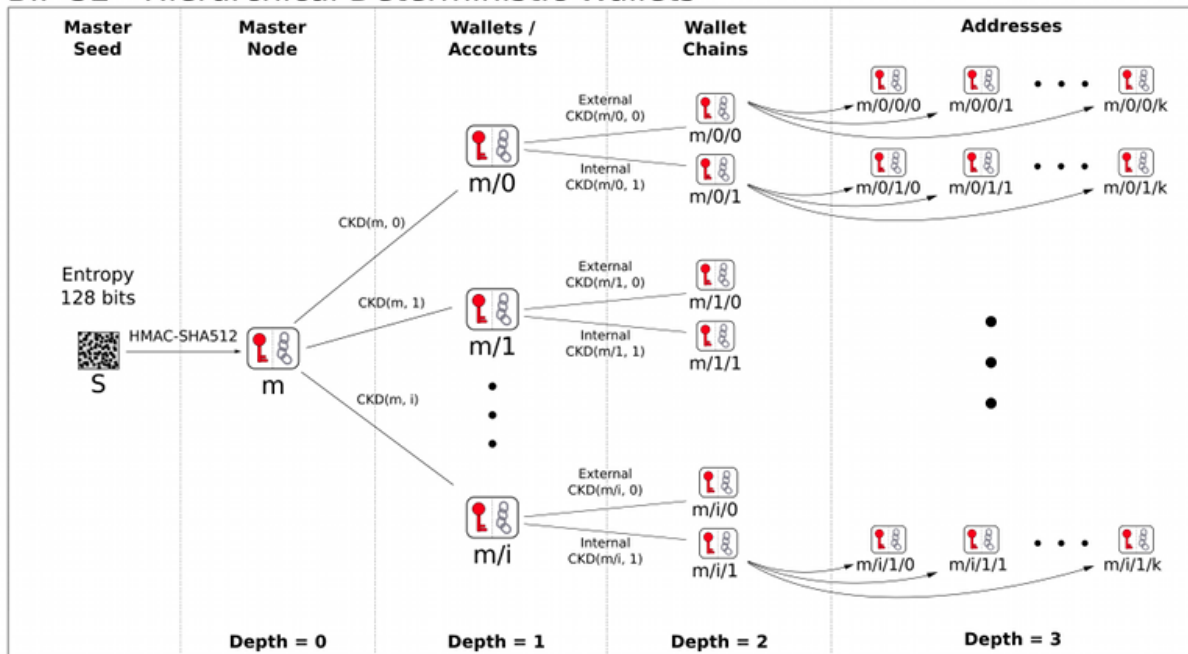
- <https://github.com/Hyperion-BT/helios>
- <https://www.hyperion-bt.org/helios-book/>
- <https://txpipe.github.io/oura/>
- <https://github.com/berry-pool/nami>
- <https://developers.cardano.org/>
- <https://docs.cardano.org/plutus/plutus-resources>
- <https://github.com/input-output-hk/cardano-addresses>
- <https://github.com/input-output-hk/cardano-node>
- <https://github.com/dendorferpatrick/nami-wallet-examples/blob/multi-wallets/Multi-Signature.md#4-hash-metadata-for-transaction>
- <https://www.youtube.com/@AndrewWestberg/videos>
- <https://www.essentialcardano.io/>
- <https://github.com/input-output-hk/plutus-pioneer-program>
- <https://medium.com/@blainemalone01/hd-wallets-why-hardened-derivation-matters-89efcdc71671>

# Appendix

## Address Key Derivation

Key derivation is a complex subject and the main idea behind it is that a private and public key is used to derive different types of keys that are used for different purposes. The starting point for a private key in blockchain cryptography is the BIP-39 encoded seed phrase. The following diagram<sup>1</sup> illustrates the steps involved in key derivation.

### BIP 32 - Hierarchical Deterministic Wallets



Child Key Derivation Function ~  $CKD(x, n) = \text{HMAC-SHA512}(x_{\text{Chain}}, x_{\text{PubKey}} || n)$

<sup>1</sup> Source: <https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>



Here are the steps of obtaining a private key that can be used with cardano-cli from a newly generated passphrase.

