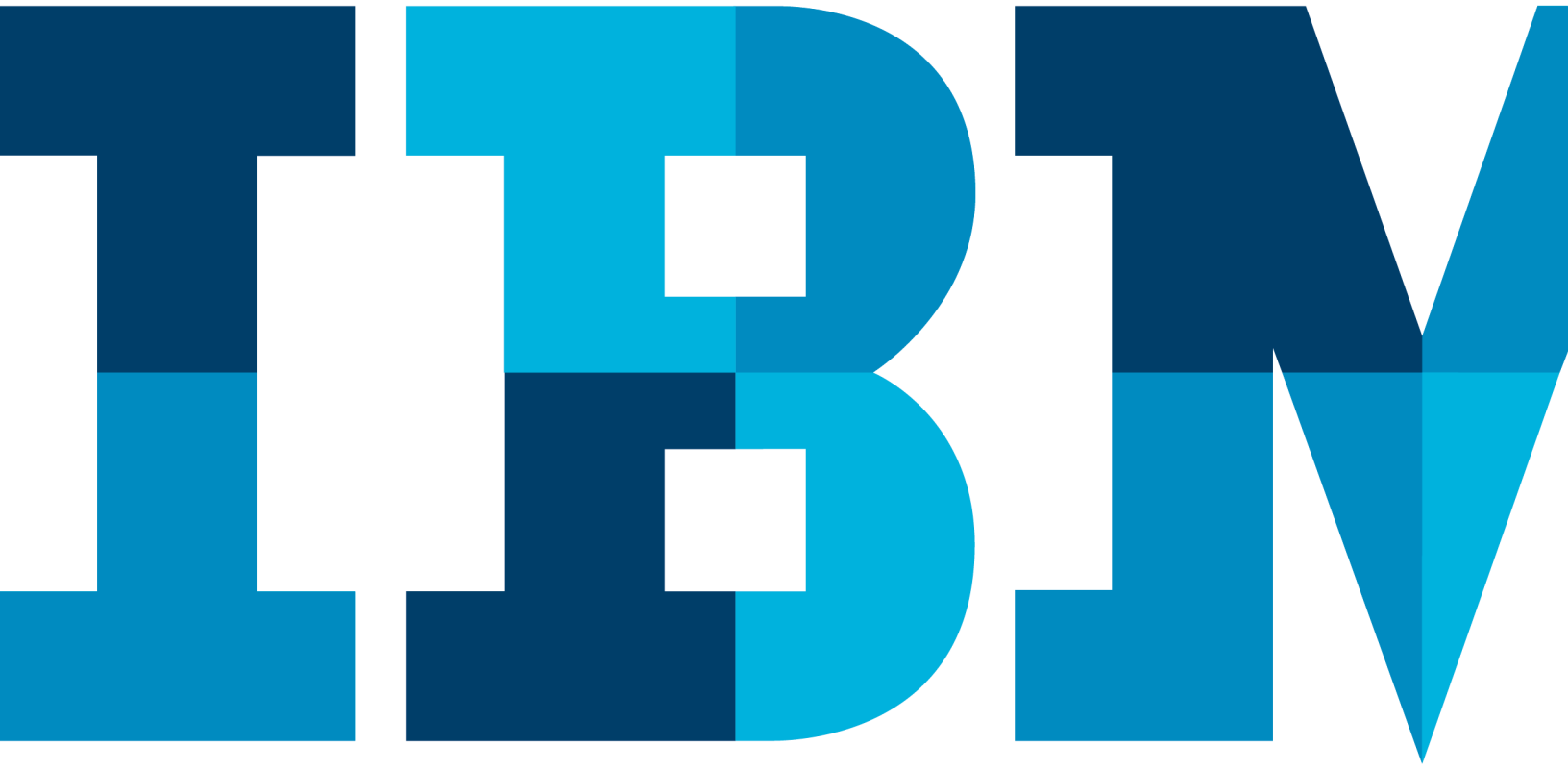
IBM Blockchain
Hyperledger Fabric Hands-On

Develop a smart contract in go (shim, putstate, getstate)

Lab



IBM

1 Contents

2 Overview

The aim of this lab is to examine explore basics of smartcontract development for Hyperledger Fabric 1.x.

2.1 Introduction

This lab has been tested on Windows 7, Windows 10 and OS X workstations.
The required tools are:

- Git
- Go
- Visual Studio Code with the Go extension.

