

INTERNSHIP REPORT

Utilizing blockchain to manage and monitor the supply chain

Loukas Litsos

The Scenario

This project is about showing how we can leverage blockchain technology to build an efficient and reliable supply chain network. Specifically, we utilized Hyperledger Fabric and we studied the supply chain of fuel and gas. A typical chain in the fuel-gas industry looks like this:

Oil-Pumper -> Crude oil Transporter -> Refiner -> Fuel & Gas Trasporter -> Retailer (fuel stations).

So, as we can see, there are at least 5 organizations involved, each of them having a completely separte role in the supply chain. The Oil Pumper pumps huge amounts of crude oil from the ground and gets paid by the Refiner, whose role is to refine the crude oil and store it.

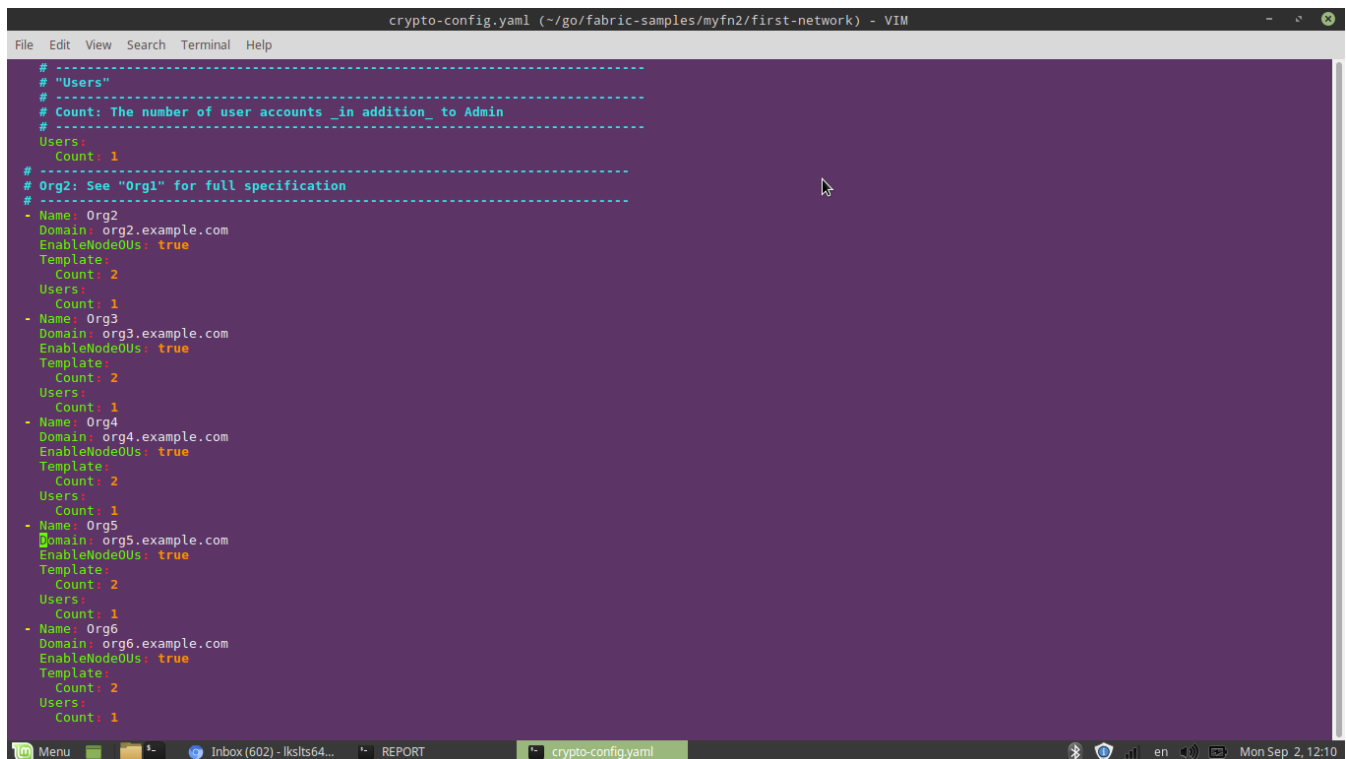
The trasportation of the crude is made by the Crude oil Transporter via tankers. When crude is refined, it's time to be send to the Retailers (fuel stations). This is done with the help of the Fuel & Gas Transporter via trucks. For every route,each truck has a Delivery Plan that specifies which Retailers should be visited in order to receive their orders.

Implementation

In order to implement the above scenario, we used Hyperledger Fabric v1.4. All transactions take place in a single channel consisting of 6 Organizations and each organization has 2 peers. The role of each organization is the same as described above but here we have added an extra organization (at the above example there were 5 in total) which acts as a second Retailer, so we have two fuel stations. Because all data in the blockchain are public,ideally one could create multiple channels (or use Fabric's Private Data feature) to retain privacy and confidentiality, but this project lacks this feature.

In order to launch a fabric network with six organizations, we made some changes at the [byfn tutorial](#) files. Specifically, the tutorial builds a network with 2 organizations so we added 4 orgs on top of the two existing ones. The configuration files that we've changed are:

- crypto-config.yaml



```
crypto-config.yaml (~/go/fabric-samples/myfn2/first-network) - VIM
File Edit View Search Terminal Help
# -----
# "Users"
# Count: The number of user accounts _in addition_ to Admin
# -----
Users:
  Count: 1
# -----
# Org2: See "Org1" for full specification
# -----
- Name: Org2
  Domain: org2.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1
- Name: Org3
  Domain: org3.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1
- Name: Org4
  Domain: org4.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1
- Name: Org5
  Domain: org5.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1
- Name: Org6
  Domain: org6.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1
```

The Template field specifies how many peers each organization will have.

