chaincode/chaincode_tokens/chaincode_tokens.go (71.3%) ▼   not tracked   **not covered**   covered

---

```go
package main

import (
        "bytes"
        "encoding/json"
        "fmt"
        "strconv"
        "time"

        "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)

// Account represents account of a member
type Account struct {
        RecordType string // RecordType is used to distinguish the various types of objects in state database
        AccountID  string // unique id of the account
        Name       string // name of the account holder
        Tokens     int64  // amount of tokens (money)
}

// limitTokens - limits the highest number of tokens that can be transfered
// from account without immediate verification of available tokens.
// This provides high throughput required for IoT data an many transactions per sec
var LimitTokens int64 = 1

// Main function
//////////////////
func main() {
        // increase max CPU
        // runtime.GOMAXPROCS(runtime.NumCPU())
        err := shim.Start(new(Chaincode))
        if err != nil {
                fmt.Printf("Error starting Simple chaincode: %s", err)
        }
}

// Init initialises chaincode - Creates initial amount of tokens in two accounts
//////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        // create initial ammount of tokens
        var err error
        argsCount := 1
        // Set number of init accounts to create
        noOfAccounts := 1
        //            1
        // "Initial amount of tokens"

        args := stub.GetStringArgs()
        if len(args) != argsCount {
                return shim.Error(`Incorect number of arguments.
                        Expectiong number of accounts and tokens to create`)
        }
        // 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
                }
        }

        tokens, err := strconv.ParseInt(args[0], 10, 64)
        if err != nil || tokens < 0 {
                return shim.Error("Expecting positiv integer or zero as number of tokens to init.")
        }

        // Create account objects in array
        accounts := make([]*Account, noOfAccounts)
        for i := 0; i < noOfAccounts; i++ {
                accounts[i] = &Account{"ACCOUNT", strconv.Itoa(i + 1), "Init_Account", tokens}
        }

        // marshal each account object and save to the blockchain
        var accountJSONasBytes []byte
```

not tracked   **not covered**   covered

```go
                if err != nil {
                        return shim.Error(err.Error())
                }
                err = stub.PutState(accounts[i].AccountID, accountJSONasBytes)
                if err != nil {
                        return shim.Error(err.Error())
                }
        }

        // Index the transactions that creates tokens
        // 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.
        txID := stub.GetTxID()
        for i := 0; i < noOfAccounts; i++ {
                // Maintain index "Account~op~Tok~TxID"
                txRecipientIDCompositeKey, err := stub.CreateCompositeKey("Account~op~Tok~TxID",
                        []string{strconv.Itoa(i + 1), "+", strconv.FormatInt(tokens, 10), txID})
                if err != nil {
                        return shim.Error(err.Error())
                }

                // Note - passing a 'nil' value will effectively delete the key from state, therefore we pass n
                value := []byte{0x00}
                stub.PutState(txRecipientIDCompositeKey, value)
                if err != nil {
                        return shim.Error(err.Error())
                }

                // Maintain index "TxID~Sender~Recipient~Tok"
                txParticipantsTokCompositeKey, err := stub.CreateCompositeKey("TxID~Sender~Recipient~Tok",
                        []string{txID, "Init", strconv.Itoa(i + 1), strconv.FormatInt(tokens, 10)})
                if err != nil {
                        return shim.Error(err.Error())
                }
                stub.PutState(txParticipantsTokCompositeKey, value)
                if err != nil {
                        return shim.Error(err.Error())
                }

                // Tx entry saved and indexed
                // Index the account to enable name-based range queries
                nameIDIndexKey, err := stub.CreateCompositeKey("Name~AccountID", []string{accounts[i].Name, acc
                if err != nil {
                        return shim.Error(err.Error())
                }
                stub.PutState(nameIDIndexKey, value)
        }

        // Return the TxID
        return shim.Success([]byte(txID))
}

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

        // Handle different functions
        if function == "createAccount" { //create a new account
                return cc.createAccount(stub, args)
        } else if function == "deleteAccountByID" { // delete an account by account Id
                return cc.deleteAccountByID(stub, args)
        } else if function == "getAccountByID" { // get an account by its Id
                return cc.getAccountByID(stub, args)
        } else if function == "getAccountByName" { // find an account base on name of account holder
