Minting tokens on OSX using cardano-cli

Jack O'Brien

Table of Contents

Introduction

The aim of these tutorial notes are to demostrate how to mint tokens on Cardano using the commandline tool `cardano-cli` on Apple Mac OSX. The notes are a companion for a series of tutorial videos.

Everything is done on a Cardano **Testnet** so you don't need to buy Ada or spend Ada to complete the steps.

Everything shown works on the Cardano Mainnet too. You will need to make a few changes to the commandline options.

The tutorial shows how to take advantage of Cardano's native asset support. **Tokens** are the accounting units of **Assets** in Cardano. They are equivalent to ERC20 or ERC721 tokens on Ethereum.

We'll be using the **Cardano Node** that is installed with Daedalus, the full-node wallet for Cardano. Daedalus works on Microsoft Windows, Apple Mac OSX, and Linux.

This tutorial is for Apple Mac OSX, and there is a companion tutorial for Microsoft Windows.

> If you are using Linux the steps and command-line details should be the same as Linux shares a simliar syntax. However, the path to the **Cardano Node** and socket will be different.

To make the tutorials consistent across platforms I'll be using Microsoft Visual Studio (VSCode). By opening this file in VSCode, or in your browser you can copy/paste and change the parameters around directly within VSCode. I'll be using VSCode in future tutorials so this is a way to introduce things. This does however mean you'll need to install VSCode.

If you don’t want to do that, you will still be able to follow along easily enough using just a terminal window.

Videos

This document is a companion for a series of short tutorial videos.

Getting Started - Setting up the tools

To get started we need to install two main applications. Download and install:

1. Daedalus Testnet - The Cardano full-node Wallet for the Testnet.
2. Microsoft Visual Studio Code - A code development environment with text editors and integrated terminals.

I’m using VSCode as my base of operations. Hopefully that will make it easier for you to follow along with the code and commandline steps I take.

> When downloading Daedalus Testnet make sure to verify the signatures of the binary release you’re about to install. **Before you install it!** Use either SHA256 or GPG to do this. Instructions for doing your operating system can be found below the download button. Verifying the signatures makes sure what you’re about to install is the official release. This is a habbit you should get into, keeping you and your tokens secure.

Syncing with the Testnet

As a full-node Wallet, Daedalus includes a complete installation of the **Cardano Node** and supporting tools. We will be taking advantage of that and interacting with the **Cardano Node** via the commandline. Using a GUI also makes it easy to see the results of submitting your transactions.

After installing Daedalus Testnet, open it and let the Wallet sync with the Testnet. This will pull down the Testnet Blockchain locally. It may take 30 minutes or more to do this.

> Completing the tutorial successfully requires a fully sync’d Testnet node, **be patient** and let it sync up.

Creating a Bash Terminal

Open up VSCode then `Ctrl` + `Shift` + `` ` `` to open up a Bash shell terminal.

> By default your terminal is typically set to using the default shell. This may be Bash or ZShell. Either is fine and I’ll use **shell session** or **Bash** to mean either.

Next we need to setup the filesystem locations for the `cardano-cli`, the communication channel between `cardano-cli` and the running **Cardano Node**, and a magic number identifying the Testnet we’re using. These settings will be used in our shell session.

```
alias cardano-cli="/Applications/Daedalus\ Testnet.app/Contents/MacOS/cardano-cli" ❶

export CARDANO_NODE_SOCKET_PATH=~/Library/Application\ Support/Daedalus\ Testnet/cardano-node.socket ❷

export MAGICID=1097911063 ❸
```

❶ Linking our `cardano-cli` that came with Daedalus.

❷ Mapping the socket id to interface with the Cardano node on Daedalus.

❸ Creating an environment variable with the magic number for the Testnet.

> If you installed Daedalus Testnet into a different location than the default used above, you'll need to change the paths to reflect the alternative location.

These commands must be re-entered everytime you start a new shell. To save some time you can set up the alias, path, and MAGICID variables in the your `` .bashrc`or `.zshrc `` file found in your `$HOME` directory. That way you don't have to enter them into the shell everytime.

Testing our setup

To make sure everything is working as expected, and the **Cardano Node** has synced, we need to check the tip of the Testnet blockchain. Enter the following in the shell:

```
cardano-cli query tip --testnet-magic $MAGICID
```

You should get a positive result, indicating we are ready to go.

Generating Address Key Pairs

Forming transactions using the `cardano-cli` requires using addresses. An **Address** is derived from a payment key (often called a Public key). To create an **Address** we first need to generate the payment and signing keys then generate the address from the payment key.

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

cardano-cli address build --payment-verification-key-file payment.vkey --out-file payment.addr --testnet-
magic $MAGICID ❸
```

❶ Generating your payment and signing keys.

❷ Generating a wallet **Address** from the payment key

Testing the Generated Address

You can test that the **Address** is valid and working by querying the blockchain for any unspent transaction outputs associated with the address. If you've followed the steps above there will be no unspent outputs since the **Address** is new.

