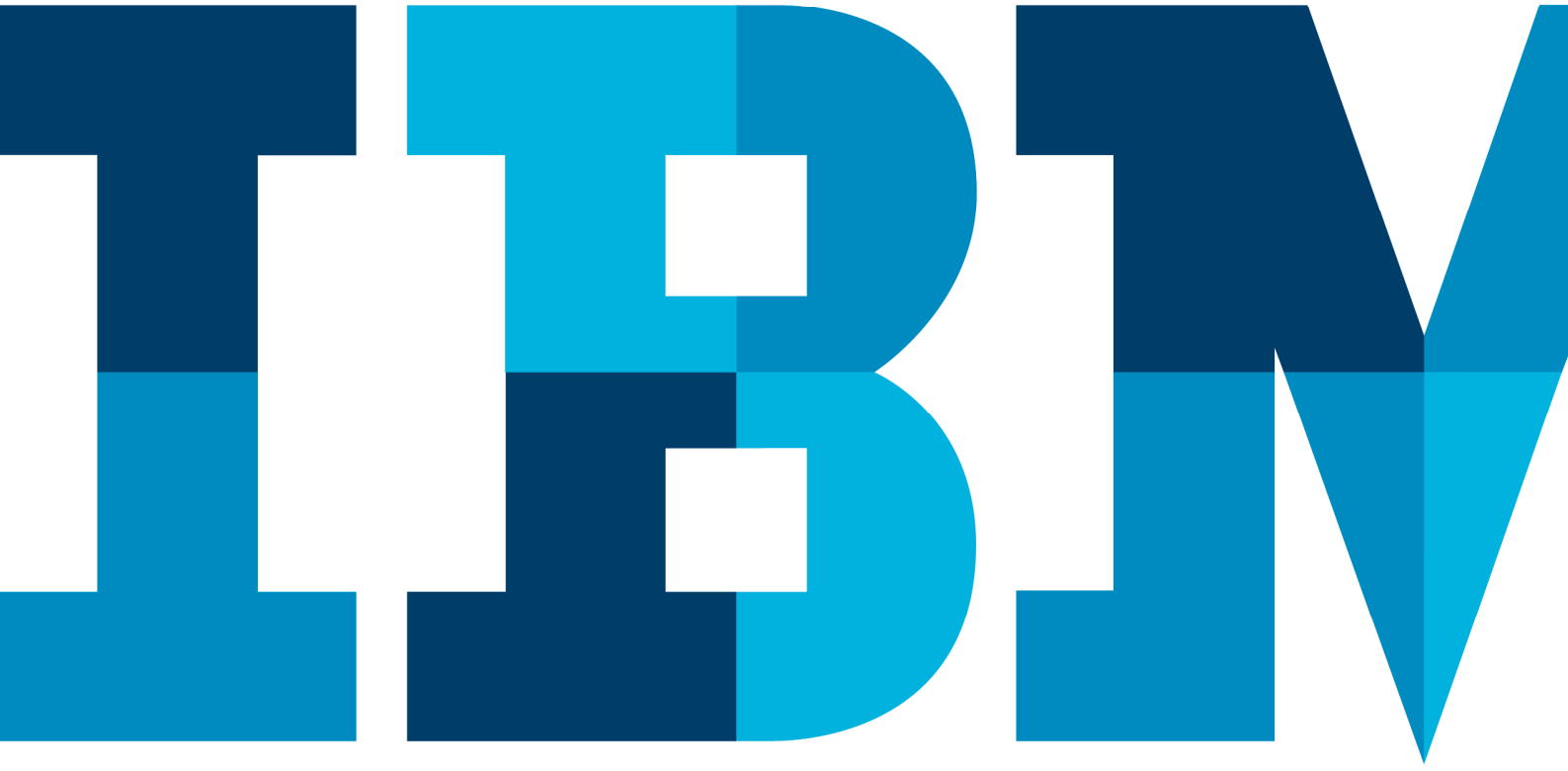
IBM Blockchain

IBM Blockchain Hands-On Introduction to Chaincode

Lab One



IBM

Contents

Overview

The aim of this lab is to teach you the basics of writing chaincode in Go and how to deploy and interact with this chaincode when it is uploaded to the network.

Introduction

Pre-requisites:

- 2 cores
- 4GB RAM
- VMWare V12+
- The lab virtual machine

The virtual machine is based on Linux Ubuntu 16.04 and contains Hyperledger Fabric V1.0, Golang, Git, Visual Studio Code and Firefox.

A network needs to be visible to the virtual machine (even if the network is just to the host environment). If you do not see the up/down arrows in the status bar at the top of the screen, or if you receive errors about no network being available, please tell the lab leader. The virtual machine might need to be reconfigured in NAT mode.

There are no additional files or software that is proprietary to the lab in the virtual machine. This means that the lab may be run on a machine without the without a lab virtual machine if Hyperledger Fabric and the other pre-requisites have been installed.

It is recommended that students have previously completed the Blockchain Explained and Blockchain Explored labs.

Section 1. Deploying and interacting with chaincode

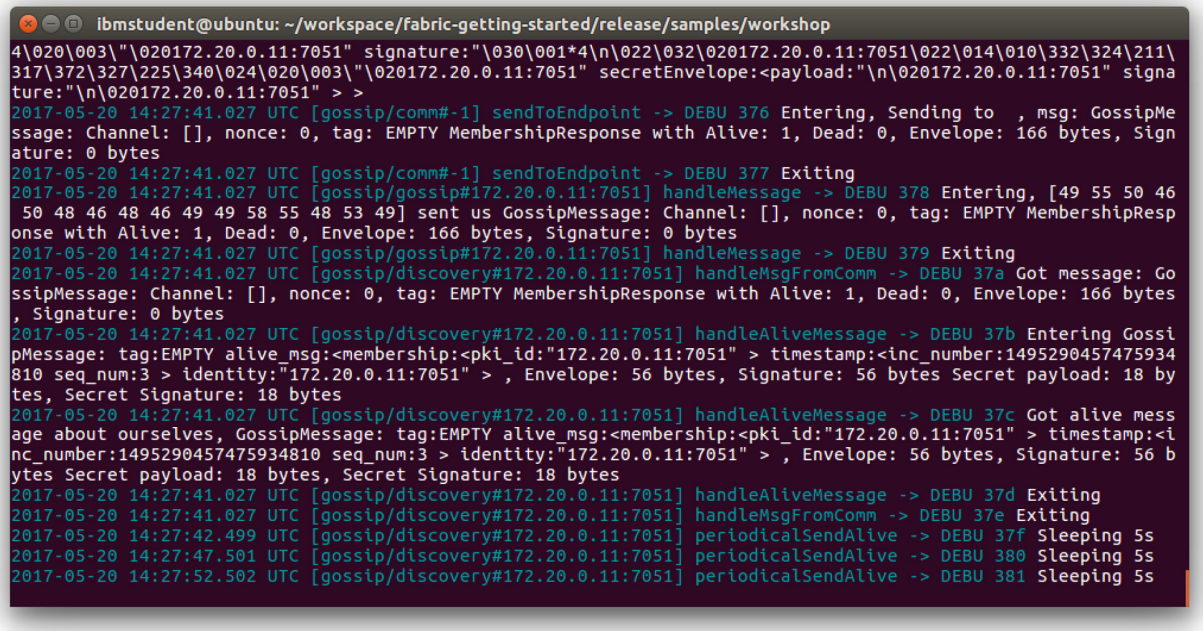
This section will briefly cover the process of standing up a network and go over how to install chaincode on the test peer.

