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 (particularly Hyperledger Fabric) to build an efficient and reliable supply chain network. Specifically, in this project 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 seperate 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.

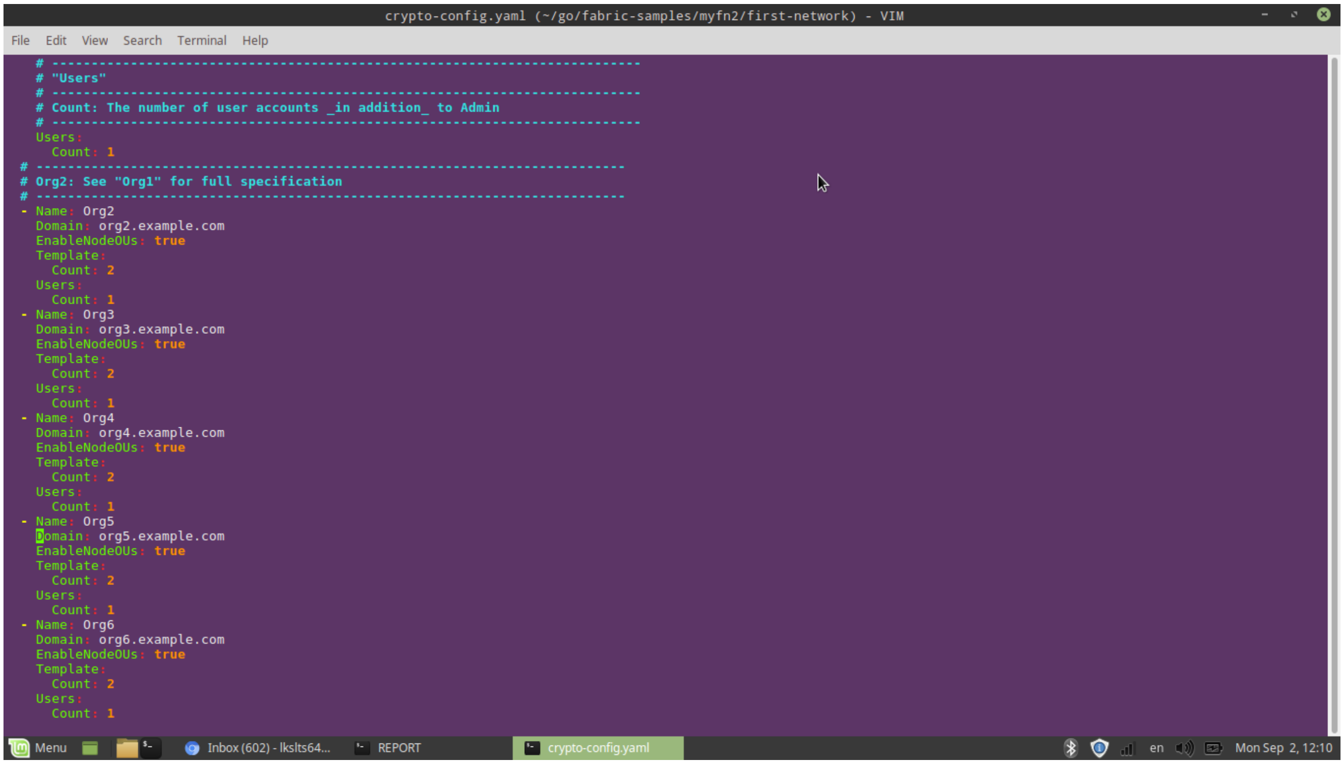
Implementation

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.

Building the network

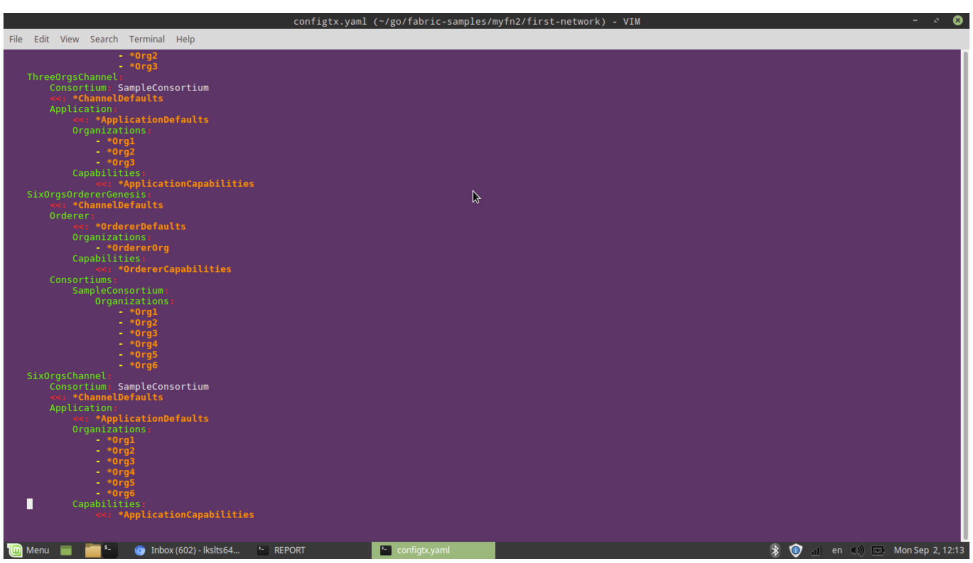
In order to launch a fabric network with six organizations, we made some changes at the byfn tutorial files. Specifically, the tutorial builds a network with 2 organizations so we added 4 orgs on top of the two existing ones. The configuration files that we've changed are:

- crypto-config.yaml

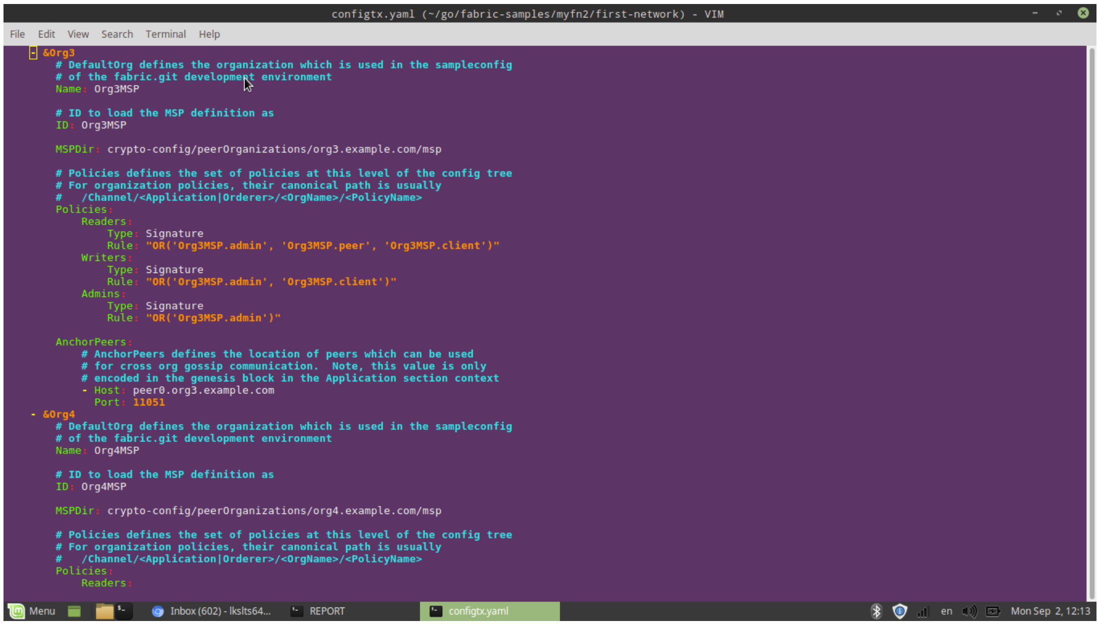


The Template field specifies how many peers each organization will have.

- configtx.yaml:



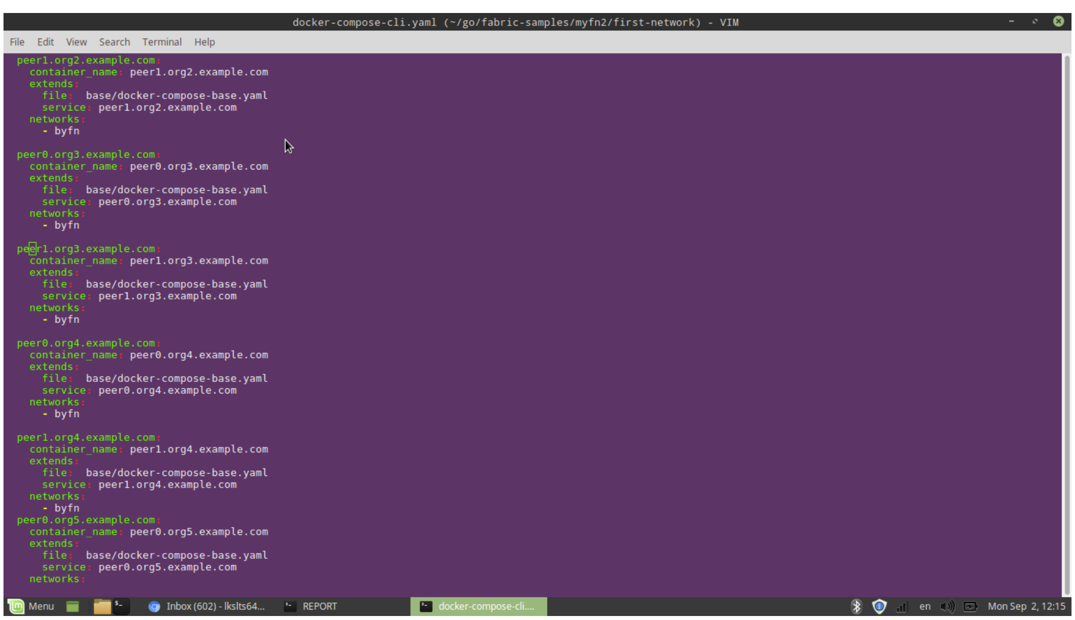
We added SixOrgsOrdererGenesis and SixOrgsChannel fields