```
cardano-cli query utxo --address $(cat payment.addr) --testnet-magic $MAGICID --mary-era
```

We'll use this command repeatedly in the tutorial to look at the wallet address. `cardano-cli query utxo …` will list the unspent outputs for the address. Hence you can also replace the `$(cat payment.addr)` in the above command with an explicit address from the Testnet.

Building Transactions

> ! The commands shown below use **Address**, **Transaction IDs**, and **Asset IDs** that are specific to a set of payment and signing keys and the given session. Where possible the tutorial abstracts these details out into environnment variables so they can be easily changed for your session. But you will need to change them!

Getting some Testnet Ada

Before we move onto making our first transaction, I've already transfered some Ada into the generated **Address**. I did this via the Daedalus Testnet GUI by copy and pasting the generated **Address** into the **Send to** field when making a payment. You can get your newly generated address by:

```
cat payment.addr
```

If you have no **Testnet ADA** you can get some from the Testnet Faucet. Either use your newly generated **Address** as receipent, or use an **Address** from your Daedalus Testnet Wallet. Then transfer a small amount, enough to pay fees etc. to your generated **Address** as I have done.

Network Parameters

To form transactions you need some information about the Testnet protocol parameters.

```
cardano-cli  query protocol-parameters --testnet-magic $MAGICID --out-file protocol.json --mary-era (1)
```

The file `protocol.json` will be used throughout the following steps to ensure the generated transactions can only be used on the designated Testnet network. In our case the current Mary Era Testnet.

Helper Environment variables

To make it easier and more readable to use the commandline, and prevent errors, we can setup and use some helper environment variables. We'll use these environment variables when building and signing a transaction. Modifying them as and when needed.

```
export LOVELACE1=10000000 (1)

export OUTFILE=matx.raw (2)

export SIGNEDFILE=matx.signed (3)

export FEE=0 (4)
```

1. The amount of Lovelace is in that unspent output.
2. Information of the transaction we plan to build.
3. Our signed file we created with our signing key.
4. How big a fee is needed to process the transaction.

Some of the environment variables will have values specific to a particular step in our workflow. We'll be updating them as we proceed. For the moment these are the values I'm using. Plug in your settings.

```
export TXINID=f5ed8592d6c733f8942c2bff7714be90f466148c70f17e6c55138950c025d2a7 ❶

export
OUTADDR=addr_test1qzz2l4gmm29rg0lmweh39x7lcwm6yxmha5806slfks8mxz83zg5yyt7lc4wuekkks0pefg468s8nhy2e4srz7lu
2dssqqej8pg ❷
```

❶ The address we want to send to an output to. In this scenario I've used an address from my Daedalus Wallet.

❷ ID of the unspent output we plan to spend. It is a combination of the Transaction Id and Output Index, the `#0` at the end. It changes everytime we spend an output.

Our first transaction

Now that our environment is setup we can proceed with building a transaction. It takes multiple steps to do that. First we create a basic raw version of the transaction, calculate the fee, sign it, then submit the transaction.

The first step is to build the raw transaction.

```
cardano-cli transaction build-raw \
  --mary-era \ ❶
  --fee $FEE \ ❷
  --tx-in "$TXINID"#0 \ ❸
  --tx-out $OUTADDR+$(expr $LOVELACE1 - $FEE) \ ❹
  --out-file $OUTFILE ❺
```

❶ For the Testnet we must specifiy the **Era** of the network we are using. In our case it is the **Mary Era**.

❷ The Transaction fee is set to zero, we're using the `$FEE` environment variable we set up earlier to do this.

❸ Specify the Transaction Output we want to spend. Again this is being pulled in from the `$TXIND` environment variable set up previously.

❹ Instead of trying to manually calculate the output, we're using an inline expression instead. We're only creating a single output in this transaction.

❺ The raw transaction information is written to a file.

Calculate Transaction Fee

To figure out the minimum transaction fee for our transaction, we need to calculate it. Using the raw transaction information we just generated.

```
cardano-cli transaction calculate-min-fee \
  --tx-body-file $OUTFILE \ ❶
  --tx-in-count 1 \ ❷
  --tx-out-count 1 \ ❸
  --witness-count 1 \ ❹
  --testnet-magic $MAGICID \ ❺
  --protocol-params-file protocol.json ❻
```

❶ The raw transaction we generated in the prior step.

❷ We're only using a single transaction input.

❸ Only a single Transaction Output is being generated.

❹ This is a simple pay-to-address transaction so we only need to construct a witness with the signing key of the input.

5. Only required for the Testnet, this specifies what test network we are using.
6. Specific network protocol parameters that include details about fees.

Running the above will output a fee in our terminal. Take that value and set the `$FEE` variable with it, then rebuild the raw transaction.