1.1. Standing Up

Navigate to `~/workspace/fabric-getting-started/release/samples/workshop` and issue the following command:

```
./network_setup.sh up
```

When the output starts to look like this:



Exit out with control + c Issue the following:

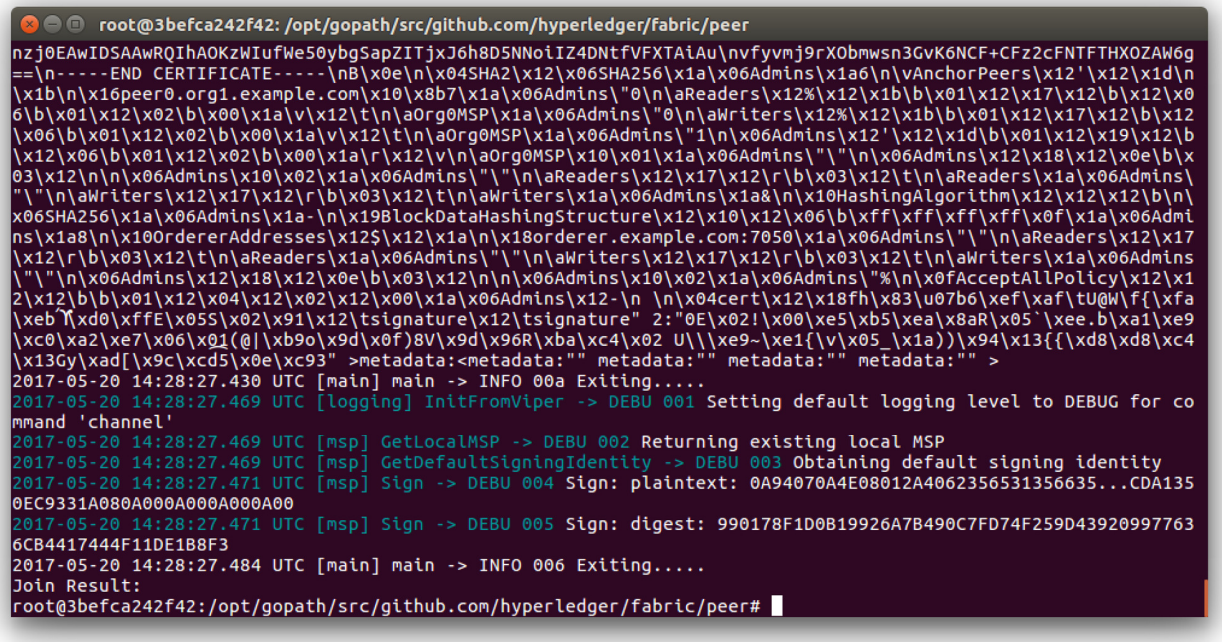
```
./cli-connect.sh
```

These scripts will create the bootstrap artefacts, stand up a single peer network and set it up with a channel ready for deploying chaincode to.

The cli-connect.sh script then opens up an interactive terminal window with the cli container which is a Docker container with a collection of tools for connecting to the peers (the tools can be found and compiled from the Hyperledger Fabric source). The install, instantiate, invoke and query operations will all be executed from within the CLI container. If at any point you exit the CLI container by accident, simply issue the following:

```
docker exec -it cli bash
```

The CLI should then look like this



Finally, set some environment variables to let you issue commands to one of the peers, issue the following:

```
CORE_PEER_LOCALMSPID=Org0MSP
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com
```

Note that if you exit and re-run the cli-connect.sh script you will also need to re-set these variables. We're going to be aiming more on development in this workshop rather than the intricacies of how Hyperledger Fabric works but for more information please consult the "Getting started" section of the fabric read-the-docs site: https://hyperledger-fabric.readthedocs.io/en/latest/getting_started.html

1.2. Installing

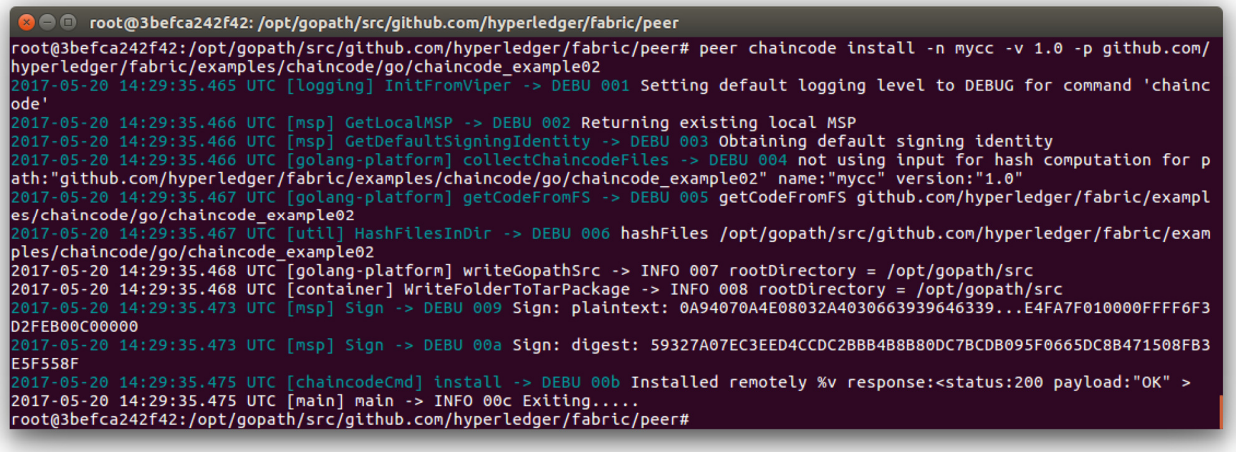
Installing chaincode means that you are allowing a peer the ability to endorse it, whether or not it will be asked for endorsement is down to the endorsement policy of the channel. Irrespective of whether chaincode is installed on a peer or not, it will still receive updates about the chaincode through the gossip network.

To install chaincode on a peer, issue the following:

```
peer chaincode install -n mycc -v 1.0 -p
github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02
```

If successful, the peer will print out a collection of installation logs finished with the following line:

```
Installed remotely %v response:<statys:200 payload:"OK">
```



`peer` in this instance refers to the peer client application that is used to issue commands to peers. The command syntax is as follows:

```
peer chaincode install –n <chaincode id> -v <version> -p <path to the code>
```

The chaincode id is a string chosen by you to act as an identifier of this chaincode. When the chaincode is invoked you will supply this id so that Fabric knows which chaincode you wish to invoke.

The path to the chaincode is a path inside the peer – if you inspect any of the `docker-compose.yaml` files you will find that the `chaincode` directory in `workshop/chaincodes` is mounted to `github.com/hyperledger/fabric/examples/chaincode/go` in the peer.

A version number is also supplied, we'll see what is used for when we look at upgrading chaincode.

Chaincode must be installed separately on each peer you wish to run it. For the purposes of this workshop however we will only bother installing it on one peer.

1.3. Instantiating

Instantiating chaincode means that you are making the chaincode available to be run, it is a channel-wide command so only needs to be run from one peer.

To instantiate the chaincode run the following:

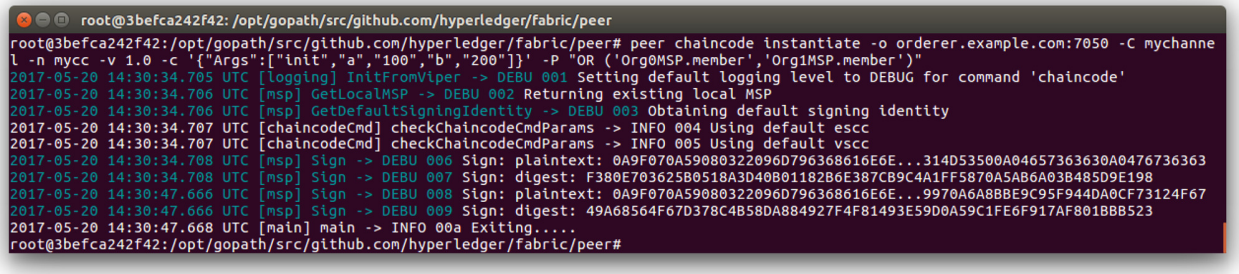
```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org0MSP.member','Org1MSP.member')"
```

The syntax is as follows:

```
peer chaincode instantiate –o <orderer address> -C <channel id> -n <chaincode id> -v <version string> -c <Init function arguments> -P <endorsement policy>
```

More information on endorsement policies can be found here.