The installation procedures for these tools are described in the 'Lab HLF – preparation' document.
When instructed to open a terminal, proceed as follows :

- On a Windows workstation (7 or 10) without administrative privileges, the terminal is located in the Git installation folder, and is named git-bash.exe.
- Microsoft Windows 10 workstation with administrative privileges, you will operate through a bash terminal in the Ubuntu subsystem: open a command prompt and issue the "bash" command.
- On OS X, you will operate through a terminal window that you can launch via Launchpad, selecting the "Other" icon, then clicking on Terminal:

In the rest of the document, "bash terminal" "Linux terminal" or "terminal window" will refer to the terminal window described above.

2.2 Prerequisites

2.2.1 Go language

This section is optional if you are familiar with Go programming language.

Hyperledger Fabric Smartcontract (also called chaincode) are developed by using several programming languages: Java, Go and JavaScript.

Go programming language will be used for this hands-on. You need to be familiar with functional programming concepts and you need to know basics of Go programming language.

We advise the reader to take some time in the Golang tour on the Golang.org website (at https://golang.org/basics/1) to become familiar with functions, variables and available types.

2.2.2 Hyperledger Fabric source code

Compiling smartcontracts depends on some hyperledger fabric packages. It is necessary to retrieve Hyperledger Fabric source code prior to starting the labs.
From a terminal, issue the following commands to checkout a copy of the Hyperledger Fabric source code repository to your local filesystem:

```
cd $GOPATH
mkdir -p src/github.com/hyperledger
cd src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric.git
cd fabric
git checkout tags/v1.2.1
```

3 My first SmartContract

In this section, you will edit and compile a smartcontract
.

3.1 Create a 'HelloWorld' SmartContract

Create the folder **labhlf** in the $GOPATH/src folder. In this, create the subfolder **hlf-<xx>-sc** (replace <xx> by your initials).
Go to the labhlf-<xx>-sc folder and create a file **labhlf-<xx>-sc.go** (replace <xx> by your initials). You should use your code editor to do this.

The structure of your go code will be the following:

```
//name of the package – used if you want to define you module as library
Package <name of package>
//list of libraries to import
import (
     "name of package to import"
     "name of package to import"
      ….
)
//structure of data you will use in your module
type <structure name> struct {
}
//variables
var …
//functions of your module
func (t *<structure name>) <function name>(…) <returned structure> {
    …
return <returned object>
}
…
//main routine
func main() {
    …
}
```

We start creating a smartcontract which will return "Hello World".

The Hyperledger Fabric smartcontract must implement the following interface:

- **init()** that is called once when the smart contract is instantiated.
- **invoke()** that will be called when a transaction is executed.

The Go package called shim, shipped as part of Hyperledger Fabric, provides the smartcontract API. It includes the `shim.Success()` *function that allows to returns some data to the application. The function inputs have to be passed as an array of bytes.*

Edit the **labhlf-<xx>-sc.go** file and input the following code:

```
package main

import (
    "fmt"

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

type Labhlfsc struct {
}

var chaincodeVersion = "2019-03-nn"

func (t *Labhlfsc) Init(stub shim.ChaincodeStubInterface) pb.Response {
    return shim.Success([]byte(chaincodeVersion))
}

func (t *Labhlfsc) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    return shim.Success(nil)
}

func main() {
    err := shim.Start(new(Labhlfsc))
    if err != nil {
        fmt.Printf("Error starting chaincode: %s", err)
    }
}
```

Save the file. Now we are going to compile it.

3.2 Compile the SmartContract

- Open your Linux terminal
- Change your current directory to $GOPATH/src/labhlf/labhlf-<xx>-sc/

Issue the command:

```
go build
```

You may have some warning messages that can ignore.
In case of error du to pkcs11 package, you should try with the option –tags nopkcs11
(without the option –tags nopkcs11, you may have errors relative to the lack of library <ltdl.h>)

Your module should be compiled. In your work directory, you should have 2 files:

- The source code with a ".go" xtension
- The run module, named after the folder name.

Run the smartcontract :

```
./labhlf-<xx>-sc
```

The result should be

```
2018-10-09 12:01:59.633 CEST [shim] SetupChaincodeLogging -> INFO 001 Chaincode log level
not provided; defaulting to: INFO
2018-10-09 12:01:59.633 CEST [shim] SetupChaincodeLogging -> INFO 002 Chaincode (build
level: ) starting up ...
Error starting chaincode: error chaincode id not provided
```

This error occurs because we are running the transaction without a blockchain network.
We are now creating a test module to test our Smartcontract.

