howest
university of applied sciences

Bachelor of Applied Informatics @ home (Postgraduate)

Blockchain Developer & Architect

BLOCKCHAIN PROJECT MORTGAGE REGISTER

Karel De Smet
Thomas Ykman

06/01/2020

Business analysis

Problem

What problem do you want to solve?

Granting a Flemish housing bonus for existing mortgage loans is a slow process and an administrative burden for the government. Therefore, we want to replace the current mortgage register[1] by a register based on blockchain and smart contracts.

For which target group?

- *Banks*: want to register mortgages (= transaction) and check existing mortgages (e.g. to see whether a person has a mortgage);
- *Notaries*: must validate transactions and check existing mortgages (e.g. to see whether the sale of a house can go ahead, because it must be free of a mortgage upon sale);
- *Flemish government*: wants to check existing mortgages in the context of annual payment of regional housing bonus (does the applicant have a (first and only) home with a mortgage?);
- *Federal government*: wants to check existing mortgages in the context of annual payment federal housing bonus (does the applicant have a second home with a mortgage?).

Which solution do you propose?

- A person goes to a bank for a mortgage loan and records the details;
- The bank makes a transaction on the blockchain to register the mortgage;
- At the time of signature at the notary, 1 or more notaries validate the transaction and thereby confirm the accuracy of the data;
- Upon cancellation / modification of the mortgage, a new transaction is sent by the bank and validated by one or more notaries;
- On a monthly / annual basis, a smart contract is invoked that calculates the payment for all persons associated with a mortgage and makes a transaction for this (which serves as a payment order for the government).

How will blockchain technology help your solution?

- tamper-proof;
- speed (calculation of the subvention/payment is done automatically every month / year).

[1] https://www.kadaster.be/Kadaster/Hypotheekregister

What makes that users will use your solution and not something else?

- No trusted third party required;
- The principle of "openness" can be better guaranteed by decentralization;
- Cost for consultation may go down;
- Search is also possible via data other than personal name.

What effort will it cost to work out the solution?

- Develop a decentralized application (dApp) with front-end, back-end and blockchain components;
- Convince the different parties involved (banks, notaries, governments, borrowers) to use and trust this solution.
- Migrate the existing data of the mortgage register to the blockchain solution

POC: Functional analysis

To demonstrate the feasibility of the mortgage register described in the chapter Business analysis, we build a proof of concept (POC). This POC consists of:

- a blockchain network of 3 organisations (actors): Bank, Notary, Government;
- a digital mortgage register, i.e. a register of assets (mortgage loans) which can be created, read and updated by the organisations of the blockchain network.

In the current chapter, we describe the main features of this POC, i.e. what these organisations can do with the digital loan asset.

Organisations

The POC foresees three main actors (organisations):

ID	Organisation	Roles
Org1	Bank	2 endorsing peers, 1 certificate authority (CA)
Org2	Notary	2 endorsing peers, 1 CA
Org3	Government	2 endorsing peers, 1 CA

Besides, there are also 5 orderers (organisations with ordering service) on the blockchain network. In the future (see Next steps), we could add other organisations to this network, e.g. other banks, notaries, governments and buyers.

Asset

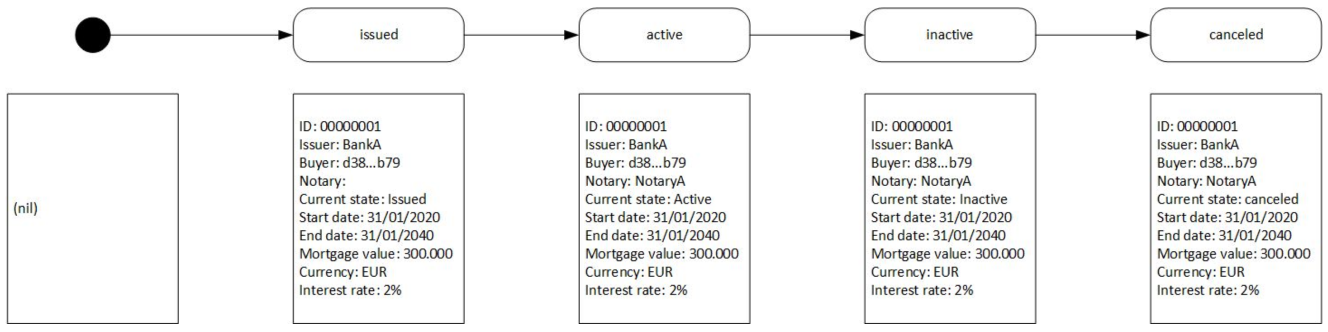
The digital asset represents a mortgage loan. It contains the following properties:

- *Loan UID*: a unique identifier of the loan, e.g. *loan1*;
- *Issuer*: the bank issuing the loan, i.e. *Org1*;
- *Buyer*: the name (or the hash of the national registry number) of the loan borrower who is buying a house, e.g. *Thomas*;
- *Notary*: the notary involved in the loan, i.e. *Org2*;
- *Status*: the current status of the loan, i.e. *issued*, *active*, *inactive*, or *cancelled*;
- *Start date*: the start date of the loan, e.g. *31/01/2020*;
- *End date*: the end date of the loan, e.g. *31/01/2040*;
- *Loan value*: expressed in a specific currency (see below), e.g. *300000*;
- *Currency*: the currency of the loan value, e.g. *EUR*;
- *Interest rate*: the interest rate of the loan (in percent), e.g. *2*.

The status of the loan can be changed in a predefined order by some actors. The table and the state transition diagram below illustrate this evolution of the loan status.

Step	Status	Who can set this status ?
1	issued	Bank
2	active	Notary
3	inactive	Bank or Notary
4	canceled	Notary

State transition diagram:



(nil)

ID: 00000001
Issuer: BankA
Buyer: d38...b79
Notary:
Current state: Issued
Start date: 31/01/2020
End date: 31/01/2040
Mortgage value: 300.000
Currency: EUR
Interest rate: 2%

ID: 00000001
Issuer: BankA
Buyer: d38...b79
Notary: NotaryA
Current state: Active
Start date: 31/01/2020
End date: 31/01/2040
Mortgage value: 300.000
Currency: EUR
Interest rate: 2%

ID: 00000001
Issuer: BankA
Buyer: d38...b79
Notary: NotaryA
Current state: Inactive
Start date: 31/01/2020
End date: 31/01/2040
Mortgage value: 300.000
Currency: EUR
Interest rate: 2%

ID: 00000001
Issuer: BankA
Buyer: d38...b79
Notary: NotaryA
Current state: canceled
Start date: 31/01/2020
End date: 31/01/2040
Mortgage value: 300.000
Currency: EUR
Interest rate: 2%

Use case

The POC aims to demonstrate the following use cases (UC):

- UC1: issuing a loan on the mortgage register and changing its status;
- UC2: reading the properties of this loan.