If successful, the command will return nothing but will exit with code `00a` like so:



When you install chaincode you are simply uploading it to the peer and attaching to it a version string. If upon instantiation the code is found to have errors then it must be re-installed with a new version string (1.1 following 1.0 for example).

To spot this in advance it is advisable to navigate into the directory containing the chaincode source and issue go `build` – this will compile the chaincode into an executable (same process that happens on the peer) and allow you to spot and compile errors.

You must remove the compiled executable with `rm <executable name>` otherwise Fabric will attempt to upload the executable too and this will cause errors.

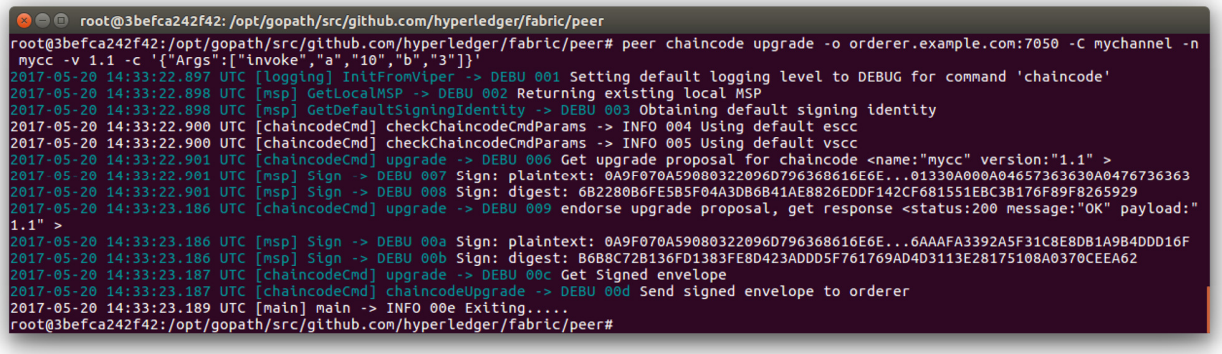
1.4. Updating

To update chaincode you first need to install the new version. This is done by issuing the install command again, supplying the id of the chaincode you want to update, a higher version number and the code you wish to push:

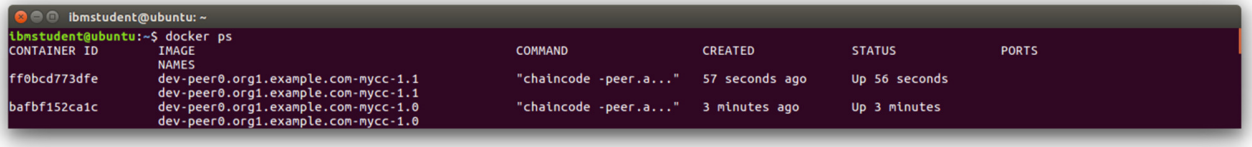
```
peer chaincode install -n mycc -v 1.1 -p
github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02
```

When the new code is installed, issue the upgrade command to deploy the new:

```
peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n mycc -v 1.1 -c
'{"Args":["invoke","a","10","b","3"]}'
```



It should be noted that the docker containers for the old chaincode will still be running despite the update. It is recommended to remove these and doing so will not affect the more recent containers.



Do so by issuing `docker rm <container id of the previous chaincode>`

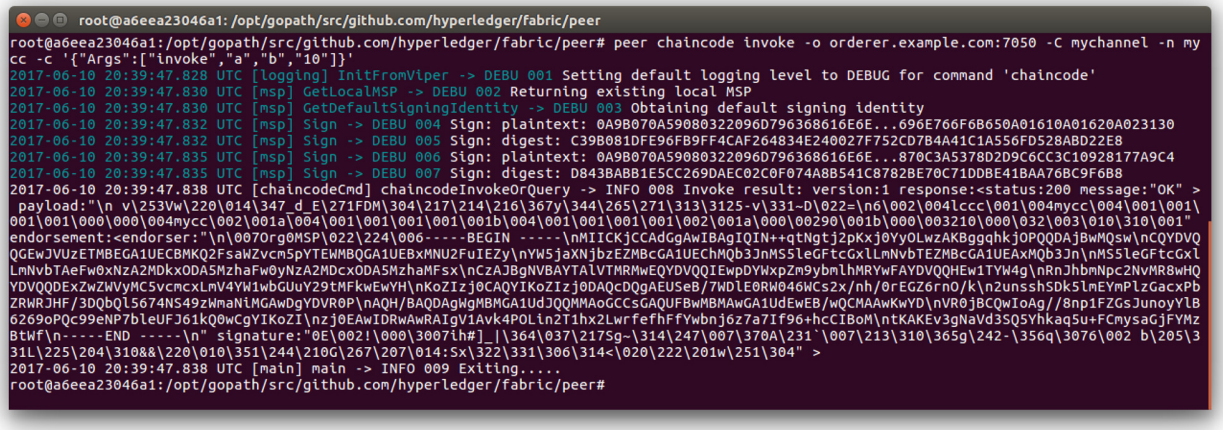
1.5. Invoking & Querying

Invoking and Querying chaincode are the two means by which you can interact with it.

Invoke is used to make calls to chaincode that will modify the world state (i.e. Created, Update or Delete data):

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n mycc -c
'{"Args":["invoke","a","b","10"]}'
```

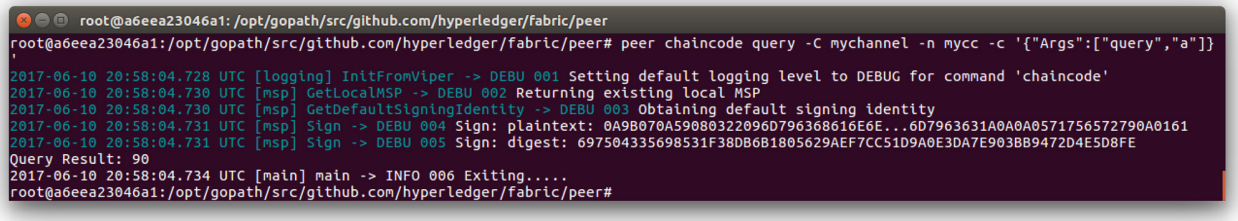
Invokes return nothing apart from a success or fail response, below you can see in the line below the green debugging output a familiar structure: `<status: 200 message: "OK">` this denotes the invoke was successful:



Query is used to make calls to chaincode that read data from the world state and return it:

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

Queries return data in JSON format, either simply as a string or returning a more complex data structure:



1.6. Troubleshooting

a. Creator Certificate is not valid (`peer chaincode install`)

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = The creator certificate is not valid, err The supplied identity is not valid, Verify() returned x509: certificate signed by unknown authority
```

To resolve this set the environment variables in 3.1.

b. Failed to deserialize creator identity (`peer chaincode instantiate`)

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = Failed to deserialize creator identity, err MSP Org0MSP is unknown
```

To resolve this have the peer you are deploying to join a channel with the chaincode on it.

c. Error starting container (`peer chaincode instantiate`)

```
Error endorsing chaincode: rpc error: code = 2 desc = Error starting container: API
error (404): {"message":"network e2e_default not found"}
```

Navigate into `peer_base` and open `peer-base-no-tls.yaml`. Examine the CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE variable, the format should be `<dirname>_default` where `dirname` is the `samples/<dirname>/peer_base` you are currently in.

This section will examine the contents of a simple chaincode file to illustrate basic structure.

Section 2. Introduction to Chaincode

2.1. Open up the sample file

Navigate to:
`~/workspace/fabric-getting-started/release/samples/workshop/chaincodes/go/strings`
and open `strings.go`.

2.2. Overview of Structure

a. Imports

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

The basic imports include the shim and the definition of the Peer Response. The shim is what translates the API calls you make into commands the peer can understand. The Peer Response is a data structure sent back by the peer through the event service triggering Success and Failure events.

b. Struct

```
type StringsChaincode struct {
}
```

The shim expects the chaincode to be structured as an object with an Init() and an Invoke() function, as such an object needs to be created for this purpose.

c. Main

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

The main function tells the shim to spin up a new instance of this chaincode when the code is executed, or rather a new instance of the object described in **b**.

It should be noted that the error check performed in this function can be applied to most `shim` functions (in that almost all return an `err` object that can be inspected). For the purposes of keeping code to a minimum and easiest to read these error checks have been omitted from the sample code in this Lab however it is recommended to insert them regardless as it aids with bug fixing.

d. Init()

```
func (t *StringsChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

    // Declare variable
    fmt.Printf("Declaring")
    key := "Test"
    value := "Hello, World!"