1. Go to your Web VS Code in your browser
2. Select the hamburger menu (top left) and Terminal -> New Terminal
3. `mkdir ~/.local/keys`
4. `cd ~/workspace`
5. `wget https://github.com/input-output-hk/cardano-wallet/releases/download/v2022-12-14/cardano-wallet-v2022-12-14-linux64.tar.gz`
6. `tar -xvzf cardano-wallet-v2022-12-14-linux64.tar.gz`
7. `cd cardano-wallet-v2022-12-14-linux64`
8. `./cardano-address recovery-phrase generate --size 24 > ~/.local/keys/key.prv`
9. `./cardano-address key from-recovery-phrase Shelley < ~/.local/keys/key.prv > ~/.local/keys/key.xprv`
10. `./cardano-address key child 1852H/1815H/0H/0/0 < ~/.local/keys/key.xprv > ~/.local/keys/key.xsk`
11. `./cardano-cli key convert-cardano-address-key --shelley-payment-key --signing-key-file ~/.local/keys/key.xsk --out-file ~/.local/keys/key.skey`
12. `./cardano-cli key verification-key --signing-key-file ~/.local/keys/key.skey --verification-key-file ~/.local/keys/key.vkey`
13. `./cardano-cli address key-hash --payment-verification-key-file ~/.local/keys/key.vkey --out-file ~/.local/keys/key.pkh`
14. `./cardano-cli address build --payment-verification-key-file ~/.local/keys/key.vkey --out-file ~/.local/keys/key.addr --testnet-magic 1`

You can then see the address created by your passphrase with a usable private & public key for cardano-cli.

```
$ more ~/.local/keys/key.addr  
addr_test1v83ynr979e4xpjj28922y4t3sh84d0n08juy58am7jxmp4g6cgxr4
```

## Installing Next.js

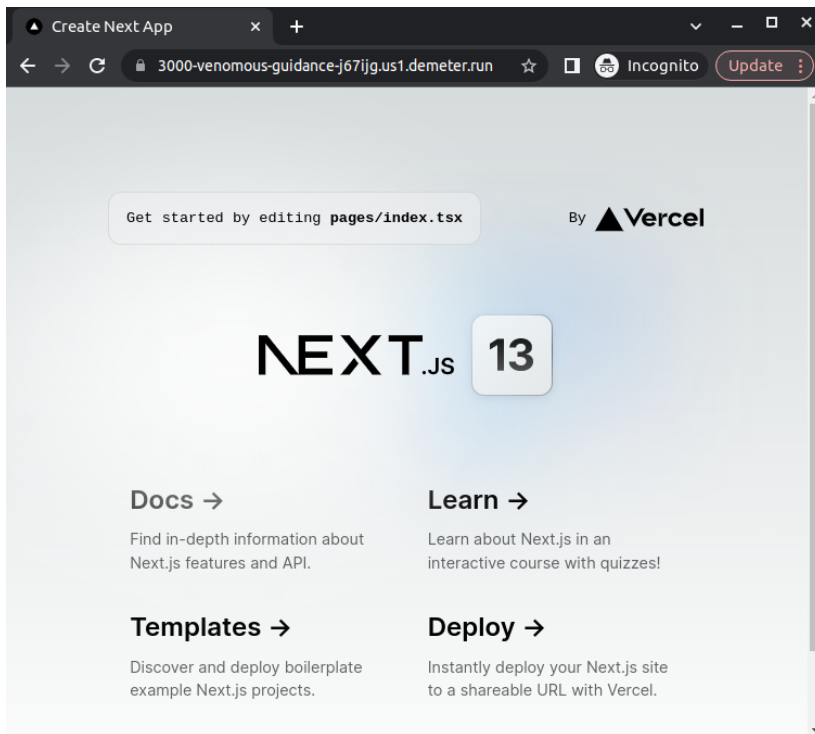
Go to your workspace using the terminal window in the Visual Studio Web.

```
$ npx create-next-app@latest --typescript .
Need to install the following packages:
create-next-app@13.1.2
Ok to proceed? (y) y
✓ Would you like to use ESLint with this project? ... Yes
✓ Would you like to use `src/` directory with this project? ... No
✓ Would you like to use experimental `app/` directory with this project? ... No
Creating a new Next.js app in /config/workspace/repo/helios-builder/app.
Using npm.
Installing dependencies:
- react
- react-dom
- next
- @next/font
- typescript
- @types/react
- @types/node
- @types/react-dom
- eslint
- eslint-config-next
added 271 packages, and audited 272 packages in 28s
102 packages are looking for funding
run `npm fund` for details
found 0 vulnerabilities
Initializing project with template: default
Success! Created app at /config/workspace/repo/helios-builder/app
npm notice
npm notice New major version of npm available! 8.19.3 -> 9.3.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v9.3.0
npm notice Run npm install -g npm@9.3.0 to update!
npm notice
abc@venomous-guidance-j67ijg-0:~/workspace/repo/helios-builder/app$
```

## Startup Next.js

```
$ npm run dev
```

Then select the URL on the Exposed Ports tab for your workstation to launch the application. You should see the default Next.js page like this.