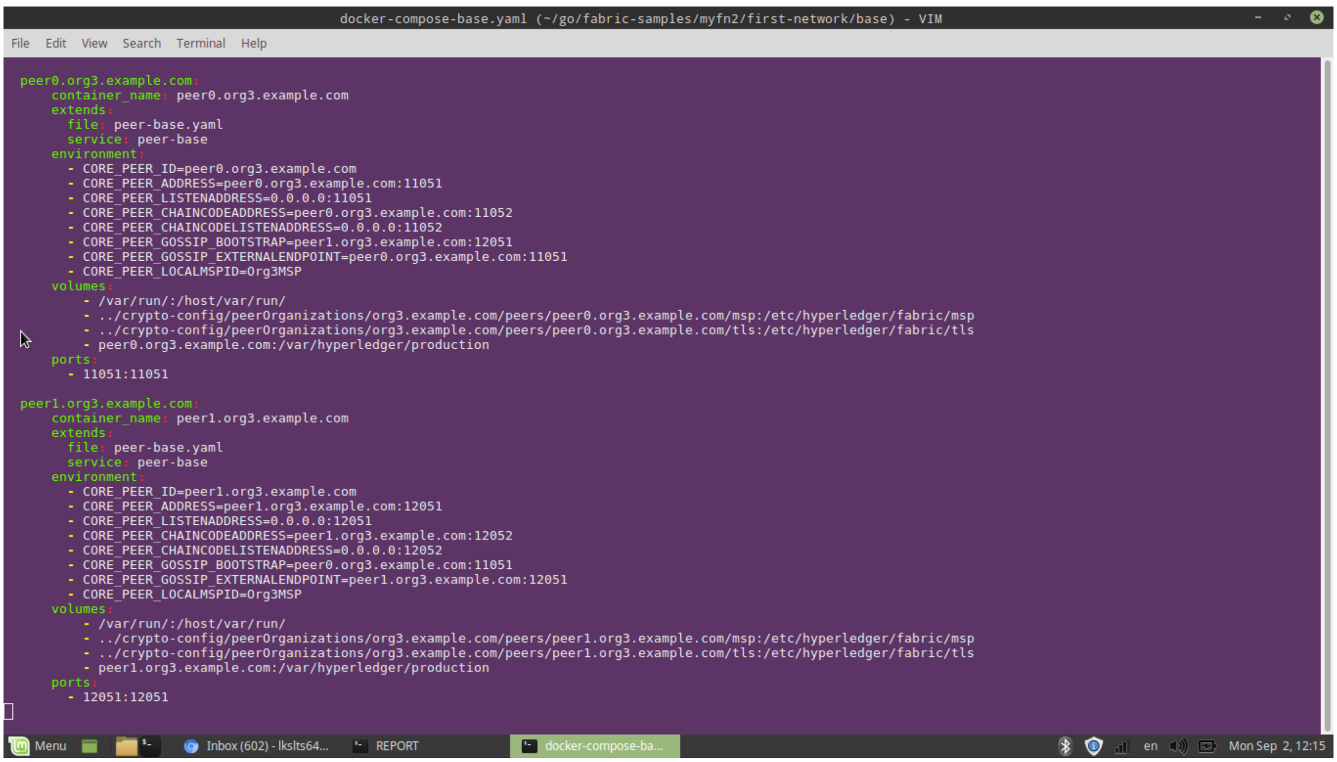
We defined the organizations 3-6.

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

- cli.yaml



- base.yaml



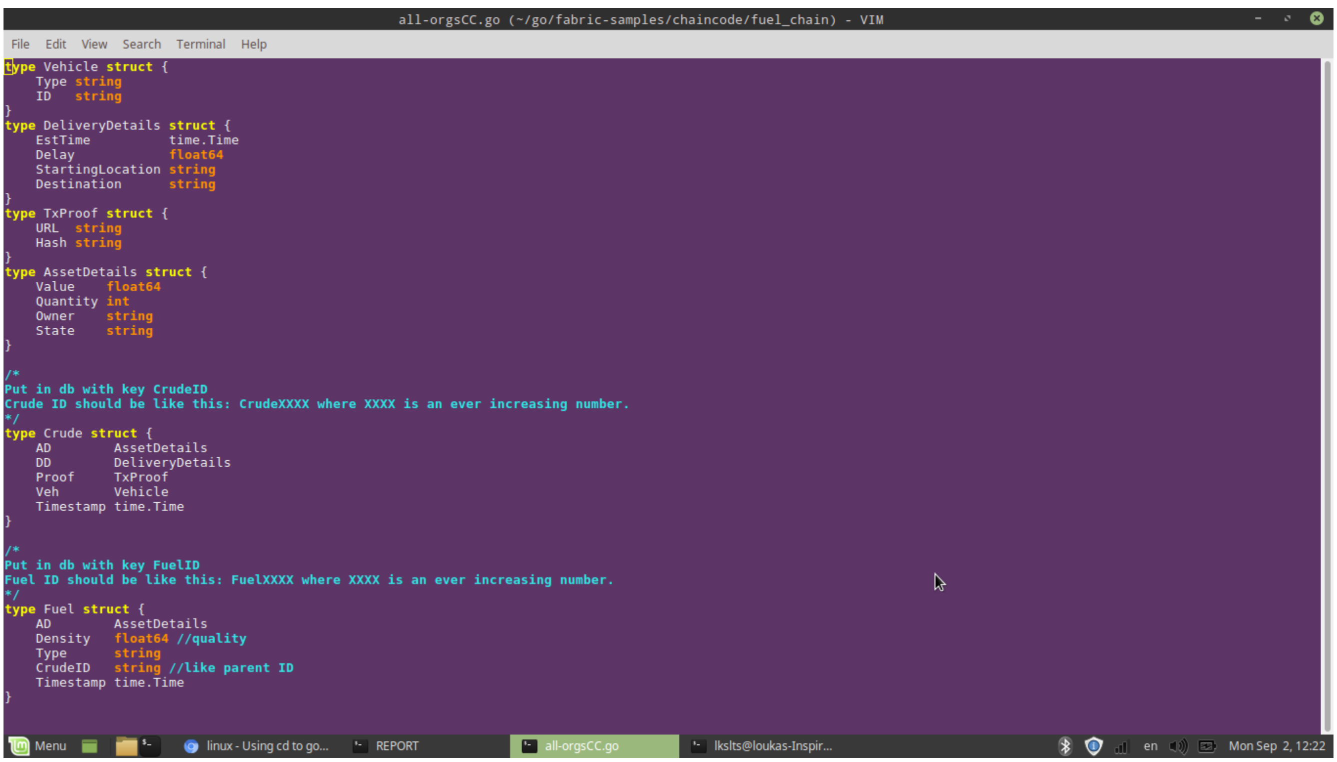
In essence, we added 4 orgs with the same configuration as the two prevísously existing ones, so we just extended the network participants but not its funcionality.

