CARDANO DEVELOPMENT COURSE (Week 4)

Module 4: Front-end Integration with Mesh JS.

4.0. Integrating vesting and minting Aiken contract code with mesh js

4.1. Interacting with wallets

4.2. Creating and submitting transactions

4.3. Locking and unlocking funds in vesting contract

4.4. Minting tokens to wallet

4.0. Integrating vesting and minting Aiken contract code with mesh js

4.0.1. Vesting contract with mesh

As a reminder, this is the vesting contract:

```
use cardano/transaction.{OutputReference, Transaction}
use cocktail/vodka_extra_signatories.{key_signed}
use cocktail/vodka_validity_range.{valid_after}
use aiken/crypto.{VerificationKeyHash}

pub type VestingDatum {
  /// POSIX time in milliseconds, e.g. 1672843961000
  lock_until: Int,
  /// Owner's credentials
  owner: VerificationKeyHash,
  /// Beneficiary's credentials
  beneficiary: VerificationKeyHash,
}

validator vesting {
  // In principle, scripts can be used for different purpose (e.g.
minting
  // assets). Here we make sure it's only used when 'spending' from a
eUTxO
  spend(
    datum_opt: Option<VestingDatum>,
    _redeemer: Data,
    _input: OutputReference,
    tx: Transaction,
  ) {
    expect Some(datum) = datum_opt
    or {
      key_signed(tx.extra_signatories, datum.owner),
      and {
        key_signed(tx.extra_signatories, datum.beneficiary),
```

```
        valid_after(tx.validity_range, datum.lock_until),
      },
    }
  }

  else(_) {
    fail
  }
}
```

To use the contract, we first build it with `aiken b` to get the `plutus.json` file.

The plutus JSON file contains information about each validator and types in your Aiken project. It is an object with different fields, the most important of which is the `validators` array. Each validator is represented as an object in the array. Each validator object contains the compiledCode field, which is the Plutus Core (PLC) bytecode of the validator.

The PLC bytecode, when Doubly CBOR encoded, is used to interact directly with the validator on-chain. We will see how that is done shortly.

Utilizing the smart contract with the off-chain library, mesh js:

```
const vestingValidator = blueprint.validators.filter(v => (
    v.title.includes("vesting.vesting.spend")
));
```

The code above gets the validator object from the imported blueprint which is the plutus.json file's object.

```
const vestingValidatorScript = applyParamsToScript(
    vestingValidator[0].compiledCode,
    [],
    "JSON",
);
```

We apply Double CBOR encoding to the validator's compiled code. This is done by using mesh's `applyParamsToScript` function. Under the hood, the function applies CBOR encoding to the compiledCode.

```
const vestingValidatorHash =
resolveScriptHash(vestingValidatorScript, "V3");
```

We can then get the hash/script_hash of the validator with the code above.

This is how you get your validator from your plutus file, apply CBOR encoding, and get its hash.
In the subsequent modules, you are going to see how you interact with wallets, create and submit vesting transactions.

4.0.2. Minting contract with mesh

All the steps used for the vesting contract above apply to the minting contract.

Something to note about the minting contract is its script hash is also referred to as the minting policy. The minting policy, along with the asset name of the tokens or NFTs minted, serves as the identity for that particular asset which is referred to as the "asset unit".

4.1. Interacting with wallets

What are wallets?
Wallets are software tools that let users manage their ADA and native tokens (Send and receive, stake, use in DApps), interact with the Blockchain (smart contracts, DApps), and perform transactions.

Mesh JS can use wallets to submit transactions we create to the blockchain.

Setting up a wallet with mesh JS:

Mesh provides 2 wallet APIs for interacting with wallets:

- BrowserWallet, and
- MeshWallet

We are going to be using MeshWallet. You can explore the wallet APIs here.

```
const walletPassphrase = process.env.WALLET_PASSPHRASE_ONE;
if (!walletPassphrase) {
    throw new Error("WALLET_PASSPHRASE_ONE does not exist");
}

const wallet = new MeshWallet({
    networkId: 0,
    fetcher: blockchainProvider,
    submitter: blockchainProvider,
    key: {
        type: "mnemonic",
        words: walletPassphrase.split(' ')
    },
});
```

MeshWallet accepts mnemonic, private key, cli keys, or an address (read-only) for its `key` field. In the code above, a mnemonic from the environment variable is used. Note that mnemonic required here is an array of words.

Mesh can be used to mimic the simple send transaction that a traditional software wallet application handles. This is how it's done:

```
const receiverAddress = "addr_test1...";

const unsignedTx = await txBuilder
  // send 50 ADA
  .txOut(receiverAddress, [ { unit: "lovelace", quantity: "50000000"
} ])
  .changeAddress(walletAddress)
  .selectUtxosFrom(walletUtxos)
  .complete()

const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);

console.log(`txHash ${txHash}`);
```

The code above sends 50 ADA to `receiverAddress`. It uses UTxOs from `wallet` to fill in the request. The remaining changes of the UTxOs returns to `walletAddress`. `wallet` signs and submits the transaction to the blockchain. The transaction hash is logged to the console.