UC1: Issuing a loan and changing its status

Step	Actor	Action	On blockchain
1	Borrower (person who wants to buy a house)raft	Goes to Bank and asks for mortgage loan	nothing
2	Bank	Checks mortgage register and creates new asset	New mortgage loan asset is created, status = issued
3	Notary	Checks if the mortgage loan is correct (e.g. data from the Bank/ borrower) and activate the mortgage loan	mortgage loan status: issued → active
4	Bank or Notary	Bank or notary deactivates the loan when the loan is paid back	mortgage loan status: active →

		and/or when the term has expired	inactive
5	Notary	Cancels the loan	mortgage loan status: inactive → cancelled

UC2: Reading the properties of a loan

After the creation of a loan on the mortgage register, the organisations can read the following loan properties:

ID	Organisation	Readable properties of a loan
Org1	Bank	Loan UID, Issuer, Buyer, Notary, Status, Start date, End date; Loan value, Currency, Interest rate
Org2	Notary	Loan UID, Issuer, Buyer, Notary, Status, Start date, End date; Loan value, Currency, Interest rate
Org3	Government	Loan UID, Issuer, Buyer, Notary, Status, Start date, End date

For the current POC, we assume that only the Bank and Notary involved in the loan can see (read) the fields loan value, currency and interest rate. In reality, other fields might be accessible only by these two parties.

POC: Technical analysis

To demonstrate the feasibility of the mortgage register described in the chapter Business analysis, we build a proof of concept (POC). This POC consists of a full stack application based on Hyperledger Fabric.
The current chapter explains:

- the choice of the blockchain framework (Hyperledger Fabric) used for the POC;
- the architecture of the POC from different perspectives (network, applications and technologies);
- the four steps to develop and run the POC (blockchain network, chaincode, client application back-end and front-end).

Blockchain framework : Hyperledger Fabric

In order to choose the blockchain framework for our POC, we take the following criteria into account:

- The mortgage register should not be open to anyone. Only the actors (e.g. banks, notaries and governmental organisations) involved in the mortgage process can access the mortgage register. Moreover, these actors should be identified. Therefore, we need a private blockchain network: an actor needs consent from membership authorities to join the blockchain network.
- The actors involved in the blockchain network need different access rights (e.g. create, read, modify the mortgage register), thus the blockchain framework should be permissioned.
- Some data shared between the bank and the notary involved in the mortgage loan should not be visible by other parties, e.g. a governmental organisation or another bank.
- The actors consist of business organisations, not citizens, and do not want to pay a fee per transaction.

Based on these criteria, we choose the blockchain framework Hyperledger Fabric (instead of Ethereum) to build our POC. Hyperledger Fabric is an “open source enterprise-grade permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts”[2]. On the other hand, Ethereum is rather used for public, permissionless blockchain.

Technical architecture

Application architecture

Our POC consists of a web application composed of the following components:

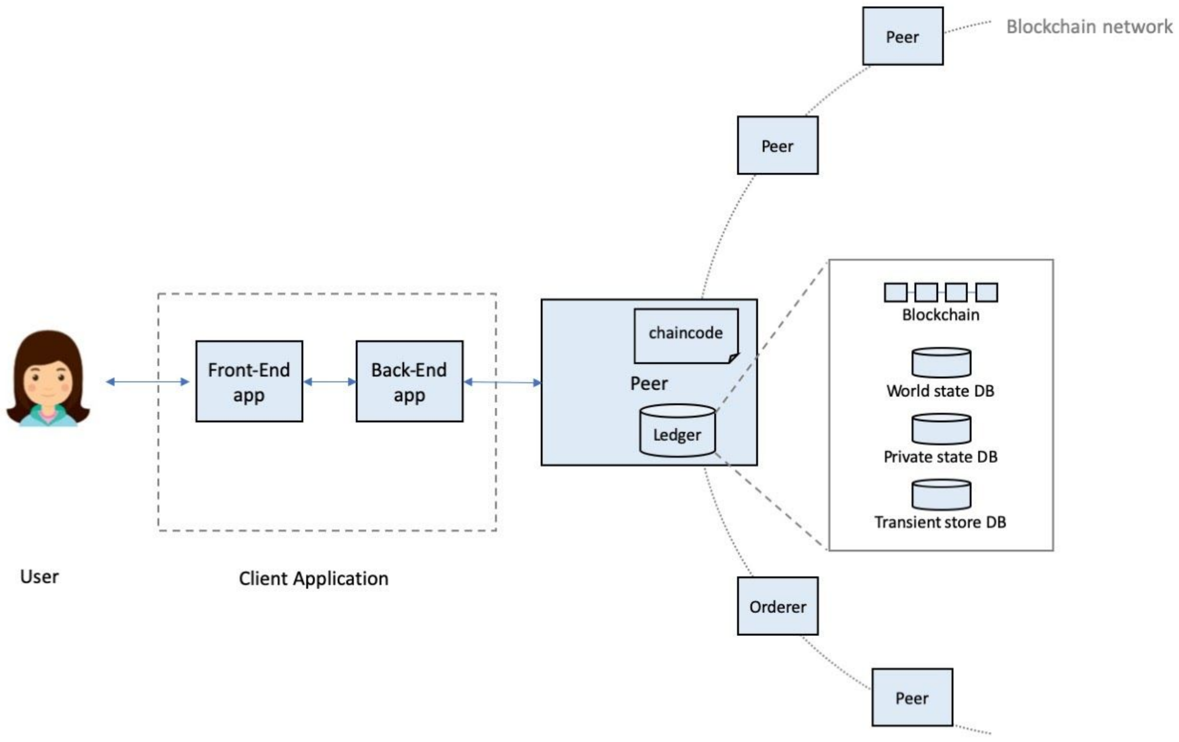
- *Front-end application*: provides an interface to the user (i.e. an agent from the Bank, the Notary or the Government) to interact with the mortgage register. Via this interface, the user can create a new loan on the mortgage register, change the status

[2] https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.html#hyperledger-fabric

of this loan or query information about this loan. The front-end component differs for the Bank, Notary and Government in order to reflect the actions each organisation's users may do.
We use Bootstrap and jQuery frameworks and HTML, CSS and Javascript languages to implement this component.

- *Back-end application*: enables to interact with the Fabric blockchain network, i.e. to access the blockchain network and invoke/query the chaincode deployed in the network. It consists of an API server.
  We use Hyperledger Fabric software development kit (SDK) for Node.js, NodeJS and ExpressJS frameworks and Javascript language to implement this component.
- *Peer*: belongs to the Fabric blockchain network. In our POC, each organisation (Bank, Notary, Government) consists of two peers. Each peer joins the channel *mychannel*, contains the installed chaincode *mortgageregister* (made of a smart contract) and stores the transactions made on this channel. Besides, a peer can also contain a private data collection shared only with some other peers.
  A network administrator must previously install this chaincode on the peers and instantiate the chaincode on the channel.
  Fabric distinguishes two kinds of peers. The endorsing peers, specified by an *endorsement policy,* consist of an installed chaincode and a local ledger. The committing peers only consist of the local ledger.
  We use Hyperledger Fabric release 1.4 framework, CouchDB database, Go language and JSON data interchange format to implement this component.
- *Orderers*: order transactions into a block. See also section Hyperledger Fabric network architecture and Hyperledger Fabric glossary[3].
- *Certificate Authorities*: assign an identity to the peers via a digital certificate. See section Hyperledger Fabric network architecture and Hyperledger Fabric key concepts[4].