3.3 Create the test module of the SmartContract

We are going to create a test module which will allow to test the smart contract without deploying it in a Hyperledger Fabric.

Create and edit a file called labhlf-<xx>-sc_test.go in the same folder as your smartcontract.
Input the following code in this file:

```go
package main

import (
    "fmt"
    "testing"
    "github.com/hyperledger/fabric/core/chaincode/shim"
)

var loggertest = shim.NewLogger("SmartContract labhlf-<xx>-sc UNIT_TEST")

var chaincodeVersionTest = "2019-03-nn"

//This function call the init function of the smart contrat and check the returned value
func checkInit(t *testing.T, stub *shim.MockStub, args [][]byte) {
    res := stub.MockInit("1", args)
    if res.Status != shim.OK {
        fmt.Println("Init failed", string(res.Message))
        t.FailNow()
    }
    if string(res.Payload) != chaincodeVersionTest {
        fmt.Println("Init returned wrong payload, \nexpected: ", chaincodeVersionTest, "
\nreturned: ", string(res.Payload))
```

```
        t.FailNow()
    }
}

//function to test the init of labhlf
func Test_HhlfscInit(t *testing.T) {
    scc := new(Labhlfsc)
    stub := shim.NewMockStub("Test_labhlfsc_Init", scc)
    loggertest.Infof(" Starting Test_labhlfsc_Init...")

    checkInit(t, stub, [][]byte{[]byte("init1"), []byte("INITTEST")})
    loggertest.Infof("Test_labhlfsc_Init completed")
}
```

Pay attention to the code of this test module.
The **checkInit** function allows to test the Init function of the smartcontract.
This checkInit is called by the **Test_HhlfscInit** function which instantiates the smartcontract (scc := new(Labhlfsc)) then runs the checkInit.
When the test module is run, it processes sequentially all the functions starting with the string “Test_”.

To execute this test module, open a Linux terminal, move to the folder of your test module and run

```
go test
```

You should get similar result (you may have warning messages that you can ignore):

```
2018-10-10 09:29:32.631 CEST [SmartContract hlf-<xx>-sc UNIT_TEST] Infof -> INFO 001  Starting Test_labhlfsc_Init...
2018-10-10 09:29:32.631 CEST [SmartContract hlf-<xx>-sc UNIT_TEST] Infof -> INFO 002 Test_labhlfsc_Init completed
PASS
ok      labhlf/labhlf-ov-sc     0.053s
```

Now you can change the test module in order to get an error. For example change

```
var chaincodeVersionTest = "2019-03-nn"
```

to

```
var chaincodeVersionTest = "2019-03-nx"
```

Save your test module and run the test again:

```
Go test
```

You should get similar result.

```
# labhlf/labhlf-ov-sc.test
2018-10-10 10:28:56.557 CEST [SmartContract hlf-<xx>-sc UNIT_TEST] Infof -> INFO 001
Starting Test_labhlfsc_Init...
Init returned wrong payload,
expected:  2018-11-nx
returned:  2019-03-nn
--- FAIL: Test_HhlfscInit (0.00s)
FAIL
exit status 1
FAIL   labhlf/labhlf-ov-sc 0.070s
```

Now, you have a good idea on how to build a smartcontract and the test module that will help to achieve the unit tests.

Let’s move forward with implementation of the business use case.

4 Implementation of the business use case

4.1 Use case

A consumer buys a product to Company A.
Company A packs the product and asks to Company B to ship the product.
Company A hands over the package to Company B.
Company B ships the package to the consumer location.
The consumer receives the package and acknowledges the reception of the product.

The problem to solve is the tracking of the product shipment.

We are creating one asset, the Package with the following attributes

- Identifier
- Description
- Status (Ready, Shipment, Shipped, Delivered, Received)
- Destination

The transactions will be:

- OrderShipment: the supplier (from Company A), once the order received from the consumer and then once the product packed, orders the shipment to the carrier (Company B)
- Ship(PackageId, status): the carrier (Company B) accounts the package corresponding to the order shipment, then provide the status of the shipment (Shipment, Shipped, Delivered).
- Acknowledgement(PackageId): the consumer, as soon as he has received the package, acknowledges the reception.

4.2 Implementation

We are updating the smart contract to account the previous use case. Then we will complete the test module accordingly to validate the implementation.