- configtx.yaml:

```
configtx.yaml (~/.go/fabric-samples/myfn2/first-network) - VIM
File Edit View Search Terminal Help

ThreeOrgsChannel:
  Consortium: SampleConsortium
  <<: *ChannelDefaults
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *Org1
      - *Org2
      - *Org3
    Capabilities:
      <<: *ApplicationCapabilities
SixOrgsOrdererGenesis:
  <<: *ChannelDefaults
  Orderer:
    <<: *OrdererDefaults
    Organizations:
      - *OrdererOrg
    Capabilities:
      <<: *OrdererCapabilities
  Consortiums:
    SampleConsortium:
      Organizations:
        - *Org1
        - *Org2
        - *Org3
        - *Org4
        - *Org5
        - *Org6
SixOrgsChannel:
  Consortium: SampleConsortium
  <<: *ChannelDefaults
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *Org1
      - *Org2
      - *Org3
      - *Org4
      - *Org5
      - *Org6
    Capabilities:
      <<: *ApplicationCapabilities
```

We added SixOrgsOrdererGenesis and SixOrgsChannel fields

```
configtx.yaml (~/.go/fabric-samples/myfn2/first-network) - VIM
File Edit View Search Terminal Help

# DefaultOrg defines the organization which is used in the sampleconfig
# of the fabric.git development environment
Name: Org3MSP

# ID to load the MSP definition as
ID: Org3MSP

MSPDir: crypto-config/peerOrganizations/org3.example.com/msp

# Policies defines the set of policies at this level of the config tree
# For organization policies, their canonical path is usually
# /Channel/<Application/<Orderer/<OrgName/<PolicyName>
Policies:
  Readers:
    Type: Signature
    Rule: "OR('Org3MSP.admin', 'Org3MSP.peer', 'Org3MSP.client')"
  Writers:
    Type: Signature
    Rule: "OR('Org3MSP.admin', 'Org3MSP.client')"
  Admins:
    Type: Signature
    Rule: "OR('Org3MSP.admin')"

AnchorPeers:
  # AnchorPeers defines the location of peers which can be used
  # for cross org gossip communication. Note, this value is only
  # encoded in the genesis block in the Application section context
  - Host: peer0.org3.example.com
    Port: 11051

- Org4
  # DefaultOrg defines the organization which is used in the sampleconfig
  # of the fabric.git development environment
  Name: Org4MSP

  # ID to load the MSP definition as
  ID: Org4MSP

  MSPDir: crypto-config/peerOrganizations/org4.example.com/msp

  # Policies defines the set of policies at this level of the config tree
  # For organization policies, their canonical path is usually
  # /Channel/<Application/<Orderer/<OrgName/<PolicyName>
  Policies:
    Readers:
```

We defined the organizations 3-6.

Each org in fabric runs in a docker container so we modified some docker-compose files.

- cli.yaml

```
docker-compose-cli.yaml (~/.go/fabric-samples/myfn2/first-network) - VIM
File Edit View Search Terminal Help

peer1.org2.example.com:
  container_name: peer1.org2.example.com
  extends:
    file: base/docker-compose-base.yaml
  service: peer1.org2.example.com
  networks:
    - byfn

peer0.org3.example.com:
  container_name: peer0.org3.example.com
  extends:
    file: base/docker-compose-base.yaml
  service: peer0.org3.example.com
  networks:
    - byfn

peer1.org3.example.com:
  container_name: peer1.org3.example.com
  extends:
    file: base/docker-compose-base.yaml
  service: peer1.org3.example.com
  networks:
    - byfn

peer0.org4.example.com:
  container_name: peer0.org4.example.com
  extends:
    file: base/docker-compose-base.yaml
  service: peer0.org4.example.com
  networks:
    - byfn

peer1.org4.example.com:
  container_name: peer1.org4.example.com
  extends:
    file: base/docker-compose-base.yaml
  service: peer1.org4.example.com
  networks:
    - byfn

peer0.org5.example.com:
  container_name: peer0.org5.example.com
  extends:
    file: base/docker-compose-base.yaml
  service: peer0.org5.example.com
  networks:
    - byfn
```

.

- `base.yaml`

```
docker-compose-base.yaml (~/.go/fabric-samples/myfn2/first-network/base) - VIM
File Edit View Search Terminal Help

peer0.org3.example.com:
  container_name: peer0.org3.example.com
  extends:
    file: peer-base.yaml
    service: peer-base
  environment:
    - CORE_PEER_ID=peer0.org3.example.com
    - CORE_PEER_ADDRESS=peer0.org3.example.com:11051
    - CORE_PEER_LISTENADDRESS=0.0.0.0:11051
    - CORE_PEER_CHAINCODEADDRESS=peer0.org3.example.com:11052
    - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:11052
    - CORE_PEER_GOSSIP_BOOTSTRAP=peer1.org3.example.com:12051
    - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org3.example.com:11051
    - CORE_PEER_LOCALMSPID=Org3MSP
  volumes:
    - /var/run:/host/var/run/
    - ../crypto-config/peerOrganizations/org3.example.com/peers/peer0.org3.example.com/msp:/etc/hyperledger/fabric/msp
    - ../crypto-config/peerOrganizations/org3.example.com/peers/peer0.org3.example.com/tls:/etc/hyperledger/fabric/tls
    - peer0.org3.example.com:/var/hyperledger/production
  ports:
    - 11051:11051

peer1.org3.example.com:
  container_name: peer1.org3.example.com
  extends:
    file: peer-base.yaml
    service: peer-base
  environment:
    - CORE_PEER_ID=peer1.org3.example.com
    - CORE_PEER_ADDRESS=peer1.org3.example.com:12051
    - CORE_PEER_LISTENADDRESS=0.0.0.0:12051
    - CORE_PEER_CHAINCODEADDRESS=peer1.org3.example.com:12052
    - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:12052
    - CORE_PEER_GOSSIP_BOOTSTRAP=peer0.org3.example.com:11051
    - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.org3.example.com:12051
    - CORE_PEER_LOCALMSPID=Org3MSP
  volumes:
    - /var/run:/host/var/run/
    - ../crypto-config/peerOrganizations/org3.example.com/peers/peer1.org3.example.com/msp:/etc/hyperledger/fabric/msp
    - ../crypto-config/peerOrganizations/org3.example.com/peers/peer1.org3.example.com/tls:/etc/hyperledger/fabric/tls
    - peer1.org3.example.com:/var/hyperledger/production
  ports:
    - 12051:12051
```

In essence, we added 4 orgs with the same configuration as the two previously existing ones, so we just extended the network participants but not its functionality.

Finally, we needed to extend some scripts to be able to launch the network. Specifically, `byfn.sh`, `utils.sh` and `script.sh` need to be changed in order to serve our needs.