                return cc.getAccountByName(stub, args)
        } else if function == "sendTokensFast" { // transfer tokens from one account to another without check
                return cc.sendTokensFast(stub, args)
        } else if function == "sendTokensSafe" { // transfer tokens from one account to another with check
                return cc.sendTokensSafe(stub, args)
        } else if function == "updateAccountTokens" { // update state of account (value of tokens)
                return cc.updateAccountTokens(stub, args)
        } else if function == "getAccountTokens" { // get the current value of tokens on account
                return cc.getAccountTokens(stub, args)
        } else if function == "getAccountHistoryByID" { // get history for an account by its Id
                return cc.getAccountHistoryByID(stub, args)
        } else if function == "getTxDetails" { // get transaction details (sender->recipient->tokens->[Pending|
                return cc.getTxDetails(stub, args)
        } else if function == "changePendingTx" { // change tx pending to tx valid so recipient can use the tok
                return cc.changePendingTx(stub, args)
```

not tracked   **not covered**   covered

```
        }

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

// createAccount - create a new account and store into chaincode state
////////////////////////////////////////////////////////////////
func (cc *Chaincode) createAccount(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 2
        //       0        1
        // "AccountID", "Name"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting account Id and name")
        }

        // 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
        accountID := args[0]
        name := args[1]

        // Check if an account already exists
        accountAsBytes, err := stub.GetState(accountID)
        if err != nil {
                return shim.Error("Failed to get account: " + err.Error())
        } else if accountAsBytes != nil {
                return shim.Error("This account already exists: " + accountID)
        }

        // Create Account object and marshal to JSON
        recordType := "ACCOUNT"
        accountEntry := &Account{recordType, accountID, name, 0}
        accountEntryJSONasBytes, err := json.Marshal(accountEntry)
        if err != nil {
                return shim.Error(err.Error())
        }

        // Save account entry to state
        err = stub.PutState(accountID, accountEntryJSONasBytes)
        if err != nil {
                return shim.Error(err.Error())
        }

        //  Index the account to enable name-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.
        indexName := "Name~AccountID"
        nameIDIndexKey, err := stub.CreateCompositeKey(indexName, []string{accountEntry.Name, accountEntry.Acco
        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 da
        //  Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null cha
        value := []byte{0x00}
        stub.PutState(nameIDIndexKey, value)

        // Account saved and indexed. Return success
        return shim.Success([]byte("Account created"))
}

// deleteAccountByID - deletes the account if number of tokens is 0
////////////////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) deleteAccountByID(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 1
        //       0
        // deleteAccountId
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting AccountID.")
        }

        // Input sanitization
        for i := 0; i < argsCount; i++ {
```

not tracked   **not covered**   covered

```go
        }
}

// Extract args
accountID := args[0]

// Get the account entry from chaincode state
accountAsBytes, err := stub.GetState(accountID)
var account Account
err = json.Unmarshal(accountAsBytes, &account)
if err != nil {
        return shim.Error(err.Error())
}

// Check if the account have any tokens
responseTokens := cc.getAccountTokens(stub, []string{accountID})
if responseTokens.Status != shim.OK {
        return shim.Error(responseTokens.Message)
}

// convert Payload to int64
remainingTokensStr := string(responseTokens.Payload)
remainingTokens, err := strconv.ParseInt(remainingTokensStr, 10, 64)
if err != nil {
        return shim.Error(err.Error())
}
if remainingTokens != 0 {
        return shim.Error("Account cannot be deleted. Amount of tokens is not 0.")
}

// Prune all Tx
responsePruneTx := cc.pruneAccountTx(stub, []string{accountID})
if responsePruneTx.Status != shim.OK {
        return shim.Error(responsePruneTx.Message)
}

// Update Account state before delete
responseAccUpdate := cc.updateAccountTokens(stub, []string{accountID})
if responseAccUpdate.Status != shim.OK {
        return shim.Error(responseAccUpdate.Message)
}

// Delete the account state
err = stub.DelState(accountID)
if err != nil {
        return shim.Error("Failed to delete state:" + err.Error())
}

// Maintain the index Name~AccountID
indexName := "Name~AccountID"
nameIDIndexKey, err := stub.CreateCompositeKey(indexName, []string{account.Name, account.AccountID})
if err != nil {
        return shim.Error(err.Error())
}

//  Delete index entry to state.