4.2.1 Declare the asset

We will start by describing the asset in the smart contract.
It consists in creating a structure containing the different attributes of the asset.
In the smartcontract, replace the following code:

```
type Labhlfsc struct {
}

var chaincodeVersion = "2019-03-nn"
```

By this one:

```
// Labhlfsc Describes the structure of the labhlf
type Labhlfsc struct {
}

// Package Describes the package that will be shipped to the destination
type Package struct {
    PackageID   string `json:"packageId"`
    Description string `json:"description"`
    Status      string `json:"status"`
    Destination string `json:"destination"`
}

var chaincodeVersion = "2019-03-nn"
var logger = shim.NewLogger("SmartContract Labhlfsc")
```

Observe that we have added a variable logger. It will allow to log all the steps of transactions.
The structure is described in JSON format. It requires to import the “encoding/json” package.
To do so, add this in the import section in the beginning of the file as described here:

```
import (
    "encoding/json"
…
```

4.2.2 Write the structure of the transactions

All the transactions will be called through the **Invoke** function. We are providing the name of the transaction in the first argument of the function, and the parameters of the transaction as additional arguments.
The structure of the Invoke function is the following. The args[0] which contains the name of the transaction is tested in a <switch … case…> structure, each case will contains the code of the different transactions.

```
    var args = stub.GetArgs()
    var err error

    logger.Info("Function: ", string(args[0]))

    // test the first argument which correspond to the transaction name
    switch string(args[0]) {
    case "OrderShipment":

       return shim.Success(nil)
    case "Ship":

       return shim.Success(nil)

    case "GetPackageStatus":
```

```
    return shim.Success(nil)

case "Acknowledgement":

    return shim.Success(nil)
default:
    //The transaction name is not known
    return shim.Error("unkwnon function")

}
```

Copy and paste the previous code in the body of the Invoke function, replacing the <return shim.success(nil)> instruction.

4.2.3 Write the transaction OrderShipment

In this transaction, we have as parameter the Package (which is provided by the supplier).
The Package information is provided in JSONformat. We are checking the Package information, before writing the Package in the ledger.

An asset is stored in the Ledger in a DB called Worldstate. The asset is stored in JSON format as the value of a (key:value) record.

To write assets in the ledger, the Hyperledger Fabric API provides the following function:

- `PutState(key string, value []byte)` *will create or will update the key attribute with the value passed as second parameter. It will return an error (type error in Go) in case of issue.*
-

If the transaction is successful, then it returns shim.Success(<ReturnedValue>). ReturnedValue is the output of the transaction in the shape of array of byte ([] byte).
If the transaction fails, then it returns shim.Error(<ErrorMessage>).

Here is the code of the transaction. Copy and paste it in the <case “OrderShipment”> section of the Invoke function.

```
    // Create a Package asset
    logger.Info(" OrderShipment function, Value:  ", string(args[1]))

    // Check input format
    var pack Package
    err = json.Unmarshal(args[1], &pack)
    if err != nil {
        fmt.Println(err)
        jsonResp := "Failed to Unmarshal package input data"
        return shim.Error(jsonResp)
    }
    // verify mandatory fields
```

```
        missingFields := false
        missingFieldsList := "Missing or empty attributes: "
        if pack.Destination == "" {
            missingFieldsList += "Destination"
            missingFields = true
        }
        // ..... Complete the fields checking
        if missingFields {
            fmt.Println(missingFieldsList)
            return shim.Error(missingFieldsList)
        }
        pack.Status = "READY"

        var packageBytes []byte
        packageBytes, err = json.Marshal(pack)
        if err != nil {
            fmt.Println(err)
            return shim.Error("Failed to marshal Package object")
        }

        err = stub.PutState(pack.PackageID, packageBytes)
        if err != nil {
            fmt.Println(err)
            return shim.Error("Failed to write packageBytes data")
        }

        logger.Info(" OrderShipment function, txid: ", stub.GetTxID())
        return shim.Success([]byte(`{"txid":"` + stub.GetTxID() + `","err":null}`))
```

Complete the transaction, adding the checking of the fields of the Package structure (Description).

You can now compile your SmartContract.