[3] https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#ordering-service
[4] https://hyperledger-fabric.readthedocs.io/en/release-1.4/peers/peers.html



To illustrate how these components interact with each other, we look at the creation of a new loan on the mortgage register (*issueLoan* transaction). As described by Phuwanai Thummavet[5], the components take the following steps to invoke the chaincode *mortgageregister* with the transaction *issueLoan* with private data:

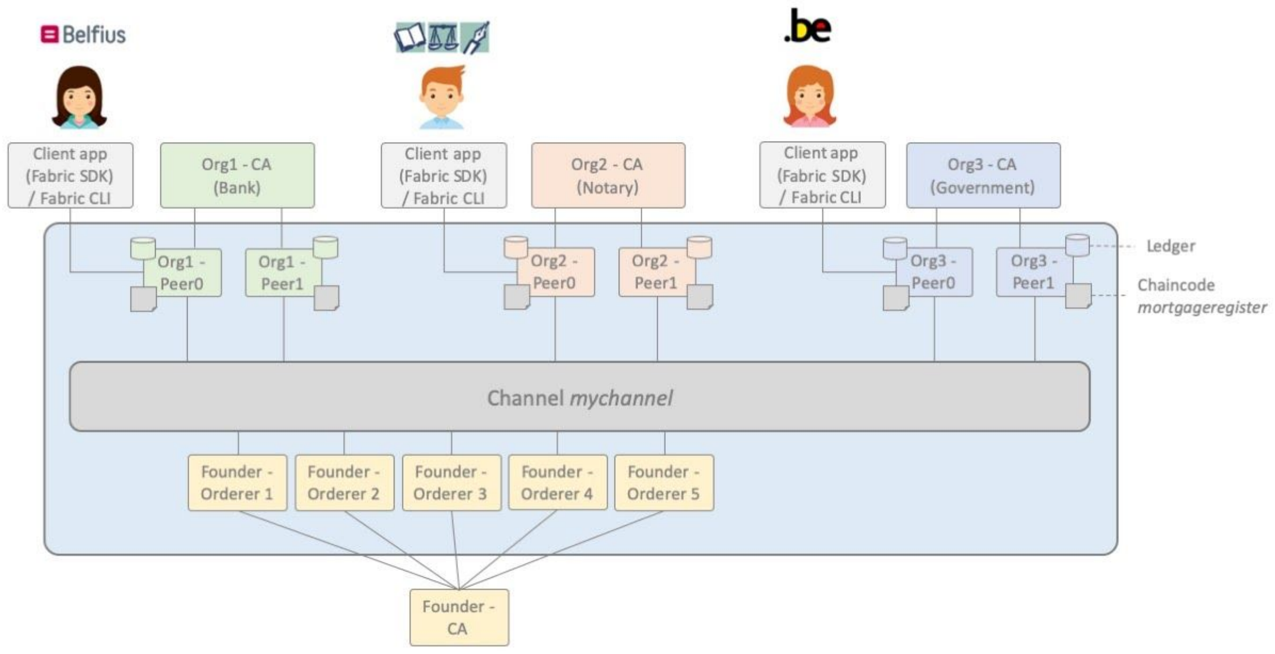
1. The *user* (a Bank agent) connects with her Web browser on the Front-end application. She goes to the page *Issue loan*, fills in the form with the different fields (e.g. loan unique ID (UID), buyer, notary) and submits this form by clicking on the button “Create loan”.
2. The *Front-end application* sends the information filled in on the form by the user (key-value pairs) to the Back-end application.
3. The *Back-end application* makes a transaction proposal, signs the proposal with the user’s certificate, and sends the transaction proposal to endorsing peers on the channel *mychannel*.
4. The *endorsing peers* verify the user’s identity and authorization. If the verification is successful, the peers simulate the transaction. The part of the simulation results containing private data (called “private read-write set”) is stored in a t*ransient store database* in the peers’ ledger.
5. The *endorsing peers* disseminate the private read-write set to other authorized peers via the gossip protocol. When the endorsing peers have disseminated this private read-write set to enough peers (see the property *requiredPeerCount* in the private data collection definition), the endorsing peers generate a response including the hash of the private read-write set. Then, they endorse this response and send it to the Back-end application.

[5] https://www.serial-coder.com/post/demystifying-hyperledger-fabric-private-data-collection/ and https://www.serial-coder.com/post/demystifying-hyperledger-fabric-fabric-architecture/

6. The *Back-end application* receives and checks the endorsed responses. The Back-end application sends the transaction attached with the endorsed responses to the Orderers.
7. The *Orderers* order the transactions, generate and sign a new block with these transactions. Then they broadcast this block to all peers on the channel *mychannel*.
8. The peers authorized to see the private data collection (see the property *policy* in the private data collection definition) check a.o. the private read-set with the hash of this private read-write set. If the checks are successful, these peers update their *private state database* and remove the private read-write set from their *transient store databases*.
9. The *Back-end application* is notified about the update of the mortgage register and can then notify the *Front-end application* about this update.

Hyperledger Fabric network architecture

The diagram below illustrates the main components involved with the Fabric blockchain network:



Running the POC

In the steps below, we describe how we build and run the POC:

1. Set up the blockchain network and the channel *mychannel*.
2. Install the chaincode *mortgageregister* on all peers and instantiate it on the channel *mychannel*.
3. Invoke the transactions *issueLoan*, *readLoan* and *changeLoanStatus* either from the Terminal (with Fabric CLI), either from the client application (Front-end and Back-end, with Fabric NodeJS SDK).

The components of our POC are structured as follows :

- **mortgageregister/fabric-samples/chaincode/mortgageregister**:
  contains the chaincode *mortgageRegister_chaincode.go* and private data collection definition *collections_config.json*.
- **mortgageregister/fabric-samples/fabcar/javascript**:
  contains the back-end application (API server), which mainly consists of the files *app.js, enrollAdmin.js, registerUser.js*
- **mortgageregister/fabric-samples/fabcar/javascript/front-end:**
  contains the front-end application, which is served by running *index.js* as a Node.js process. The connection files are found in the first-network folder.

