chaincode/chaincode_ad/chaincode_ad.go (66.3%) ▾ not tracked not covered covered

```
package main

import (
	"encoding/json"
	"strconv"
	"strings"

	"github.com/hyperledger/fabric/core/chaincode/shim"
	pb "github.com/hyperledger/fabric/protos/peer"
)

// Chaincode implements Chaincode interface
type Chaincode struct {
}

// Variable names in a struct must be capitalised. Otherwise they are not exported (also to JSON)

// DataEntry represents data created on IoT device
type DataEntry struct {
	RecordType   string // RecordType is used to distinguish the various types of objects in state database
	DataEntryID  string // ID of the entry
	Description  string // human readable description
	Value        string // data value
	Unit         string // optional units for the data value
	CreationTime uint64 // Time when the data was created. It can differ from the blockchain entry time
	Publisher    string // publisher of the data
}

// DataEntryAd - represents data created by publisher and advertised for specific price
type DataEntryAd struct {
	DataEntry        // anonymous field
	Price     int64  // Price for data value
	AccountNo string // account number where to transfer tokens
}

// Main
//////////
func main() {
	// increase max CPU
	// runtime.GOMAXPROCS(runtime.NumCPU())
	err := shim.Start(new(Chaincode))
	if err != nil {
		shim.Error(err.Error())
	}
}

// Init initializes chaincode
////////////////////////////
func (cc *Chaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
	return shim.Success(nil)
}

// Invoke - Our entry point for Invocations
////////////////////////////////////////
func (cc *Chaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
	function, args := stub.GetFunctionAndParameters()

	// Handle functions
	if function == "createDataEntryAd" { //create a new data entry
		return cc.createDataEntryAd(stub, args)
	} else if function == "getDataAdByIDAndTime" { //read specific data by DataEntryID and creationTime
		return cc.getDataAdByIDAndTime(stub, args)
	} else if function == "getAllDataAdByID" { // invoke other chaincode and reveal values
		return cc.getAllDataAdByID(stub, args)
	} else if function == "getLatestDataAdByID" { // invoke other chaincode and reveal values
		return cc.getLatestDataAdByID(stub, args)
	} else if function == "getDataAdByPub" { //find data created by publisher using compound key
		return cc.getDataAdByPub(stub, args)
	} else if function == "revealPaidData" { // invoke other chaincode and reveal values
		return cc.revealPaidData(stub, args)
	} else if function == "checkTXState" { // check if TxID is used for data purchase
		return cc.checkTXState(stub, args)
	}

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

// createDataEntryAd - create a new data entry, store into chaincode state
//////////////////////////////////////////////////////////////////////////
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) ▾ not tracked not covered covered

```
	argsCount := 8
	//       0          1             2        3          4             5         6               7
	// "DataEntryID", "Description", "Value", "Unit", "CreationTime", "Publisher", "Price", "AccountNo"
	if len(args) != argsCount {
		return shim.Error("Incorrect number of arguments. Expecting 8")
	}

	// Input sanitization
	for i := 0; i < argsCount; i++ {
		if len(args[i]) <= 0 {
			return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
		}
	}

	// Get args and check if they are correct
	dataEntryID := args[0]
	description := args[1]
	value := args[2]
	unit := args[3]
	creationTime := args[4]
	creationTimeUint, err := strconv.ParseUint(creationTime, 10, 64)
	if err != nil {
		return shim.Error("Expecting positiv integer or zero as creation time.")
	}
	publisher := args[5]
	price, err := strconv.ParseInt(args[6], 10, 64)
	if err != nil {
		return shim.Error("Expecting positiv integer or zero as price.")
	}
	// Check if price is positive number
	if price < 0 {
		return shim.Error("Price cannot be negative number.")
	}
	accountNo := args[7]

	// Create composite key
	idTimeCompositeKey, err := stub.CreateCompositeKey("ID~Time", []string{dataEntryID, creationTime})
	if err != nil {
		return shim.Error(err.Error())
	}

	// Check if data entry already exists
	dataAsBytes, err := stub.GetState(idTimeCompositeKey)
	if err != nil {
		return shim.Error("Failed to get data entry: " + err.Error())
	} else if dataAsBytes != nil {
		return shim.Error("This data entry already exists: " + dataEntryID + "~" + creationTime)
	}

	// Create data entry object and marshal to JSON
	recordType := "DATA_ENTRY_AD"
	dataEntryAd := &DataEntryAd{DataEntry{recordType, dataEntryID, description, value,
		unit, creationTimeUint, publisher}, price, accountNo}
	dataEntryAdJSONasBytes, err := json.Marshal(dataEntryAd)
	if err != nil {
		return shim.Error(err.Error())
	}

	// Save data entry to state
	err = stub.PutState(idTimeCompositeKey, dataEntryAdJSONasBytes)
	if err != nil {
		return shim.Error(err.Error())
	}