Add the following function in the test module. It creates a Package identified by P1, then it simulates the transaction OrderShipment through the **stub.MockInvoke** instruction. Then it compares the result of the transaction with the expected result.

```
//function to test the transaction OrderShipment
func Test_Labhlfsc_OrderShipment(t *testing.T) {
    scc := new(Labhlfsc)
    stub := shim.NewMockStub("Test_Labhlfsc_OrderShipment", scc)
    loggertest.Infof(" Starting Test_Labhlfsc_OrderShipment...")

    var packageValue string
    packageValue = `{"packageID":"P1","description":"Package for product P001","destination":
"Montpellier, FRANCE, 34006"}`
    //var txId string
```

```
    txID := "PACK_1"
    res := stub.MockInvoke(txID, [][]byte{[]byte("OrderShipment"), []byte(packageValue)})
    if res.Status != shim.OK {
        //fmt.Println(string(res.Message))
        fmt.Println("Invoke OrderShipment failed ", res.Message)
        t.FailNow()
    }

    // check returned value
    var testvalue string
    testvalue = `{"txid":"PACK_1","err":null}`
    // check returned payload
    if res.Payload == nil {
        fmt.Println("Invoke OrderShipment returned wrong payload, expected txid => nil
returned !!!")
        t.FailNow()
    }
    if string(res.Payload) != testvalue {
        fmt.Println("Invoke OrderShipment returned wrong payload, \nexpected: ", testvalue, "
\nreturned: ", string(res.Payload))
        t.FailNow()
    }

    // check stored value in worldstate
    testvalue = `{"packageId":"P1","description":"Package for product
P001","status":"READY","destination":"Montpellier, FRANCE, 34006"}`
    checkState(t, stub, "P1", testvalue)


    loggertest.Infof(" Test_Labhlfsc_OrderShipment completed")

}
```

At the end of the test, it does a checkState: it checks the assets stored in the WorldState.
This function is described here after. It retrieves the content of the ledger using the stub.State function.
Copy this code and paste it in the test module before the Test function.

```
// checkState This function checks the value of an attribute of the worldstate
func checkState(t *testing.T, stub *shim.MockStub, name string, value string) {
    bytes := stub.State[name]
    if bytes == nil {
        //fmt.Println("State", name, "failed to get value")
        loggertest.Error(stub.Name, " State", name, "failed to get value")
        t.FailNow()
    }
    if string(bytes) != value {
        //fmt.Println("State value", name, "was not", value, "as expected")
```

```
        loggertest.Error(stub.Name, " State value", name, " \nvalue: ", string(bytes), "
\nexpected: ", value)
        t.FailNow()
    }
}
```

Test the SmartContract: open a Linux terminal, move to the folder of your test module and run

```
go test
```

You should get similar results:

```
# labhlf/labhlf-ov-sc.test
ld: warning: text-based stub file /System/Library/Frameworks//CoreFoundation.framework/CoreFoundation.tbd and
library file /System/Library/Frameworks//CoreFoundation.framework/CoreFoundation are out of sync. Falling back to
library file for linking.
ld: warning: text-based stub file /System/Library/Frameworks//Security.framework/Security.tbd and library file
/System/Library/Frameworks//Security.framework/Security are out of sync. Falling back to library file for linking.
ld: warning: text-based stub file /System/Library/Frameworks//IOKit.framework/Versions/A/IOKit.tbd and library file
/System/Library/Frameworks//IOKit.framework/Versions/A/IOKit are out of sync. Falling back to library file for
linking.
2018-10-11 12:00:03.235 CEST [SmartContract labhlf-<xx>-sc UNIT_TEST] Infof -> INFO 001  Starting
Test_labhlfsc_Init...
2018-10-11 12:00:03.236 CEST [SmartContract labhlf-<xx>-sc UNIT_TEST] Infof -> INFO 002 Test_labhlfsc_Init completed
2018-10-11 12:00:03.236 CEST [SmartContract labhlf-<xx>-sc UNIT_TEST] Infof -> INFO 003  Starting
Test_Labhlfsc_OrderShipment...
2018-10-11 12:00:03.236 CEST [SmartContract Labhlfsc] Info -> INFO 004 Function:  OrderShipment
2018-10-11 12:00:03.236 CEST [SmartContract Labhlfsc] Info -> INFO 005  OrderShipment function, Value:
{"PackageID":"P1","Description":"Package for product P001","Destination": "Montpellier, FRANCE, 34006"}
2018-10-11 12:00:03.236 CEST [SmartContract Labhlfsc] Info -> INFO 006  OrderShipment function, txid:  PACK_1
2018-10-11 12:00:03.236 CEST [SmartContract labhlf-<xx>-sc UNIT_TEST] Infof -> INFO 007  Test_Labhlfsc_OrderShipment
completed
PASS
ok      labhlf/labhlf-ov-sc     0.059s
```

We observe that some logs are provided by the smartcontract and some by the test module.
You can change some values (for example change the testValue) in the test module to see the behavior in case of errors. Then execute again the test.