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.

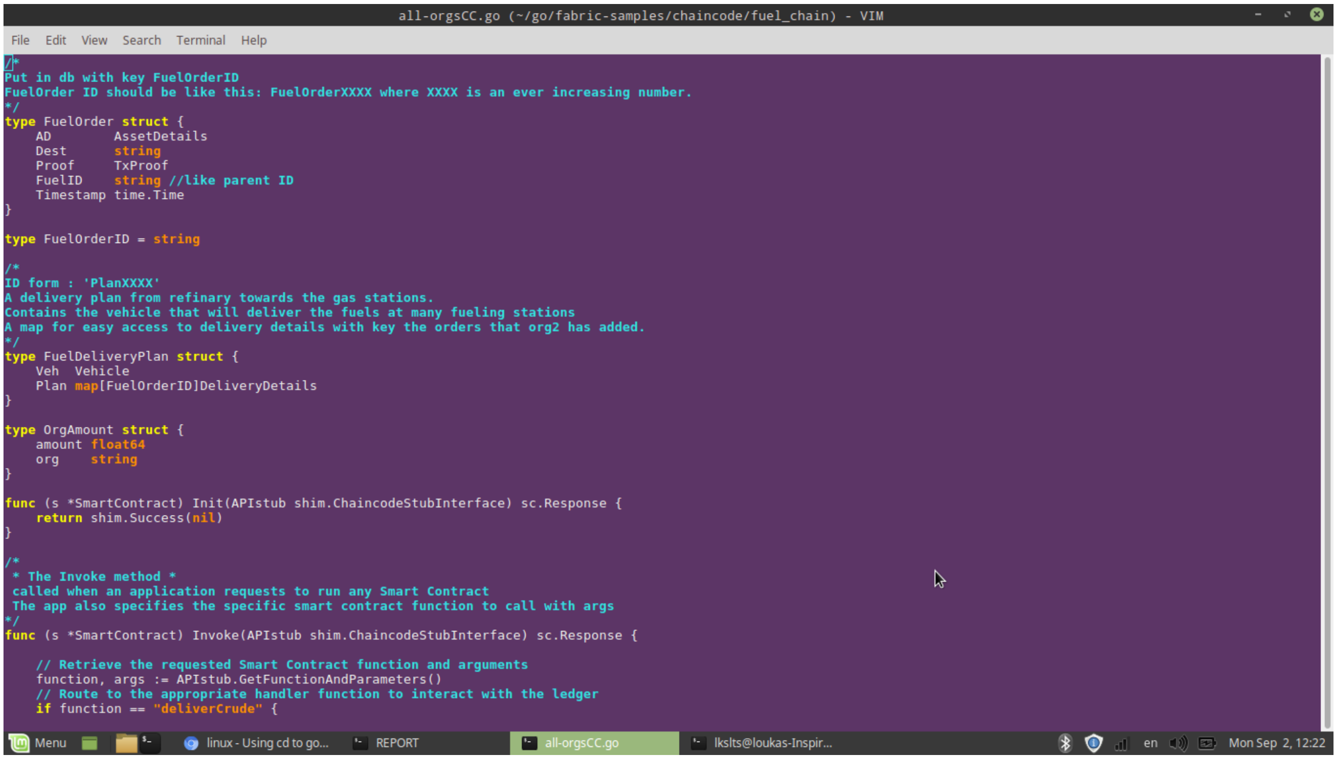
- Structs



```
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
}
```



- 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 e.g. crude oil. So, this struct cointains 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 "7cb0d761a60f4968299cda86c333dafe318fbf87b0979f60befd0499e39e21d6" (the SHA-256 Hash of 'www.ait.gr') respectively.
- Fuel Delivery Plan: A delivery plan from the refinary 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 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 (100000) . 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 take 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 invokation, a new fuelOrder will be created and stored at 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 submited 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 quering a range of assets of the same type (e.g. Crude1-999, Plan1-999).
- queryHistoryForKey: API for quering a single asset for its complete update history in the database (maybe there are some bugs at this method).