	// Index the data to enable publisher-based range queries
	// An 'index' is a normal key/value entry in state.
	// The key is a composite key, with the elements that you want to range get on listed first.
	pubIDIndexKey, err := stub.CreateCompositeKey("Publisher~DataEntryID~CreationTime",
		[]string{dataEntryAd.Publisher, dataEntryAd.DataEntryID, creationTime})
	if err != nil {
		return shim.Error(err.Error())
	}

	// Save index entry to state. Only the key name is needed, no need to store a duplicate copy of the dat
	// Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null char
	valueNull := []byte{0x00}
	stub.PutState(pubIDIndexKey, valueNull)

	// Data entry saved and indexed
	return shim.Success(nil)
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) not tracked not covered covered

```
// getDataAdByIDAndTime - read data entry from chaincode state based its Id
////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getDataAdByIDAndTime(stub shim.ChaincodeStubInterface, args []string) pb.Response {
	var err error
	argsCount := 2
	//  0         1
	// "ID" "creationTime"
	if len(args) != argsCount {
		return shim.Error("Incorrect number of arguments. Expecting data entry Id and creationTime")
	}

	// Input sanitization
	for i := 0; i < argsCount; i++ {
		if len(args[i]) <= 0 {
			return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
		}
	}

	// Get args
	dataEntryID := args[0]
	creationTime := args[1]
	_, err = strconv.ParseUint(creationTime, 10, 64)
	if err != nil {
		return shim.Error("Expecting positiv integer or zero as creation time.")
	}

	// Create composite key
	idTimeCompositeKey, err := stub.CreateCompositeKey("ID~Time", []string{dataEntryID, creationTime})
	if err != nil {
		return shim.Error(err.Error())
	}
	dataAsBytes, err := stub.GetState(idTimeCompositeKey) //get the data entry from chaincode state
	if err != nil {
		return shim.Error(err.Error())
	} else if dataAsBytes == nil {
		return shim.Error(err.Error())
	}

	// Return result as bytes
	return shim.Success(dataAsBytes)
}

// getAllDataAdByID - read all data entry from chaincode state based on Id
////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getAllDataAdByID(stub shim.ChaincodeStubInterface, args []string) pb.Response {
	var err error
	argsCount := 1
	//  0
	// "ID"
	if len(args) != argsCount {
		return shim.Error("Incorrect number of arguments. Expecting data entry Id to get")
	}

	// Input sanitization
	for i := 0; i < argsCount; i++ {
		if len(args[i]) <= 0 {
			return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
		}
	}

	// Get args
	dataEntryID := args[0]
	// Create composite key
	idTimeIterator, err := stub.GetStateByPartialCompositeKey("ID~Time", []string{dataEntryID})
	if err != nil {
		return shim.Error(err.Error())
	}
	defer idTimeIterator.Close()

	// Iterate through result set and create JSON array
	var dataAsBytes []byte
	for idTimeIterator.HasNext() {
		// Note that we don't get the value (2nd return variable)
		responseRange, err := idTimeIterator.Next()
		if err != nil {
			return shim.Error(err.Error())
		}

		// get the dataEntryID and creationTime from ID~Time composite key
		_, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) not tracked not covered covered

```
                }
                returnedTime := compositeKeyParts[1]

                // Retriev the data from the state
                response := cc.getDataAdByIDAndTime(stub, []string{dataEntryID, returnedTime})
                if response.Status != shim.OK {
                        return shim.Error("Retrieval of data entry failed: " + response.Message)
                }

                // Append the retrieved data to the array
                dataAsBytes = append(dataAsBytes, response.Payload...)
                if idTimeIterator.HasNext() {
                        dataAsBytes = append(dataAsBytes, []byte(",")...)
                }
        }

        // At the end insert and append [] to create JSON array
        dataAsBytes = append([]byte("["), dataAsBytes...)
        dataAsBytes = append(dataAsBytes, []byte("]")...)

        // It returns results as JSON array
        return shim.Success(dataAsBytes)
}

// getLatestDataAdByID - read all data entry from chaincode state based on Id
////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getLatestDataAdByID(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 1
        //  0
        // "ID"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting data entry Id to get")
        }

        // Input sanitization
        for i := 0; i < argsCount; i++ {
                if len(args[i]) <= 0 {
                        return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
                }
        }

        // Get args
        dataEntryID := args[0]
        // Create composite key
        idTimeIterator, err := stub.GetStateByPartialCompositeKey("ID~Time", []string{dataEntryID})
        if err != nil {
                return shim.Error(err.Error())
        }
        defer idTimeIterator.Close()

        // Iterate through result set and return the latest data
        var latestTime uint64
        for idTimeIterator.HasNext() {
                // Note that we don't get the value (2nd return variable)
                responseRange, err := idTimeIterator.Next()
                if err != nil {
                        return shim.Error(err.Error())
                }