4.2.4 Write the transaction Ship

In this transaction, we have 2 parameters: the PackageID and the status of the shippement.
The principle is to retrieve the concerned Package from the ledger, then update it with the new status (passed in parameter).

To retrieve an asset from the ledger, the Hyperledger Fabric API provides the following function:

- `GetState(key string)` *that will return the value of the attribute as an array of byte ([] byte) as first returned argument and an error as second returned argument.*

The following code implements the transaction.
Copy and paste it in the <case “Ship”> section of the Invoke function.

```
// Change the status of the package
// parameters : packageId, status
logger.Info(" Shipp function, Package id:  ", string(args[1]), " - Status ", string(args[2]))
```

```
// Retrieve the parameters
var packageID, status string
var packageBytes []byte
var pack Package
packageID = string(args[1])
status = string(args[2])

// Test the value of the status : it should be SHIPMENT, SHIPPED, DELIVERED
// ... TBC .... Implement the test

// Get the Package in the ledger based on the PackageId
packagebytes, err := stub.GetState(packageID)
if err != nil {
      return shim.Error("Failed to get package " + packageID)
}

err = json.Unmarshal(packagebytes, &pack)
if err != nil {
        fmt.Println(err)
        jsonResp := "Failed to Unmarshal package data " + packageID
        return shim.Error(jsonResp)
}
// Test the value of the current status:
// if the new status is SHIPMENT, the current one should be READY
// if the new status is SHIPPED, the current one should be SHIPMENT
// if the new status is DELIVERED, the current one should be SHIPPED
// ... TBC .... Implement the test

//Update the status
pack.Status = status
packageBytes, err = json.Marshal(pack)
if err != nil {
        fmt.Println(err)
        return shim.Error("Failed to marshal Package object")
}
// Update the package in the ledger
err = stub.PutState(pack.PackageID, packageBytes)
if err != nil {
        fmt.Println(err)
        return shim.Error("Failed to write packageBytes data")
}
// Submit an event to inform about the status change
 err = stub.SetEvent("Shipp", packageBytes)
if err != nil {
        fmt.Println(err)
        return shim.Error("Failed to raise enregistrePalette event!!!")
}
```

```
logger.Info(" shipp function, txid: ", stub.GetTxID())
return shim.Success([]byte(`{"txid":"` + stub.GetTxID() + `","err":null}`))
```

Complete the missing tests described in comment in the code.
Then create the test function in the Test module, duplicating the Test_Labhlfsc_OrderShipment function.

4.2.5 Write the transaction GetPackageStatus

In this transaction, there is 1 parameter, the PackageID.
The principle is to retrieve the concerned Package from the ledger, then return the PackageID and the status of the Package.

The following code implements the transaction.
Copy and paste it in the <case “GetPackageStatus”> section of the Invoke function.

```
// Get the status of the package
        // parameters : packageId
        logger.Info(" GetPackageStatus function, Package id:  ", string(args[1]))

        // Retrieve the parameters
        var packageID string
        var pack Package
        packageID = string(args[1])

        // Get the Package in the ledger based on the PackageId
        packagebytes, err := stub.GetState(packageID)
        if err != nil {
            return shim.Error("Failed to get package " + packageID)
        }

        err = json.Unmarshal(packagebytes, &pack)
        if err != nil {
            fmt.Println(err)
            jsonResp := "Failed to Unmarshal package data " + packageID
            return shim.Error(jsonResp)
        }

        return shim.Success([]byte(`{"PackageID":"` + pack.PackageID + `","Status":"` +
pack.Status + `"}`))
```

When the smartcontract is updated and compiled, update the test module to add the test function of this transaction.

4.2.6 Write the transaction Acknowledgment

In this transaction, there is 1 parameter the PackageID.

The principle is to retrieve the concerned Package from the ledger, then to update the status to RECEIVED only if the current status is DELIVERED.

Update the test module accordingly implementing the test function of this transaction.

Notices

This information was developed for products and services offered in the U.S.A.
IBM may not offer the products, services, or features discussed in this document in other countries.

Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.
IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.
For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.
This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.
Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.
Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM

products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.
All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.
COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Appendix A. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	AIX	CICS	ClearCase	ClearQuest	Cloudscape
Cube Views	DB2	developerWorks	DRDA	IMS	IMS/ESA
Informix	Lotus	Lotus Workflow	MQSeries	OmniFind	
Rational	Redbooks	Red Brick	RequisitePro	System i	
System z	*Tivoli*	*WebSphere*	*Workplace*	*System p*	

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.
IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.
Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

NOTES

NOTES

IBM Software