Step 0: Prerequisites

Import chaincode dependencies

We use *govendor* to import the external dependencies of the chaincode *mortgageRegister_chaincode.go* into a local vendor directory:

```
cd
/home/fabric/blockchain/src/mortgageRegisterBC/fabric-samples/chaincode/
mortgageregister/go/
govendor init
govendor add +external
```

Clean up the old docker containers

We stop any previous blockchain network and kill any stale or active containers:

```
cd
/home/fabric/blockchain/src/mortgageRegisterBC/fabric-samples/first-netw
ork
./byfn.sh down
docker kill $(docker ps -aq)
docker rm $(docker ps -aq)
```

Then, we check that there is no active Docker container anymore:

```
docker ps -a
```

This command gives this outcome:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

Step 1: Set up the blockchain network

We use the scripts *Build Your First Network* (*byfn.sh*) and *Extend Your First Network* (*eyfn.sh*) to set up the blockchain network:

- 3 organisations : Bank (org1), Notary (org2) and Government (org3);
- each organisation consists of 2 peers: peer0 and peer1;
- 5 orderers, RAFT consensus.

First, we generate the certificates and genesis block for the channel *mychannel*:

```
./byfn.sh generate
```

Secondly, we launch the blockchain network with 2 organisations (org1 and org2), RAFT consensus, 5 orderers and CouchDB state database (see also appendix Explanation of the script Build your first network (byfn.sh) for more details about the script *byfn.sh*):

```
./byfn.sh up -c mychannel -s couchdb -o etcdraft -a
```

Finally, we bring the third organisation (org3) into the channel *mychannel*:

```
./eyfn.sh up
```

We check the active Docker containers:

```
docker ps -a


CONTAINER ID         IMAGE
COMMAND              CREATED              STATUS               PORTS                                    NAMES
4681399715fc         dev-peer0.org2.example.com-mycc-2.0-c7aee9ad18dddc18319f5f00199f05d866f9e61dca40c9af3e226d434ac4a63c
"chaincode -peer.add..."   About an hour ago   Up About an hour
dev-peer0.org2.example.com-mycc-2.0
07409105d11b         dev-peer0.org3.example.com-mycc-2.0-156223788c3ef42ff3094c6cf1d2f71284c36f2074cc4d1f09a7065cb903d192
"chaincode -peer.add..."   About an hour ago   Up About an hour
dev-peer0.org3.example.com-mycc-2.0
cc5972e781ab         dev-peer0.org1.example.com-mycc-2.0-2732cd4d96a0b88594aefca15581eaa0fb481ad15beeb86cc79931b2a90ee621
"chaincode -peer.add..."   About an hour ago   Up About an hour
dev-peer0.org1.example.com-mycc-2.0
bd7a2a73fd9d         hyperledger/fabric-tools:latest
"/bin/bash"          About an hour ago   Up About an hour                                              Org3cli
6133da62075a         hyperledger/fabric-peer:latest
"peer node start"    About an hour ago   Up About an hour       0.0.0.0:12051->12051/tcp                 peer1.org3.example.com
00146668d65b         hyperledger/fabric-peer:latest
"peer node start"    About an hour ago   Up About an hour       0.0.0.0:11051->11051/tcp                 peer0.org3.example.com
7743941233d2         dev-peer1.org2.example.com-mycc-1.0-26c2ef32838554aac4f7ad6f100aca865e87959c9a126e86d764c8d01f8346ab
"chaincode -peer.add..."   About an hour ago   Up About an hour
dev-peer1.org2.example.com-mycc-1.0
da127a5c2fd7         dev-peer0.org1.example.com-mycc-1.0-384f11f484b9302df90b453200cfb25174305fce8f53f4e94d45ee3b6cab0ce9
"chaincode -peer.add..."   About an hour ago   Up About an hour
dev-peer0.org1.example.com-mycc-1.0
80f6baaecd5b         dev-peer0.org2.example.com-mycc-1.0-15b571b3ce849066b7ec74497da3b27e54e0df1345daff3951b94245ce09c42b
```

```
"chaincode -peer.add..."   About an hour ago   Up About an hour
dev-peer0.org2.example.com-mycc-1.0
ee43baeb474f          hyperledger/fabric-tools:latest
"/bin/bash"           About an hour ago   Up About an hour                                                   cli
09541f67dad6          hyperledger/fabric-peer:latest
"peer node start"     About an hour ago   Up About an hour       0.0.0.0:9051->9051/tcp                      peer0.org2.example.com
9358ee49dc1d          hyperledger/fabric-peer:latest
"peer node start"     About an hour ago   Up About an hour       0.0.0.0:7051->7051/tcp                      peer0.org1.example.com
0113905730de          hyperledger/fabric-peer:latest
"peer node start"     About an hour ago   Up About an hour       0.0.0.0:10051->10051/tcp                    peer1.org2.example.com
d121294c80b3          hyperledger/fabric-peer:latest
"peer node start"     About an hour ago   Up About an hour       0.0.0.0:8051->8051/tcp                      peer1.org1.example.com
defeb9e600da          hyperledger/fabric-orderer:latest
"orderer"             About an hour ago   Up About an hour       0.0.0.0:11050->7050/tcp                     orderer5.example.com
1346197cfe74          hyperledger/fabric-couchdb
"tini -- /docker-ent..."   About an hour ago   Up About an hour  4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp   couchdb2
36c9ae92866a          hyperledger/fabric-orderer:latest
"orderer"             About an hour ago   Up About an hour       0.0.0.0:10050->7050/tcp                     orderer4.example.com
1d10f498709c          hyperledger/fabric-couchdb
"tini -- /docker-ent..."   About an hour ago   Up About an hour  4369/tcp, 9100/tcp, 0.0.0.0:8984->5984/tcp   couchdb3
097e7e54ed12          hyperledger/fabric-couchdb
"tini -- /docker-ent..."   About an hour ago   Up About an hour  4369/tcp, 9100/tcp, 0.0.0.0:6984->5984/tcp   couchdb1
bfc42461e3ab          hyperledger/fabric-orderer:latest
"orderer"             About an hour ago   Up About an hour       0.0.0.0:7050->7050/tcp                      orderer.example.com
a00853e27119          hyperledger/fabric-couchdb
"tini -- /docker-ent..."   About an hour ago   Up About an hour  4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp   couchdb0
341c1c3cefbc          hyperledger/fabric-orderer:latest
"orderer"             About an hour ago   Up About an hour       0.0.0.0:8050->7050/tcp                      orderer2.example.com
5c306b70a3c6          hyperledger/fabric-orderer:latest
"orderer"             About an hour ago   Up About an hour       0.0.0.0:9050->7050/tcp                      orderer3.example.com
```

Step 2: Install and instantiate the chaincode

Private data collections definition

In the file *collections_config.json*, we define two private data collections:

- *collectionLoanPrivateInfo*: can be seen and edited only by the Bank (Org1) and the Notary (Org2). It contains the "private" information about the mortgage loan that should be accessible only by the Bank and the Notary involved with this loan, not by any third party (e.g. the Government). For our POC, we assumed that the loan value, currency and interest rate were private information.
- *collectionLoans*: can be seen and modified by the Bank, Notary and Government organisations (Org1, Org2, Org3). It includes "public" information such as the issuer, buyer (lender) and notary names, the start and end dates of the loan, etc. We could also record this information on the public world state instead of in a private data collection, since this information should be accessible by any organisation connected to the blockchain network.