                // get the dataEntryID and creationTime from ID~Time composite key
                _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
                if err != nil {
                        return shim.Error(err.Error())
                }
                returnedTime := compositeKeyParts[1]
                creationTime, err := strconv.ParseUint(returnedTime, 10, 64)
                if err != nil {
                        return shim.Error("Retrieved composite key conversion to uint64 failed: " + err.Error()
                }

                // compare if the time is later than existing one
                if creationTime > latestTime {
                        latestTime = creationTime
                }
        }

        // Retriev the data from the state only if it is the latest entry
        response := cc.getDataAdByIDAndTime(stub, []string{dataEntryID, strconv.FormatUint(latestTime, 10)})
        if response.Status != shim.OK {
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) not tracked not covered covered

```
        // It returns result
        return shim.Success(response.Payload)
}

// getDataAdByPub - get data entry from chaincode state by publisher
////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getDataAdByPub(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 1
        //    0
        // "Publisher"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting publisher to get")
        }

        // Input sanitization
        for i := 0; i < argsCount; i++ {
                if len(args[i]) <= 0 {
                        return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
                }
        }

        // Get args
        publisher := args[0]

        // get the Publisher~DataEntryID index by publisher
        // This will execute a key range get on all keys starting with 'Publisher'
        pubIDResultsIterator, err := stub.GetStateByPartialCompositeKey("Publisher~DataEntryID~CreationTime", [
        if err != nil {
                return shim.Error(err.Error())
        }
        defer pubIDResultsIterator.Close()

        // Iterate through result set
        var dataAsBytes []byte
        for pubIDResultsIterator.HasNext() {
                // Note that we don't get the value (2nd return variable)
                responseRange, err := pubIDResultsIterator.Next()
                if err != nil {
                        return shim.Error(err.Error())
                }

                // get the publisher and dataEntryID from Publisher~DataEntryID composite key
                _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
                if err != nil {
                        return shim.Error(err.Error())
                }
                returnedDataEntryID := compositeKeyParts[1]
                returnedCreationTime := compositeKeyParts[2]

                // Get the data from the ledger
                response := cc.getDataAdByIDAndTime(stub, []string{returnedDataEntryID, returnedCreationTime})
                if response.Status != shim.OK {
                        return shim.Error("Retrieval of data entry failed: " + response.Message)
                }

                // Append the retrieved data to the array
                dataAsBytes = append(dataAsBytes, response.Payload...)
                if pubIDResultsIterator.HasNext() {
                        dataAsBytes = append(dataAsBytes, []byte(",")...)
                }
        }

        // At the end insert and append [] to create JSON array
        dataAsBytes = append([]byte("["), dataAsBytes...)
        dataAsBytes = append(dataAsBytes, []byte("]")...)

        // It returns results as JSON array
        return shim.Success(dataAsBytes)
}

// revealPaidData - invokes chaincode in different channel. Data entry
//                  is paid, first check transaction.
////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) revealPaidData(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 7
        //      0                1                2              3                4                   5
        // "channelData", "chaincodeDataName", "dataEntryID", "creationTime", "channelTokens", "chaincodeTokens
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) not tracked not covered covered

```
	}

	// Input sanitization
	for i := 0; i < argsCount; i++ {
		if len(args[i]) <= 0 {
			return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
		}
	}

	// Get args
	channelData := args[0]
	chaincodeDataName := args[1]
	dataEntryID := args[2]
	creationTime := args[3]
	_, err = strconv.ParseUint(creationTime, 10, 64)
	if err != nil {
		return shim.Error("Expecting positiv integer or zero as creation time.")
	}
	channelTokens := args[4]
	chaincodeTokensName := args[5]
	txID := args[6]

	// check if the dataEntryID is present in this ledger
	responseAd := cc.getDataAdByIDAndTime(stub, []string{dataEntryID, creationTime})
	if responseAd.Status != shim.OK {
		return shim.Error(err.Error())
	}

	// unmarshal
	var dataEntryAd DataEntryAd
	err = json.Unmarshal(responseAd.Payload, &dataEntryAd)
	if err != nil {
		return shim.Error(err.Error())
	}

	// Check if the txID is already in state used for some data entry purchase.
	// If not then add and index it as used transaction
	txIDResultsIterator, err := stub.GetStateByPartialCompositeKey("Tx~DataEntryID~CreationTime", []string{
	if err != nil {
		return shim.Error("Error while getting partial composite key for Tx~DataEntryID~CreationTime: "
	}
	defer txIDResultsIterator.Close()

	// Check if in the index Tx~DataEntryID~CreationTime is the TxID already
	if txIDResultsIterator.HasNext() {
		return shim.Error("Transaction was already used for data entry ID: " + dataEntryID + " Creation
	}