Chaincode

We used Go (Golang) to develop the chaincode (smart contracts). The whole chaincode leaves in a single file so let's have a closer look at this file. First and foremost, the structs that we created give a good insight into the functionality of the chaincode.

```
all-orgsCC.go (~/.go/fabric-samples/chaincode/fuel_chain) - VIM
File Edit View Search Terminal Help

type Vehicle struct {
    Type string
    ID    string
}

type DeliveryDetails struct {
    EstTime    time.Time
    Delay      float64
    StartingLocation string
    Destination string
}

type TxProof struct {
    URL    string
    Hash   string
}

type AssetDetails struct {
    Value    float64
    Quantity int
    Owner    string
    State    string
}

/*
Put in db with key CrudeID
Crude ID should be like this: CrudeXXXX where XXXX is an ever increasing number.
*/
type Crude struct {
    AD      AssetDetails
    DD      DeliveryDetails
    Proof   TxProof
    Veh     Vehicle
    Timestamp time.Time
}

/*
Put in db with key FuelID
Fuel ID should be like this: FuelXXXX where XXXX is an ever increasing number.
*/
type Fuel struct {
    AD      AssetDetails
    Density float64 //quality
    Type    string
    CrudeID  string //like parent ID
    Timestamp time.Time
}
```

```
all-orgsCC.go (~/.go/fabric-samples/chaincode/fuel_chain) - VIM
File Edit View Search Terminal Help

/*
Put in db with key FuelOrderID
FuelOrder ID should be like this: FuelOrderXXXX where XXXX is an ever increasing number.
*/
type FuelOrder struct {
    AD      AssetDetails
    Dest    string
    Proof   TxProof
    FuelID  string //like parent ID
    Timestamp time.Time
}

type FuelOrderID = string

/*
ID form : 'PlanXXXX'
A delivery plan from refinery towards the gas stations.
Contains the vehicle that will deliver the fuels at many fueling stations
A map for easy access to delivery details with key the orders that org2 has added.
*/
type FuelDeliveryPlan struct {
    Veh Vehicle
    Plan map[FuelOrderID]DeliveryDetails
}

type OrgAmount struct {
    amount float64
    org     string
}

func (s *SmartContract) Init(APIStub shim.ChaincodeStubInterface) sc.Response {
    return shim.Success(nil)
}

/*
* The Invoke method *
called when an application requests to run any Smart Contract
The app also specifies the specific smart contract function to call with args
*/
func (s *SmartContract) Invoke(APIStub shim.ChaincodeStubInterface) sc.Response {
    // Retrieve the requested Smart Contract function and arguments
    function, args := APIStub.GetFunctionAndParameters()
    // Route to the appropriate handler function to interact with the ledger
    if function == "deliverCrude" {

```

- Vehicle: Each transportation vehicle has a type (e.g Vessel, Truck) and an ID
- Asset Details : Every Asset (e.g. Crude , Fuel , FuelOrder) has a State (e.g. READY_FOR_DISTRIBUTION, DELIVERED, REFINED) and some other essential fields like Value etc.
- Delivery Details: Contains the starting point of the asset's route and its destination. Also, it contains the Estimated Delivery Time (EstTime) . This is the approximate time that the Transporter believes he can deliver the asset. The Delay field is how much the delivery of the asset was actually delayed relatively to the EstTime field. If Delay is negative, this means that the Transporter delivered the asset before the EstTime.
- TxProof: A realistic application would need to couple the transaction data with some 'real' data. What I mean by real is for example, the original documents which were signed by two or more orgs in order to make an order for an asset. So, this struct contains the URL of an original document and its Hash in order to ensure its integrity. This is a proof-of-concept project so

no 'real' documents were created nor signed by anyone. For every asset, this struct contains the same dummy URL and Hash fields which is "www.ait.gr" and "7cb0d761a60f4968299cda86c333dafa318fbf87b0979f60befd0499e39e21d6" (the SHA-256 Hash of 'www.ait.gr') respectively.

- Fuel Delivery Plan: A delivery plan from the refinery towards the gas stations. Contains the vehicle that will deliver the fuelOrders at many fueling stations. Each fuelOrder has some delivery details and when the order is eventually delivered, the state of the order will become 'DELIVERED'.

Chaincode API

- InitLedger: Every organization is associated with an escrow account. For example, the account for organization 1 is named 'org1'. By calling initLedger we initialize all accounts with some starting balance (1000000) . If an asset in the blockchain changes ownership, all organizations involved will send/receive money from these escrow accounts.
- deliverCrude: When this function is called , it creates a new Crude Asset based on the supplied arguments and starts the delivery of the Crude.
- refine: When the Crude gets eventually delivered , the refine process takes place. The crude is transformed into some other useful products (e.g. fuel , gas).
- addFuelOrder: When a Retailer makes an order for fuel from the Refiner, this method should be called. By the end of this invocation, a new fuelOrder will be created and stored in the blockchain.
- deliverFuel: When this function is called, a new Delivery Plan is made. The caller specifies which fuelOrders will be delivered and the Plan is made based on these orders. There is no limit to how many orders a Delivery Plan can hold.
- transfer : This method should be called when an asset is going to change ownership. For example, if the Oil-Pumper sells crude oil to Refiner then after the delivery, a transfer transaction should be submitted on the blockchain. When calling this method, all payments associated with the transfer of the asset will take place. Transporters will have their payment decreased based on the value of Delay field (see Delivery Details above). Sellers will get paid based on quantity and value of the asset they sold and buyers will pay both the transporter and the seller as usual.
- queryAsset: API for quering a single asset or an account balance

- (e.g. Crude423 , Fuel212).
- queryAssetByRange: API for querying a range of assets of the same type (e.g. Crude1-999, Plan1-999).
 - queryHistoryForKey: API for querying a single asset for its complete update history in the database (maybe there are some bugs at this method).

Transact with the network

Note: All instructions should be run under first-network/app/application/ directory

A client can transact with the blockchain with the help of a node.js app that has been developed for this purpose.