Don't know what `txBuilder` is? Don't worry as it will be explained in the next section.

4.2. Creating and submitting transactions

This section will be a walkthrough on the entire processes required to create and submit a transaction with Mesh JS.

First step is to set up the `txBuilder`:

```
// Create transaction builder
const txBuilder = new MeshTxBuilder({
    fetcher: blockchainProvider,
    submitter: blockchainProvider,
});

txBuilder.setNetwork('preview');
```

The `txBuilder` is initialized using the `MeshTxBuilder` class. We can see that both the `fetcher` and `submitter` fields are set to `blockchainProvider`.

The txBuilder network is set to `preview` (the network we want to execute our transactions in).

What is the blockchainProvider? We will see that **next**:

```
// Setup blockchain provider as Maestro
const maestroKey = process.env.MAESTRO_KEY;
if (!maestroKey) {
    throw new Error("MAESTRO_KEY does not exist");
}
const blockchainProvider = new MaestroProvider({
    network: 'Preview',
    apiKey: maestroKey,
});
```

A blockchain provider allows you to query and submit transactions to the cardano blockchain without the need of having a local copy of the blockchain (Cardano node).

In the code above, we used one of Cardano's several blockchain providers, Maestro,

setting the network to “Preview”.

We need a wallet to sign and submit transactions to the Cardano blockchain. That’s where the previous section comes in. Here is the code that was used:

```
const walletPassphrase = process.env.WALLET_PASSPHRASE_ONE;
if (!walletPassphrase) {
    throw new Error("WALLET_PASSPHRASE_ONE does not exist");
}

const wallet = new MeshWallet({
    networkId: 0,
    fetcher: blockchainProvider,
    submitter: blockchainProvider,
    key: {
        type: "mnemonic",
        words: walletPassphrase.split(' ')
    },
});
```

Here are some wallet related functions:

```
// Returns the wallet's address
const walletAddress = await wallet.getChangeAddress();

// Returns a list of the UTxOs residing on the wallet
const walletUtxos = await wallet.getUtxos();

// Returns a list of the collateral UTxOs residing on the wallet
const walletCollateral: UTxO = (await wallet.getCollateral())[0]
if (!walletCollateral) {
    throw new Error('No collateral utxo found');
}
```

At this stage, we have everything set up to create and submit transactions: the blockchain provider, tx (transaction) builder, and wallet.

Now, we can gracefully execute this transaction:

```
const receiverAddress = "addr_test1...";

const unsignedTx = await txBuilder
  // send 50 ADA
  .txOut(receiverAddress, [ { unit: "lovelace", quantity: "50000000"
} ])
  .changeAddress(walletAddress)
  .selectUtxosFrom(walletUtxos)
  .complete()

const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);

console.log(`txHash ${txHash}`);
```

Good job coming this far!
In the next sections, we shall see how mesh is used to create transactions with smart contracts.

4.3. Locking and unlocking funds in vesting contract

4.3.1. Locking funds

We are going to see how funds are locked to a smart contract using mesh JS. The smart contract referenced in sub-section `4.0.1` is going to be utilized.

A datum is going to be created first:

```
const lockDatum = mConStr0([
    new Date().getTime() + (10 * 60 * 1000), // 10 mins
    walletVK,
    wallet2VK,
]);
```

For demonstration purposes, the funds are going to be locked for just 10 minutes. Note above that wallet2 VK (verification key hash) is used as the beneficiary.

Then a UTxO of 100 ADA is sent together with the datum (`lockDatum`) to the vesting smart contract address; in short, locking:

```
const unsignedTx = await txBuilder
  // lock 100 ADA
  .txOut(vestingAddress, [ { unit: "lovelace", quantity: "100000000"
} ])
  .txOutInlineDatumValue(lockDatum)
  .changeAddress(walletAddress)
  .selectUtxosFrom(walletUtxos)
  .complete()

const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);

console.log(`txHash: ${txHash}`);
```

The vesting address is created like so:

```
const vestingAddress = serializePlutusScript(
    { code: vestingValidatorScript, version: "V3" },
    undefined,
    0,
).address;
```

4.0.2. Unlocking funds

To unlock funds, we first identify the UTxO on which the funds we are willing to unlock is on:

```
const vestingUtxo = (await
blockchainProvider.fetchUTxOs("63ea9e734c42109f1fdad04739565a0becda94
a8e5af2958c94b70b298b54196"))[0];
```

The smart contract needs the current time to compare against the vesting lock time. It is provided in the transaction through this:

```
const invalidBefore = unixTimeToEnclosingSlot(
    (Date.now() - 35000),
    SLOT_CONFIG_NETWORK.preview
)
```

