Chaincode

Chaincode

- Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.
- A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a “smart contract”. State created by a chaincode is scoped exclusively to that chaincode and can’t be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state

- Install Chaincode Using Node.js Chaincode Chaincode is software that encapsulates the business logic and transactioninstructions for creating and modifying assets in the ledger. Chaincode can be written in different languages, and Hyperledger Platform supports: Go and Node.js Chaincode.
- A Chaincode runs in a docker container that is associated with any peer that needs to interact with it. Chaincode is installed on a peer, then instantiated on a channel. All members that want to submit transactions or read data by using a Chaincode need to install the Chaincode on their peer.
- IMPORTANT: A Chaincode is defined by its name and version. Both the name and version of the installed Chaincode need to be consistent across the peers on a channel.

- After Chaincode is installed on the peers, a single network member instantiates the Chaincode on the channel. The network member needs to have joined the channel in order to perform this action.
- If a peer with a Chaincode installed joins a channel where it has already been instantiated, the Chaincode container will start automatically.
- The combination of installation and instantiation is a powerful feature because it allows for a peer to use a single Chaincode across many channels.
- Peers may want to join multiple channels that use the same Chaincode, but with different sets of network members able to access the data.
- A peer can the install the Chaincode once, and then use the same Chaincode container on any channel where it has been instantiated

Chaincode Interface

- Chaincode interface must be implemented by all chaincodes. Below is the syntax of that. As you can see there are two functions defined one is Init and other is Invoke so every chaincode need to implement these functions.

```
type Chaincode interface {
        // Init is called during Instantiate transaction after the chaincode container
        // has been established for the first time, allowing the chaincode to
        // initialize its internal data
        Init(stub ChaincodeStubInterface) pb.Response

        // Invoke is called to update or query the ledger in a proposal transaction.
        // Updated state variables are not committed to the ledger until the
        // transaction is committed.
        Invoke(stub ChaincodeStubInterface) pb.Response
}
```

- **Init**
  Called when a chaincode receives
  an *instantiate* or *upgrade* transaction. This is where you will initialize any application state.
- **Invoke**
  Called when the *invoke* transaction is received to process any transaction proposals.

What is ChaincodeStubInterface ?

- ChaincodeStubInterface provides the functions that are used to access and modify the ledger, and to make invocations between chaincodes. Below is short syntax of that.

```
type ChaincodeStubInterface interface {
        GetArgs() [][]byte
        GetStringArgs() []string
        GetFunctionAndParameters() (string, []string)
        GetArgsSlice() ([]byte, error)
        GetTxID() string
        GetChannelID() string
        InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response
        GetState(key string) ([]byte, error)
        PutState(key string, value []byte) error
        DelState(key string) error
        GetStateValidationParameter(key string) ([]byte, error)
        GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)
        --------
}
```

Main Section of Chaincode

- Import section
- Struct section
- Init function section
- Invoke function section
- Custom functions section
- Main function section

Import Section

```
package main

import (

"fmt"

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

)
```

- **fmt** - contains **Println** for debugging/logging
- **github.com/hyperledger/fabric/core/chaincode/shim** - contains the definition for the chaincode interface and the chaincode stub, which you will need to interact with the ledger, as we described in the *Chaincode Key APIs* section

Struct Section - define the Chaincode type

```
type TestChaincode struct {
}
```

Init function Section

Init function of Chaincode interface is implemented. Init is called during the chaincode instantiation to initialize data required by the application.

```
func (t *TestChaincode) Init(stub shim.ChainCodeStubInterface) peer.Response {
//write required logic here
return shim.Success(nil)
//Below statement can be used in case of error
//return shim.Error("Error Message")
}
```

Invoke function Section

- Invoke function of Chaincode is implemented. Invoke method, which gets called when a transaction is proposed by a client application.

```
func (t *TestChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
// Extract the function and args from the transaction proposal
fn, args := stub.GetFunctionAndParameters()
var result string
var err error
if fn == "setValue" {
result, err = setValue(stub, args)
} else if  fn == "getValue" {
result, err = getValue(stub, args)
}
if err != nil { //Failed to get function and/or arguments from transaction proposal
return shim.Error(err.Error())
}
// Return the result as success payload
return shim.Success([]byte(result))
}
```

GetFunctionAndParameters function from shim package helps to get the function name and arguments to that function.

In this example, SDK can either call setValue() function or getValue() function with required arguments.
function name is getting saved in fn and arguments in args.

Later you can check the function name and based upon that invoke custom functions such as setValue or getValue

Custom function Section

- Invoke function can further call custom defined functions. These custom functions can be defined as per the requirement.

```
func setValue(stub shim.ChaincodeStubInterface, args []string) (string, error) {
if len(args) != 2 {
return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
}
err := stub.PutState(args[0], []byte(args[1]))
if err != nil {
return "", fmt.Errorf("Failed to set values:
{1d6e369f27b3b55fac77a13f30e13d2b04ce9713b9295394d4a423074fd2099f}s", args[0])
}
return args[1], nil
}
```

The *setValue* method will modify the world state to include the key/value pair specified. If the key exists, it will override the value with the new one, using the **PutState** method; otherwise, a new asset will be created with the specified value.

```
func getValue(stub shim.ChaincodeStubInterface, args []string) (string, error) {

if len(args) != 1 {

return "", fmt.Errorf("Incorrect arguments. Expecting a key")

}

value, err := stub.GetState(args[0])

if err != nil {

return "", fmt.Errorf("Failed to get value:
{1d6e369f27b3b55fac77a13f30e13d2b04ce9713b9295394d4a423074fd2099f}s with error:
{1d6e369f27b3b55fac77a13f30e13d2b04ce9713b9295394d4a423074fd2099f}s", args[0], err)

}

if value == nil {

return "", fmt.Errorf("Value not found: {1d6e369f27b3b55fac77a13f30e13d2b04ce9713b9295394d4a423074fd2099f}s",
args[0])

}

return string(value), nil

}
```

getValue method will attempt to retrieve the value for the specified key. If key is not passed in the arguments then error is returned and if key is there then GetState function is called and that in turn query the state database and get the value for that key.