    // Place into world state
    fmt.Printf("PutState")
    stub.PutState(key, []byte(value))

    // Exit with success
    return shim.Success(nil)
}
```

`Init` is called when the code is instantiated and again whenever it is updated.

In the above, the initial value of the test key is set. While in this situation this is fine it is recommended that any initial values for keys and other world state data should not be placed here as they will be reset every time the chaincode is upgraded. A better practice is to write a supplementary init function that is called by an invoke, as such it can be called only when the chaincode starts.

e. Invoke()

```
func (t *StringsChaincode) read(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
      valueAsBytes, _ := stub.GetState(args[0])
      return shim.Success(valueAsBytes)
}

func (t *StringsChaincode) overwrite(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
      stub.PutState(args[0], []byte(args[1]))
      return shim.Success(nil)
}

func (t *StringsChaincode) append(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
      oldValueAsBytes, _ := stub.GetState(args[0])
      var newValue bytes.Buffer
      newValue.Write(oldValueAsBytes)
      newValue.WriteString(args[1])
      stub.PutState(args[0], newValue.Bytes())
      return shim.Success(nil)
}

func (t *StringsChaincode) delete(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
      stub.DelState(args[0])
      return shim.Success(nil)
```

```
}

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

	// Grab the inputs to the transaction
	function, args := stub.GetFunctionAndParameters()
	fmt.Println("invoke has been called with", function, "and", args)

	// Call the apropriate function
	if function == "read" {
		return t.read(stub, args)
	} else if function == "overwrite" {
		return t.overwrite(stub, args)
	} else if function == "append" {
		return t.append(stub, args)
	} else if function == "delete" {
		return t.delete(stub, args)
	}

	// If 'function' doesn't match any of the above then return an error
	return shim.Error("Invalid invoke function name.")
}
```

`Invoke` is called whenever the user makes Invoke and Query requests.

The normal structure is to have a series of `if` and `else if` statements that route the transactions to the functions they are requesting. As seen in the above, the `if` chain checks if the transaction is asking to change or read the value I the world state.

It is possible to supply 'invoke' function names to query transactions and vice versa. However in calling an invoke function with a query no edits will be made and in calling a query function with an invoke will not return any content (only a success/fail message).

2.3. Using Strings

a. Install

Issue the following to install Strings:

```
peer chaincode install -n strings -v 1.0 -p
github.com/hyperledger/fabric/examples/chaincode/go/strings
```

b. Instantiate

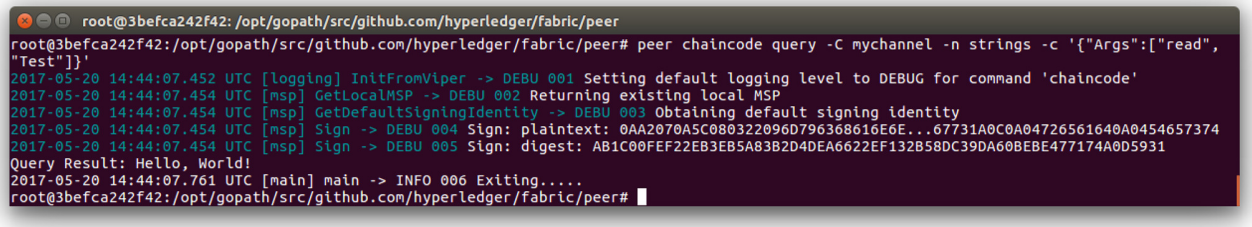
Issue the following to instantiate Strings:

```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n strings -v
1.0 -c '{"Args":["init"]}' -P "OR ('Org0MSP.member','Org1MSP.member')"
```

c. Query

Query the current state, it should come back with ‘Hello, World!’:

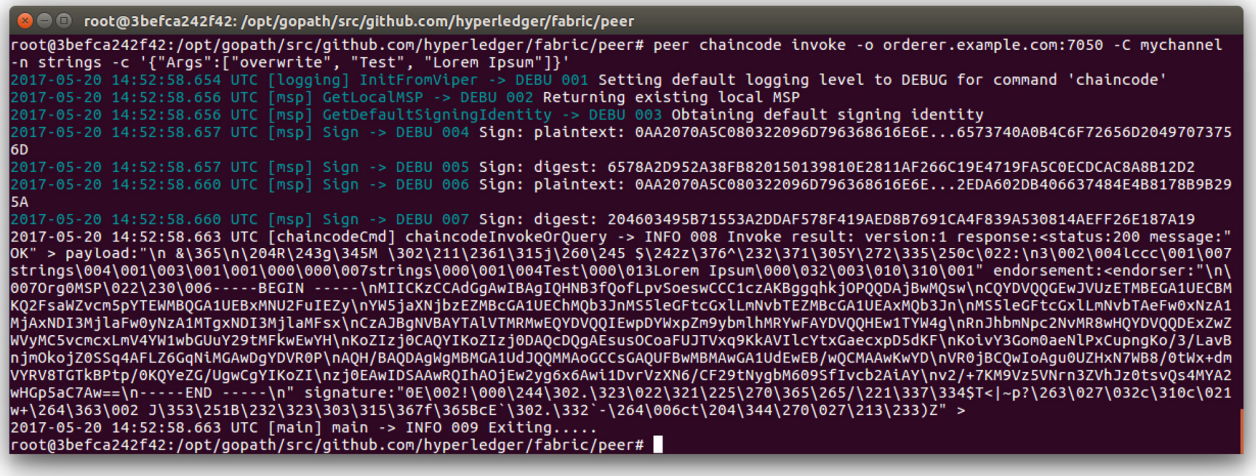
```
peer chaincode query -C mychannel -n strings -c '{"Args":["read", "Test"]}'
```

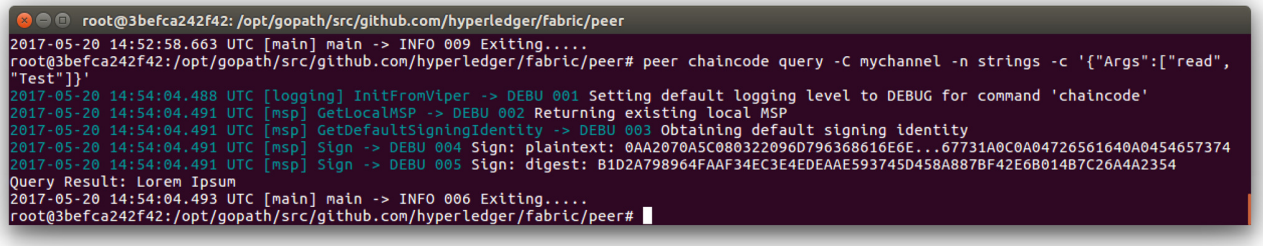


d. Invoke

Now change the value, substitute ‘Lorem Ipsum’ with alternative text:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n strings -c
'{"Args":["overwrite", "Test", "Lorem Ipsum"]}'
```



