Migration of Auction Chaincode from v0.6 to v1.0 of Chaincode

Introduction:

The auction test application was developed early last year to understand chaincode development on the Fabric 0.1 or later and support fabric testing. The chain-code has a number of functions such as registering a user, registering an art asset, placing it on auction or transferring its ownership. The following are the list of functions in the application and are representative of how an auction process works.

<table>
<thead>
<tr>
<th>Invoke Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- PostUser: Register a user such as an owner, auction house, bank or shipper</td>
</tr>
<tr>
<td>- PostItem: Register an Art asset owned by a registered user</td>
</tr>
<tr>
<td>- PostAuctionRequest: A user requests an Auction House to place an art item owned by him/her on auction</td>
</tr>
<tr>
<td>- OpenAuctionForBids: An Auction House opens the Auction Request for Bids for a set duration once the asset has been verified and assessed</td>
</tr>
<tr>
<td>- PostBid: A registered buyer (user) places a bid against the asset during the window when the auction is open. All bids are registered</td>
</tr>
<tr>
<td>- CloseOpenAuctions: Check all open auctions that have gone past the open duration and close them (process the bids)</td>
</tr>
<tr>
<td>- PostTransaction: Post an auction transaction based on the bids received against an auction that has closed</td>
</tr>
<tr>
<td>- BuyItNow: If the opened auction allows a “buy-it-now”, then a buyer can place a buy-it-now bid.</td>
</tr>
<tr>
<td>- TransferItem : Transfers an item from one owner to another</td>
</tr>
</tbody>
</table>

In addition to the above functions, there are a number of Query functions:

<table>
<thead>
<tr>
<th>Query Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- GetItem: Retrieves art item information by item-id</td>
</tr>
<tr>
<td>- GetUser: Retrieves user information by user-id</td>
</tr>
<tr>
<td>- GetAuctionRequest: Retrieves an auction record</td>
</tr>
<tr>
<td>- GetTransaction: Fetches the final transaction record posted against an auction</td>
</tr>
<tr>
<td>- GetBid: Retrieve a bid by auction-id/bid-no</td>
</tr>
<tr>
<td>- GetLastBid: Get the last bid posted against an open auction</td>
</tr>
<tr>
<td>- GetHighestBid: Get the highest bid against the auction</td>
</tr>
<tr>
<td>- GetNoOfBidsReceived: Get number of bids received against an auction</td>
</tr>
<tr>
<td>- GetListOfBids: Get a list of all bids received for an auction</td>
</tr>
<tr>
<td>- GetItemLog: Get the asset history (Can use the history feature in Fabric 1.0)</td>
</tr>
<tr>
<td>- GetItemListByCat: Retrieve registered assets by category</td>
</tr>
<tr>
<td>- GetUserListByCat: Get user list by category</td>
</tr>
<tr>
<td>- GetListOfInitAucs: Get a list of auctions waiting to be processed</td>
</tr>
<tr>
<td>- GetListOfOpenAucs: Get a list of auctions that are currently open</td>
</tr>
<tr>
<td>- ValidateItemOwnership: Validate ownership of an asset using the users provided private-key</td>
</tr>
</tbody>
</table>
The application uses custom database tables, a feature available in Fabric 0.6. A set of database functions were written to perform CRUD operations. Some of the pilot applications developed by us use the database API and the concepts were tested while writing this application.

We recently explored how easy would it be to migrate as-is the auction application chaincode to Fabric 1.0. There are a number of new features and architectural advantages provided by Fabric V1.0. After performing an as-is migration, we recommend that the application architecture be reviewed, redesigned, redone or enhanced as appropriate to take advantage of those features.

**Migration from v0.6 to v1.0**

**The code modifications were in 6 areas:**

1. Add to imports: `pb "github.com/hyperledger/fabric/protos/peer``
2. Consolidate “Invoke” and “Query” in 0.6 into “Invoke” in v1.0
3. Change the return values of “Invoke” and “Init” from ([] byte , error) to pb.Response
   a. It is not necessary to replace the return values in every function with pb.Response. It is sufficient to fix the Invoke and Init functions
   b. All stub.FunctionName(args...) return pb.Response
   c. Obtain the function and arguments for “Init” and “Invoke” using "stub.GetFunctionAndParameters()"

   ```
   func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {``

   ```
   function, args := stub.GetFunctionAndParameters()
   ```
4. The Return values inside “Init” and “Invoke” must be either shim.Success([]byte) or shim.Error(...)  
5. Databases tables are not supported, instead use a compound key where the Object(aka table) is the prefix e.g. User100key2 key3 where User is the table name
6. Partial key Queries return iterators and the list of rows (usually in []byte) are extracted in a loop and converted into the appropriate object type.

   In v0.6 version of the auction chaincode, we used a helper function GetList() to return shim.rows whereas, in v1.0 version of the chaincode GetList() returns a result Iterator(resultset). We will need to process the resultset.

**Processing pb.Response**

- `pb.Response` can be processed as follows....

