

chaincode/chaincode_data/chaincode_data.go (84.6%) ▼

not tracked not covered covered

```

package main

import (
    "encoding/json"
    "strconv"

    "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 // unique compound key that is ID~CreationTime
    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
}

// 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 == "createData" { //create a new data entry
        return cc.createData(stub, args)
    } else if function == "getDataByIDAndTime" { //read specific data by DataEntryID
        return cc.getDataByIDAndTime(stub, args)
    } else if function == "getAllDataByID" { //read all data by DataEntryID
        return cc.getAllDataByID(stub, args)
    } else if function == "getLatestDataByID" { //read latest data by DataEntryID
        return cc.getLatestDataByID(stub, args)
    } else if function == "getDataByPub" { //find data created by publisher using rich get
        return cc.getDataByPub(stub, args)
    }

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

// createData - create a new data entry, store into chaincode state
/////////
func (cc *Chaincode) createData(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var err error
    argsCount := 6
    //      0          1          2          3          4          5
    // "DataEntryID", "Description", "Value", "Unit", "CreationTime", "Publisher",
    if len(args) != argsCount {
        return shim.Error("Incorrect number of arguments. Expecting 6")
    }

    // Input sanitization
    for i := 0; i < argsCount; i++ {
        if len(args[i]) <= 0 {

```

```

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

// 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())
}

// 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"
creationTimeUint, err := strconv.ParseUint(creationTime, 10, 64)
if err != nil {
    return shim.Error("Error while parse Uint: " + err.Error())
}
dataEntry := &DataEntry{recordType, dataEntryID, description, value, unit, creationTimeUint, publisher}
dataEntryJSONasBytes, err := json.Marshal(dataEntry)
if err != nil {
    return shim.Error("Error while Marshal dataEntry: " + err.Error())
}

// Save data entry to state
err = stub.PutState(idTimeCompositeKey, dataEntryJSONasBytes)
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{dataEntry.Publisher, dataEntry.DataEntryID, creationTime})
if err != nil {
    return shim.Error("Error while creating composite key for Publisher~DataEntryID: " + 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 nil
return shim.Success(nil)
}

// getDataByIDAndTime - read data entry from chaincode state based on Id and time creation
////////////////////////////////////
func (cc *Chaincode) getDataByIDAndTime(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 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
        }
    }
}

```

chaincode/chaincode_data/chaincode_data.go (84.6%) ▾

not tracked not covered covered

```

    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 retrieved result
    return shim.Success(dataAsBytes)
}

// getAllDataByID - read all data entry from chaincode state based on Id
////////////////////////////////////
func (cc *Chaincode) getAllDataByID(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)
        if err != nil {
            return shim.Error(err.Error())
        }
        returnedTime := compositeKeyParts[1]

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

        // Append data to 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...)

```

chaincode/chaincode_data/chaincode_data.go (84.6%) ▾

not tracked not covered covered

```

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

// getLatestDataByID - read all data entry from chaincode state based on Id
////////////////////////////////////
func (cc *Chaincode) getLatestDataByID(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
        }
    }

    // Retrieve the data from the state only if it is the latest entry
    response := cc.GetDataByIDAndTime(stub, []string{dataEntryID, strconv.FormatUint(latestTime, 10)})
    if response.Status != shim.OK {
        return shim.Error("Retrieval of data entry failed: " + response.Message)
    }

    // It returns the retrieved result with latest time
    return shim.Success(response.Payload)
}

// getDataByPub - get data entry from chaincode state by publisher
////////////////////////////////////
func (cc *Chaincode) getDataByPub(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

```

chaincode/chaincode_data/chaincode_data.go (84.6%) ▼

not tracked not covered covered

```

        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.GetDataByIDAndTime(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)
}

```