Transact with the network

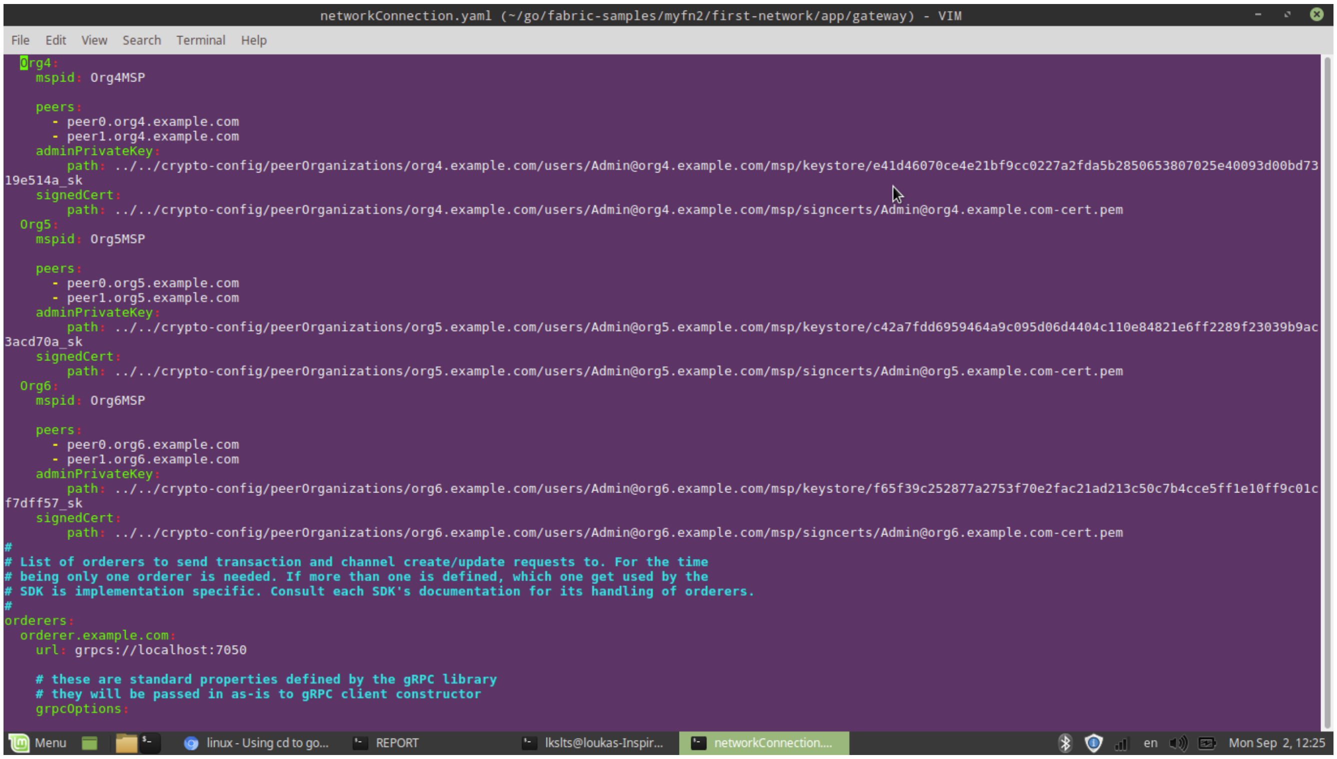
Note: All instructions should be run under first-network/app/application/ directory

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:



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)