Both collections also contain a unique ID (UID) for each loan. This enables the *readLoan* function to query both "private" and "public" information based on this UID.

```
[{
      "name": "collectionLoans",
      "policy": "OR('Org1MSP.member', 'Org2MSP.member',
'Org3MSP.member')",
      "requiredPeerCount": 3,
      "maxPeerCount": 3,
      "blockToLive": 0,
      "memberOnlyRead": true
      },
      {
      "name": "collectionLoanPrivateInfo",
      "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
      "requiredPeerCount": 3,
      "maxPeerCount": 3,
      "blockToLive": 0,
      "memberOnlyRead": true
      }
]
```

Chaincode

We highlight below four aspects of the chaincode *mortgageregister* defined in the file *mortgageRegister_chaincode.go*:

- the dependency with the Client Identity Chaincode Library (cid);
- the structures representing the loan asset;
- the functions to issue a loan, read a loan, and change its status;
- the transient data.

More details can be found in the comments of the file *mortgageRegister_chaincode.go*.

Client Identity Chaincode Library

We import the Client Identity Chaincode Library[6] (cid) to identify the client invoking the chaincode and make access control decisions based on the client identity's MSP (Membership Service Provider) ID:

```
import (
     "github.com/hyperledger/fabric/core/chaincode/lib/cid"
)
```

More specifically, we use the function *GetMSPID* from this library in two functions of our chaincode:

- in *issueLoan* to define the issuer of the new loan;
- in *changeLoanStatus* to ensure that only the authorized organisation (notary or bank) change the status of the loan.

Structures

We define two structures representing the “public” and “private” information of a loan asset:

```
// struct representation of a mortgage loan
type loan struct {
    LoanUID   string `json:"loanUID"`   // unique ID of the loan, e.g. loan1, loan2, loan3
    Issuer      string `json:"issuer"`   // bank issuing the loan
    Buyer       string `json:"buyer"`    // hash of the national registry number of the person
borrowing the loan (buyer of the house)
```

[6] https://github.com/hyperledger/fabric/tree/release-1.1/core/chaincode/lib/cid

```
    Notary      string `json:"notary"`   // notary of the buyer
    Status      string `json:"status"`   // status of the loan
    StartDate string `json:"startDate"` // start date of the loan, e.g. 31/01/2020
    EndDate   string `json:"endDate"`   // end date of the loan, e.g. 31/01/2040
}

// struct representation of the private info of a loan, visible only by the bank and notary involved
in the loan
type loanPrivateInfo struct {
    LoanUID      string  `json:"loanUID"`         // unique ID of the loan, e.g. loan1, loan2, loan3
    LoanValue    int     `json:"loanValue"`       // mortgage loan value (eg 300000)
    Currency     string  `json:"currency"`        // mortgage loan currency (eg EUR)
    InterestRate float64 `json:"interestRate"` // mortgage loan interest rate (eg 0.02)
}
```

Functions
We implement the business logic in our chaincode mainly with three functions:
- *issueLoan*: to create a new mortgage loan asset and store it into chaincode private states;
- *readLoan*: to read a loan from chaincode state for a specific private data collection (*collectionLoans* or *collectionLoanPrivateInfo*);
- *changeLoanStatus* - change the status of a loan to a new status

Transient map
During the invocation of the transactions *issueLoan* and *changeLoanStatus*, the private data (e.g. loan value, status) are sent in a transient field (transient map of arguments in JSON encoding) to the endorsing peers part of authorized organisations of the private data collection. These peers simulate the transaction and store the private data in a *transient data store*, i.e. a temporary local storage.

For instance, to create a new loan, we invoke *issueLoan* and pass the transient field *{"loan": { "loanUID": "loan1", ..., "interestRate": 5 }}* :

```
export LOAN=$(echo -n
"{\"loanUID\":\"loan1\",\"buyer\":\"thomas\",\"notary\":\"Org2\",\"start
Date\":\"26-12-2020\",\"endDate\":\"26-12-2040\",\"loanValue\":100000,\"
currency\":\"EUR\",\"interestRate\":5}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["issueLoan"]}'  --transient "{\"loan\":\"$LOAN\"}"
```

Install chaincode on all peers of org1 and org2

Firstly, we enter the CLI container for org1 and org2:

```
docker exec -it cli bash
```

By default, we are connected with the peer0 of org1. We install the chaincode *mortgageregister* on this peer:

```
peer chaincode install -n mortgageregister -v 1.0 -p
github.com/chaincode/mortgageregister/go/
```

Secondly, we move to peer1 of org1 and install the chaincode:

```
export CORE_PEER_ADDRESS=peer1.org1.example.com:8051
peer chaincode install -n mortgageregister -v 1.0 -p
github.com/chaincode/mortgageregister/go/
```

Thirdly, we move to peer0 of org2:

```
export CORE_PEER_LOCALMSPID=Org2MSP
export
PEER0_ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.c
rt
export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG2_CA
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/pe
er/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.co
m/msp
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
```

where we install the chaincode:

```
peer chaincode install -n mortgageregister -v 1.0 -p
github.com/chaincode/mortgageregister/go/
```

Finally, we move to peer1 of org2 and install the chaincode:

```
export CORE_PEER_ADDRESS=peer1.org2.example.com:10051
peer chaincode install -n mortgageregister -v 1.0 -p
github.com/chaincode/mortgageregister/go/
```

Install chaincode on all peers of org3

We exit the CLI container of org1 and org2 and enter the Org3-specific CLI container for org3:

```
exit
docker exec -it Org3cli bash
```

By default, we are connected with the peer0 of org3. We install the chaincode on this peer:

```
peer chaincode install -n mortgageregister -v 1.0 -p
```

```
github.com/chaincode/mortgageregister/go/
```

Eventually, we move to peer1 of org3 and install the chaincode:

```
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org3.example.com/peers/peer1.org3.exampl
e.com/tls/ca.crt
export CORE_PEER_ADDRESS=peer1.org3.example.com:12051
peer chaincode install -n mortgageregister -v 1.0 -p
github.com/chaincode/mortgageregister/go/
```

Instantiate chaincode on channel mychannel

We exit the Org3 CLI container and enter into the CLI container of Org1 and Org2:

```
exit
docker exec -it cli bash
```

Then, we instantiate the chaincode *mortgageregister* on the channel *mychannel*:

```
export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ord
ererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacert
s/tlsca.example.com-cert.pem
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile
$ORDERER_CA -C mychannel -n mortgageregister -v 1.0 -c
'{"Args":["init"]}' -P "OR('Org1MSP.member','Org2MSP.member')"
--collections-config
$GOPATH/src/github.com/chaincode/mortgageregister/collections_config.jso
n
```

Step 3a: Invoke and query the chaincode with the Fabric CLI

In the current step, we invoke and query the chaincode with the Fabric CLI container for different transactions:

- *issueLoan*: allows the Bank (Org1) to issue the loan *loan1*;
- *readLoan*: allows the Bank (Org1), the Notary (Org2) and the Government (Org3) to query the information of a specific loan (*loan1*);
- *changeLoanStatus*: allows the Bank (Org1) and the Notary (Org2) to change the status of the loan *loan1*.