Config & Wallet

The app uses the Node SDK of Hyperledger Fabric to connect to the network and submit transactions. In order to connect to a Fabric network with the Node SDK, a client should first create a config file called Connection Profile. Here's what ours looks like:

```
networkConnection.yaml (~/go/fabric-samples/myfn2/first-network/app/gateway) - VIM
File Edit View Search Terminal Help

Org4:
  mspid: Org4MSP
  peers:
    - peer0.org4.example.com
    - peer1.org4.example.com
  adminPrivateKey:
    path: ../../crypto-config/peerOrganizations/org4.example.com/users/Admin@org4.example.com/msp/keystore/e41d46070ce4e21bf9cc0227a2fda5b2850653807025e40093d00bd7319e514a_sk
  signedCert:
    path: ../../crypto-config/peerOrganizations/org4.example.com/users/Admin@org4.example.com/msp/signcerts/Admin@org4.example.com-cert.pem
Org5:
  mspid: Org5MSP
  peers:
    - peer0.org5.example.com
    - peer1.org5.example.com
  adminPrivateKey:
    path: ../../crypto-config/peerOrganizations/org5.example.com/users/Admin@org5.example.com/msp/keystore/c42a7fdd6959464a9c095d06d4404c110e84821e6ff2289f23039b9ac3acd70a_sk
  signedCert:
    path: ../../crypto-config/peerOrganizations/org5.example.com/users/Admin@org5.example.com/msp/signcerts/Admin@org5.example.com-cert.pem
Org6:
  mspid: Org6MSP
  peers:
    - peer0.org6.example.com
    - peer1.org6.example.com
  adminPrivateKey:
    path: ../../crypto-config/peerOrganizations/org6.example.com/users/Admin@org6.example.com/msp/keystore/f65f39c252877a2753f70e2fac21ad213c50c7b4cce5ff1e10ff9c01cf7dfff57_sk
  signedCert:
    path: ../../crypto-config/peerOrganizations/org6.example.com/users/Admin@org6.example.com/msp/signcerts/Admin@org6.example.com-cert.pem
#
# List of orderers to send transaction and channel create/update requests to. For the time
# being only one orderer is needed. If more than one is defined, which one get used by the
# SDK is implementation specific. Consult each SDK's documentation for its handling of orderers.
#
orderers:
  orderer.example.com:
    url: grpc://localhost:7050
# these are standard properties defined by the gRPC library
# they will be passed in as-is to gRPC client constructor
grpcOptions:
```

The Connection Profile has the information needed for the SDK to establish a connection between the peers of the network.

Because Fabric uses CAs (Certificate Authorities) to authenticate every action on the network, a client should also be equipped with some crypto material. For this purpose, a wallet should be made. A script has been developed to create a wallet and it is available under `first-network/app/application/addToWallet.js`. (anyone can run the script with this command : `node addToWallet.js`)

Update the ledger

After creating the Connection Profile and the Wallet, we are ready to interact with the network (assuming there is one running). The very first think we need to do is call the `initLedger` method to initialize the escrow accounts. So, we run the following command:

```
$ node init.js
```

This script establishes a connection with the network by using the Connection Profile file and submits a transaction with a single argument that is `'initLedger'` i.e the name of the method we want to call. Here's what the file looks like:

```
init.js (~/.fabric-samples/myfn2/first-network/app/application) - VIM
File Edit View Search Terminal Help
>async function main() {
  // A gateway defines the peers used to access Fabric networks
  const gateway = new Gateway();

  // Main try/catch block
  try {
    // Specify userName for network access
    const userName = 'Admin@org1.example.com';

    // Load connection profile; will be used to locate a gateway
    let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/networkConnection.yaml', 'utf8'));
    //
    //let client = Client.loadFromConfig('../gateway/networkConnection.yaml')

    // Set connection options; identity and wallet
    let connectionOptions = {
      identity: userName,
      wallet: wallet,
      discovery: { enabled:false, asLocalhost: true }
    };

    // Connect to gateway using application specified parameters
    console.log('Connect to Fabric gateway.');
```

```
    await gateway.connect(connectionProfile, connectionOptions);
    console.log('Use network channel: mychannel.');
```

```
    const network = await gateway.getNetwork('mychannel');
```

```
    console.log('Use scthreadiff6 smart contract.');
```

```
    const contract = await network.getContract('scthreadiff6');
```

```
    console.log('Submit initLedger transaction.');
```

```
    let resp = await contract.submitTransaction('initLedger');
```

```
    console.log(resp)
```

```
  } catch (error) {
    console.log('Error processing transaction. ${error}');
```

```
    console.log(error.stack);
  }
}
```

```
"init.js" 113L, 2752C written
```

Menu 12:39 Mon Sep 2, 13:39

The above picture shows the steps that a client should make to connect to a Fabric network and submit a transaction.

If we want to call other methods than `initLedger`, then we should run another script called `issue.js`. This script offers a wide variety of methods to call, so we should pass as command line args the specific method we would like to invoke. We can run this script like this:

```
$ node issue.js <method_name> <arg0> <arg1> <arg2> ... <argN>
```

`method_name` can be any of the Chaincode API methods (see above) except the last three, so we can only update the ledger with `issue.js`. These are the available `method_names` and the args one should give after:

- `deliverCrude`: needs one more arg that is the ID of the Crude that will be created. Example :
\$ node issue.js deliverCrude 9842
- `transferCrude`: needs one more arg that is the ID of the Crude that will be transferred. Example:
\$ node issue.js transferCrude 4324
- `refineRand`: needs two more args that is the ID of the Crude that will be transformed to fuel and the ID of the newly created fuel (the Crude ID should pre-exist in the database).Example:
\$ node issue.js refineRand 342 98
- `addFuelOrderRand`: needs two more args that is the ID of the refined fuel that the order originated and the ID of the newly created fuel order(the fuel ID should pre-exist in the database).Example:
\$ node issue.js addFuelOrderRand 321 908
- `deliverFuelRand`: can have infinite arguments. The first one should be the ID of the newly created Plan and the rest should be the FuelOrder IDs that the Delivery Plan will contain (FuelOrderIDs should pre-exist in db). Example:
\$ node issue.js deliverFuelRand 4 1 2 3 4 5 6 ...

- transferFuel: should have two args . The first one should be the FuelOrderID and the second the PlanID (both args should pre-exist in db). Example:
\$ node issue.js transferFuel 5423 6546.

Alternatively, we can run issue.js without any args (\$ node issue.js) if we want to invoke all previous methods N times (N is random). If someone looks at the chaincode file, he/she will notice that in order to create/deliver/transfer aasset, more arguments should be supplied on top of IDs. Hence, a reasonable question will be: Where the issue.js script finds all the extra arguments that are needed? The answer is that it generates them randomly. That's why some method_names have a Rand suffix.