err = stub.DelState(nameIDIndexKey)
if err != nil {
        return shim.Error("Failed to delete state:" + err.Error())
}

// Maintain the index "Account~op~Tok~TxID"
// Get the single account transactions for the account ID
accountTxIterator, err := stub.GetStateByPartialCompositeKey("Account~op~Tok~TxID",
        []string{accountID})
if err != nil {
        return shim.Error(err.Error())
}
defer accountTxIterator.Close()

// If it is created account without any Tx then do not have to delete aby Tx in index
if accountTxIterator.HasNext() {
        // Get the row
        responseAccTxRange, err := accountTxIterator.Next()
        if err != nil {
                return shim.Error(err.Error())
        }

        // Get separate parts of composite key
        _, compositeKey, err := stub.SplitCompositeKey(responseAccTxRange.Key)
```

not tracked   **not covered**   covered

```go
                        return shim.Error(err.Error())
                }

                //  Delete index entry in the ledger.
                err = stub.DelState(responseAccTxRange.Key)
                if err != nil {
                        return shim.Error(err.Error())
                }

                // Maintain the index "TxID~Sender~Recipient~Tok"
                txParticipantsTokIterator, err := stub.GetStateByPartialCompositeKey("TxID~Sender~Recipient~Tok
                        []string{lastTxToDel})
                if err != nil {
                        return shim.Error(err.Error())
                }
                defer txParticipantsTokIterator.Close()

                // Check if the TxID is in the TxID~Sender~Recipient~Tok index
                if !txParticipantsTokIterator.HasNext() {
                        return shim.Error("DeleteAccount: There was an entry in one index but no entry in the s
                }

                // Get the row
                responseTxParticipantsRange, err := txParticipantsTokIterator.Next()
                if err != nil {
                        return shim.Error(err.Error())
                }

                // // Delete the Tx entry in "TxID~Sender~Recipient~Tok"
                err = stub.DelState(responseTxParticipantsRange.Key)
                if err != nil {
                        return shim.Error(err.Error())
                }
                if !txParticipantsTokIterator.HasNext() {
                        return shim.Error("DeleteAccount: Two Transactions with the same ID? Impossible!")
                }
        }

        // Return Success
        return shim.Success([]byte("Account deleted"))
}

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

        // 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
        accountID := args[0]

        // Get the account entry from chaincode state
        accountAsBytes, err := stub.GetState(accountID)
        if err != nil {
                return shim.Error(err.Error())
        } else if accountAsBytes == nil {
                return shim.Error(err.Error())
        }

        // Return as JSONasBytes
        return shim.Success(accountAsBytes)
}

// getAccountByName - get data entry from chaincode state by name
////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getAccountByName(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 1
```

```go
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting name of account holder")
        }

        // 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
        name := args[0]

        // get the Name~AccountID index by name
        // This will execute a key range get on all keys starting with 'name'
        nameIDResultsIterator, err := stub.GetStateByPartialCompositeKey("Name~AccountID", []string{name})
        if err != nil {
                return shim.Error(err.Error())
        }
        defer nameIDResultsIterator.Close()

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

                // get the Name and AccountID from Name~AccountID composite key
                _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
                if err != nil {
                        return shim.Error(err.Error())
                }
                returnedAccountID := compositeKeyParts[1]

                // Get the account from state
                response := cc.getAccountByID(stub, []string{returnedAccountID})
                if response.Status != shim.OK {
                        return shim.Error("Retrieval of account entry failed: " + response.Message)
                }

                // Append account to array of bytes if there is more accounts
                accountsAsBytes = append(accountsAsBytes, response.Payload...)
                if nameIDResultsIterator.HasNext() {
                        accountsAsBytes = append(accountsAsBytes, []byte(",")...)
                }
        }

        // Create JSON array from it
        accountsAsBytes = append([]byte("["), accountsAsBytes...)
        accountsAsBytes = append(accountsAsBytes, []byte("]")...)

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

// sendTokensFast - transfer tokens from one account to another without check of sender's tokens
////////////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) sendTokensFast(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 4
        //        0              1              2              3
        // "fromAccountId" "toAccountId" "Amount" "dataPurchase"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting FromAccountId, ToAccountId, Amount,
        }

        // 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
        fromAccountID := args[0]
        toAccountID := args[1]
```

```go
		if fromAccountID == toAccountID {
			return shim.Error("From account and to account cannot be the same.")