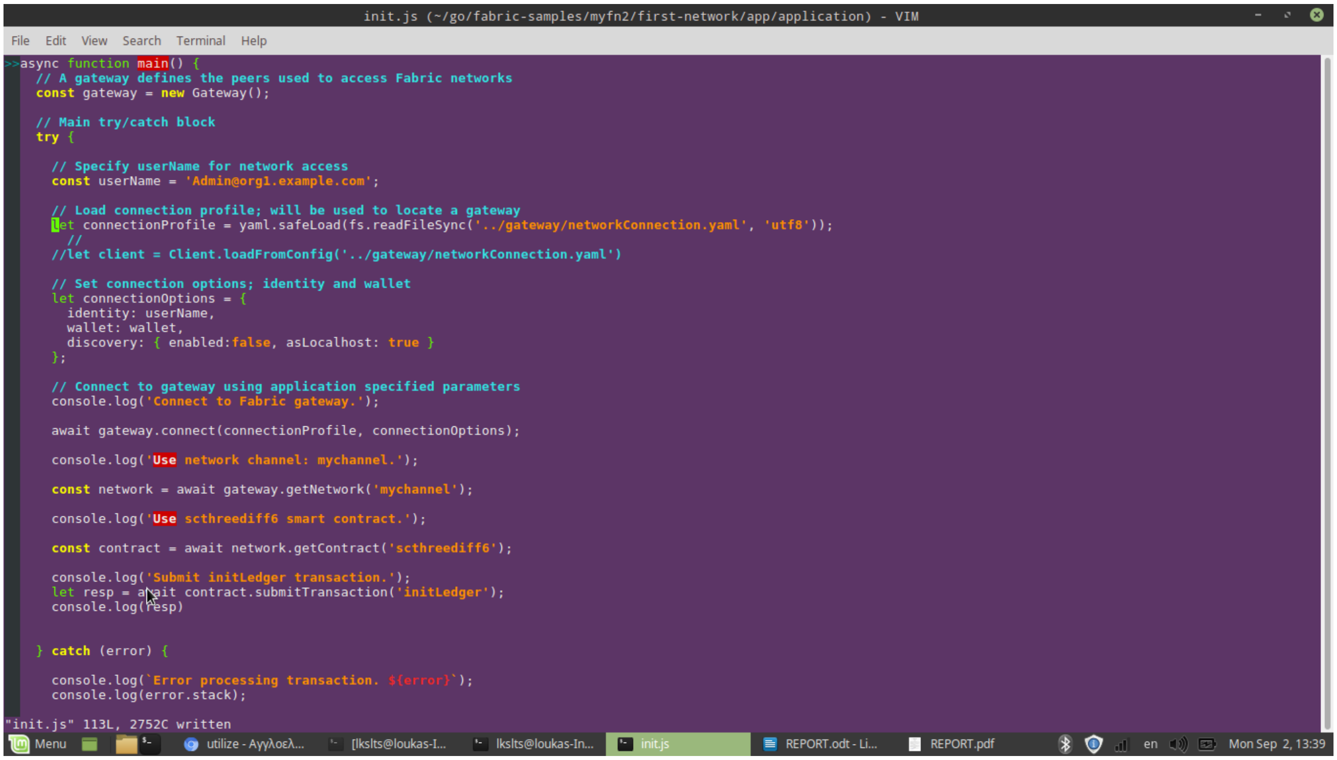
<u>Update the ledger</u>

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:



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:

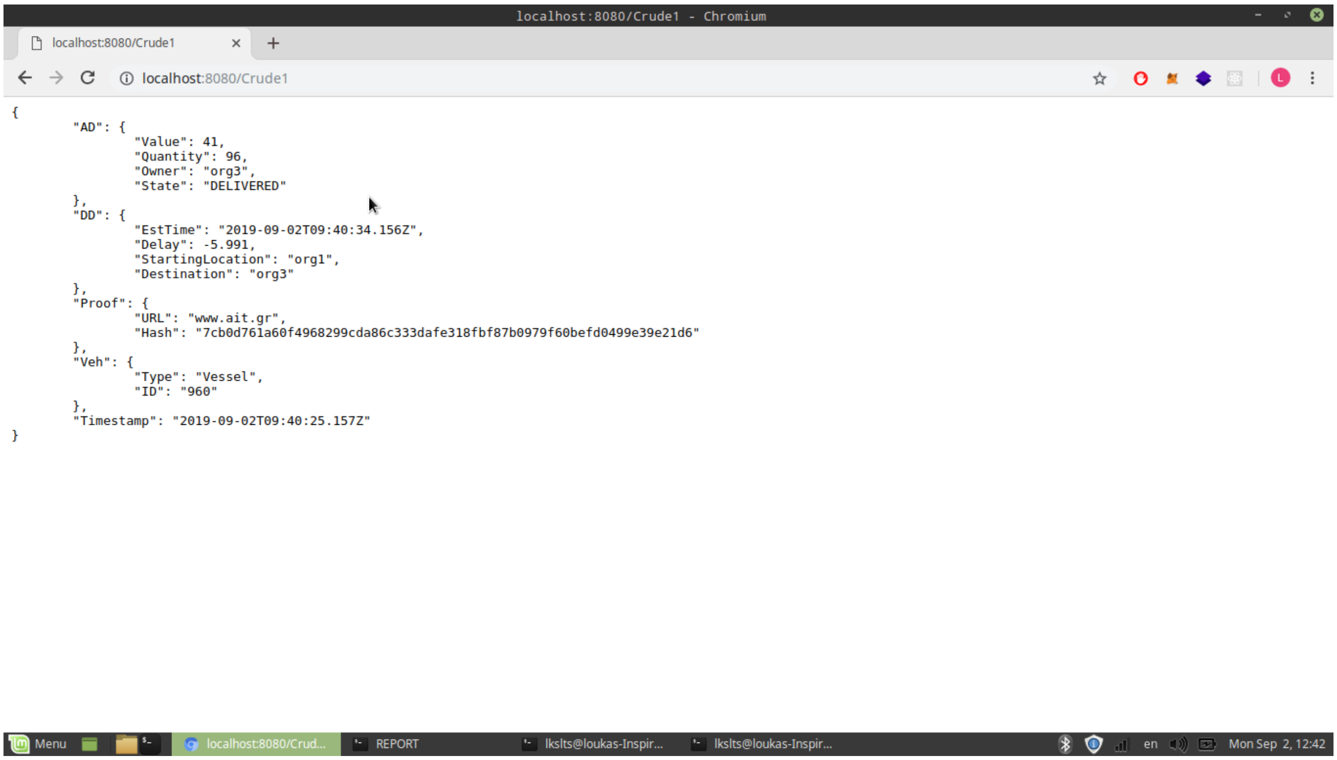
- deliverCrude: needs one more arg that is the ID of the Crude that will be created. Example :
  $ node issue.js deliverCrude 9842
- transferCrude: needs one more arg that is the ID of the Crude that will be transfered. Example:
  $ node issue.js transferCrude 4324
- refineRand: needs two more args that 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.