```
export FEE=0 <1>  # <-- CHANGE ME (1)

cardano-cli transaction build-raw \ (2)
  --mary-era \
  --fee $FEE \
  --tx-in "$TXINID"#0 \
  --tx-out $OUTADDR+$(expr $LOVELACE1 - $FEE) \
  --out-file $OUTFILE
```

1. Make sure to change this with the fee calculated in the previous step.
2. This is just the same command used to generate the raw transaction. Use the up arrow to quickly repeat the command from your terminal history.

Sign and Submit the Transaction

Assuming everything worked above, you’re now ready to sign the transaction using the signing key you generated in the first few steps above. That assumes the **Output** we are spending is coming from the **Address** we generated.

```
cardano-cli transaction sign \
  --signing-key-file payment.skey \ (1)
  --testnet-magic $MAGICID \
  --tx-body-file $OUTFILE \
  --out-file $SIGNEDFILE (2)
```

1. The payment key we generated previously.
2. The signed transaction is written to this file.

Once the transaction is signed it can be submitted to the local **Cardano Node** and eventually confirmed and added into the blockchain.

```
cardano-cli transaction submit \
  --tx-file $SIGNEDFILE \
  --testnet-magic $MAGICID
```

Congratulations on making your first transaction! The transaction will appear in Daedalus TestNet or via the `cardano-cli` in about twenty seconds.

To query your address from the commandline:

```
cardano-cli query utxo --address $OUTADDR --testnet-magic $MAGICID --mary-era
```

You should see the new unspent output appearing in the list. Next up let’s mint a token.

Minting Native Tokens with meta-data

Now that we have a basic understanding of how transactions are built with `cardano-cli` let's try minting **Tokens** for an **Asset**. To do this we need to create a policy script (a type of multisignature script) that governs the minting and burning of **Tokens**. The hash of the policy script is the **Asset ID**. In addition we're going to add some transaction meta-data. Hinting at the ability for Cardano to create NFTs (more capable version of ERC721 tokens if you're familiar with Ethereum).

Creating a policy script

The policy script sets the rules around how a **Token** for a given **Asset** can be minted and burnt. Without it we can't do either. Keep in mind that the **Asset ID** is the hash of the policy script, hence changing anything in the script and we have a new **Asset**. We are creating a basic policy script for the moment, but a policy script is either a multi-signature script or a Plutus script (After Alonzo is released). There is lot more to scripts that I'm not going to cover at all.

```
mkdir policy ❶

cardano-cli address key-gen --verification-key-file policy/policy.vkey --signing-key-file
policy/policy.skey ❷

touch policy/policy.script && echo "{" > policy/policy.script

echo "  \"keyHash\": \"$(cardano-cli address key-hash --payment-verification-key-file
policy/policy.vkey)\"," >> policy/policy.script

echo "  \"type\": \"sig\"" >> policy/policy.script

echo "}" >> policy/policy.script ❸
```

❶ Making a new directory named `policy`

❷ Generating our new keys needed for the **Asset** policy script

❸ Creating a policy script file

Check to see if it worked in our terminal.

```
cat policy/policy.script ❶
```

❶ Looking into the script file to see if the data we put in worked.

```
cardano-cli transaction policyid --script-file ./policy/policy.script ❶

policy script output: 84061ca10033c03618948a25790a7d103feb2ef25c0fd388f8c28c34 ❸
```

❶ Generating our unique policy ID from our script file needed to mint our tokens.

❷ The Policy ID output.

Creating Transaction meta-data

We're not going to dive into any details of transaction meta-data. I just want to demonstrate the capability more than anything else. To do this we'll just use some simple meta-data encoded as a JSON file.

First create the file.

```
touch metadata.json
```

Open it in our editor `code metdata.json` and paste the following in. None of the values are important, so feel free to change them to suit.

```
{
    "6969":{
       "ticker": "MELON",
       "name": "meloncoin",
       "description": "This is a description about watermelons.",
       "homepage": "www.melons.com",
       "address": "addr_test1vq0ghmsf2n4vqd8sv5c0emht0mmfpc47zdt3rzql447g8vgmfcwkz"
    }
  }
```

With our metadata set up, we can now go ahead and reset some of our environment variables as well as make some new ones.

```
export TXINID=6d5b3511d5c2831ed46cf23fd566a4952de9352722e47a1efb92bd4176de340d

export LOVELACE1=10000000

export FEE=0

export ASSET1="10 84061ca10033c03618948a25790a7d103feb2ef25c0fd388f8c28c34.melonCoin"

export METADATA=metadata.json
```

Now let’s build a new transaction with our new variables set.