		}
		tokensToSend, err := strconv.ParseInt(args[2], 10, 64)
		if err != nil || tokensToSend < 1 {
			return shim.Error("Expecting positive integer as number of tokens to transfer.")
		}

		// Check if the amount of tokens does not exceed limit for fast transfer
		if tokensToSend > LimitTokens {
			return shim.Error("Exceeded max number of tokens for fast transaction. Use safe token transfer
		}

		// Is it payment for data purchase
		dataPurchase, err := strconv.ParseBool(args[3])
		if err != nil {
			return shim.Error("Expecting boolean value. If this transfer is for data purchase or not.")
		}

		// Index txID and sender accounts ID
		// this is required for quick lookup and transaction aggregation.
		txID := stub.GetTxID()
		var SenderIDOpTokCompositeKey, recipientIDOpTokCompositeKey, txParticipantsTokCompositeKey string
		if dataPurchase {
			SenderIDOpTokCompositeKey, err = stub.CreateCompositeKey("Account~op~Tok~TxID",
				[]string{fromAccountID, "-", strconv.FormatInt(tokensToSend, 10), txID})
			if err != nil {
				return shim.Error(err.Error())
			}
			txParticipantsTokCompositeKey, err = stub.CreateCompositeKey("PendingTxID~Sender~Recipient~Tok"
				[]string{txID, fromAccountID, toAccountID, strconv.FormatInt(tokensToSend, 10)})
			if err != nil {
				return shim.Error(err.Error())
			}
		} else {
			SenderIDOpTokCompositeKey, err = stub.CreateCompositeKey("Account~op~Tok~TxID",
				[]string{fromAccountID, "-", strconv.FormatInt(tokensToSend, 10), txID})
			if err != nil {
				return shim.Error(err.Error())
			}
			recipientIDOpTokCompositeKey, err = stub.CreateCompositeKey("Account~op~Tok~TxID",
				[]string{toAccountID, "+", strconv.FormatInt(tokensToSend, 10), txID})
			if err != nil {
				return shim.Error(err.Error())
			}
			txParticipantsTokCompositeKey, err = stub.CreateCompositeKey("TxID~Sender~Recipient~Tok",
				[]string{txID, fromAccountID, toAccountID, strconv.FormatInt(tokensToSend, 10)})
			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
		value := []byte{0x00}
		stub.PutState(SenderIDOpTokCompositeKey, value)
		if err != nil {
			return shim.Error(err.Error())
		}
		stub.PutState(recipientIDOpTokCompositeKey, value)
		if err != nil {
			return shim.Error(err.Error())
		}
		stub.PutState(txParticipantsTokCompositeKey, value)
		if err != nil {
			return shim.Error(err.Error())
		}

		// Tx entry saved and indexed
		return shim.Success([]byte(txID))

}

// sendTokensSafe - transfer tokens from one account to another with check of sender's tokens
//////////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) sendTokensSafe(stub shim.ChaincodeStubInterface, args []string) pb.Response {
	var err error
	argsCount := 4
	//       0              1            2           3
	// "fromAccountId" "toAccountId" "Amount" "dataPurchase"
	if len(args) != argsCount {
```

**not tracked**   **not covered**   **covered**

```go
	// 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
	fromAccountID := args[0]
	toAccountID := args[1]

	// Check if sender and recipient args are the same
	if fromAccountID == toAccountID {
		return shim.Error("From account and to account cannot be the same.")
	}
	tokensToSend, err := strconv.ParseInt(args[2], 10, 64)
	if err != nil || tokensToSend < 1 {
		return shim.Error("Expecting positive integer as number of tokens to transfer.")
	}

	// Is it payment for data purchase
	dataPurchase, err := strconv.ParseBool(args[3])
	if err != nil {
		return shim.Error("Expecting boolean value. If this transfer is for data purchase or not.")
