Analyzing a Chaincode

This tutorial analyzes the implementation of the blue coin chaincode.

Quick Setup

1. Run the quick setup.

```
chaincode> ./quick-setup.sh blue-coin-no-acl blue-coin 1.0
```

Analyze the chaincode

1. Open the following source in an IDE.

```
IDE> blockchain-tutorial/chaincode/blue-coin-no-acl/lib/BlueCoinContract.js
```

2. The NodeJS SDK `fabric-contract-api` is used.

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

3. A utility class called `Utility.js` is also used.

```
const Utility = require("./Utility.js");
```

   The source code of this class is found in the same folder as `BlueCoinContract.js`.

   This utility class is used to abstract some of the complexities of frequently called codes.

4. Using the `Contract` as a superclass, a `BlueCoinContract` subclass is created.

```
class BlueCoinContract extends Contract {
```

5. An `init` function is used for any world state initialization that needs to be done.

```
async init(ctx){
```

   The `init` function checks first if the function is executed already before it proceeds with running the rest of the function.

The `init` function is meant to be executed only once in any chaincode.

For the blue coin chaincode, there are no world state initialization that is needed.

6. A `generateInitialCoin` function is used to give an organization an initial set of blue coins.

```
async generateInitialCoin(ctx, mspId) {
```

The function accepts as a parameter an `mspId` which corresponds to the `mspId` of an organization (e.g., `Org1MSP` or `Org2MSP`).

To ensure the organization `mspId` cannot call this function several times, the existence of blue coins for the `mspId` is checked first.

```
let json = await Utility.getState(ctx, mspId);

//check if organization has already generated blue coins before
if (json != null)
  throw new Error("Organization " + mspId + " has generated already initial
blue coins before.  Organization can generate blue coins only once.")
```

The information related to the blue coins of the `mspId` is saved in the world state.

```
json = {
  mspId: mspId,
  amt: 500
}

await Utility.putState(ctx, mspId, json);
```

7. A `getBalance` function is used to query the remaining blue coins of a particular organization.

```
async getBalance(ctx, mspId) {
```

8. A `transferCoin` function is used to transfer a specified amount of blue coins from one organization to another.

```
async transferCoin(ctx, srcMspId, dstMspId, amount){
```

This function checks first if both the `srcMspId` and `dstMspId` have records (i.e., called the `generateInitialCoin`) in the world state.

In addition, it checks if the blue coins of `srcMspId` is sufficient to transfer the specified `amount`.

If the conditions above are satisfied, the blue coins of `srcMspId` and `dstMspId` are updated and saved.

```
  srcBCOINJson.amt -= parseInt(amount);
  dstBCOINJson.amt += parseInt(amount);

  await Utility.putState(ctx, srcMspId, srcBCOINJson);
  await Utility.putState(ctx, dstMspId, dstBCOINJson);
```

9. A `getAllAbove` function is used to get all blue coins that go are above a particular value.

```
async getAllAbove(ctx, val) {
```

This function is ONLY created to demonstrate the use of rich queries.

```
const jsonQuery = {
  "selector": {
    "amt": {"$gt": parseInt(val)}
  }
}
```

Rich queries are used to directly query the contents of couchdb.

10. A `getTransactionHistory` function is used to get all transactions related to a particular organization.

```
async getTransactionHistory(ctx, mspId){
```

This function can be used to trace the provenance of all the blue coins received by `mspId`.

Generate Initial Blue Coins

1. Query the balance of org1.

```
chaincode> docker exec cli0.org1 peer chaincode query \
  -C mychannel -n blue-coin \
-c '{"function":"getBalance","Args":["Org1MSP"]}'
```

Notice that the function returned a `status` of `500` to denote that the query is not successful. Since org1 has not invoked the `generateInitialCoin` function, org1 has no blue coins yet.

2. Generate initial coins for org1.

```
chaincode> docker exec cli0.org1 peer chaincode invoke \
  -o orderer.example.com:7050 \
  -C mychannel -n blue-coin \
  -c '{"function":"generateInitialCoin","Args":["Org1MSP"]}'
```

3. Query again the balance of org1.

```
chaincode> docker exec cli0.org1 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getBalance","Args":["Org1MSP"]}'
```

This time it shows that org1 has a balance of 500 blue coins.

4. Try to genenerate again initial coins for org1.

```
chaincode> docker exec cli0.org1 peer chaincode invoke \
  -o orderer.example.com:7050 \
  -C mychannel -n blue-coin \
  -c '{"function":"generateInitialCoin","Args":["Org1MSP"]}'
```

Notice that the function returned a `status` of `500` to denote that org1 cannot generate initial coins more than once.

5. Generate initial coins for org2.

```
chaincode> docker exec cli0.org2 peer chaincode invoke \
  -o orderer.example.com:7050 \
  -C mychannel -n blue-coin \
  -c '{"function":"generateInitialCoin","Args":["Org2MSP"]}'
```

6. Query the balance of org2.

```
chaincode> docker exec cli0.org2 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getBalance","Args":["Org2MSP"]}'
```

View Logs

1. View the logs generated by the chaincode of peer0.org1.example.com

```
chaincode> ./view-chaincode-logs.sh 1 0 blue-coin 1.0
```

The `view-chaincode-logs.sh` script accepts two parameters:

- `org index` - index of organization
- `peer index` - index of peer
- `chaincode name` - name of the chaincode
- `chaincode version` - version of the chaincode

Transfer Blue Coins Between Organizations

1. Transfer 150 blue coins from org1 to org2.

```
chaincode> docker exec cli0.org1 peer chaincode invoke \
  -o orderer.example.com:7050 \
  -C mychannel -n blue-coin \
  -c '{"function":"transferCoin","Args":["Org1MSP", "Org2MSP", "150"]}'
```

2. Query the balance of org1 and org2.

```
chaincode> docker exec cli0.org1 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getBalance","Args":["Org1MSP"]}'

chaincode> docker exec cli0.org2 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getBalance","Args":["Org2MSP"]}'
```

Confirm that org1 now has only 350 while org2 has 650 blue coins.