  ```
  Var response  pb.Response
  Response = stub.SomeFunction(... Args ...)
  if response.Status != shim.OK {
      fmt.Println("Query() Object not found : ", args[0])
  ```
```
response_str := "Query() : Object not found : " + args[0]
return shim.Error(response_str)
}

Avalbytes := response.Pay

// Processing Logic
// return response is also valid
Return shim.Success(nil)
```

**Auction Chain Code Modification (art_app.go)**

**INVOKE AND QUERY FUNCTIONS: (ROUGHLY 8 HOURS TO MAKE THE CHANGES TO 3000 LINES OF CODE)**

In the application, we try to avoid the “Case” statements within the “Invoke” and “Query” functions by calling the `InvokeFunction` and `QueryFunction` as shown below in the left column.

- To preserve the same approach and continue to use the existing “Invoke” and “Query” chain code functions, we prefixed the function names with an “i” and a “q” respectively.
- Additionally, the return values were changed from ([] byte, error) to pb.Response

<table>
<thead>
<tr>
<th>0.6</th>
<th>1.0</th>
</tr>
</thead>
</table>
| ```
func InvokeFunction(fname string) func(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte, error) {
InvokeFunc := map[string]func(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte, error) {
  "PostItem": PostItem,
  "PostUser": PostUser,
  "PostAuctionRequest": PostAuctionRequest,
  "PostTransaction": PostTransaction,
  "PostBid": PostBid,
  "OpenAuctionForBids": OpenAuctionForBids,
  "BuyItNow": BuyItNow,
  "TransferItem": TransferItem,
  "CloseAuction": CloseAuction,
  "CloseOpenAuctions": CloseOpenAuctions,
}
return InvokeFunc[fname]
``` | ```
func InvokeFunction(fname string) func(stub shim.ChaincodeStubInterface, function string, args []string) pb.Response {
InvokeFunc := map[string]func(stub shim.ChaincodeStubInterface, function string, args []string) pb.Response {
  "IPostItem": PostItem,
  "IPostUser": PostUser,
  "IPostAuctionRequest": PostAuctionRequest,
  "IPostTransaction": PostTransaction,
  "IPostBid": PostBid,
  "IOpenAuctionForBids": OpenAuctionForBids,
  "IBuyItNow": BuyItNow,
  "ITransferItem": TransferItem,
  "ICloseAuction": CloseAuction,
  "ICloseOpenAuctions": CloseOpenAuctions,
}
return InvokeFunc[fname]
``` |
In Fabric v1.0, both query and invokes are processed by the chaincode API “Invoke”. To preserve the old approach, we renamed the chaincode “Invoke” and “Query” to “invoke” and “query” respectively and inserted a new “Invoke”. This was done to not redo any of the old code that had separate “Invoke” and “Query” functions, each with specific pre/post processing logic. The changes are shown in the following table.
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte, error) {
    ...
    InvokeRequest := InvokeFunction(function)
    if InvokeRequest != nil {
        response := InvokeRequest(stub, function, args)
        return (response)
    }
    ...
}

func (t *SimpleChaincode) Query(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte, error) {
    ...
    QueryRequest := QueryFunction(function)
    if QueryRequest != nil {
        response := QueryRequest(stub, function, args)
        return (response)
    }
    ...
}

changes to 0.6 changes are shown in bold:

- Return value is changes to pb.Response i.e. shim.Success or Shim.Error(..) from ( []byte, error)
- Prefixed the Invoke function names with “i” and Query function names with “q” and retained old logic as is
• Changes to 0.6 are shown in bold

**DATABASE CHANGES: (table_api1.0.go) – ROUGHLY 24 HOURS OF EFFORT TO REWRITE AND TEST**

• In V0.5/0.6, the application specific data was recorded in rocksDB by creating a table i.e TableName and writing key/value as key1, key2..keyn, []byte The []byte was a marshalled JSON structure of the data. With LevelDb and CouchDb, the migration was thus very easy
• All the calls to RocksDb in the 0.5/0.6 used a custom CRUD API we wrote using the chaincode database API. The arguments included ObjectName (aka TableName), keys, arguments or ObjectData. The v1.0 modifications retained the signature but changed the implementation logic. Signature was not changed to support migration of all apps that used our custom table API built using the chaincode supported database API functions like PutState, GetState, etc.
• PutState and GetState is simplified using the composite key approach