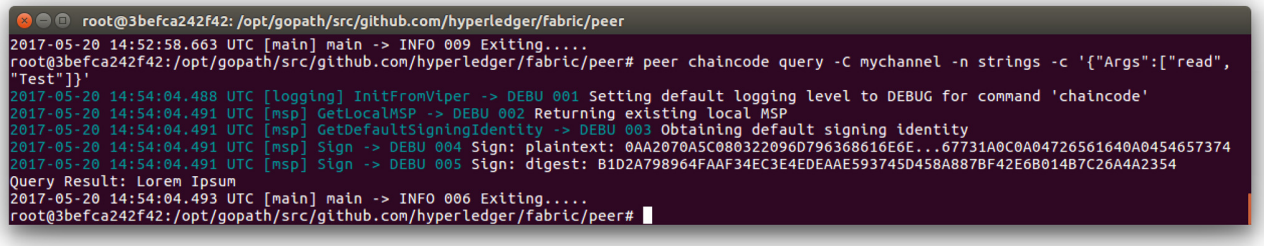


a. Check the Invoke worked

Issue the query command again to check that the invoke transaction changed the world state:

```
peer chaincode query -C mychannel -n strings -c '{"Args":["read", "Test"]}'
```

The response should now be `"Lorem Ipsum"` or something else if you used a different string:

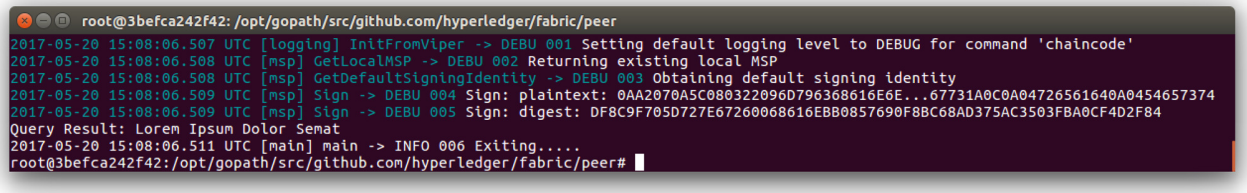


b. More Invokes

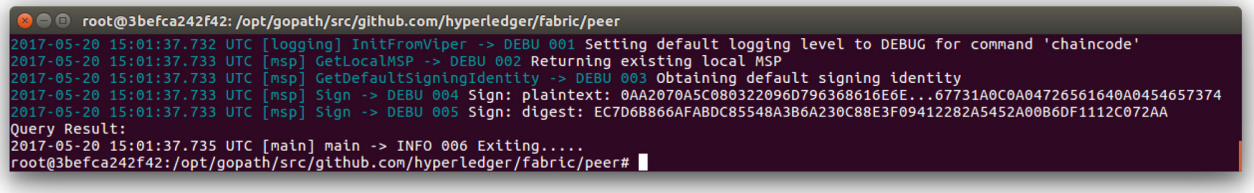
Try reading the definitions of `append()` and `delete()` from `strings.go` and issuing Invokes for them. Remember, the syntax for an invoke is as follows:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n strings -c
'{"Args":["overwrite", "Test", "Lorem Ipsum"]}'
```

The output of a query issued after an `append` with `"Lorem Ipsum"` supplied as the argument would be the following:



The output of a query issued after a `delete` with `"Test"` supplied as the argument would be the following:



Section 3. Introduction to Chaincode

In this section you will write out the code in Strings that was seen in the last section. When this is complete the codebase will be revised to handle data structures. The chaincode will be written in Go, for more information about the language see https://gobyexample.com/.

3.1. Create a basic file

cd to ~/workspace/fabric-getting-started/release/samples/workshop/chaincodes/go Create a new folder called `marbles` and copy into it the `skeleton.go` file in ~/workspace/fabric-getting-started/release/samples/workshop/chaincodes/go/skeleton, and rename it `marbles.go`.

This skeleton code will serve as the basis for this section. We will begin by writing `strings.go` and then transforming it into a program that stores complex data structures.

Visual Studio Code has a Go plugin that should automatically detect missing dependencies and add them to the import clause. In the event that code is throwing compile errors it is worth checking the import statement for some of the following:

"`bytes`" – Handles byte buffers in `append()`
"`encoding/json`" – Handles un/marshalling complex data structures
"`fmt`" – Roughly akin to `stdio` in C
"`strconv`" – String conversion library

3.2. Storing Basic Data

a. Adding Data

In the body of `Init`, add a line that stores a key and a string value when the chaincode is instantiated:

```
func (t *Chaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
      stub.PutState("Test", []byte("Hello, World!"))
      return shim.Success(nil)
}
```

Hyperledger Fabric stores data as Key/Value pairs. To store a key use shim's `PutState` method where a key is a `string` and value is a `[]byte` representing the value:

```
stub.PutState(<key>, <value>)
```

b. Reading data

Add a new function above `Invoke` called `read`:

```
func (t *Chaincode) read(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
      valueAsBytes, _ := stub.GetState(args[0])
      return shim.Success(valueAsBytes)
```

```
}
```

Retrieving data from the world state uses the shim's `GetState` method:

```
buffer, errorBuffer := stub.GetState(args[0])
```

This reads the contents into the buffer specified and also returns an error object if there was any trouble. Note that in the above we do not convert the data from it's `byte` array format, the shim takes a `byte` array as an argument so we simply pass it in.

We'll also need to update `Invoke`. Currently, `read` will not called, for this to happen we need to add some code to `Invoke`: to route allow any incoming queries to make use of this function:

```
func (t *Chaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
      function, args := stub.GetFunctionAndParameters()
      if function == "read" {
            return t.read(stub, args)
      }
      return shim.Error("Invalid invoke function name.")
}
```

Now when an invoke is called, the request will be funnelled into the `read` function. Deploy the code using the instructions in Section 2.3 and issue a query transaction for `read`. You should receive "`Hello, World!`" in response.

c. Testing read

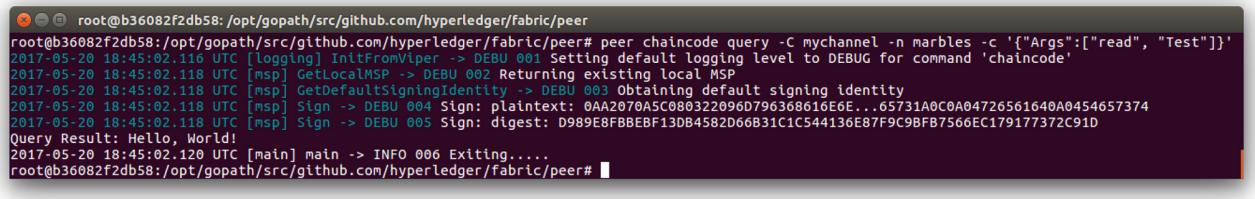
At the docker commandline deploy the marbles code you have created:

```
peer chaincode install -n marbles -v 1.0 -p
github.com/hyperledger/fabric/examples/chaincode/go/marbles

peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n marbles -v
1.0 -c '{"Args":["init"]}' -P "OR ('Org0MSP.member','Org1MSP.member')"
```

Run a `read` query:

```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "Test"]}'
```

If successful, this should output the value written in by `Init`:



d. Modifying data

Add the following underneath `read`:

```
func (t *Chaincode) overwrite(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
	stub.PutState(args[0], []byte(args[1]))
	return shim.Success(nil)
}
```

The simplest way of modifying data is to simply write over it, if you issue PutState with an existing key and new data it will overwrite the old data irrespective of what it was in relation to the new data. However, to modify existing data and put it back a combination of `GetState` and `PutState` are used. Add the following below `overwrite`:

```
func (t *Chaincode) append(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
	oldValueAsBytes, _ := stub.GetState(args[0])
	var newValue bytes.Buffer
	newValue.Write(oldValueAsBytes)
	newValue.WriteString(args[1])
	stub.PutState(args[0], newValue.Bytes())
	return shim.Success(nil)
}
```

Here the old value is read from the world state, edited and then written back into the world state.

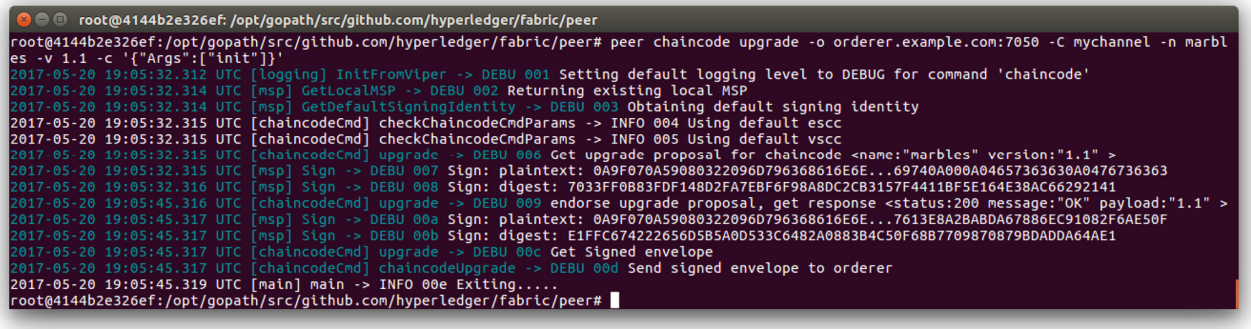
Make the following edit to the `if` chain in `Invoke`:

```
	if function == "read" {
		return t.read(stub, args)
	} else if function == "overwrite" {
		return t.overwrite(stub, args)
	} else if function == "append" {
		return t.append(stub, args)
	}
```

e. Testing overwrite

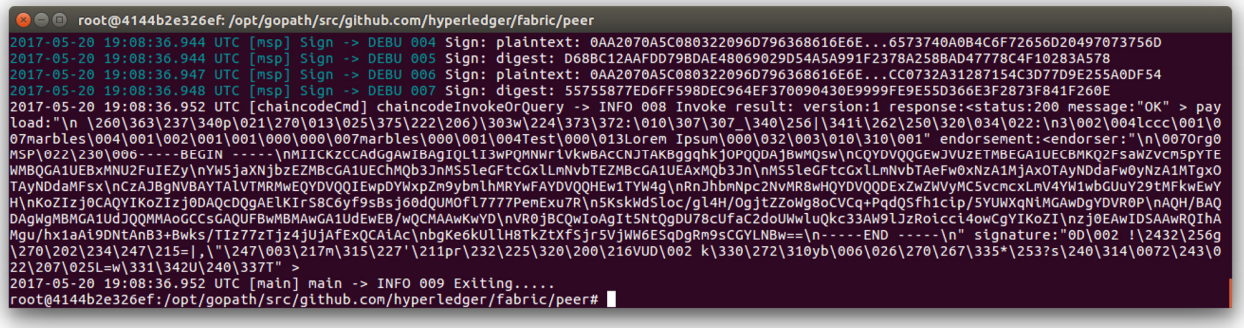
Install and upgrade the marbles chaincode by issuing the following:

```
peer chaincode install -n marbles -v 1.1 -p
github.com/hyperledger/fabric/examples/chaincode/go/marbles

peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n marbles -v 1.1 -c
'{"Args":["init"]}'
```



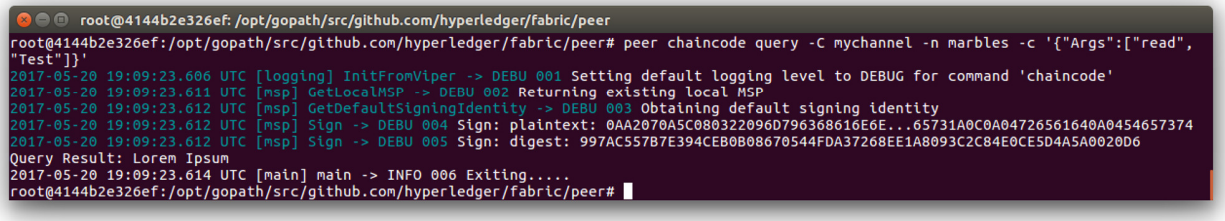
While the only data in the world state is re-written by the code in `Init`, it is worth noting that the data is preserved between versions of the chaincode, i.e. no data will be lost from the world state because the chaincode it is attached to has changed. We will see this in the testing phase for **f**.
Issue an overwrite modifying the data:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["overwrite","Test","Lorem Ipsum"]}'
```



Issuing a read query will show the data has been modified:

```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "Test"]}'
```



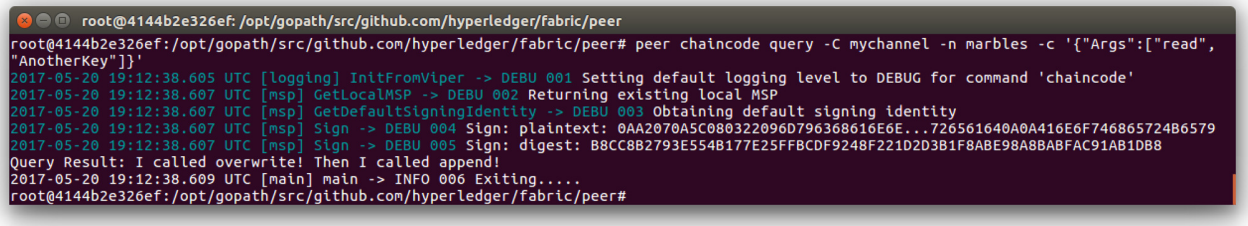
Now add a new key by using overwrite and append something to it:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["overwrite","AnotherKey","I called overwrite!"]}'
```

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["append","AnotherKey"," Then I called append!"]}'
```

Query the new key to see the effects:

```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "AnotherKey"]}'
```



f. Deleting data

Finally, add the following function:

```
func (t *Chaincode) delete(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
      stub.DelState(args[0])
      return shim.Success(nil)
}
```

Also add the delete function to the `if` chain in `Invoke`. To delete keys from the world state we use the shim's `DelState` function, supplying the key you wish to delete as an argument.

Please note that this does not contravene the Immutability characteristic – Blockchains are immutable and the Blockchain recording state changes to this channel will maintain a permanent record that details the lifecycle of any asset you create. **This function edits the World State, not the Blockchain** – the blockchain simply records what edits are made.

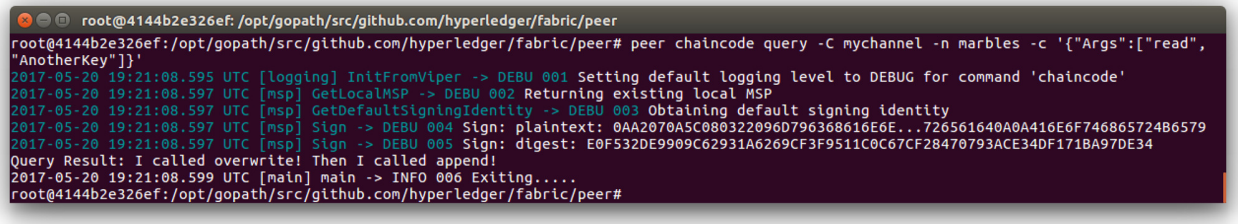
g. Testing Deletions

Install and upgrade the chaincode:

```
peer chaincode install -n marbles -v 1.2 -p
github.com/hyperledger/fabric/examples/chaincode/go/marbles

peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n marbles -v 1.2 -c
'{"Args":["init"]}'
```

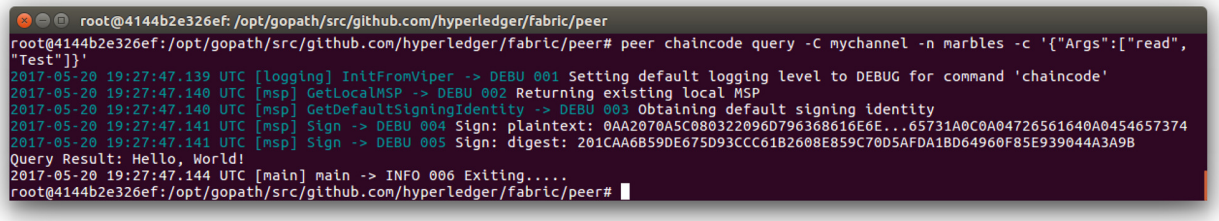
Try issuing a read query on the Key we defined last time:

```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "AnotherKey"]}'
```



