Lab 8.1: Balance Transfer Chaincode

In this exercise, we will implement a new chaincode to demonstrate the possibilities of identity-based access control via the `BalanceTransfer` chaincode. The main purpose of the chaincode is to manage different accounts granting full permissions to modify your own accounts, but denying all attempts to access an account that does not belong to you.

To keep it simple, the `BalanceTransfer` chaincode will implement the following methods:

1) `initAccount(id, balance)` to create a new account with a specified ID and balance owned by a transaction creator.
2) `setBalance(id, newBalance)` to set the balance of the specified account to `newBalance`. The transaction creator must be the owner of the account.
3) `transfer(idFrom, idTo, amount)` to transfer `amount` from the `idFrom` account to the `idTo` account. The transaction creator must be the owner of the `idFrom` account.
4) `listAccounts()` to list all the accounts belonging to the transaction creator.

Let's create a folder structure for the `BalanceTransfer` chaincode. In a chaincode folder, make a `lib` directory and put inside the `balanceTransfer.js` file containing a contract skeleton.

```
'use strict';

const { Contract } = require('fabric-contract-api');

class BalanceTransfer extends Contract {
   async initAccount(ctx, id, balance) {}
   async setBalance(ctx, id, newBalance) {}
   async transfer(ctx, idFrom, idTo, amount) {}
   async listAccounts (ctx) {}
```

```
module.exports = BalanceTransfer;
```

In a chaincode folder, create an `index.js` file exporting the `BalanceTransfer` contract.

```
'use strict';

const balanceTransfer = require('./lib/balanceTransfer');

module.exports.BalanceTransfer = balanceTransfer;
module.exports.contracts = [balanceTransfer];
```

As you may have noticed, our ledger state is an account that can be identified by the ID and has two additional properties: an account owner and a balance. Before proceeding to the `BalanceTransfer` business logic implementation, let's introduce several helper functions to simplify accounts processing.

We are going to identify the account by ID. It means that our composite key should contain the account ID along with an object type. Knowing this, we can implement helpers to check account existence, get and put account details to the ledger.

```
const accountObjType = "Account";

class BalanceTransfer extends Contract {

   // <...>

   async _accountExists(ctx, id) {
       const compositeKey = ctx.stub.createCompositeKey(accountObjType,
[id]);
       const accountBytes = await ctx.stub.getState(compositeKey);
       return accountBytes && accountBytes.length > 0;
   }

   async _getAccount(ctx, id) {
       const compositeKey = ctx.stub.createCompositeKey(accountObjType,
[id]);

       const accountBytes = await ctx.stub.getState(compositeKey);
       if (!accountBytes || accountBytes.length === 0) {
           throw new Error(`the account ${id} does not exist`);
       }
```

```
        return JSON.parse(accountBytes.toString());
    }

    async _putAccount(ctx, account) {
        const compositeKey = ctx.stub.createCompositeKey(accountObjType,
[account.id]);
        await ctx.stub.putState(compositeKey,
Buffer.from(JSON.stringify(account)));
    }
}
```

From the specification of the business logic, we can see that in each function, we should extract identifying information about the transaction creator. Moreover, it's essential that identifying information stored in the account details should be unique across network users. To meet this requirement, we can use a combination of `ClientIdentity.getMSPID()` and `ClientIdentity.getID()` that gives us a unique user identifier within the blockchain network. The `_getTxCreatorUID` helper function is an example of how the IDs can be combined.

```
_getTxCreatorUID(ctx) {
    return JSON.stringify({
        mspid: ctx.clientIdentity.getMSPID(),
        id: ctx.clientIdentity.getID()
    });
}
```

Now, we can easily implement all chaincode functions. In `initAccount`, we should ensure that the balance passed is not negative and the account ID is not in use. If both checks pass, a new account can be created.

```
async initAccount(ctx, id, balance) {
    const accountBalance = parseFloat(balance);
    if (accountBalance < 0) {
        throw new Error(`account balance cannot be negative`);
    }

    const account = {
        id: id,
        owner: this._getTxCreatorUID(ctx),
        balance: accountBalance
    }

    if (await this._accountExists(ctx, account.id)) {
```

```
        throw new Error(`the account ${account.id} already exists`);
    }

    await this._putAccount(ctx, account);
}
```

There are three more functions left to complete the implementation of the `BalanceTransfer` chaincode. We leave this as an exercise, so you can finalize the `BalanceTransfer` chaincode on your own. The file with our version of the `BalanceTransfer` chaincode implementation is attached (you can check it out and compare with your implementation).

Now, it is time to test our chaincode in action! We'll start `test-network` from the `fabric-samples` repository, but you can use any locally deployed Hyperledger Fabric network. For the complete chaincode life cycle description, see the **The Chaincode Life Cycle** chapter.