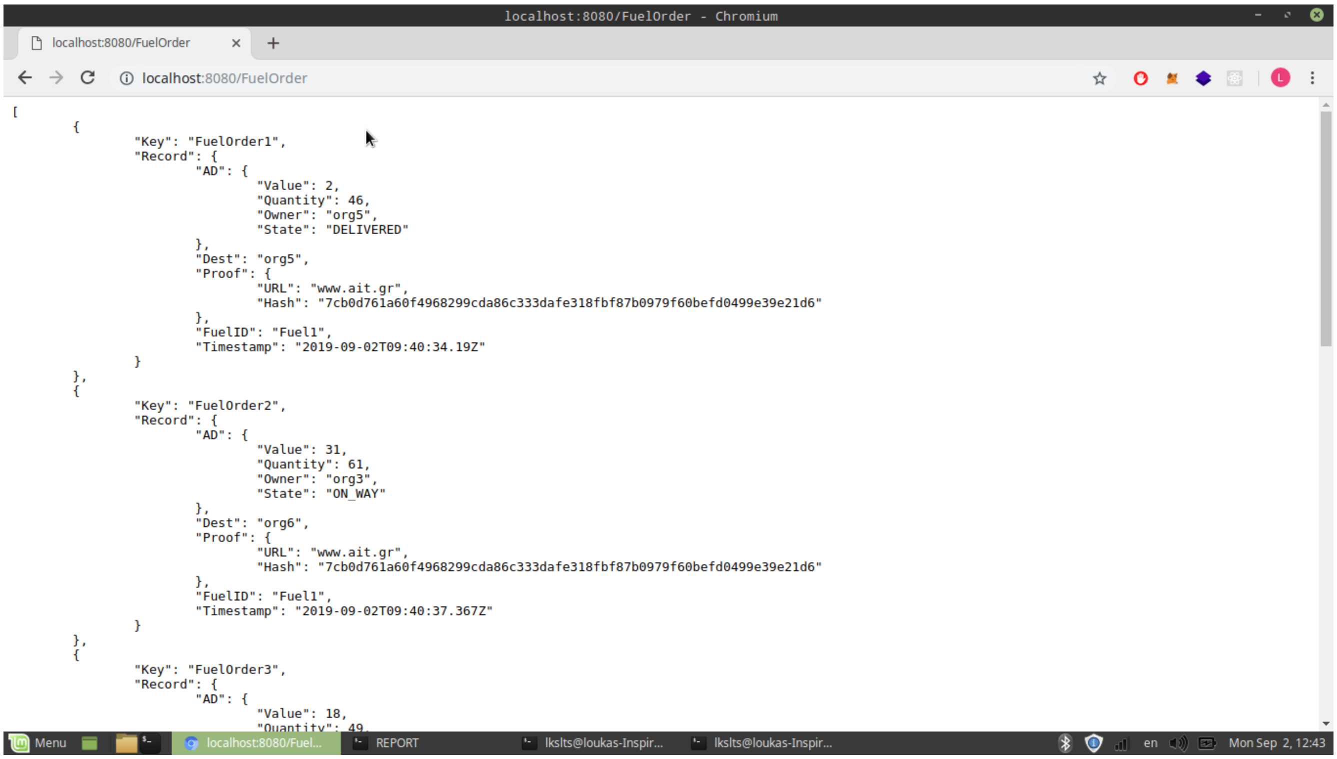
Query the ledger

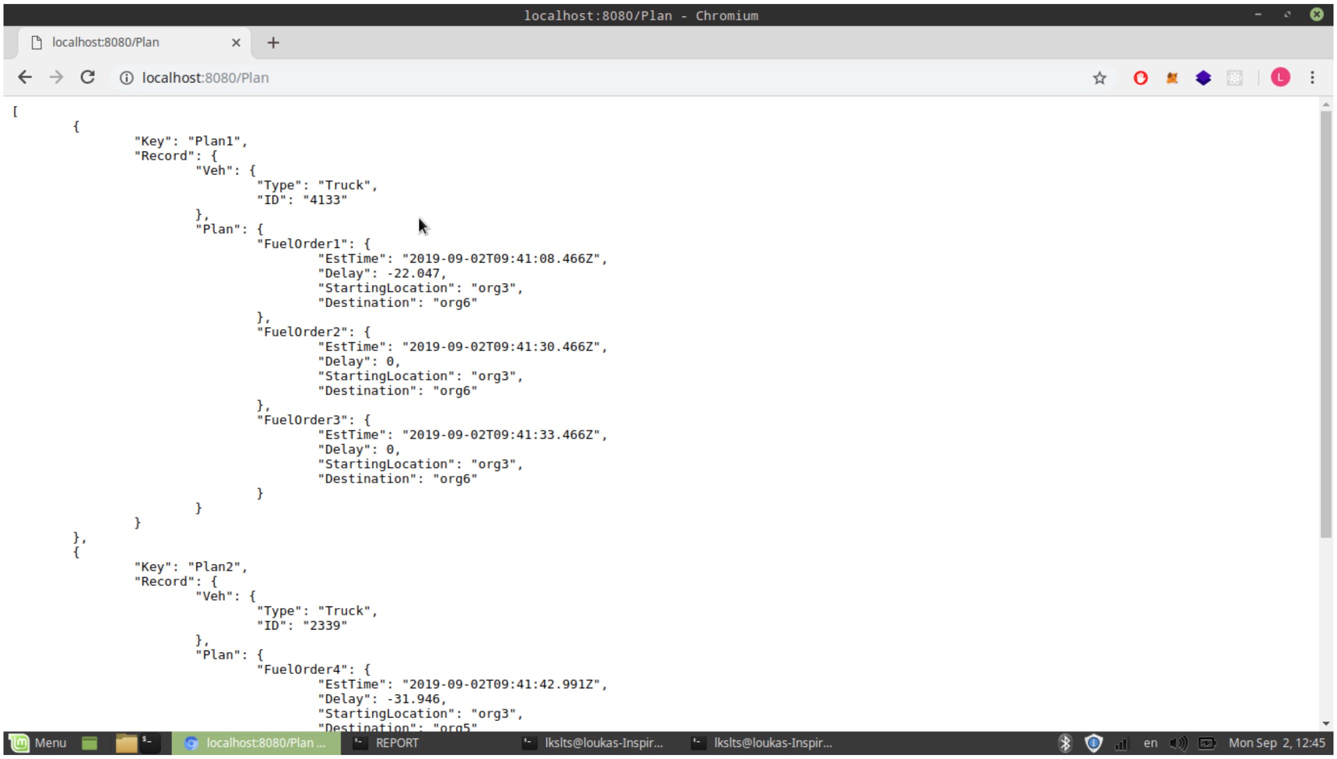
In order to query the ledger, we should run the serve.js file:
$ node serve.js
This command will start up a server on your machine (localhost) listening at port 8080.We can make GET requests to the server from our browser and query the blockchain:

- Query a specific asset at /<Asset><ID> . Asset can be any one of {Crude,Plan,Fuel,FuelOrder} and ID can by any positive integer number.

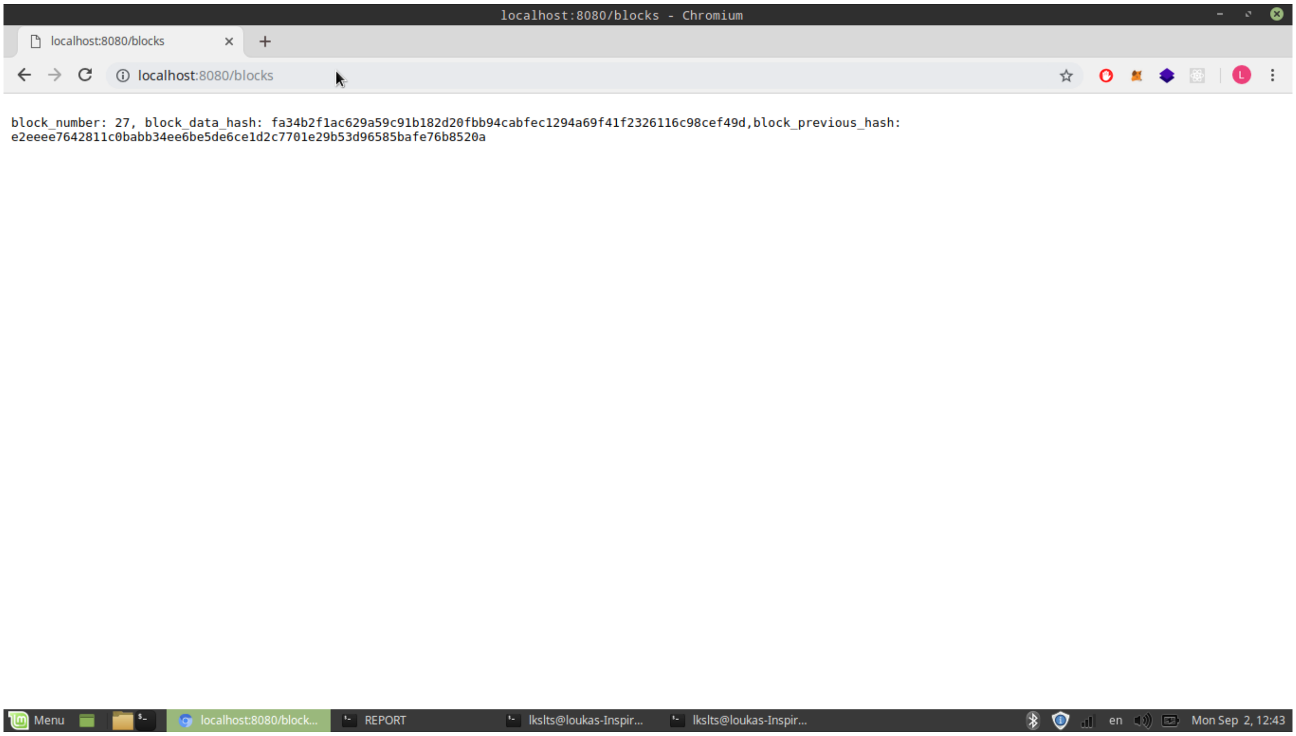


- Query a range of assets /<Asset> . Asset here is defined as above.





- Query for the last commited block at /blocks



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