List the Transaction History

1. List the transaction history related to the blue coins of org1.

```
chaincode> docker exec cli0.org1 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getTransactionHistory","Args":["Org1MSP"]}'
```

Notice that the payload has two transaction IDs (`TxId`'s). These refer to the two transactions related to the blue coins of org1:

- `generateInitialCoin` - that generated the initial 500 blue coins of org1
- `transferCoin` - that transferred 150 blue coins from org1 to org2, giving a balance of 350 blue coins to org1.

2. List the transaction history related to the blue coins of org2.

```
chaincode> docker exec cli0.org2 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getTransactionHistory","Args":["Org2MSP"]}'
```

Similar to org1, the payload has two transaction IDs (`TxId`'s). These refer to the two transactions related to the blue coins of org2:

- `generateInitialCoin` - that generated the initial 500 blue coins of org2
- `transferCoin` - that transferred 150 blue coins from org1 to org2, giving a balance of 650 blue coins to org2.

Notice that there is a `TxId` in the transaction history of org1 that matches the `TxId` of org2. This refers to the same transaction `transferCoin` that transferred 150 blue coins from org1 to org2.

Perform a Rich Query

1. List all organizations with more than 400 blue coins.

   ```
   chaincode> docker exec cli0.org2 peer chaincode query \
     -C mychannel -n blue-coin \
     -c '{"function":"getAllAbove","Args":["400"]}'
   ```

   Notice that the payload is only an array with one element since only org2 has a balance of more than 400 coins.

2. List all organizations with more than 100 blue coins.

   ```
   chaincode> docker exec cli0.org2 peer chaincode query \
     -C mychannel -n blue-coin \
     -c '{"function":"getAllAbove","Args":["100"]}'
   ```

   This time the payload is an array with two elements since both org1 and org2 have more than 100 blue coins.

Use Org2 to Transfer Blue Coins of Org1

The use of the functions of the chaincode is demonstrated in the previous section.

In the previous steps, the CLI of org1 is used to generate initial blue coins of org1. Similarly, the CLI of org2 is used to generate the initial blue coins of org2.

When it comes to the transfering of blue coins, the CLI of org1 is used to transfer blue coins from org1 to org2.

However, since the organization MSP IDs are passed as parameter in the functions, we can technically use org2 to transfer the blue coins from org1 to org2.

1. Transfer 300 blue coins from org1 to org2 using the CLI of peer0.org2.example.com.

```
chaincode> docker exec cli0.org2 peer chaincode invoke \
  -o orderer.example.com:7050 \
  -C mychannel -n blue-coin \
  -c '{"function":"transferCoin","Args":["Org1MSP", "Org2MSP", "300"]}'
```

2. Query the balance of org1 and org2 using the CLI of peer0.org2.example.com.

```
chaincode> docker exec cli0.org2 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getBalance","Args":["Org1MSP"]}'

chaincode> docker exec cli0.org2 peer chaincode query \
  -C mychannel -n blue-coin \
  -c '{"function":"getBalance","Args":["Org2MSP"]}'
```

   Confirm that org1 now has only 50 while org2 has 950 blue coins.

   This demonstates that even if you are not the owner of a particular set of blue coins you can transfer these blue coins to another owner.

   In an actual deployment, this is a very undesirable limitation.

Limit the Generation of Initial Blue Coins and Transfer of Blue Coins to the Owning Organization

1. Open the following source in an IDE.

```
IDE> blockchain-tutorial/chaincode/blue-coin-no-acl/lib/Utility.js
```

2. Take a look at the two functions related to MSP IDs.

```
static getMspId(ctx){
    const cid = new ClientIdentity(ctx.stub);
    return cid.getMSPID();
}

static assertMspId(ctx, mspId){
    const cid = new ClientIdentity(ctx.stub);
    return mspId == cid.getMSPID();
}
```

   The `getMspId` function returns the MSP ID of the calling organization.

For example, if `getMspId` is called inside `generateInitialCoin` and the latter is called using the CLI of peer0.org1.example.com, the `getMspId` function will return `Org1MSP`.

The `assertMspId` function has a similar code to `getMspId`. However, it does not return an MSP ID. It returns `true` if the MSP ID of the calling organization matches the parameter `mspId`, otherwise it returns `false`.

3. Open the following source in an IDE.

```
IDE> blockchain-tutorial/chaincode/blue-coin-no-acl/lib/BlueCoinContract.js
```

Modify `generateInitialCoin` such that it can only be called by a particular organization if the call to the function is meant to generate initial blue coins for said organization.

Similarly, modify `transferCoin` such that the organization that calls the function is the owner of the blue coins that will be transferred.

4. Test the updated chaincode by running the `quick-setup.sh` script and invoking the `generateInitialCoin` and `transferCoin` functions.