Assuming we have running `test-network`, let's test our chaincode:

1) Create the `package.json` file in the chaincode folder.

```
{
   "name": "balance_transfer",
   "version": "1.0.0",
   "main": "index.js",
   "scripts": {
       "start": "fabric-chaincode-node start"
   },
   "engines": {
       "node": ">=12",
       "npm": ">=5"
   },
   "dependencies": {
       "fabric-contract-api": "^2.0.0",
       "fabric-shim": "^2.0.0"
   }
}
```

2) Navigate to the `test-network` folder and deploy the `BalanceTransfer` chaincode.

```
# cd $HOME/go/src/github.com/hyperledger/fabric-samples/test-network
# ./network.sh deployCC -ccn balance_transfer -ccv 1.0 -ccp
../lfd272/chaincodes/balance_transfer -ccl javascript
```

3) Set all necessary environmental variables to run the `peer` binary on behalf of `Org1MSP Admin`.

```
# export PATH=${PWD}/../bin:$PATH
# export FABRIC_CFG_PATH=$PWD/../config/

# export CORE_PEER_TLS_ENABLED=true
# export CORE_PEER_LOCALMSPID="Org1MSP"
# export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1
.example.com/peers/peer0.org1.example.com/tls/ca.crt
# export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.exa
mple.com/users/Admin@org1.example.com/msp
# export CORE_PEER_ADDRESS=localhost:7051
```

Note: There are two predefined users for each organization in `test-network`: `Admin` and `User1`. We can change `CORE_PEER_MSPCONFIGPATH` and `CORE_PEER_LOCALMSPID` environmental variables to switch between users.

4) Initialize an account for `Admin` and test the `setBalance` function. To make invocation commands more straightforward, we are going to introduce a couple of additional environmental variables.

```
# export orderer="-o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.
example.com/msp/tlscacerts/tlsca.example.com-cert.pem"

# export peer_org1="--peerAddresses localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org
1.example.com/tls/ca.crt"

# export peer_org2="--peerAddresses localhost:9051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org
2.example.com/tls/ca.crt"

# peer chaincode invoke ${orderer} -C mychannel -n balance_transfer
${peer_org1} ${peer_org2} -c '{"function":"initAccount","Args":["A1",
"100"]}'
```



```
# peer chaincode invoke ${orderer} -C mychannel -n balance_transfer
${peer_org1} ${peer_org2} -c '{"function":"setBalance","Args":["A1",
"150"]}'
```

5) Check the A1 account existence and balance.

```
# peer chaincode query -C mychannel -n balance_transfer -c
'{"function":"listAccounts", "Args":[]}'
```

The expected output is a JSON-array containing one account.

```
{
      "id": "A1",
      "owner": "{\"mspid\":\"Org1MSP\",\"id\":\"x509::/C=US/ST=North
Carolina/O=Hyperledger/OU=admin/CN=org1admin::/C=US/ST=North
Carolina/L=Durham/O=org1.example.com/CN=ca.org1.example.com\"}",
      "balance": 150
}
```

6) Switch to User1 from Org1MSP, initialize the U1 account on the user's behalf, and check its existence and balance.

```
# export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.exa
mple.com/users/User1@org1.example.com/msp

# peer chaincode invoke ${orderer} -C mychannel -n balance_transfer
${peer_org1} ${peer_org2} -c '{"function":"initAccount","Args":["U1",
"200"]}'
# peer chaincode query -C mychannel -n balance_transfer -c
'{"function":"listAccounts", "Args":[]}'
```

The expected output is a JSON array containing a single account.

```
{
      "id": "U1",
      "owner": "{\"mspid\":\"Org1MSP\",\"id\":\"x509::/C=US/ST=North
Carolina/O=Hyperledger/OU=client/CN=user1::/C=US/ST=North
Carolina/L=Durham/O=org1.example.com/CN=ca.org1.example.com\"}",
      "balance": 200
}
```

7) Transfer 100 units from U1 to A1 on behalf of User1 and display the U1 account again.

```
# peer chaincode invoke ${orderer} -C mychannel -n balance_transfer
${peer_org1} ${peer_org2} -c '{"function":"transfer","Args":["U1",
"A1", "100"]}'
# peer chaincode query -C mychannel -n balance_transfer -c
'{"function":"listAccounts", "Args":[]}'
```

The expected output is a JSON array, containing a single account.

```
{
      "id": "U1",
      "owner": "{\"mspid\":\"Org1MSP\",\"id\":\"x509::/C=US/ST=North
Carolina/O=Hyperledger/OU=client/CN=user1::/C=US/ST=North
Carolina/L=Durham/O=org1.example.com/CN=ca.org1.example.com\"}",
      "balance": 100
}
```

8) Try to transfer 100 units back from A1 to U1, still on behalf of User1.

```
# peer chaincode invoke ${orderer} -C mychannel -n balance_transfer
${peer_org1} ${peer_org2} -c '{"function":"transfer","Args":["A1",
"U1", "100"]}'
```

We should receive an error message about an unauthorized access, because only Admin can transfer values from the A1 account.

9) Switch back to Admin and check the A1 balance—it should be equal to 250.

```
# export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.exa
mple.com/users/Admin@org1.example.com/msp
# peer chaincode query -C mychannel -n balance_transfer -c
'{"function":"listAccounts", "Args":[]}'
```