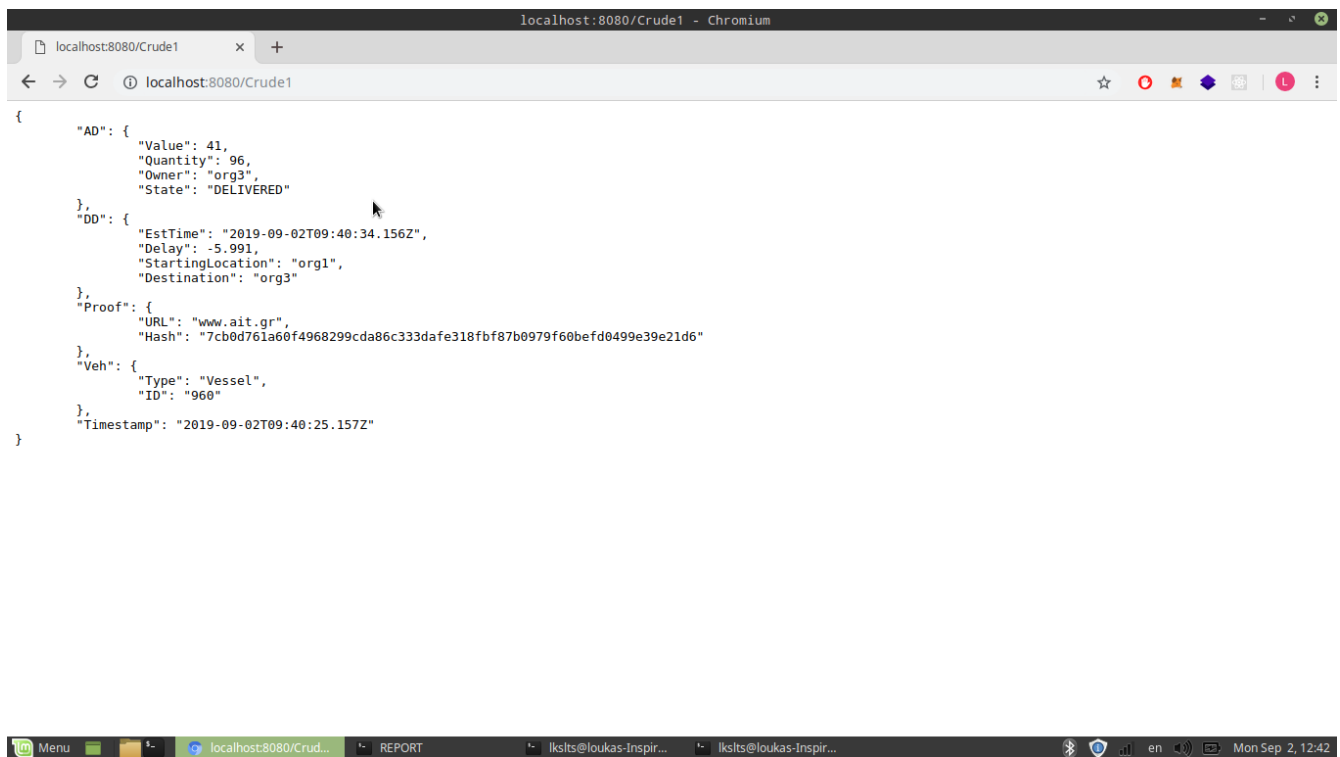
HTTP Server

In order to query/update the ledger, we should run the serve.js file:

```
$ node serve.js
```

This command will start up a server on your machine (localhost) listening at port 8080. We can make GET requests to the server from our browser and query/update the blockchain:

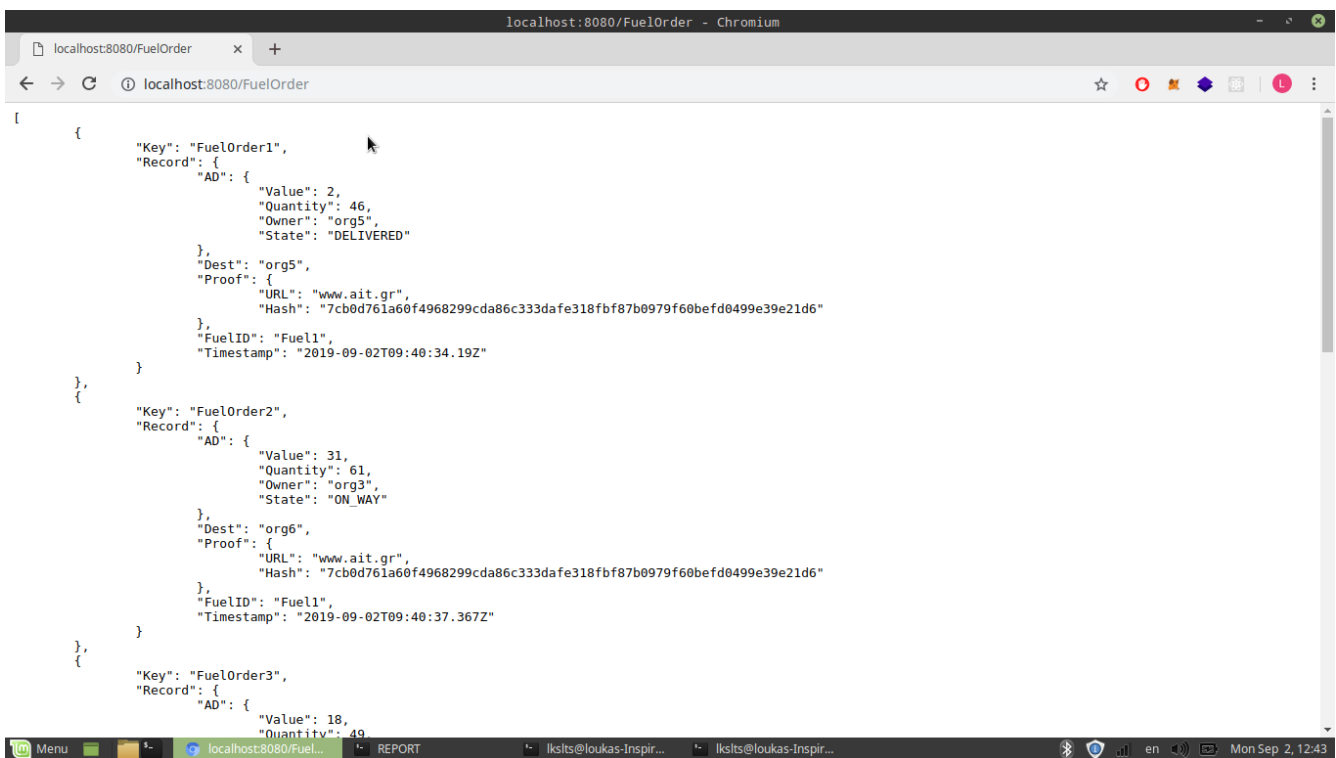
Query for a specific asset : localhost:8080/<Asset><ID> , where Asset = {Crude,Fuel,FuelOrder,Plan} and ID is a positive integer.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/Crude1'. The page content is a JSON object representing a crude oil asset. The JSON structure includes an 'AD' (Asset Data) object with 'Value', 'Quantity', 'Owner', and 'State' fields; a 'DD' (Delivery Data) object with 'EstTime', 'Delay', 'StartingLocation', and 'Destination' fields; a 'Proof' object with 'URL' and 'Hash' fields; and a 'Veh' (Vehicle) object with 'Type' and 'ID' fields. The 'Timestamp' field at the bottom indicates the time of the query.