	}

	// If account retrieval from state does not fail then accounts exist. Just check
	// No need to unmarshal
	fromAccountAsBytes, err := stub.GetState(fromAccountID)
	if err != nil {
		return shim.Error(err.Error())
	} else if fromAccountAsBytes == nil {
		return shim.Error(err.Error())
	}
	toAccountAsBytes, err := stub.GetState(toAccountID)
	if err != nil {
		return shim.Error(err.Error())
	} else if toAccountAsBytes == nil {
		return shim.Error(err.Error())
	}

	// Get the latest state of tokens for sender's account
	argsTok := []string{fromAccountID}
	fromAccTokResponse := cc.getAccountTokens(stub, argsTok)
	if fromAccTokResponse.Status != shim.OK {
		return shim.Error("Retrieval of account tokens failed: " + fromAccTokResponse.Message)
	}

	// Extract the values from Payload
	fromAccTokStr := string(fromAccTokResponse.Payload)
	fromAccTok, err := strconv.ParseInt(fromAccTokStr, 10, 64)
	if err != nil {
		return shim.Error(err.Error())
	}

	// Check if sender has enough tokens
	if fromAccTok < tokensToSend {
		return shim.Error("Not enough tokens on the sender's account")
	}
	// Index txID and sender accounts ID
	// this is required for quick lookup and transaction aggregation.
	txID := stub.GetTxID()
	var SenderIDOpTokCompositeKey, recipientIDOpTokCompositeKey, txParticipantsTokCompositeKey string
	if dataPurchase {
		SenderIDOpTokCompositeKey, err = stub.CreateCompositeKey("Account~op~Tok~TxID",
			[]string{fromAccountID, "-", strconv.FormatInt(tokensToSend, 10), txID})
		if err != nil {
			return shim.Error(err.Error())
		}
		txParticipantsTokCompositeKey, err = stub.CreateCompositeKey("PendingTxID~Sender~Recipient~Tok"
			[]string{txID, fromAccountID, toAccountID, strconv.FormatInt(tokensToSend, 10)})
		if err != nil {
			return shim.Error(err.Error())
		}
	} else {
		SenderIDOpTokCompositeKey, err = stub.CreateCompositeKey("Account~op~Tok~TxID",
			[]string{fromAccountID, "-", strconv.FormatInt(tokensToSend, 10), txID})
		if err != nil {
			return shim.Error(err.Error())
```

chaincode/chaincode_tokens/chaincode_tokens.go (71.3%) ▼   not tracked   **not covered**   covered

```go
                        []string{toAccountID, "+", strconv.FormatInt(tokensToSend, 10), txID})
                if err != nil {
                        return shim.Error(err.Error())
                }
                txParticipantsTokCompositeKey, err = stub.CreateCompositeKey("TxID~Sender~Recipient~Tok",
                        []string{txID, fromAccountID, toAccountID, strconv.FormatInt(tokensToSend, 10)})
                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
        value := []byte{0x00}
        stub.PutState(SenderIDOpTokCompositeKey, value)
        if err != nil {
                return shim.Error(err.Error())
        }
        stub.PutState(recipientIDOpTokCompositeKey, value)
        if err != nil {
                return shim.Error(err.Error())
        }
        stub.PutState(txParticipantsTokCompositeKey, value)
        if err != nil {
                return shim.Error(err.Error())
        }

        // Tx entry saved and indexed
        return shim.Success([]byte(txID))
}

// updateAccountTokens - updates the account entry in state with the latest values of tokens
///////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) updateAccountTokens(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 1
        //        0
        // "accountID"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting account ID")
        }

        // 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
        accountID := args[0]

        // Get the account entry from chaincode state
        accountAsBytes, err := stub.GetState(accountID)
        if err != nil {
                return shim.Error(err.Error())
        } else if accountAsBytes == nil {
                return shim.Error(err.Error())
        }
        var account Account
        err = json.Unmarshal(accountAsBytes, &account)
        if err != nil {
                return shim.Error("Some error: " + err.Error())
        }

        // Get the latest state of tokens for sender's account
        argsTok := []string{accountID}
        accTokResponse := cc.getAccountTokens(stub, argsTok)
        if accTokResponse.Status != shim.OK {
                return shim.Error("Retrieval of account tokens failed: " + accTokResponse.Message)
        }

        // Extract values from Payload
        accTokStr := string(accTokResponse.Payload)
        accTok, err := strconv.ParseInt(accTokStr, 10, 64)
        if err != nil {
                return shim.Error(err.Error())
        }

        // update values of tokens
```

```
chaincode/chaincode_tokens/chaincode_tokens.go (71.3%) ▼    not tracked   not covered   covered
```

```go
        // Marshal object
        accountAsBytesNew, err := json.Marshal(&account)
        if err != nil {
                return shim.Error("Some error: " + err.Error())
        }

        // Write state back to the ledger
        err = stub.PutState(accountID, accountAsBytesNew)
        if err != nil {
                return shim.Error("Some error: " + err.Error())
        }

        // return JSON object Account with updated token values
        return shim.Success(accountAsBytesNew)
}

// getAccountTokens - returns current state of tokens in a specific account
///////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getAccountTokens(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error
        argsCount := 1
        //      0
        // "accountID"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting account ID")
        }

        // 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
        accountID := args[0]

        // Get all account transactions for the account ID
        accountTxIterator, err := stub.GetStateByPartialCompositeKey("Account~op~Tok~TxID", []string{accountID}