	// it only indexes if this transaction is commited. Atomicity...
	// therefore this statement does not have to be at the end.
	txIDIndexKey, err := stub.CreateCompositeKey("Tx~DataEntryID~CreationTime", []string{txID, dataEntryID,
	if err != nil {
		return shim.Error("Error while creating composite key for Tx~DataEntryID~CreationTime: " + err.
	}

	// Save index entry to state. Only the key name is needed, no need to store a duplicate copy of the dat
	// Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null char
	value := []byte{0x00}
	stub.PutState(txIDIndexKey, value)
	// txId entry saved and indexed

	// Invoke chaincode and get the recipient of Tx
	fTokens := []byte("getTxDetails")
	argsToChaincodeTokens := [][]byte{fTokens, []byte(txID)}
	responseTxDetails := stub.InvokeChaincode(chaincodeTokensName, argsToChaincodeTokens, channelTokens)
	if responseTxDetails.Status != shim.OK {
		return shim.Error(responseTxDetails.Message)
	}

	// Check if recipient of the Tx is the data entry account No.
	txDetails := strings.Split(string(responseTxDetails.Payload), "->")
	recipientAccID := txDetails[1]
	tokensPaid := txDetails[2]
	txStatus := txDetails[3]
	if recipientAccID != dataEntryAd.AccountNo {
		return shim.Error("This transaction does not have the same recipient account ID as required by
	}
	if tokensPaid != strconv.FormatInt(dataEntryAd.Price, 10) {
		return shim.Error("Price for the data and tokens sent in this Tx are not the same amount.")
	}
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) not tracked not covered covered

```
	}

	// Invoke chaincode in channel where data entry with value is
	// this prevent from indexing TxID as used if data entry is not present on another channel
	fData := []byte("getDataByIDAndTime")
	argsToChaincodeData := [][]byte{fData, []byte(dataEntryID), []byte(creationTime)}
	responseData := stub.InvokeChaincode(chaincodeDataName, argsToChaincodeData, channelData)
	if responseData.Status != shim.OK {
		return shim.Error(responseData.Message)
	}

	// Unmarshal data entry
	var dataEntry DataEntry
	err = json.Unmarshal(responseData.Payload, &dataEntry)
	if err != nil {
		return shim.Error(err.Error())
	}

	/*
		// This may work in the future if we get function that can invoke PutState into another chainco
			// At this stage we know that Tx recipient is correct and data entry present
			// invoke chaincode and check/add Tx to the central index as valid/spent
			fTokens = []byte("changePendingTx")
			argsToChaincodeTokens = [][]byte{fTokens, []byte(txID)}
			responseTxchange := stub.InvokeChaincode(chaincodeTokensName, argsToChaincodeTokens, ch
			if responseTxchange.Status != shim.OK {
				return shim.Error(responseTxchange.Message)
			}
	*/

	// Update the value
	dataEntryAd.Value = dataEntry.Value
	dataEntryAdAsBytes, err := json.Marshal(dataEntryAd)
	if err != nil {
		return shim.Error(err.Error())
	}

	// Create composite key
	idTimeCompositeKey, err := stub.CreateCompositeKey("ID~Time", []string{dataEntryID, creationTime})
	if err != nil {
		return shim.Error("Error while creating composite key for ID~Time: " + err.Error())
	}

	// Update the ledger
	err = stub.PutState(idTimeCompositeKey, dataEntryAdAsBytes)
	if err != nil {
		return shim.Error(err.Error())
	}

	// No need to index. DataEntryAd already indexed.
	return shim.Success(dataEntryAdAsBytes)
}

func (cc *Chaincode) checkTXState(stub shim.ChaincodeStubInterface, args []string) pb.Response {
	var err error
	argsCount := 1
	//    0
	// "txID"
	if len(args) != argsCount {
		return shim.Error("Incorrect number of arguments. Expecting TxID")
	}

	// Input sanitization
	for i := 0; i < argsCount; i++ {
		if len(args[i]) <= 0 {
			return shim.Error("Argument at position " + strconv.Itoa(i+1) + " must be a non-empty s
		}
	}

	// Extract args
	txID := args[0]
	txIDResultsIterator, err := stub.GetStateByPartialCompositeKey("Tx~DataEntryID~CreationTime", []string{
	if err != nil {
		return shim.Error("Error while getting partial composite key for Tx~DataEntryID~CreationTime: "
	}
	defer txIDResultsIterator.Close()

	// Check if the TxID is already used for data purchase
	if txIDResultsIterator.HasNext() {
		// Return that the TxID is used for data purchase in this ledger
```

chaincode/chaincode_ad/chaincode_ad.go (66.3%) ▾ not tracked not covered covered

```
        // Return that the TxID is unused for data purchase in this ledger
        return shim.Success([]byte("Unused"))
}
```