It's time to see the whole unlocking transaction:

```
const unsignedTx = await txBuilder
  .spendingPlutusScriptV3()
  .txIn(
      vestingUtxo.input.txHash,
      vestingUtxo.input.outputIndex,
      vestingUtxo.output.amount,
      vestingUtxo.output.address,
  )
```

```
  .txInScript(vestingValidatorScript)
  .spendingReferenceTxInInlineDatumPresent()
  .spendingReferenceTxInRedeemerValue("")
  .txOut(wallet2Address, vestingUtxo.output.amount)
  .changeAddress(wallet2Address)
  .selectUtxosFrom(wallet2Utxos)
  .txInCollateral(
    wallet2Collateral.input.txHash,
    wallet2Collateral.input.outputIndex,
    wallet2Collateral.output.amount,
    wallet2Collateral.output.address,
  )
  .invalidBefore(invalidBefore)
  .requiredSignerHash(wallet2VK)
  .complete()

const signedTx = await wallet2.signTx(unsignedTx);
const txHash = await wallet2.submitTx(signedTx);

console.log(`txHash ${txHash}`);
```

Going through the unlocking transaction:

- The vesting UTxO is spent using the vesting smart_contract/validator
- All vested ADA is sent to the beneficiary
- A collateral UTxO is required as a smart contract is involved in the transaction
- The beneficiary signs and submits the transaction

4.4. Minting tokens to wallet

Minting and burning is going to be demonstrated using the smart contract below

```
use aiken/collection/dict
use cardano/transaction.{Transaction}
use cardano/assets.{PolicyId}

pub type Action {
  Minting
  Burning
}

validator mint_nft() {
  mint(redeemer: Action, policy_id: PolicyId, self: Transaction) {
      // It checks that only one minted asset exists and will fail
otherwise
      expect [Pair(_asset_name, quantity)] = self.mint
        |> assets.tokens(policy_id)
        |> dict.to_pairs()

      when redeemer is {
        Minting -> (quantity == 1)?

        Burning -> (quantity == -1)?
      }
    }
  }
```

Minting:

```
const assetNameHex = stringToHex("exNFT");

const unsignedTx = await txBuilder
  .mintPlutusScriptV3()
  .mint("1", mintingPolicy, assetNameHex)
  .mintingScript(mintingValidatorScript)
  .mintRedeemerValue(mConStr0([]))
  .txOut(walletAddress, [ { unit: mintingPolicy + assetNameHex,
```

```
quantity: "1" } ])
  .changeAddress(walletAddress)
  .selectUtxosFrom(walletUtxos)
  .txInCollateral(
    walletCollateral.input.txHash,
    walletCollateral.input.outputIndex,
    walletCollateral.output.amount,
    walletCollateral.output.address,
  )
  .complete()

const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);

console.log(`txHash: ${txHash}`);
```

The Mesh Typescript code above mints an NFT and sends it to `walletAddress`.

Another variant is the **Burning** with just minor changes in the above code:

```
const assetNameHex = stringToHex("exNFT");

const unsignedTx = await txBuilder
  .mintPlutusScriptV3()
  .mint("-1", mintingPolicy, assetNameHex)
  .mintingScript(mintingValidatorScript)
  .mintRedeemerValue(mConStr1([]))
  .changeAddress(walletAddress)
  .selectUtxosFrom(walletUtxos)
  .txInCollateral(
    walletCollateral.input.txHash,
    walletCollateral.input.outputIndex,
    walletCollateral.output.amount,
    walletCollateral.output.address,
  )
  .complete()

const signedTx = await wallet.signTx(unsignedTx);
const txHash = await wallet.submitTx(signedTx);
```

```
console.log(`txHash: ${txHash}`);
```

The changes are:

- Mint is specified with “-1” instead of “1”
- The redeemer is `mConStr1([])` instead of `mConStr0([])`

Exercise

Using Mesh JS, interact with the one-shot minting policy. Mint and Burn an NFT.

Here is the one-shot minting policy as a reference:

```
use aiken/collection/dict
use aiken/collection/list
use cardano/transaction.{OutputReference, Transaction}
use cardano/assets.{PolicyId}

pub type Action {
  Mint
  Burn
}

validator one_shot(utxo_ref: OutputReference) {
  mint(redeemer: Action, policy_id: PolicyId, self: Transaction) {
    let Transaction {inputs, mint ..} = self

    // It checks that only one minted asset exists and will fail
otherwise
    expect [Pair(_asset_name, quantity)] = mint
      |> assets.tokens(policy_id)
      |> dict.to_pairs()

    // Check if the specified UTxO reference (utxo_ref) is consumed
by any input
    let is_utxo_spent = list.any(inputs, fn(input) {
input.output_reference == utxo_ref })

    when redeemer is {
      Mint ->
        is_utxo_spent? && (quantity == 1)?

      Burn -> (quantity == -1)? // No need to check if output is
consumed for burning
    }
```

```
    }
}
```

Good luck!