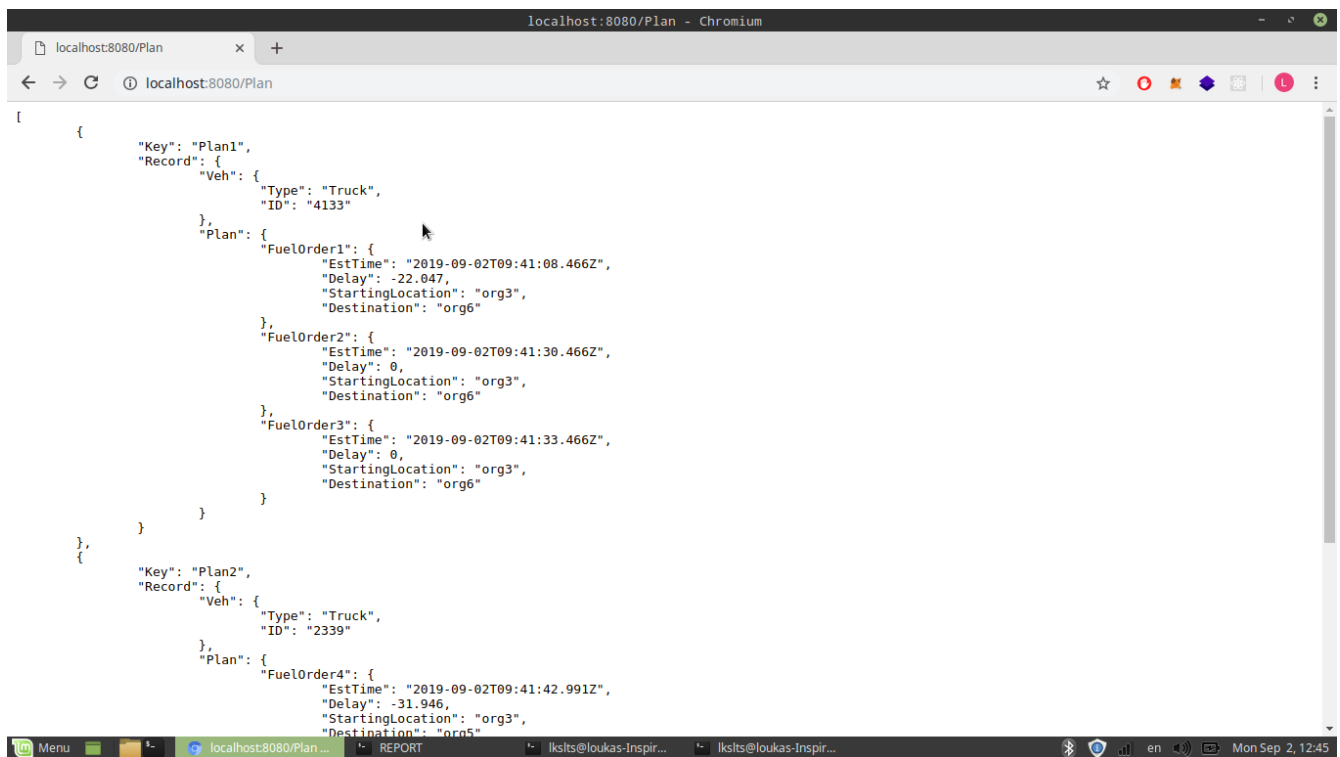
```
{
  "AD": {
    "Value": 41,
    "Quantity": 96,
    "Owner": "org3",
    "State": "DELIVERED"
  },
  "DD": {
    "EstTime": "2019-09-02T09:40:34.156Z",
    "Delay": -5.991,
    "StartingLocation": "org1",
    "Destination": "org3"
  },
  "Proof": {
    "URL": "www.ait.gr",
    "Hash": "7cb0d761a60f4968299cda86c333dfe318fbf87b0979f60befd0499e39e21d6"
  },
  "Veh": {
    "Type": "Vessel",
    "ID": "960"
  },
  "Timestamp": "2019-09-02T09:40:25.157Z"
}
```

- Query a range of assets /<Asset> . Asset here is defined as above.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/FuelOrder'. The main content area displays a JSON array of three fuel order records. The first record, 'FuelOrder1', has a value of 2, quantity of 46, and is in a 'DELIVERED' state. The second record, 'FuelOrder2', has a value of 31, quantity of 61, and is in an 'ON_WAY' state. The third record, 'FuelOrder3', has a value of 18 and a quantity of 49. Each record includes a destination, a proof URL, a proof hash, a fuel ID, and a timestamp.