        if err != nil {
                return shim.Error(err.Error())
        }
        defer accountTxIterator.Close()

        // Iterate through result set and compute final amount of tokens
        var finalTok int64
        for accountTxIterator.HasNext() {
                // Get the next row
                responseRange, err := accountTxIterator.Next()
                if err != nil {
                        return shim.Error(err.Error())
                }

                // Split the composite key into its component parts
                _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
                if err != nil {
                        return shim.Error(err.Error())
                }

                // Retrieve the amount of tokens and operation
                operation := compositeKeyParts[1]
                tokensStr := compositeKeyParts[2]

                // Convert the tokensStr string and perform the operation
                tokens, err := strconv.ParseInt(tokensStr, 10, 64)
                if err != nil {
                        return shim.Error(err.Error())
                }

                // calculate the delta
                switch operation {
                case "+":
                        finalTok += tokens
                case "-":
                        finalTok -= tokens
                default:
                        return shim.Error(fmt.Sprintf("Unrecognized operation %s", operation))
                }
        }
```

```go
        // Return result
        return shim.Success([]byte(res))
}

// getAccountHistoryByID - get the whole history of specific account number even if it was deleted from state.
///////////////////////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getAccountHistoryByID(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        argsCount := 1
        //     0
        // "accountID"
        if len(args) != argsCount {
                return shim.Error("Incorrect number of arguments. Expecting AccountID")
        }

        // 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
        accountID := args[0]

        // Get history for the account ID
        resultsIterator, err := stub.GetHistoryForKey(accountID)
        if err != nil {
                return shim.Error(err.Error())
        }
        defer resultsIterator.Close()

        // buffer is a JSON array containing historic values for the account
        var buffer bytes.Buffer
        buffer.WriteString("[")

        // for each entry write to buffer
        for resultsIterator.HasNext() {
                response, err := resultsIterator.Next()
                if err != nil {
                        return shim.Error(err.Error())
                }

                // This create structure of the output
                // Write transaction ID
                buffer.WriteString("{\"TxId\":")
                buffer.WriteString("\"")
                buffer.WriteString(response.TxId)
                buffer.WriteString("\"")

                // Write Value
                buffer.WriteString(", \"Value\":")

                // if it was a delete operation on given key, then we set the
                // value to null. Else, we will write the response.Value
                // as-is (as the Value itself a JSON)
                if response.IsDelete {
                        buffer.WriteString("null")
                } else {
                        buffer.WriteString(string(response.Value))
                }

                // Write Timestamp of the transaction
                buffer.WriteString(", \"Timestamp\":")
                buffer.WriteString("\"")
                buffer.WriteString(time.Unix(response.Timestamp.Seconds, int64(response.Timestamp.Nanos)).Strin
                buffer.WriteString("\"")

                // Write if it was delete transaction
                buffer.WriteString(", \"IsDelete\":")
                buffer.WriteString("\"")
                buffer.WriteString(strconv.FormatBool(response.IsDelete))
                buffer.WriteString("\"")

                // Close the single entry
                buffer.WriteString("}")

                // Add a comma in front of an array member
                if resultsIterator.HasNext() {
                        buffer.WriteString(",")
```

```go
            // Close JSON array
            buffer.WriteString("]")

            // Return JSON array of history transactions for the account ID
            return shim.Success(buffer.Bytes())
}

// getTxDetails - returns participants' account IDs of transaction and amount
//////////////////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) getTxDetails(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]

            // Get the state from TxID~Sender~Recipient~Tok index
            txIDResultsIterator, err := stub.GetStateByPartialCompositeKey("TxID~Sender~Recipient~Tok",
                        []string{txID})
            if err != nil {
                        return shim.Error(err.Error())
            }
            defer txIDResultsIterator.Close()

            // Initialise txState as validTx
            txState := "ValidTx"

            // If there is not such TxID then check the PendingTxID~Sender~Recipient~Tok index
            if !txIDResultsIterator.HasNext() {
                        txIDResultsIterator, err = stub.GetStateByPartialCompositeKey("PendingTxID~Sender~Recipient~Tok
                                    []string{txID})
                        if err != nil {
                                    return shim.Error(err.Error())
                        }

                        // There is not such transaction in both indices
                        if !txIDResultsIterator.HasNext() {
                                    return shim.Error("Transaction was not found.")