```
cardano-cli transaction build-raw \
  --mary-era \
  --fee $FEE \
  --tx-in "$TXINID"#0 \
  --tx-out $OUTADDR+$(expr $LOVELACE1 - $FEE)+"$ASSET1" \ 1
  --mint "$ASSET1" \ 2
  --json-metadata-no-schema \ 3
  --metadata-json-file $METADATA \ 4
  --out-file $OUTFILE
```

1. The new tokens are added to the **Output’s** token bundle.
2. Forge the new tokens for the **Asset**.
3. The meta-data is not using any JSON schema.
4. The Meta-data file.

With the raw transaction created we now need to calculate the fee, recreate the raw transaction, sign it, and submit it as we did with our first simple transactions.

```
cardano-cli transaction calculate-min-fee \
  --tx-body-file $OUTFILE \
  --tx-in-count 1 \
  --tx-out-count 1 \
  --witness-count 1 \
  --testnet-magic $MAGICID \
  --protocol-params-file protocol.json
```

Set the `$FEE` environment variable then rebuild the raw transaction.

```
export FEE=0 <1>  # <-- CHANGE ME ❶

cardano-cli transaction build-raw \
  --mary-era \
  --fee $FEE \
  --tx-in "$TXINID"#0 \
  --tx-out $OUTADDR+$(expr $LOVELACE1 - $FEE) \
  --out-file $OUTFILE
```

❶ Change this to reflect the calculated fee.

Now sign the raw transaction and submit it… then check the unspent outputs.

```
cardano-cli transaction sign \
  --signing-key-file payment.skey \
 --signing-key-file ./policy/policy.skey \
 --script-file ./policy/policy.script \
 --testnet-magic $MAGICID \
 --tx-body-file $OUTFILE \
  --out-file $SIGNEDFILE

cardano-cli transaction submit \
  --tx-file $SIGNEDFILE \
  --testnet-magic $MAGICID

cardano-cli query utxo --address $(cat payment.addr) --testnet-magic $MAGICID --mary-era ❶
```

❶ Checking to see if it worked, do give it a few seconds.

Congratulations you've succesfully minted some tokens with meta-data in the transaction!

If you want to see the meta-data, get the transaction ID and look up the transaction with the Testnet Transaction Explorer.

Burning Tokens

The process of burning (deleting or disposing of tokens) is more-less identitical to minting tokens. Instead of using a positive value which is for minting, we use a negative value to burn **Tokens**. You need the policy script we generated for minting in order for this work.

To get started let's reset our environment variables, the ones that count at least.

```
export TXINID=3fd9dfe8d42562af8bf373c6bcff3918df0a386dc8e3c216992871d07a770f2a

export ASSET1="-10 84061ca10033c03618948a25790a7d103feb2ef25c0fd388f8c28c34.melonCoin" ❶

export LOVELACE1=5000000

export FEE=0
```

❶ Using a negative integer to burn ten tokens.

Now it's as simple as building, signing and submiting the transaction as before.

```
cardano-cli transaction build-raw \
    --mary-era \
    --fee $FEE \
    --tx-in "$TXINID"#0 \
    --tx-out $OUTADDR+$(expr $LOVELACE1 - $FEE) \
    --mint "$ASSET1" \
    --out-file $OUTFILE
```

Calculate a fee then set the `$FEE` environment variable then rebuild the raw transaction again.

```
cardano-cli transaction calculate-min-fee \
  --tx-body-file $OUTFILE \
  --tx-in-count 1 \
  --tx-out-count 1 \
  --witness-count 1 \
  --testnet-magic $MAGICID \
  --protocol-params-file protocol.json

export FEE=0 <1>  # <-- CHANGE ME (1)

cardano-cli transaction build-raw \
  --mary-era \
  --fee $FEE \
  --tx-in "$TXINID"#0 \
  --tx-out $OUTADDR+$(expr $LOVELACE1 - $FEE) \
  --out-file $OUTFILE
```

(1) Change this to reflect the calculated fee.

Sign and submit the transaction.

```
cardano-cli transaction sign \
  --signing-key-file payment.skey \
 --signing-key-file ./policy/policy.skey \
 --script-file ./policy/policy.script \
 --testnet-magic $MAGICID \
 --tx-body-file $OUTFILE \
  --out-file $SIGNEDFILE

cardano-cli transaction submit \
  --tx-file $SIGNEDFILE \
  --testnet-magic $MAGICID

cardano-cli query utxo --address $OUTADDR --testnet-magic $MAGICID --mary-era (1)
```

(1) Checking to see if it worked, but do give it a few seconds.

You will see the ADA we specified in our `$LOVELACE` variable, minus the fee, shows up in the address we sent it to with the 10 **Tokens** removed and destroyed!

Congratulations you sucessfully minted and burnt native asset tokens without the need of any smart-contracts. Now if only there was someway to pack this all into a script with a nice UI… That is coming to Daedalus soon.