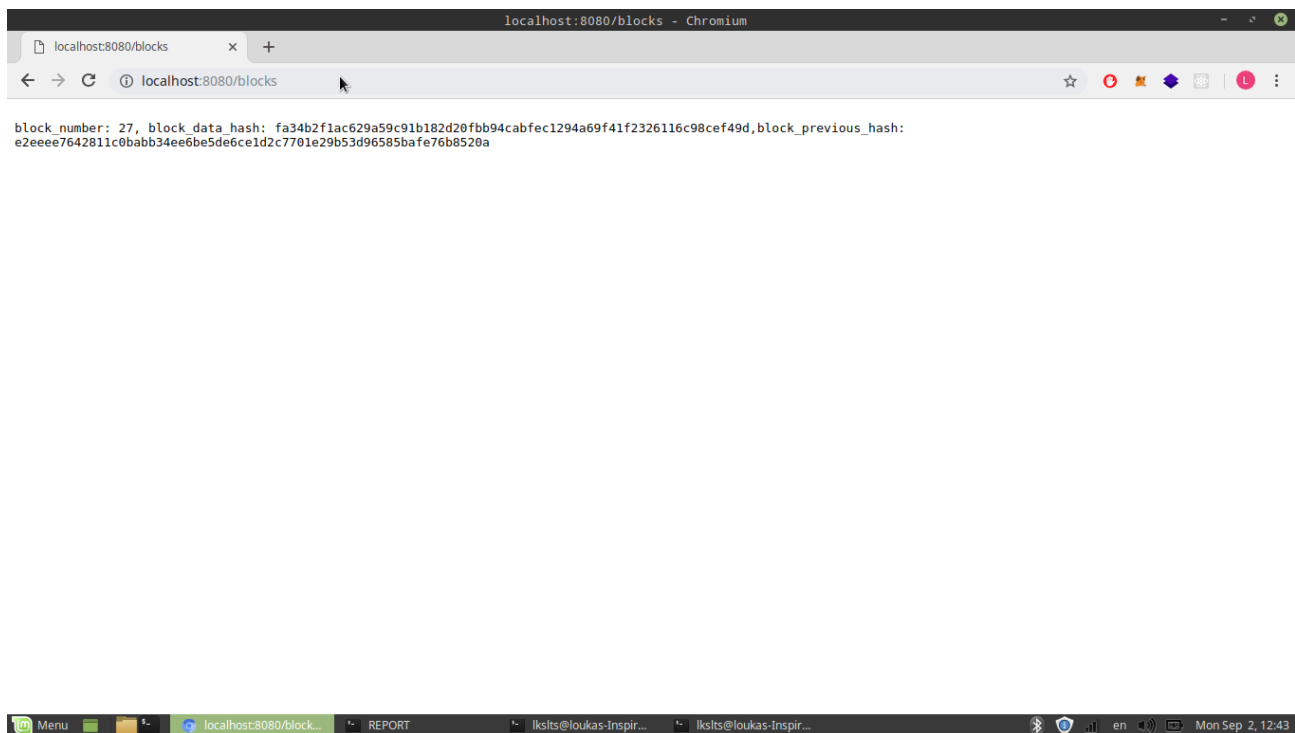
```
[
  {
    "Key": "FuelOrder1",
    "Record": {
      "AD": {
        "Value": 2,
        "Quantity": 46,
        "Owner": "org5",
        "State": "DELIVERED"
      },
      "Dest": "org5",
      "Proof": {
        "URL": "www.ait.gr",
        "Hash": "7cb0d761a60f4968299cda86c333dafa318fbf87b0979f60befd0499e39e21d6"
      },
      "FuelID": "Fuel1",
      "Timestamp": "2019-09-02T09:40:34.19Z"
    }
  },
  {
    "Key": "FuelOrder2",
    "Record": {
      "AD": {
        "Value": 31,
        "Quantity": 61,
        "Owner": "org3",
        "State": "ON_WAY"
      },
      "Dest": "org6",
      "Proof": {
        "URL": "www.ait.gr",
        "Hash": "7cb0d761a60f4968299cda86c333dafa318fbf87b0979f60befd0499e39e21d6"
      },
      "FuelID": "Fuel1",
      "Timestamp": "2019-09-02T09:40:37.367Z"
    }
  },
  {
    "Key": "FuelOrder3",
    "Record": {
      "AD": {
        "Value": 18,
        "Quantity": 49
      }
    }
  }
]
```

The screenshot shows a web browser window with the address bar displaying 'localhost:8080/Plan'. The page content is a JSON array containing two objects, each representing a plan. The first object has a 'Key' of 'Plan1' and a 'Record' object containing a 'Veh' object (Type: 'Truck', ID: '4133') and a 'Plan' object. The 'Plan' object contains three 'FuelOrder' objects with details like 'EstTime', 'Delay', 'StartingLocation', and 'Destination'. The second object has a 'Key' of 'Plan2' and a 'Record' object containing a 'Veh' object (Type: 'Truck', ID: '2339') and a 'Plan' object with one 'FuelOrder' object.

```
[
  {
    "Key": "Plan1",
    "Record": {
      "Veh": {
        "Type": "Truck",
        "ID": "4133"
      },
      "Plan": {
        "FuelOrder1": {
          "EstTime": "2019-09-02T09:41:08.466Z",
          "Delay": -22.047,
          "StartingLocation": "org3",
          "Destination": "org6"
        },
        "FuelOrder2": {
          "EstTime": "2019-09-02T09:41:30.466Z",
          "Delay": 0,
          "StartingLocation": "org3",
          "Destination": "org6"
        },
        "FuelOrder3": {
          "EstTime": "2019-09-02T09:41:33.466Z",
          "Delay": 0,
          "StartingLocation": "org3",
          "Destination": "org6"
        }
      }
    }
  },
  {
    "Key": "Plan2",
    "Record": {
      "Veh": {
        "Type": "Truck",
        "ID": "2339"
      },
      "Plan": {
        "FuelOrder4": {
          "EstTime": "2019-09-02T09:41:42.991Z",
          "Delay": -31.946,
          "StartingLocation": "org3",
          "Destination": "org5"
        }
      }
    }
  }
]
```

- Query for the last committed block at /blocks



- Query for the balance of a specific account at `/org<ID>` , where `ID = {1,2,3,4,5,6}`.