                        }

                        // Change the txState to pendingTx
                        txState = "PendingTx"
            }

            // Get the response range
            responseRange, err := txIDResultsIterator.Next()
            if err != nil {
                        return shim.Error(err.Error())
            }

            // Get the values from composite key
            _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
            if err != nil {
                        return shim.Error(err.Error())
            }

            // Construct participants as string
            participantsAccountsID := compositeKeyParts[1] + "->" + compositeKeyParts[2]
            tokens := compositeKeyParts[3]
            if txIDResultsIterator.HasNext() {
                        return shim.Error("Two TxID are same? Impossible!")
            }

            // Construct the response string
            var response []byte
            response = append(response, []byte(participantsAccountsID)...)
            response = append(response, []byte("->")...)
```

```go
            response = append(response, []byte(txState)...)

            // Return byte array with Tx details
            return shim.Success(response)
}

// changePendingTx - change pending tokens to normal tokens
////////////////////////////////////////////////////////////////////////////////////////////
func (cc *Chaincode) changePendingTx(stub shim.ChaincodeStubInterface, args []string) pb.Response {
            var err error
            argsCount := 3
            //       0                1            2
            // "channelAd" "chaincodeAdName" "txID"
            if len(args) != argsCount {
                    return shim.Error("Incorrect number of arguments. Expecting 3")
            }

            // 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
            channelAd := args[0]
            chaincodeAdName := args[1]
            txID := args[2]

            // Get the TxID from another chaincode
            pendingTxIDResultsIterator, err := stub.GetStateByPartialCompositeKey("PendingTxID~Sender~Recipient~Tok
                    []string{txID})
            if err != nil {
                    return shim.Error(err.Error())
            }
            defer pendingTxIDResultsIterator.Close()

            // If not Tx found then it was already moved as valid Tx
            if !pendingTxIDResultsIterator.HasNext() {
                    return shim.Error("Transaction was already used or does not exist.")
            }

            // Extract values
            responseRange, err := pendingTxIDResultsIterator.Next()
            if err != nil {
                    return shim.Error(err.Error())
            }

            // Get the separate values
            _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
            if err != nil {
                    return shim.Error(err.Error())
            }

            // Check if there is another Tx with the same ID
            if pendingTxIDResultsIterator.HasNext() {
                    return shim.Error("changePendingTx: Two TxID are same? Impossible!")
            }

            // check if the Tx was already used for data purchase
            fDataAd := []byte("checkTXState")
            argsToChaincodeAd := [][]byte{fDataAd, []byte(txID)}
            responseTXCheck := stub.InvokeChaincode(chaincodeAdName, argsToChaincodeAd, channelAd)
            if responseTXCheck.Status != shim.OK {
                    return shim.Error("changePendingTx: Error while invoking another chaincode: " + responseTXCheck
            }
            if string(responseTXCheck.Payload) != "Used" {
                    return shim.Error("This TxID was not used for data purchase yet.")