To use Helios, you will need to install the Helios npm module

```
$ npm install @hyperionbt/helios
```

Also install the express-validator module to sanitize inputs when required.

```
$ npm install express-validator
```

You will also need to replace the next.config.js file with the following

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  reactStrictMode: true,
  webpack: function (config, options) {
    config.experiments = {
      asyncWebAssembly: true,
      topLevelAwait: true,
    };
    return config;
  },
};
module.exports = nextConfig;
```

## IPFS (NFT Images)

The InterPlanetary File System (IPFS) is typically used to store NFT images and other media files. IPFS is a peer-to-peer file sharing network where uploaded content gets a unique key so it can be accessed.

There are a large number of service providers that can help get your images on to IPFS. An easy and free option is thirdweb.

<https://blog.thirdweb.com/guides/how-to-upload-and-pin-files-to-ipfs-using-storage/>

```
$ npx thirdweb@latest upload path/to/file.extension
```

## Shopify Ada Payments

At the time of writing, there is no Ada Shopify payment plugin. The following Shopify integration is for merchants who have a basic plan and cannot customize their checkout page. For merchants who have higher priced plans, they can do more front-end and checkout customizations which can streamline a pay with Ada process.

## Shopify Configuration

We need to update the Shopify store so when an order is placed, there is a link to pay with Ada.

1. In Settings, select Apps and sales channels
2. Select Develop apps for your store
3. Select Allow custom app development
4. Select Allow customer app development again
5. Create an app
6. Enter the name of the app, eg. "Pay With Ada"
7. Select Create app
8. Select Configure Admin API scopes
9. Scroll down and enable in the Order section, write\_orders and read\_orders
10. Scroll down and enable in the Products section, read\_products
11. Go to the bottom of the page and press Save
12. Now select the API credentials tab
13. In the Access tokens box, select Install app
14. A prompt will ask you if you want to install your app, select Install
15. Go back to the main settings menu and select Payments
16. Select Add manual payment method
17. In the dialog box, enter the name of the payment method. eg "Pay With Ada"
18. Add additional details if required. eg "Paying with Ada using your Cardano Wallet"
19. Add payment instructions. eg. "Please select to the Pay Now With Ada link below to pay using your Cardano Wallet"
20. Go back to the main settings menu and Select Checkout
21. Scroll to the bottom of the page and add the following to the Additional scripts window

```
<script>
  Shopify.Checkout.OrderStatus.addContentBox(
    '<h2 style="color:red;">Pay To Complete Your Order</h2>',
    '<a href="#" id="paynow">Pay Now In Ada</a>'
  );
  var urlStr =
"https://3000-venomous-decision-kd0ey1.us1.demeter.run/";
  var url = new URL(urlStr);
  var params = url.searchParams;
  var orderId = Shopify.checkout.order_id
  params.append("id", orderId);

  function updatePayNow () {
    document.getElementById("paynow").href=url
  }
</script>
```

```

function updateOrderNum () {
    document.querySelector("body > div > div > div > main >
div.step > div.step__sections > div:nth-child(1) > div > div >
span").innerHTML=orderId
}

document.addEventListener("DOMContentLoaded", function() {
    updatePayNow()
});
document.addEventListener("DOMContentLoaded", function() {
    updateOrderNum()
});
</script>

```

22. Select Save
23. In the Setting menu, select Apps and sales channels
24. Select Develop Apps
25. Select Pay With Ada App
26. Select Manage Credentials
27. Show the Access Token and copy it to your clipboard
28. Paste the Access Token into your ~/.bashrc file for  
NEXT\_PUBLIC\_ACCESS\_TOKEN value



# About Helios



A DSL for writing smart contracts on the Cardano blockchain.

Twitter: [@helios\\_lang](https://twitter.com/helios_lang)

Github: <https://github.com/Hyperion-BT/helios>

Discord: <http://discord.gg/XTwPrvB25q>

Web: <https://hyperion-bt.org>

## About The Author



Lawrence Ley is passionate about learning new technologies and how they can make a positive impact. His Cardano journey started out in the Plutus pioneers program 2nd cohort and has worked on a number open source impact projects. He discovered Helios and was thoroughly impressed how easy it was to pick up due to its familiar language, syntax and structure. Since onboarding new developers to Cardano is not easy, he decided to write this book.

Twitter: [@lley154](https://twitter.com/lley154)

YouTube: <https://www.youtube.com/channel/UC5R8HrDUms8XqZVWszqbA7A>

GitHub: <https://github.com/lley154>