These invocations and queries allow us to check that the blockchain network and chaincode work correctly. However, these invocations can (should) be done from the Client application (see Step 3b: Invoke transactions from the Client application (front-end and back-end) below).

issueLoan

Chaincode invoked by the Bank (org1.peer0)

Since only the Bank can issue a loan, we connect with peer0 of Org1:

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
export CORE_PEER_LOCALMSPID=Org1MSP
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.exampl
e.com/tls/ca.crt
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/pe
er/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.co
m/msp
export
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.c
rt
```

Then, we issue a new loan *loan1*:

```
export LOAN=$(echo -n
"{\"loanUID\":\"loan1\",\"buyer\":\"thomas\",\"notary\":\"Org2\",\"start
Date\":\"26-12-2020\",\"endDate\":\"26-12-2040\",\"loanValue\":100000,\"
currency\":\"EUR\",\"interestRate\":5}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["issueLoan"]}'  --transient "{\"loan\":\"$LOAN\"}"
```

readLoan

Chaincode queried by the Bank (org1.peer0)

First, we query the data about *loan1* from the private data collection *collectionLoans*:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoans"]}'
```

and we receive then the following data:

```
{"buyer":"thomas","endDate":"26-12-2040","issuer":"Org1","loanUID":"loan
1","notary":"Org2","startDate":"26-12-2020","status":"issued"}
```

Secondly, we query the data about *loan1* from the private data collection *collectionLoanPrivateInfo*:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoanPrivateInfo"]}'
```

and we receive then the following data:

```
{"currency":"EUR","interestRate":5,"loanUID":"loan1","loanValue":100000}
```

Chaincode queried by the Notary (org2.peer0)

We connect to the Notary (Org2, peer0) and perform similar commands to receive the information about the loan *loan1*:

- connect to org2.peer0:

```
export CORE_PEER_LOCALMSPID=Org2MSP
export
PEER0_ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.c
rt
export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG2_CA
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/pe
er/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.co
m/msp
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
```

- invoke *readLoan* for collection *collectionLoans*:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoans"]}'
>
{"buyer":"thomas","endDate":"26-12-2040","issuer":"Org1","loanUID":"loan
1","notary":"Org2","startDate":"26-12-2020","status":"issued"}
```

- invoke *readLoan* for collection *collectionLoanPrivateInfo*:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoanPrivateInfo"]}'
>
{"currency":"EUR","interestRate":5,"loanUID":"loan1","loanValue":100000}
```

changeLoanStatus

Chaincode invoked by the Notary (org2.peer0) to change the loan status from "issued" to "active"

As Notary, we want to change the status of the loan *loan1* from *issued* to *active*. Therefore, we define a new variable *NEWLOANSTATUS* and pass it as transient data (with *loan_status* as key) when we submit the transaction *changeLoanStatus*:

```
export NEWLOANSTATUS=$(echo -n
"{\"loanUID\":\"loan1\",\"status\":\"active\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["changeLoanStatus"]}' --transient
"{\"loan_status\":\"$NEWLOANSTATUS\"}"
```

To check if the status has been correctly adapted, we query the chaincode for the loan *loan1* on the ledger with a *readLoan* transaction:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoans"]}'
>
{"buyer":"thomas","endDate":"26-12-2040","issuer":"Org1","loanUID":"loan
1","notary":"Org2","startDate":"26-12-2020","status":"active"}
```

Chaincode invoked by the Notary (org2.peer0) to change the loan status from "active" to "inactive"

As Notary, we apply similar commands to change the status of the loan *loan1* from *active* to *inactive*:

```
export NEWLOANSTATUS=$(echo -n
"{\"loanUID\":\"loan1\",\"status\":\"inactive\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["changeLoanStatus"]}'  --transient
"{\"loan_status\":\"$NEWLOANSTATUS\"}"
```

Then, we query the chaincode to check if the status has been correctly adapted:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoans"]}'
>
```

```
{"buyer":"thomas","endDate":"26-12-2040","issuer":"Org1","loanUID":"loan
1","notary":"Org2","startDate":"26-12-2020","status":"inactive"}
```

Chaincode invoked by the Notary (org2.peer0) to change the loan status from "inactive" to "issued"

As Notary, we try to change the status of the loan to a wrong status, e.g. from *inactive* to *issued*. Since this in not allowed by the chaincode, this generates an error:

```
export NEWLOANSTATUS=$(echo -n
"{\"loanUID\":\"loan1\",\"status\":\"issued\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["changeLoanStatus"]}'  --transient
"{\"loan_status\":\"$NEWLOANSTATUS\"}"
>
Error: endorsement failure during invoke. response: status:500
message:"New status must be 'active', 'inactive' or 'cancelled'. New
status proposed is: issued"
```

Chaincode invoked by the Notary (org2.peer0) to change the loan status from "inactive" to "active"

We observe a similar error when we try to change the status from *inactive* to *active*, an update not allowed by the chaincode:

```
export NEWLOANSTATUS=$(echo -n
"{\"loanUID\":\"loan1\",\"status\":\"active\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["changeLoanStatus"]}'  --transient
"{\"loan_status\":\"$NEWLOANSTATUS\"}"
>
Error: endorsement failure during invoke. response: status:500
message:"Only 'issued' loan can be changed to 'active'. New status
is:active, but current status is: inactive"
```

Chaincode invoked by the Bank (org1.peer0) to change the loan status from "inactive" to "cancelled"

We connect to one of the Bank's peers (org1.peer0) and try to change the status from the loan from *inactive* to *cancelled*:

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
export CORE_PEER_LOCALMSPID=Org1MSP
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.exampl
e.com/tls/ca.crt
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/pe
er/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.co
m/msp
export
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.c
rt
```

Afterwards, we invoke the chaincode (function *changeLoanStatus*). Since this is not allowed by the chaincode, we receive an error. Indeed, only the notary (Org2) can update a loan from *inactive* to *cancelled*:

```
export NEWLOANSTATUS=$(echo -n
"{\"loanUID\":\"loan1\",\"status\":\"cancelled\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["changeLoanStatus"]}' --transient
“{\"loan_status\":\"$NEWLOANSTATUS\"}"
=>
Error: endorsement failure during invoke. response: status:500
message:"Only notary can change the status to 'active'. Transaction
creator is: Org1, but notary is: Org2"
```

Chaincode invoked by the Notary (org2.peer0) to change the loan status from "inactive" to "cancelled"