            }

            // create composite key to reindex
            txCompositeIndexKey, err := stub.CreateCompositeKey("TxID~Sender~Recipient~Tok",
                    []string{txID, compositeKeyParts[1], compositeKeyParts[2], compositeKeyParts[3]})
            if err != nil {
                    return shim.Error(err.Error())
            }

            // Maintain index of pending Tx
            err = stub.DelState(responseRange.Key)
            if err != nil {
```

not tracked   **not covered**   **covered**

```go
        // Add Tx to index "TxID~Sender~Recipient~Tok"
        // Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null char
        value := []byte{0x00}
        err = stub.PutState(txCompositeIndexKey, value)
        if err != nil {
                return shim.Error(err.Error())
        }

        // Create the new composite key for the index  Account~op~Tok~TxID
        recipientIDOpTokCompositeKey, err := stub.CreateCompositeKey("Account~op~Tok~TxID",
                []string{compositeKeyParts[2], "+", compositeKeyParts[3], txID})
        if err != nil {
                return shim.Error(err.Error())
        }

        // Save to the state
        err = stub.PutState(recipientIDOpTokCompositeKey, value)
        if err != nil {
                return shim.Error(err.Error())
        }

        // Tx changed and indexed in both indexes
        // Return tx ID
        return shim.Success([]byte(txID))
}

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

        // 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
        accountID := args[0]

        // Get all account transactions for the account ID
        accountTxIterator, err := stub.GetStateByPartialCompositeKey("Account~op~Tok~TxID", []string{accountID}
        if err != nil {
                return shim.Error("pruneAccountTx: " + err.Error())
        }
        defer accountTxIterator.Close()

        // Iterate through result set and compute final amount of tokens
        var finalTok int64
        for accountTxIterator.HasNext() {
                // Get the next row
                responseRange, err := accountTxIterator.Next()
                if err != nil {
                        return shim.Error("pruneAccountTx: " + err.Error())
                }

                // Split the composite key into its component parts
                _, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
                if err != nil {
                        return shim.Error("pruneAccountTx: " + err.Error())
                }

                // Retrieve the amount of tokens and operation
                operation := compositeKeyParts[1]
                tokensStr := compositeKeyParts[2]
                txID := compositeKeyParts[3]

                // Convert the tokensStr string and perform the operation
                tokens, err := strconv.ParseInt(tokensStr, 10, 64)
                if err != nil {
                        return shim.Error("pruneAccountTx: " + err.Error())
                }

                // check if the TxID is in "TxID~Sender~Recipient~Tok" index. If not then it is not valid Tx ye
```

```go
                            []string{txID})
            if err != nil {
                    return shim.Error("pruneAccountTx: " + err.Error())
            }
            defer txParticipantsTokIterator.Close()

            if !txParticipantsTokIterator.HasNext() {
                    continue
            }

            // calculate the delta
            switch operation {
            case "+":
                    finalTok += tokens
            case "-":
                    finalTok -= tokens
            default:
                    return shim.Error(fmt.Sprintf("Unrecognized operation %s", operation))
            }

            // Maintain the index of "Account~op~Tok~TxID"
            err = stub.DelState(responseRange.Key)
            if err != nil {
                    return shim.Error("pruneAccountTx: " + err.Error())
            }

            // Get the Row Tx entry in the other index
            responseTxParticipantsRange, err := txParticipantsTokIterator.Next()
            if err != nil {
                    return shim.Error("pruneAccountTx: " + err.Error())
            }

            // Maintain index of "TxID~Sender~Recipient~Tok"
            err = stub.DelState(responseTxParticipantsRange.Key)
            if err != nil {
                    return shim.Error("pruneAccountTx: " + err.Error())
            }
    }

    newTxID := stub.GetTxID()
    // Create the new composite key for the new entry
    recipientIDOpTokCompositeKey, err := stub.CreateCompositeKey("Account~op~Tok~TxID",
            []string{accountID, "+", strconv.FormatInt(finalTok, 10), newTxID})
    if err != nil {
            return shim.Error("pruneAccountTx: " + err.Error())
    }

    // Create the new composite key for the new entry
    txParticipantsTokCompositeKey, err := stub.CreateCompositeKey("TxID~Sender~Recipient~Tok",
            []string{newTxID, "pruneTx", accountID, strconv.FormatInt(finalTok, 10)})
    if err != nil {
            return shim.Error("pruneAccountTx: " + err.Error())
    }

    // Update the index with single Tx that agregates all that were deleted
    // Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null char
    value := []byte{0x00}
    err = stub.PutState(recipientIDOpTokCompositeKey, value)
    if err != nil {
            return shim.Error("pruneAccountTx: " + err.Error())
    }
    err = stub.PutState(txParticipantsTokCompositeKey, value)
    if err != nil {
            return shim.Error("pruneAccountTx: " + err.Error())
    }

    // Index updated
    // Return new Tx ID
    return shim.Success([]byte(newTxID))
}
```