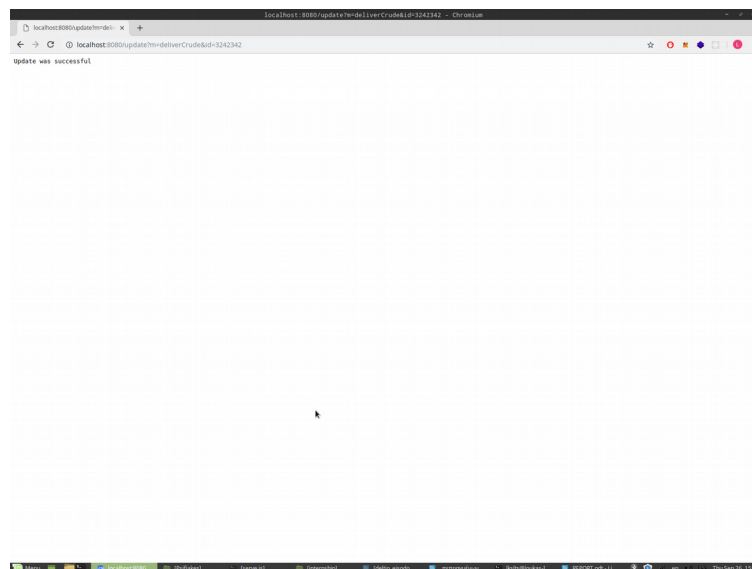
- Update ledger: A client can update the ledger by supplying a URL whose form is :

<http://localhost:8080/update?m=<method>&id=<id>&id2=<id2>&idN=<idN>>

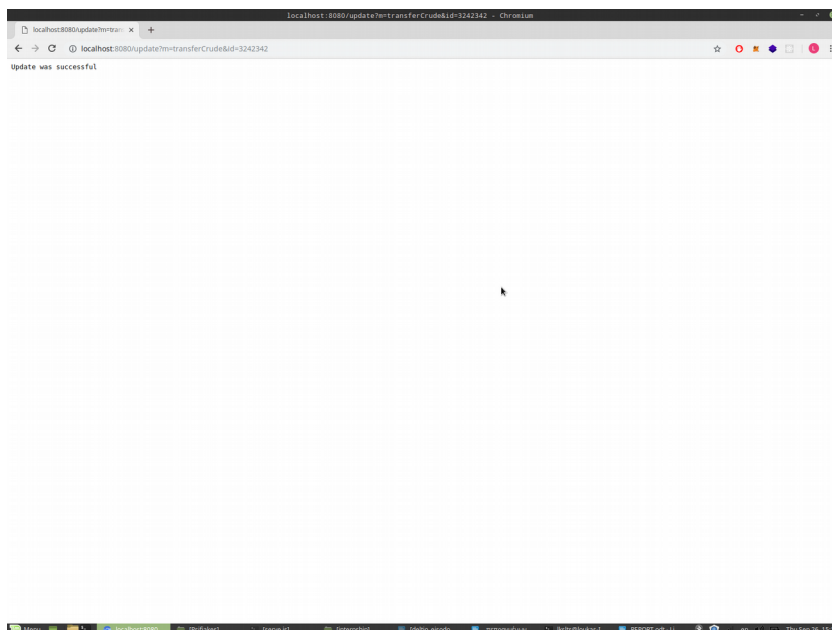
where method is the method name of the Chaincode API and id the ID of the asset that will be inserted or updated. This is the list of the available options for <method> query parameter and its corresponding IDs:

- deliverCrude: needs one more arg that is the ID of the Crude that will be created. Example :
- <http://localhost:8080/update?m=deliverCrude&id=4>
- transferCrude: needs one more arg that is the ID of the Crude that will be transfered. Example:
<http://localhost:8080/update?m=transferCrude&id=7>
- refineRand: needs two more args that is the ID of the Crude that will be transformed to fuel and the ID of the newly created fuel (the Crude ID should pre-exist in the database).Example:
<http://localhost:8080/update?m=refineRand&id=43&id2=654>
- addFuelOrderRand: needs two more args that is the ID of the refined fuel that the order originated and the ID of the newly created fuel order(the fuel ID should pre-exist in the database).Example:
<http://localhost:8080/update?m=addFuelOrderRand&id=534&id2=87>
- deliverFuelRand: can have infinite arguments. The first one should be the ID of the newly created Plan and the rest should be the FuelOrder IDs that the Delivery Plan will contain (FuelOrderIDs should pre-exist in db).
- transferFuel: should have two args . The first one should be the FuelOrderID and the second the PlanID (both args should pre-exist in db). Example:
<http://localhost:8080/update?m=transferFuel&id=132&id2=423>

→ Create and deliver crude



➔ Transfer the crude to the Refiner



→ Ensure Crude3242342 exists in the database

```
localhost:8080/Crude - Chromium
localhost:8080/Crude
[
  {
    "Key": "Crude1",
    "Record": {
      "AD": {
        "Value": 62,
        "Quantity": 35,
        "Owner": "org1",
        "State": "ON_WAY"
      },
      "DD": {
        "EstTime": "2019-09-26T11:07:58.32Z",
        "Delay": 0,
        "StartingLocation": "org1",
        "Destination": "org3"
      },
      "Proof": {
        "URL": "www.ait.gr",
        "Hash": "7cb8d761a60f4968299cda86c333dafa318fbf87b0979f60befd0499e39e21d6"
      },
      "Veh": {
        "Type": "Vessel",
        "ID": "488"
      },
      "Timestamp": "2019-09-26T11:07:15.321Z"
    }
  },
  {
    "Key": "Crude3242342",
    "Record": {
      "AD": {
        "Value": 65,
        "Quantity": 93,
        "Owner": "org3",
        "State": "DELIVERED"
      },
      "DD": {
        "EstTime": "2019-09-26T12:06:59.776Z",
        "Delay": 229.576,
        "StartingLocation": "org1",
        "Destination": "org3"
      },
      "Proof": {
        "URL": "www.ait.gr",
        "Hash": "7cb8d761a60f4968299cda86c333dafa318fbf87b0979f60befd0499e39e21d6"
      },
      "Veh": {
        "Type": "Vessel",
        "ID": "582"
      },
      "Timestamp": "2019-09-26T12:10:49.352Z"
    }
  }
]
```

Installation

Prerequisites:

- Hyperledger fabric 1.4

In order to build the network (Debian platforms) :

- 1) clone this repo under fabric samples directory
- 2) copy chaincode directory under fabric-samples/chaincode/
- 3) navigate under first-network directory
- 4) ./byfn up
- 5) docker exec -it cli bash
- 6) cd scripts && ./upgrade.sh 1.1

In order to make transactions and query the network with the SDK:

- 1) navigate under app/application directory.
- 2) npm install
- 3) run: node addToWallet.js
- 4) node init.js
- 5) Now you are ready to transact with the blockchain.
- 6) Run issue.js to update the blockchain and after serve.js to query/update the blockchain.