We connect to a Notary’s peer (org2.peer0):

```
export CORE_PEER_LOCALMSPID=Org2MSP
export
PEER0_ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.c
rt
export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG2_CA
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/pe
er/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.co
```

```
m/msp
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
```

and change the status of the loan from *inactive* to *cancelled*:

```
export NEWLOANSTATUS=$(echo -n
"{\"loanUID\":\"loan1\",\"status\":\"cancelled\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganiz
ations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exa
mple.com-cert.pem -C mychannel -n mortgageregister -c
'{"Args":["changeLoanStatus"]}'  --transient
"{\"loan_status\":\"$NEWLOANSTATUS\"}"
```

We query the chaincode to read the loan properties (*readLoan* function) and notice that the status has been changed:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoans"]}'
>
{"buyer":"thomas","endDate":"26-12-2040","issuer":"Org1","loanUID":"loan
1","notary":"Org2","startDate":"26-12-2020","status":"cancelled"}
```

readLoan

Chaincode queried by the Government (org3.peer0)

In order to check that the Government, cannot access the private data collection*collectionLoanPrivateInfo*, we connect to the Government (org3.peer0) and query the chaincode for information about the loan *loan1*. As expected, we get an error when we query the information contained in the private data collection*collectionLoanPrivateInfo*:

- exit Fabric CLI of Org 1 and 2, and enter the Fabric CLI of Org3 (by default connected to org3.peer0):

```
exit
docker exec -it Org3cli bash
```

- invoke *readLoan* for collection *collectionLoans*:

```
peer chaincode query -C mychannel -n mortgageregister -c
'{"Args":["readLoan","loan1", "collectionLoans"]}'
>
{"loanUID":"loan1","issuer":"Org1","buyer":"thomas","notary":"Org2","sta
tus":"cancelled","startDate":"26-12-2020","endDate":"26-12-2040"}
```

- invoke *readLoan* for collection *collectionLoanPrivateInfo*:

```
peer chaincode query -C mychannel -n mortgageregister -c
```

```
'{"Args":["readLoan","loan1", "collectionLoanPrivateInfo"]}'
>
Error: endorsement failure during query. response: status:500
message:"{\"Error\":\"Failed to get state for loan loan1\"}"
```

Step 3b: Invoke and query the chaincode with the Client application (front-end and back-end)

Based on the LedgerTech example[7], we created a Node.js API server which is based on the Express framework and sits in the fabric-samples/fabcar/javascript folder. Before starting the server, we will need to create an idendity through which the Node.js SDK will interact with the blockchain. In the fabric-samples/fabcar/javascript folder, pre-made modules are available to register these identities.

To enroll an admin user for Org1 (= Bank), issue the following command:

```
node enrollAdmin.js
```

Next, create a user for Org1 by issuing:

```
node registerUser.js
```

Wallets for the admin and user are now created in the fabcar/javascript/wallet folder. These wallets contain the authentication data needed to interact with the blockchain.

Install the dependencies:

```
npm install
```

The server can now be started with the command:

```
node app.js
```

This will start the server on port 8080. Once started, the server accepts 4 possible requests, which in turn invoke a command on the blockchain.

[7] https://medium.com/@kctheservant/an-implementation-of-api-server-for-hyperledger-fabric-network-8764c79f1a87

- GET /api/readloan/private/:loanuid
  → readLoan (collection: “collectionLoanPrivateInfo”)

- GET /api/readloan/public/:loanuid
  → readLoan (collection: “collectionLoans”)

- POST /api/issueloan
  → issueLoan

- POST /api/changeloanstatus
  → changeLoanStatus

When either of these requests occur, the server responds with the following process:

- CORS policy: allow the request, regardless of the address of the requester.
  This is only for demo purposes and should ofcourse be modified for production environments.
- Check if a wallet exists for user1.
- Connect to the network through user1.
- Set the transient data (if needed).
- Invoke a transaction towards the mortgageregister smart contract.

If one of the conditions is not met, the server responds with an error.

The routes from this API are addressed by the front-end application which we will describe in the next chapter. Another option would be to address the API via the command line using curl. An example of how to invoke a query command using cURL:

```
curl http://localhost:8080/api/readloan/public/loan1
```

This will return the data from loan1 as a (JSON) string.

Front-end application

Our front-end application is started by serving a Node.js process, similar to the back-end, which makes use of the ExpressJS framework. The application can be found in the fabric-samples/fabcar/javascript/front-end folder. You will also need to install the dependencies with “npm install” first to get it running.

There are several GET requests that can be made which do not trigger interaction with the back-end, but instead are used to display static HTML (using EJS as a templating engine):

- GET /issueloan
- GET /readloan/public
- GET /readloan/private

- GET /deactivateloan
- GET /

Furthermore, there are 4 requests that interact with the back-end:

- POST /issueloan
- GET /readloan/public/:loanuid
- GET /readloan/private
- POST /deactivateloan

The requests to the back-end are made with the “request” module. Basically, the data from the front-end is posted in the body of the request and sent to the API. The API then subsequently calls the corresponding transaction on the blockchain.

This implementation has numerous problems, mainly due to time restriction to optimize this. Some examples:

- Currently, the registerUser.js module registers a user who is part of Org1. So transactions can only be made coming from the bank, other transactions will have to be made via the CLI.
- There is no error handling, the front-end is built for the “happy flow” only.
- There are no separate parts to activate or cancel a loan from the front-end. Even using the deactivation form on the front-end application is complex: a loan (of which the value for notary is “Org2”) must first be activated via the CLI. Next, a user must be created for Org2 and this user must be used to deactivate the loan. However, this will first require manual editing of the enrollAdmin and registerUser modules.

We also address these problems in “Next Steps” below.

Conclusion

Lessons learned and individual reflection

Karel's reflection

This project was the perfect exercise to learn more about the actual underlying aspects of (permissioned) blockchain technology. The complexity of creating a real-world application actually forces you to learn about aspects that we (almost) did not cover in any of the lessons such as authentication and actually connecting a front-end/back-end application to a Hyperledger blockchain network.

The hardest parts were understanding the very basics of what's going on behind the scenes. Once you get a grasp, the network set-up becomes easier but it has still been a lot of "trial and error" up until the end. I also felt that I wanted to get into the details far too quickly. This evidently led me to bumping into several issues that could have been resolved much faster if I would've taken the time to understand the bigger picture. But overall, I have learned quite a lot and it entices me to learn even more about Hyperledger Fabric. I would really look forward to building another application!

Thomas' reflection

For me, the main challenges of this project were :

- understand several concepts behind blockchain (e.g. cryptography, networking, databases and web development), Hyperledger Fabric (e.g. peers, orderers, chaincode, private data collection) and other related aspects (Go and Javascript languages, NodeJS framework, Docker containers and Linux command line interface);
- identify a business problem where blockchain makes sense and be critical;
- define the architecture of the solution;
- the constant evolution of Fabric framework, which made the documentation not always up to date.

These challenges were a real opportunity to learn and put into practice what we saw during the last 1,5 year. Not only at a theoretical level, but also at a hands-on level. This hands-on aspect forced me to get confronted with issues you don't see in slides or lectures.

In the future, I hope to use all these learnings to identify other business issues where blockchain technology makes sense and develop solutions to solve these issues.

Next steps

In order to further improve our POC, we would take the following actions:

- Optimize the infrastructure to enable multiple banks, notaries and governments to join the blockchain network and interact with the mortgage register;
- Adapt the loan properties, business logic and access rights to reflect closer an actual mortgage register;
- Use the *mortgageregister* chaincode to automate some actions, e.g. the computation of payment/subventions;
- Add a function in the *mortgageregister* chaincode to index and query all the loans of a specific bank, notary or borrower (buyer);
- Enable the borrower (buyer) to access the blockchain network and perform some transactions while preserving her anonymity, e.g. with Identity Mixer[8].
- Implement a solution to be able to access the API with multiple identities (i.e. modify/extend the enrollAdmin.js and registerUser.js modules)
- Introduce adequate error handling for both the back-end and front-end application

[8] https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html

Appendix

A1. Explanation of the script Build your first network (byfn.sh)

Quick set-up of first-network with Raft consensus

In what follows, we will explain how the Build Your First Network (BYFN) tutorial network is set up and what happens under the hood when performing 2 commands which generate a Docker-based Hyperledger Fabric network with Raft consensus:

```
cd fabric-samples/first-network
./byfn.sh up -o etcdraft
```

These simple commands generate the artifacts, certificates & configuration transaction and start 5 ordering nodes and 4 peer nodes (in 2 organizations). The nodes have the following names by default:

- `orderer.example.com`
- `orderer2.example.com`
- `orderer3.example.com`
- `orderer4.example.com`
- `orderer5.example.com`
- `peer0.org1.example.com`
- `peer1.org1.example.com`
- `peer0.org2.example.com`
- `peer1.org2.example.com`

Next, it creates a channel *mychannel* and lets all peers join this channel. The anchor peers are then designated. After that, the chaincode is installed on *peer0.org1.example.com* and *peer0.org2.example.com*.

Finally, the chaincode is instantiated on *peer0.org2.example.com* and an initial transaction is invoked which creates two key-value pairs: *a* (value: 100) and *b* (value: 200):

```
Instantiating chaincode on peer0.org2...
+ peer chaincode instantiate -o orderer.example.com:7050
--tls true --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ord
ererOrganizations/example.com/orderers/orderer.example.com/ms
p/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n mycc
-l golang -v 1.0 -c '{"Args":["init","a","100","b","200"]}'
-P 'AND ('\''Org1MSP.peer'\'','\''Org2MSP.peer'\'')'
```

```
+ res=0
+ set +x
2019-12-06 07:46:54.604 UTC [chaincodeCmd]
checkChaincodeCmdParams -> INFO 001 Using default escc
2019-12-06 07:46:54.604 UTC [chaincodeCmd]
checkChaincodeCmdParams -> INFO 002 Using default vscc
===================== Chaincode is instantiated on peer0.org2
on channel 'mychannel' =====================
```

To verify that the world state has been updated on the other peers and organizations, the value of *a* is queried on *peer0.org1.example.com*. The correct value *100* is returned:

```
Querying chaincode on peer0.org1...
===================== Querying on peer0.org1 on channel
'mychannel'... =====================
Attempting to Query peer0.org1 ...3 secs
+ peer chaincode query -C mychannel -n mycc -c
'{"Args":["query","a"]}'
+ res=0
+ set +x

100
```

Subsequently, a new transaction is invoked which transfers a value of 10 from *a* to *b*. This means the new values should be 90 and 210 respectively. The transaction is invoked on *peer0.org1.example.com* and *peer0.org2.example.com*:

```
Sending invoke transaction on peer0.org1 peer0.org2...
+ peer chaincode invoke -o orderer.example.com:7050 --tls
true --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ord
ererOrganizations/example.com/orderers/orderer.example.com/ms
p/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n mycc
--peerAddresses peer0.org1.example.com:7051
--tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/pee
rOrganizations/org1.example.com/peers/peer0.org1.example.com/
tls/ca.crt --peerAddresses peer0.org2.example.com:9051
--tlsRootCertFiles
```

```
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/pee
rOrganizations/org2.example.com/peers/peer0.org2.example.com/
tls/ca.crt -c '{"Args":["invoke","a","b","10"]}'
+ res=0
+ set +x
2019-12-06 07:47:33.918 UTC [chaincodeCmd]
chaincodeInvokeOrQuery -> INFO 001 Chaincode invoke
successful. result: status:200
===================== Invoke transaction successful on
peer0.org1 peer0.org2 on channel 'mychannel'
=====================
```

Next, the chaincode is installed on peer1.org2.example.com. As a test, the new value for *a* is read from *peer1.org2.example.com*. The returned vaue of 90 proves that the transaction has persisted and that all peers are in sync with the anchor peer, immediately after installing the chaincode:

```
Installing chaincode on peer1.org2...
+ peer chaincode install -n mycc -v 1.0 -l golang -p
github.com/chaincode/chaincode_example02/go/
+ res=0
+ set +x
2019-12-06 07:47:33.977 UTC [chaincodeCmd]
checkChaincodeCmdParams -> INFO 001 Using default escc
2019-12-06 07:47:33.977 UTC [chaincodeCmd]
checkChaincodeCmdParams -> INFO 002 Using default vscc
2019-12-06 07:47:34.178 UTC [chaincodeCmd] install -> INFO
003 Installed remotely response:<status:200 payload:"OK" >
===================== Chaincode is installed on peer1.org2
=====================

Querying chaincode on peer1.org2...
===================== Querying on peer1.org2 on channel
'mychannel'... =====================
+ peer chaincode query -C mychannel -n mycc -c
'{"Args":["query","a"]}'
Attempting to Query peer1.org2 ...3 secs
+ res=0
+ set +x

90
```

```
===================== Query successful on peer1.org2 on
channel 'mychannel' =====================
```

This steps marks the end of the “Build Your First Network” tutorial. We will use this tutorial and the accompanying scripts as a basis and extend it to accomodate for the needs of our network and the client application.

Additional set-up based on BYFN

Some additional actions we have taken to prove our understanding of the BYFN tutorial:

- The chaincode has not yet been installed for *peer1.org1.example.com*. To do this manually, we execute the following commands in our terminal:

```
docker exec -it cli bash
CORE_PEER_ADDRESS=peer1.org1.example.com:8051 peer chaincode install -n
mycc -v 1.0 -l golang -p github.com/chaincode/chaincode_example02/go/
```

- In the terminal excerpts above, we can see that the value for *a* is indeed 90 after the transaction was executed. However, we have not verified whether the value of *b* has actually incremented by 10. We can do this by querying *peer1.org1.example.com*, on which we have just installed the chaincode:

```
docker exec -it cli bash
CORE_PEER_ADDRESS=peer1.org1.example.com:8051 peer chaincode query -C
mychannel -n mycc -c '{"Args":["query","b"]}'
```

This returns the expected value 210.