```go
compositeKey, _ := stub.CreateCompositeKey(objectName, keys)  //
```

**Initialize, Insert and Update Objects**

- func InitObject(stub shim.ChaincodeStubInterface, objectName string, keys []string) error {
- func UpdateObject(stub shim.ChaincodeStubInterface, objectName string, keys []string, objectData []byte) error {
- func ReplaceObject(stub shim.ChaincodeStubInterface, objectName string, keys []string, objectData []byte) error {

**Delete Objects**

- func DeleteObject(stub shim.ChaincodeStubInterface, objectName string, keys []string) error {
- func DeleteAllObjects(stub shim.ChaincodeStubInterface, objectName string) error {

**Query Object(s)**

- func QueryObject(stub shim.ChaincodeStubInterface, objectName string, keys []string) ([]byte, error) {
- func QueryObjectWithProcessingFunction(stub shim.ChaincodeStubInterface, objectName string, keys []string, fname func(shim.ChaincodeStubInterface, []byte, []string)(error)) ([]byte, error) {

**Query a List of Objects based on Key (Returns an Iterator instead of shim.rows)**

- func GetList(stub shim.ChaincodeStubInterface, objectName string, keys []string) (shim.StateQueryIteratorInterface, error) {

This function verifies if the number of key provided is at least 1
<table>
<thead>
<tr>
<th>Check for Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>- func GetNumberOfKeys(objectName string) int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Check the incoming args for a valid record</th>
</tr>
</thead>
<tbody>
<tr>
<td>- func VerifyObjectArguments(objectName, string, args []string) (string, error)</td>
</tr>
</tbody>
</table>

**Key changes to the database code that implements the above functions:**
<table>
<thead>
<tr>
<th>0.6</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InitLedger</strong></td>
<td><strong>InitObject</strong></td>
</tr>
</tbody>
</table>
| var columnDefsForTbl []*shim.ColumnDefinition  
for i := 0; i < nKeys; i++ {  
columnDef := shim.ColumnDefinition{Name: "keyName" + strconv.Itoa(i),  
Type: shim.ColumnDefinition_STRING, Key: true}  
columnDefsForTbl = append(columnDefsForTbl, &columnDef)}  
columnLastTblDef := shim.ColumnDefinition{Name: "Details", Type:  
shim.ColumnDefinition_BYTES, Key: false}  
columnDefsForTbl = append(columnDefsForTbl, &columnLastTblDef)  
// Create the Table (Nil is returned if the Table exists or if the table is created successfully)  
err := stub.CreateTable(tableName, columnDefsForTbl) | **InitLedger renamed to InitObject.**  
There is no concept of creating tables, so the InitObject function does nothing |
| **UpdateObject, ReplaceObject** | **UpdateObject, ReplaceObject** |
| nKeys := GetNumberOfKeys(tableName)  
if nKeys < 1 {  
fmt.Println("Atleast 1 Key must be provided \n") }  
var columns []*shim.Column  
for i := 0; i < nKeys; i++ {  
col := shim.Column{Value: &shim.Column_String_{String_: keys[i]}}  
columns = append(columns, &col)}  
lastCol := shim.Column{Value: &shim.Column_Bytes{Bytes: []byte(args)}}  
columns = append(columns, &lastCol)  
row := shim.Row{columns}  
ok, err := stub.InsertRow(tableName, row)  
if err != nil {  
return fmt.Errorf("InsertRow into \"table\" Table operation failed. \s", err)  
} | // Check how many keys  
err := VerifyAtLeastOneKeyIsPresent(objectName, keys)  
if err != nil {  
return err  
}  
// Convert keys to compound key  
compositeKey, _ := stub.CreateCompositeKey(objectName, keys)  
// Add Object JSON to state  
err = stub.PutState(compositeKey, objectData)  
if err != nil {  
fmt.Println("UpdateObject() : Error inserting Object into State Database \s", err)  
return err  
}  
return nil |
```go
if !ok {
    return errors.New("UpdateLedger: InsertRow into " + tableName + " Table failed. Row with given key " + keys[0] + " already exists")
}
return nil

