**CHAINCODE**

Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

Chaincode is the smart contract of hyperledger fabric. It controls the business logic and decides who can invoke particular function in the business network. Chaincode does the logical operations in the blockchain network.

A chaincode may invoke another chaincode either in same channel or in different channel, to access its state. if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. For writing chaincode we have chaincode shim API which consists of all the APIs needed to execute the operations of the chaincode.

[Chaincode Shim API](https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim)

There are 3 important functions in chaincode:

1. Init
2. Invoke
3. Query

**Init:**

Init function is called during initialization or instantiation of data. This function is also called if any update is done on the chaincode.

Init function takes ChaincodeStubInterface as parameter. When we call this function using CLI, it responds back with the peer response.

Syntax:

func (t \*OrderChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response

where pb is the peer which is imported to the chaincode.

Example of Init function:

func (t \*OrderChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

    function, args := stub.GetFunctionAndParameters()

    fmt.Println("init is running " + function)

    OrderId := args[0]

    Buyer := args[1]

    Seller := args[2]

    CurrentLocation := args[3]

    DestinationCity := args[4]

    OriginCity := args[5]

    OrderCondition := args[6]

    Temperature := args[7]

    Humidity := args[8]

    Luminosity := args[9]

    // ==== Create order object and marshal to JSON ====

    objectType := "order"

    order := &order{objectType, OrderId, Buyer, Seller, CurrentLocation, DestinationCity, OriginCity, OrderCondition, Temperature, Humidity, Luminosity}

    orderJSONasBytes, err := json.Marshal(order)

    if err != nil {

        return shim.Error(err.Error())

    }

    err = stub.PutState(OrderId, orderJSONasBytes)

    if err != nil {

        return shim.Error(err.Error())

    }

    fmt.Println("- end init Order")

    return shim.Success(nil)

}

Here OrderChaincode is a struct. Stub.GetFunctionAndParameters take the function name and parameters needed for the initialization. The Marshal function is used to convert the go data types to JSON format. For converting from Json to datatypes UnMarshal is used.

The PutState function is used to store the data to the ledger. It works as a key-value pair. Here the OrderId is the key and the value inside orderJSONasBytes will be stored to ledger.

All the Init function should return shim.Sucess. otherwise it will throw error.

**Invoke:**

Invoke function is like a post call. All the implementation function needed for the chaincode is written here

Syntax:

func (t \*OrderChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response

Example of Invoke:

func (t \*OrderChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

    function, args := stub.GetFunctionAndParameters()

    fmt.Println("invoke is running " + function)

    // Handle different functions

    if function == "registerOrder" {

        return t.registerOrder(stub, args)

    } else if function == "getOrderDetails" {

        return t.getOrderDetails(stub, args)

    }

    fmt.Println("invoke did not find func: " + function) //error

    return shim.Error("Received unknown function invocation")

}

Here we have written the implementation needed for this code. This functions will be implemented separately. We can also write the complete implementation inside the above function. Example of registerOrder is below.

Here we have registered an order with a unique OrderId

func (t \*OrderChaincode) registerOrder(stub shim.ChaincodeStubInterface, args []string) pb.Response {

    var err error

    if len(args) != 10 {

        return shim.Error("Incorrect number of arguments. Expecting 10")

    }

    // ==== Input sanitation ====

    fmt.Println("- start register order")

    OrderId := args[0]

    Buyer := args[1]

    Seller := args[2]

    CurrentLocation := args[3]

    DestinationCity := args[4]

    OriginCity := args[5]

    OrderCondition := args[6]

    Temperature := args[7]

    Humidity := args[8]

    Luminosity := args[9]

    // ==== Check if order already exists ====

    orderAsBytes, err := stub.GetState(OrderId)

    if err != nil {

        return shim.Error("Failed to register order: " + err.Error())

    } else if orderAsBytes != nil {

        fmt.Println("This order already exists: " + OrderId)

        return shim.Error("This order already exists: " + OrderId)

    }

    // ==== Create order object and marshal to JSON ====

    objectType := "order"

    order := &order{objectType, OrderId, Buyer, Seller, CurrentLocation, DestinationCity, OriginCity, OrderCondition, Temperature, Humidity, Luminosity}

    fmt.Println(order)

    orderJSONasBytes, err := json.Marshal(order)

    fmt.Println(orderJSONasBytes)

    if err != nil {

        return shim.Error(err.Error())

    }

    // === Save order to state ===

    err = stub.PutState(OrderId, orderJSONasBytes)

    if err != nil {

        return shim.Error(err.Error())

    }

    fmt.Println("- end register order")

    return shim.Success(nil)

}

**Query:**

Query returns the string which will be array of results. This function is used to query the ledger and, filter results based on particular logic etc.