Main function Section

```
func main() {
err := shim.Start(new(TestChaincode))
if err != nil {
fmt.Println("Could not start TestChaincode")
} else {
fmt.Println("TestChaincode successfully started")
}
}
```

Last section is main function section and this is the starting point of chaincode.

https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim

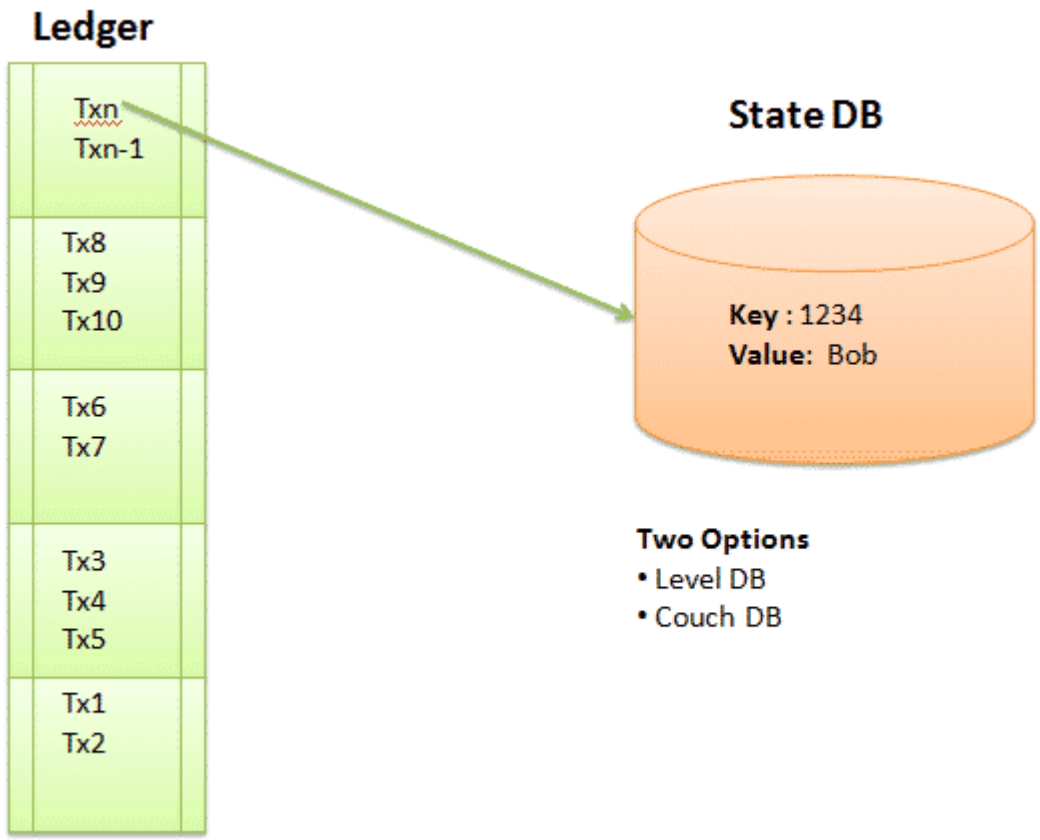
State Database- Level and Couch DB

- One of the major challenge with Ethereum Blockchain is the scale-ability and throughput. Hypeledger Fabric try to address that with state database so this post is related to that.
- You will get to understand about state database, it's usage and types of state db in hyperledger fabric .

State database and why it is needed ?

- Mark purchased new car from company with car Id **1234**. We store that ownership details in blockchain block with key as 1234 and value as Mark. Here key represent the Car Id and value represent the car owner.
- After few months, Mark sell that car to John and we store that change in ownership in blockchain block with key as 1234 and value as John.
- Again after few months John sell that car to Bob and we store that change in ownership in blockchain block with key as 1234 and value as Bob.

- Whenever anyone wants to check who is the current owner of the car with car id 1234, that query will go through the blockchain (all the blocks) and then get the latest value and this query surely going to take some time.
- What if, we maintain a separate storage (database) and store only the latest record (key value pair) in that storage. In this example, this storage only contains latest value which is key as 1234 and value as Bob so that if anyone wants to check current owner of the car, they need not to go to blockchain rather read that from that storage.



Why ?

- As Latest key value pair get stored in state database along with blockchain so all the queries for latest records will be address by state database instead of going to blockchain. This is more efficient process and improve the performance.

Function to interact with State Database

- The ChaincodeStub provides functions that allow you to interact with the underlying ledger to query, update, and delete assets as shown below.
- *func (stub *ChaincodeStub) GetState(key string) ([]byte, error)*
  This function or API helps to get the state from ledger means returns the value of the specified key from the ledger.
- *func (stub *ChaincodeStub) PutState(key string, value []byte) error*
  As the name suggests, this API or function puts the specified key and value into the transaction's Write set as a data-write proposal.
- *func (stub *ChaincodeStub) DelState(key string) error*
  This function helps to delete the specified key in the Write set of the transaction proposal.

DBs

- **Level DB**: This is Key-Value storage library where we store key and value pairs. This helps to create indexes on keys only and does not have option to have indexes on values.

- **Couch DB**: This is documented oriented storage that support rich queries. Another benefit of couch db is that, it support JSON. Couch DB does support index on value to perform rich queries.
- Chaincode interface provides set of functions that you can use for Level DB and separate set of functions for Couch DB.