func QueryObject() {
    row, err := stub.GetRow(tableName, columns)
    fmt.Println("Length or number of rows retrieved ", len(row.Columns))
    if len(row.Columns) == 0 {
        jsonResp := "{"Error": "Failed retrieving data " + args[0] + ".\n"
        fmt.Println("Error retrieving data record for Key = ", args[0], ", Error : ", jsonResp)
        return nil, errors.New(jsonResp)
    }
    Avalbytes := row.Columns[nCol].GetBytes()
    .......... return Avalbytes, err
}

func GetList() {
    rowChannel, err := stub.GetRows(tableName, columns)
    if err != nil {
        return nil, fmt.Errorf("GetList operation failed. %s", err)
    }
    var rows []shim.Row
    for {
        select {
            case row, ok := <-rowChannel:
                if !ok {
                    rowChannel = nil
                } else {
                    rows = append(rows, row)
                }
        }
    }
    resultIter, err := stub.PartialCompositeKeyQuery(objectName, keys)
    if err != nil {
        return nil, err
    }
    // Return iterator for result set
    // Use code above to retrieve objects
    return resultIter, nil
}
```
if rowChannel == nil {
    break
}
return rows, nil

Iteration Logic - Example of processing the iterator

tlist := make([]UserObject, len(rows))
for i := 0; i < len(rows); i++ {
    ts := rows[i].Columns[nCol].GetBytes()
    uo, err := JSONtoUser(ts)  // Convert bytes to a User Object
    if err != nil {
        return nil, fmt.Errorf("GetUserListByCat() operation failed. %s", err)
    }
    tlist[i] = uo
}

jsonRows, _ := json.Marshal(tlist)
return jsonRows, nil

Iteration Logic - Example of processing the iterator

defer rs.Close()
// Iterate through result set
var i int
for i = 0; rs.HasNext(); i++ {
    // We can process whichever return value is of interest
    myKey, myKeyVal, err := rs.Next()
    if err != nil {
        return shim.Success(nil)
    }
    bob := JSONtoUser(myKeyVal)  // Convert the bytes to a user object
    fmt.Println("GetList() : my Value : ", bob)
}

COMMAND LINE INVOCATION: ROUGHLY 16 HOURS OF TESTING ALL THE FUNCTIONS

The following are examples of changes to the CLI commands. We had to edit all CLI scripts used to invoke or query

V0.6 CLI Command

./peer chaincode invoke -l golang -n mycc -c '{"Function": "PostItem", "Args": ["1000", "ARTINV", "Shadows by Asppen", "Asppen Messer", "20140202", "Original", "Landscape", "Canvas", "15 x 15 in", "sample_7.png","$600", "100", "2016-02-02 03:00:00"]}'

V1.0 CLI Command
./peer chaincode invoke -l golang -n mycc -c '{ "Args": ["PostItem", "1000", "ARTINV", "Shadows by Asppen", "Asppen Messer", "20140202", "Original", "Landscape", "Canvas", "15 x 15 in", "sample_7.png", "$600", "100", "2016-02-02 03:000:00"]}'

**Node-SDK Code Migrations**

The Node-SDK migrations are based on the experience we had from two other applications that we were involved with.

<table>
<thead>
<tr>
<th></th>
<th>0.6</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Installation of the Node-SDK Fabric Client</strong></td>
<td><strong>SOFTWARE VERSIONS:</strong></td>
<td><strong>SOFTWARE VERSIONS:</strong></td>
</tr>
<tr>
<td></td>
<td>• NodeJS runtime version 4.4.x</td>
<td>• NodeJS runtime version 6.9.x ( 7.0 is not supported yet )</td>
</tr>
<tr>
<td></td>
<td>• npm tool version 3.10.x</td>
<td>• npm tool version 3.10.x</td>
</tr>
<tr>
<td></td>
<td>• Docker version 1.12 or higher,</td>
<td>• docker-compose version 1.7 or higher</td>
</tr>
<tr>
<td></td>
<td>• docker-compose version 1.7 or higher</td>
<td>• docker-compose version 1.9.0</td>
</tr>
<tr>
<td><strong>INSTALLATION PROCESS</strong></td>
<td>• npm install hfc@0.6.5</td>
<td>• npm install fabric-client</td>
</tr>
<tr>
<td></td>
<td>• npm install fabric-ca-client</td>
<td>• npm install fabric-ca-client</td>
</tr>
</tbody>
</table>

- Node module version used at the time of this writing is 0.2.1
- Please note that the npm node module name has been changed from hfc which is replaced with the above two modules

**CHAINCODE DEPLOYMENT**

Once the user is enrolled and registered with CA server, then they are allowed to deploy chaincode onto the blockchain network.

NodeSdk will package the chaincode and send it to the peer. The peer will then extract the chaincode and the dependent libs, builds and runs the chaincode on the docker container.