Example of query function:

func (t \*OrderChaincode) getOrderDetails(stub shim.ChaincodeStubInterface, args []string) pb.Response {

    var OrderId, jsonResp string

    var err error

    if len(args) != 1 {

        return shim.Error("Incorrect number of arguments. Expecting name of the marble to query")

    }

    OrderId = args[0]

    valAsbytes, err := stub.GetState(OrderId)

    if err != nil {

        jsonResp = "{\"Error\":\"Failed to get state for " + OrderId + "\"}"

        return shim.Error(jsonResp)

    } else if valAsbytes == nil {

        jsonResp = "{\"Error\":\"Order does not exist: " + OrderId + "\"}"

        return shim.Error(jsonResp)

    }

    return shim.Success(valAsbytes)

}

We can query the ledger to retrieve the value for the ledger. Here stub.GetState function is used. We have to pass the key (Here it is OrderId) and the GetState function will retrieve the value form the ledger. We will get the output as a JSON array in this case.

We have to import the shim package and other necessary packages while writing the code. Example of import look like below.

import (

    "bytes"

    "encoding/json"

    "fmt"

    "strconv"

    "time"

    "github.com/hyperledger/fabric/core/chaincode/shim"

    pb "github.com/hyperledger/fabric/protos/peer"

)

STEPS TO RUN THE CHAINCODE USING THE INSTANCE

**Note:** Set the GOPATH to the working directory. We can check the environment variables using the command go env or echo $GOPATH.

Make a directory for the current program and put the chaincode.

1. Building chaincode:

go get **-**u github**.**com**/**hyperledger**/**fabric**/**core**/**chaincode**/**shim

go build

1. Install hyperledger Fabric samples

While we work on developing real installers for the Hyperledger Fabric binaries, we provide a script that will download and install samples and binaries to your system.

Determine the directory where the fabric-samples to be cloned and run the following command.

curl **-**sSL http:**//**bit**.**ly**/**2ysbOFE **|** bash **-**s 1.2**.**0

We should add that to your PATH environment variable so that these can be picked up without fully qualifying the path to each binary. e.g.:

export PATH=<path to download location>/bin:$PATH

Navigate to the chaincode-docker-devmode directory of the fabric-samples clone

cd chaincode**-**docker**-**devmode

Now we need to open three terminals, first for stating the docker, second for starting the chaincode and third for running the functions implemented in the chaincode.

**Terminal 1 – Start the network**

docker**-**compose **-**f docker**-**compose**-**simple**.**yaml up

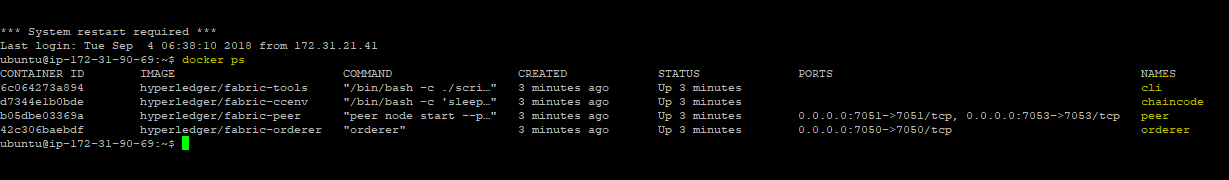
The above starts the network with the SingleSampleMSPSolo orderer profile and launches the peer in “dev mode”. It also launches two additional containers - one for the chaincode environment and a CLI to interact with the chaincode. The commands for create and join channel are embedded in the CLI container. So four containers will be running: 1) orderer 2) peer 3) chaincode 4) CLI

**Note:**

After starting the network, you can check if all the containers are started or not. For the we use the following command.

docker ps

We will be getting the following.



If the above command is not displaying all the containers mentioned in the screenshot we have to stop the docker, delete the old docker images and run the .yaml file again. The following commands will delete the docker images.

docker rm -f $(docker ps -aq)

docker rmi -f $(docker images -q)

**Terminal2 – Build and start the chaincode**

docker exec **-**it chaincode bash

You should see the following:

root@d2629980e76b:**/**opt**/**gopath**/**src**/**chaincode*#*

Now, compile your chaincode using following commands:

cd chaicodedirectoryname

go build

Now run the chaincode:

CORE\_PEER\_ADDRESS**=**peer:7052 CORE\_CHAINCODE\_ID\_NAME**=**mycc:0 **./**painting

The chaincode will be now started. Next step we have to instantiate the chaincode.

**Terminal3 – Use the chaincode**

We can test the chaincode in this step. Run the following commands on the third terminal.

docker exec **-**it cli bash

peer chaincode install -p chaincodedev/chaincode/painting -n mycc -v 0

peer chaincode instantiate -n mycc -v 0 -c '{"Args":["order","1","abc","xyz","kochi","Pune","Kochi","in\_good\_condition","100","25","10"]}' -C myc

peer chaincode invoke -n mycc -c '{"Args":["registerOrder","2","abc","xyz","kochi","Pune","Kochi","in\_good\_condition","100","25","10"]}' -C myc

peer chaincode query -n mycc -c '{"Args":["getOrderDetails","2"]}' -C myc

Note: If in any of the steps it we are getting chaincode version mismatch error, we should delete all the docker containers and restart the docker container by executing the docker-compose.yaml