Lab 6.1: Chaincode for Report Generation

Let’s now apply our knowledge about rich queries and indexes to implement `ReportsContract`. Assuming the ledger is a registry of purchases and sales, we can write chaincode logic for generating annual and custom reports. Having this in mind, let’s introduce the following three functions in `ReportsContract`:

1. The `putRecord(id, [record arguments])` function creates a new record with the specified ID and the arguments that make up a record (e.g., a date).
2. The `getAnnualReport(year)` function collects all the records for the specified year.
3. The `generateCustomReport(queryString)` function generates any custom report based on a given query string.

Let’s start the implementation with the definition of a record and clarification of what `[record arguments]` can stand for.

Let’s assume that a record in the ledger comprises a unique identifier, name of the goods, price (positive if the goods were sold and negative otherwise), and transaction date. Since we plan to generate an annual report that uses the transaction date of the records, we can store the date as a JSON object with separate fields representing a day, a month, and a year.

Considering all the above, let’s define the record object structure.

```
// Record
{
    id:    string;
    name:  string;
    price: number;
    date: {
        day:   number;
        month: number;
        year:  number;
    }
}
```

Since we have defined the record structure, we can now clarify what `[record arguments]` stands for by providing the full signature for the `putRecord` function.

```
async putRecord(ctx, id, name, price, dateString)
```

The arguments of `putRecord` contain the ID and the remaining record elements, such as a name, a price, and a string representation of a date that can be parsed into a corresponding JSON.

Now, we have an understanding of the ledger record structure and required methods signatures. That allows us to start the implementation of the chaincode itself. For this purpose, we should create the `reportsContract.js` file and put the skeleton of our chaincode there.

```
'use strict';

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

class ReportsContract extends Contract {
    async putRecord(ctx, id, name, price, dateString) {
    }

    async getAnnualReport(ctx, year) {
    }

    async generateCustomReport(ctx, queryString) {
    }
}

module.exports = ReportsContract;
```

Let's start with the implementation of `putRecord`. First, we need to parse the input parameters to compose the ledger record.

```
const date = new Date(dateString);

const record = {
    id: id,
    name: name,
    price: parseFloat(price),
    date: {
        day: date.getDate(),
        month: date.getMonth() + 1,
        year: date.getFullYear()
    }
};
```

Then, we should check if the record ID is unique to avoid data rewriting. We are going to identify the record simply by ID, so our composite key should contain the `id` field along with an object type.

```
const compositeKey = ctx.stub.createCompositeKey(recordObjType, [id]);

const recordBytes = ctx.stub.getState(compositeKey);
if (recordBytes && recordBytes.length > 0) {
    throw new Error(`The record ${record.id} already exists`);
}
```

The `recordObjType` constant can be defined right before `ReportsContract`.

```
const recordObjType = "Record";

class ReportsContract extends Contract {
   <...>
}
```

Finally, we are going to save the record in the ledger. In order to run rich queries against the records, we will store a JSON-encoded record as a value associated with its composite key.

```
await ctx.stub.putState(compositeKey, Buffer.from(JSON.stringify(record)));
```

There are two more functions to implement, and both of them have common functionality—they perform a rich query using a query string. That is why we can extract this piece of functionality into a separate `_getResultsForQueryString` function.

```
async _getResultsForQueryString(ctx, queryString) {
    const iteratorPromise = ctx.stub.getQueryResult(queryString);

    let results = [];
    for await (const res of iteratorPromise) {
        results.push(JSON.parse(res.value.toString()));
    }

    return JSON.stringify(results);
}
```

This function is very similar to the query functions that we have implemented previously in `SimpleContract`. The main difference is that `_getResultsForQueryString` uses the `getQueryResult` function (instead of `getStateByPartialCompositeKey` or `getStateByRange`) to perform a rich query. The iterator returned by `getQueryResult` has the `StateQueryIterator` type, which we have already discussed in the **Ledger Data Range Queries** chapter, so processing the results follows the same pattern.

Before implementing the `getAnnualReport` function, let's create an index for our query. Indexes should be packaged alongside chaincode in the `META-INF/statedb/couchdb/indexes` directory. Any indexes in this directory will be packaged up with the chaincode for deployment. Create the `indexYear.json` file and put an index definition inside.

```
{
  "index": {
    "fields": [
      "date.year"
    ]
  },
  "ddoc": "indexYearDoc",
  "name": "indexYear",
  "type": "json"
}
```

Now, we have everything to complete `ReportsContract`.

```
async getAnnualReport(ctx, year) {
    const queryString = `{"selector":{"date":{"year":${year}}},
"use_index":["_design/indexYearDoc", "indexYear"]}`;
    return this._getResultsForQueryString(ctx, queryString);
}

async generateCustomReport(ctx, queryString) {
    return this._getResultsForQueryString(ctx, queryString);
}
```

Finally, it is time to test our chaincode in action. To simplify the testing process, we will create the `init` function to populate the ledger with the initial data.

```
async init(ctx) {
    const records = [
        {
            id: "id1",
            name: "goods1",
            price: 100.0,
            date: {
                day: 1,
                month: 1,
                year: 2018
            }
        },
        {
            id: "id2",
            name: "goods2",
            price: -90.0,
            date: {
                day: 12,
                month: 2,
                year: 2019
            }
        },
        {
            id: "id3",
            name: "goods3",
            price: 75.0,
            date: {
                day: 27,
                month: 5,
                year: 2020
            }
        }
    ];

    for (const record of records) {
        const compositeKey = ctx.stub.createCompositeKey(recordObjType,
[record.id]);
        await ctx.stub.putState(compositeKey,
Buffer.from(JSON.stringify(record)));
```

```
    }
}
```

The final version of `ReportsContract` is attached.

Next, 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 **Chaincode Life Cycle** chapter.

Assuming we have a running `test-network`, we can test our chaincode.

1) Go to the `fabric-samples/test-network` folder and deploy `ReportsChaincode` to the `mychannel` channel using the `network.sh` script. Don't forget to specify the initialization function.

```
# ./network.sh deployCC -ccn reports -ccl javascript -ccp
~/go/src/github.com/hyperledger/fabric-samples/lfd272/chaincodes/report
s_chaincode -ccv 1.0 -ccs 1 -cci init
```

2) Set all necessary environment variables to run the `peer` binary on behalf of `Org1MSP`.

```
# 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
```

3) Generate the annual report for 2019.

```
# peer chaincode invoke -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 -C mychannel -n
reports --peerAddresses localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org
```

```
1.example.com/tls/ca.crt --peerAddresses localhost:9051
--tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org
2.example.com/tls/ca.crt -c
'{"function":"getAnnualReport","Args":["2019"]}'
```

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

```
{
  "id": "id2",
  "name": "goods2",
  "price": -90,
  "date": {
    "day": 12,
    "month": 2,
    "year": 2019
  }
}
```

4) Generate a custom report with any query string. For example, the following command generates a report containing all the records with a negative price.

```
# peer chaincode query -n reports -C mychannel -c
'{"function":"generateCustomReport", "Args":["{\"selector\":
{\"price\": {\"$lt\": 0}}}"]}'
```

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

```
{
  "id": "id2",
  "name": "goods2",
  "price": -90,
  "date": {
    "day": 12,
    "month": 2,
    "year": 2019
  }
}
```

As an exercise, try to add new records to the ledger and check out different queries.