```javascript
chain.enroll(usr.username, usr.secret, function(err, admin) {
  var deployTx = admin.deploy(deployRequest);
  // deploy request contains deploy spec, path to chaincode, function to execute, args to process the initialization etc.,

  // need to enroll it with CA server
})
```

**CHAINCODE DEPLOYMENT**

Once the user is enrolled and registered with CA server, then they are allowed to deploy chaincode onto the blockchain network.

From 1.0 onwards, it is a two-step process in deploying the chaincode

1. Install chaincode on the peer(s)
2. Instantiate the chaincode on a channel (or default channel testchainid)

```javascript
function getSubmitter(){
  var ca_client = new copService(config.caserver.ca_url);
  // need to enroll it with CA server
```
deployTx.on('submitted', function(results) {
    // deployment
    deployTx.on('complete', function(results) {
        // complete deployment
        deployTx.on('error', function(err) {
            // handle error
        });
    });
});

return ca_client.enroll({
    enrollmentID: username,
    enrollmentSecret: password
}).then(() => {
    // deploy chaincode once member is available here
});

// 1. prepare and send install proposal
var nonce = utils.getNonce();
var tx_id = chain.buildTransactionId(nonce, submitter);

// send proposal to endorser
var request = {
    targets: targets,
    chaincodePath: CHAINCODE_PATH,
    chaincodeId: testchainId,
    chaincodeVersion: 1.0,
    txId: tx_id,
    nonce: nonce
};
chain.sendInstallProposal(request);

// 2. prepare and send instantiate proposal to endorser
var request = {
    chaincodePath: chaincodePath,
    chaincodeId: chaincodeID,
   fcn: functionName,
    args: args,
    chaincodeVersion: 1.0,
    chainId: testchainid,
    txId: tx_id,
    nonce: nonce,
};
chain.sendInstantiateProposal(request);
// A registered user is allowed to send invoke transactions on to the blockchain

var invokeRequest = {
  chaincodeID: chaincodeID,
  fcn: request.fcn,
  args: request.args,
};

// Trigger the invoke transaction
var invokeTx = admin.invoke(invokeRequest);

invokeTx.on('complete', function (results) {
  ....
});

invokeTx.on('error', function (err) {
  // Invoke transaction submission failed
});

//verify responses from all endorsers

// A registered user is allowed to send a proposal to peers to obtain the required endorsements
// and then sends the transaction to the orderer service to be put into a block and forwarded
// to the peers to be committed to the ledger.

var promise = getSubmitter(client);
promise = promise.then((admin) => {
  var nonce = utils.getNonce();
  tx_id = chain.buildTransactionID(nonce, admin);
  // send proposal to endorser
  var request = {
    chaincodeId: chaincodeID,
    fcn: functionName,
    args: args,
    chainId: testchainid,
    txId: tx_id,
    nonce: nonce
  };
  // sending proposal to the endorser
  return chain.sendTransactionProposal(request);
}, (err) => {
  ...}
});

//send transaction to the orderer system
chain.sendTransaction(request);
**QUERY A TRANSACTION:**

query chaincode on blockchain

```
var queryRequest = {
    chaincodeID: chaincodeID,
    fcn: function,
    args: ....
};
var queryTx = admin.query(queryRequest);
queryTx.on('complete', function (results) {
    ....
});
queryTx.on('error', function (err) {
    ....
});
```

**QUERY A TRANSACTION:**

query chaincode on blockchain on specific targets (peers)

```
var promise = helper.getSubmitter(client);
promise.then((admin) => {
    var nonce = utils.getNonce();
    var tx_id = chain.buildTransactionID(nonce, admin);
    // chaincode query reque 
    var request = {
        targets: [peer0, peer1 , ...],
        chaincodeId: haincodeID,
        chainId: testchainid,
        txId: tx_id,
        nonce: nonce,
        fcn: functionName,
        args: a 
    };
    // Query chaincode
    var queryPromise = chain.queryByChaincode(request); 
```

**SUMMARY:**

This article is based on intermediate versions of the v1.0 fabric release. The information provided here is based on our desire to understand the potential migration paths available to 0.6 versions of applications. While it is not a big challenge to migrate the chaincode and the node application, the migration of existing data from 0.6 to 1.0 has no clearly defined path. Our view is that data extract scripts can be developed to extract the application specific data from the current data store and insert them into the v1.0 ledger using data migration contracts agreed to by all the participants to verify and validate the extract and load of data into v1.0. However, such an approach does not guarantee that source and target ledgers are identical.