Now issue a read query looking at the value set in `Init`:

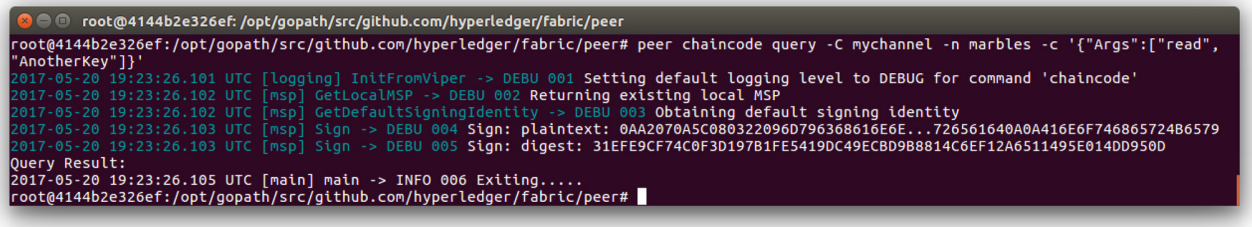
```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "Test"]}'
```



The `Init` value has been reset as each time the code is upgraded, `Init` is called again. Note that in spite of the code having been upgraded, the key persists. Let's put a stop to that, issue the following:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["delete","AnotherKey"]}'
```

Now issue the read query again:



As you can see, the value is gone.

3.3. Storing Complex Data Structures

This section will overwrite the contents of marbles.go, if you want to save it is recommended to copy it to a different directory. Do not save it to the current directory, if you do so Hyperledger Fabric will pick up the file and try to compile it which will cause issues due to duplication.

a. Marshalling Structs

Now we have a basic chaincode developed, let's expand it to use more complex objects. In Go this is done by using `structs`. You will notice that the structs have tags attached to them that denote JSON metadata, when later on we marshal these structs into byte format, the parser reads these tags and uses them to determine which keys to map the data on to.

Above `main` write the following:

```
type Marble struct {
      ObjectType string `json:"objectType"`
      Color      string `json:"color"`
      Size       int    `json:"size"`
      Owner      string `json:"owner"`
}
```

Remove the `stub.PutState("Test", []byte("Hello, World!"))` line from `Init` too.
This defines a marbles object and uses tags to denote how it should be encoded into JSON. The Go JSON library has tools for easily marshalling `struct` objects into `byte` format and vice versa.

b. Add initMarbles

Now you have the data structure in place, some dummy data is needed, add a new function above `Invoke` called `initMarbles`:

```
func (t *Chaincode) initMarbles(stub shim.ChaincodeStubInterface) pb.Response {
    marbles := []Marble{
        Marble{ObjectType: "Marble", Color: "blue", Size: 20, Owner: "Tom"},
        Marble{ObjectType: "Marble", Color: "yellow", Size: 10, Owner: "Tom"},
        Marble{ObjectType: "Marble", Color: "red", Size: 22, Owner: "Tom"},
        Marble{ObjectType: "Marble", Color: "blue", Size: 21, Owner: "Tom"},
        Marble{ObjectType: "Marble", Color: "red", Size: 13, Owner: "Tom"},
        Marble{ObjectType: "Marble", Color: "yellow", Size: 34, Owner: "Tom"},
        Marble{ObjectType: "Marble", Color: "blue", Size: 25, Owner: "Matthew"},
        Marble{ObjectType: "Marble", Color: "red", Size: 9, Owner: "Matthew "},
        Marble{ObjectType: "Marble", Color: "blue", Size: 16, Owner: "Matthew "},
        Marble{ObjectType: "Marble", Color: "blue", Size: 13, Owner: "Matthew "},
    }

    i := 0
    for i < len(marbles) {
        fmt.Println("i is ", i)
        marbleAsBytes, _ := json.Marshal(marbles[i])
        stub.PutState("MARBLE" + strconv.Itoa(i), marbleAsBytes)
```

```
        fmt.Println("Added", marbles[i])
        i = i + 1
    }

    return shim.Success(nil)
}
```

A quick aside – a quirk of Go's syntax is trailing commas on the final item of an array. This is not a syntax error but simply syntax to denote an array element.

You'll notice that this code is not placed within `Init` but instead is in an `Invoke` function which we call. As you saw earlier, the `Init` function is called each time the code is updated. This means that it should not be used to set any values that need to persist over multiple versions of the code. A better practice is to put code loading dummy data into an `Invoke` function that can be manually called. Add an entry into the `if` chain in Invoke:

```
    if function == "read" {
        return t.read(stub, args)
    } else if function == "overwrite" {
        return t.overwrite(stub, args)
    } else if function == "append" {
        return t.append(stub, args)
    } else if function == "delete" {
        return t.delete(stub, args)
    } else if function == "initMarbles" {
        return t.initMarbles(stub)
    }
```

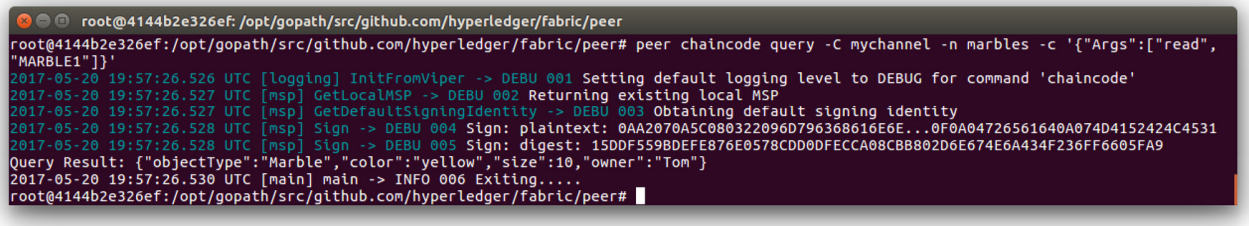
c. Test the new data structure

Read will not actually have to be modified at all to test if this works. Upgrade the code and invoke `initMarbles`:

```
peer chaincode install -n marbles -v 1.3 -p
github.com/hyperledger/fabric/examples/chaincode/go/marbles

peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n marbles -v 1.3 -c
'{"Args":["init"]}'

peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["initMarbles"]}'
```

Because the read function simply returns the `byte` array directly, we can use it to `read` one of the `Marbles`:

```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "MARBLE1"]}'
```



d. Changing ownership

Replace the append function with the following:

```
func (t *Chaincode) changeOwner(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
	marbleAsBytes, _ := stub.GetState(args[0])
	marble := Marble{}

	json.Unmarshal(marbleAsBytes, &marble)
	marble.Owner = args[1]

	marbleAsBytes, _ = json.Marshal(marble)
	stub.PutState(args[0], marbleAsBytes)

	return shim.Success(nil)
}
```

Much like append, `changeOwner` retrieves data from the World State and edits it before putting it back in. In the context of a complex data structure this requires the structure to be unmarshalled into a buffer that is the same type as the data originally was.

Once this happens the structure can be edited the same way `structs` in go are edited. Upon completion the `struct` is marshalled again and overwrites the old structure in the World State.

Edit Invoke to change the routing for append to `changeOwner`:

```
	if function == "read" {
		return t.read(stub, args)
	} else if function == "overwrite" {
		return t.overwrite(stub, args)
	} else if function == "changeOwner" {
		return t.changeOwner(stub, args)
	} else if function == "delete" {
		return t.delete(stub, args)
	} else if function == "initMarbles" {
		return t.initMarbles(stub)
	}
```

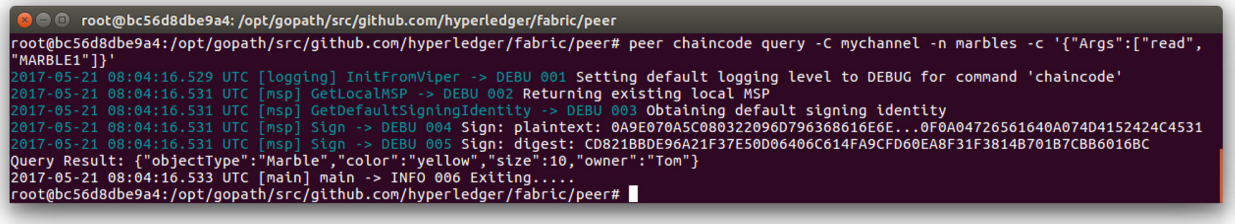
e. Test changeOwner

Install the new chaincode and upgrade what exists:

```
peer chaincode install -n marbles -v 1.4 -p
github.com/hyperledger/fabric/examples/chaincode/go/marbles

peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n marbles -v 1.4 -c
'{"Args":["init"]}'
```

Try issuing a read function on one of the `Marbles` from `initMarble`:

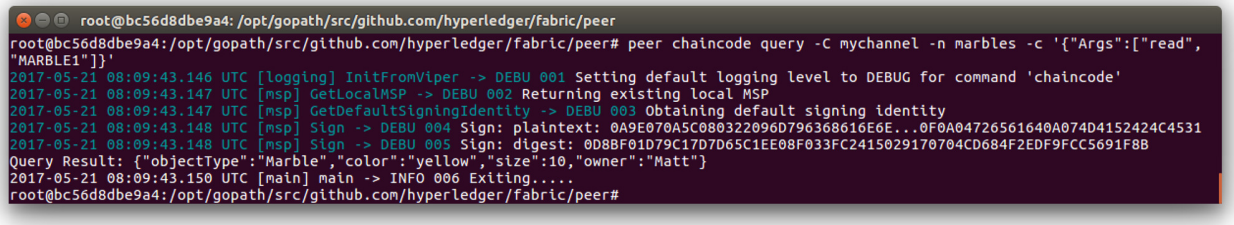
```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "MARBLE1"]}'
```



As said before, data persists upgrade to upgrade and as such we do not need to re-invoke `initMarbles` to repopulate the World State.

Issue the following to change the owner of MARBLE1:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["changeOwner", "MARBLE1", "Matthew"]}'
```



f. Add a Marble

Replace overwrite with the following:

```
func (t *Chaincode) addMarble(stub shim.ChaincodeStubInterface, args []string)
pb.Response {

      size, _ := strconv.Atoi(args[2])
```

```
	var marble = Marble{ObjectType: "Marble", Color: args[1], Size: size, Owner:
args[3]}

	marbleAsBytes, _ := json.Marshal(marble)
	stub.PutState(args[0], marbleAsBytes)

	return shim.Success(nil)
}
```

We want to be able to add `Marble` objects, as such the above function takes in the parameters for creating a `Marble`, packages them into a `struct` of type `Marble` and places this into the World State with the key we supply.

Modify the `if` chain in `Invoke` to replace `overwrite` with `addMarble`:

```
	if function == "read" {
		return t.read(stub, args)
	} else if function == "addMarble" {
		return t.addMarble(stub, args)
	} else if function == "changeOwner" {
		return t.changeOwner(stub, args)
	} else if function == "delete" {
		return t.delete(stub, args)
	} else if function == "initMarbles" {
		return t.initMarbles(stub)
	}
```

g. Test addMarble

Install and upgrade the chaincode:

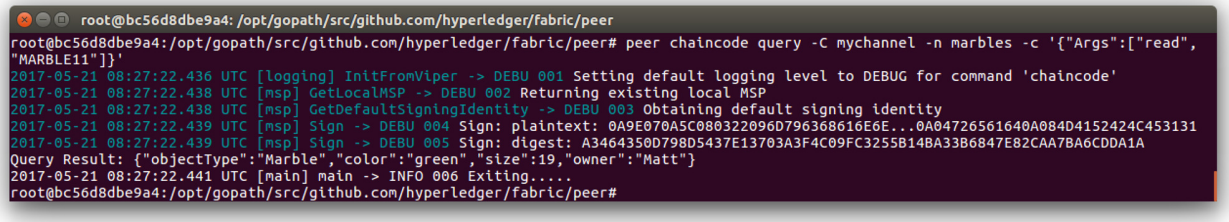
```
peer chaincode install -n marbles -v 1.5 -p
github.com/hyperledger/fabric/examples/chaincode/go/marbles

peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n marbles -v 1.5 -c
'{"Args":["init"]}'
```

Issue the following to create a `Marble`:

```
peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n marbles -c
'{"Args":["addMarble", "MARBLE11", "green", "19", "Matthew"]}'
```

Issue a read query to get the details of the new `Marble`:

```
peer chaincode query -C mychannel -n marbles -c '{"Args":["read", "MARBLE11"]}'
```


root@bc56d8dbe9a4: /opt/gopath/src/github.com/hyperledger/fabric/peer
root@bc56d8dbe9a4:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n marbles -c '{"Args":["read",
"MARBLE11"]}'
2017-05-21 08:27:22.436 UTC [logging] InitFromViper -> DEBU 001 Setting default logging level to DEBUG for command 'chaincode'
2017-05-21 08:27:22.438 UTC [msp] GetLocalMSP -> DEBU 002 Returning existing local MSP
2017-05-21 08:27:22.438 UTC [msp] GetDefaultSigningIdentity -> DEBU 003 Obtaining default signing identity
2017-05-21 08:27:22.439 UTC [msp] Sign -> DEBU 004 Sign: plaintext: 0A9E070A5C080322096D796368616E6E...0A04726561640A084D4152424C453131
2017-05-21 08:27:22.439 UTC [msp] Sign -> DEBU 005 Sign: digest: A3464350D798D5437E13703A3F4C09FC3255B14BA33B6847E82CAA7BA6CDDA1A
Query Result: {"objectType":"Marble","color":"green","size":19,"owner":"Matt"}
2017-05-21 08:27:22.441 UTC [main] main -> INFO 006 Exiting.....
root@bc56d8dbe9a4:/opt/gopath/src/github.com/hyperledger/fabric/peer#

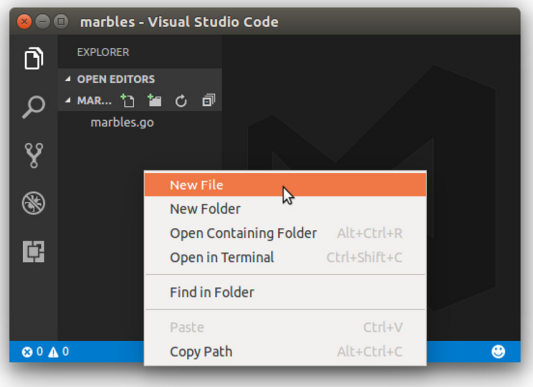
Section 4. Unit Testing (optional)

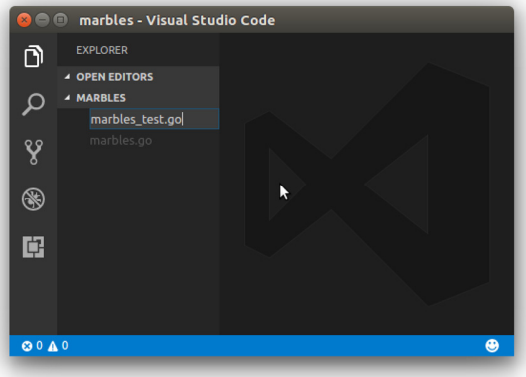
Fabric allows you to test chaincode offline with Go's inbuilt testing framework and the 'MockStub' – a spoofed World State that behaves the same as an actual LevelDB or CouchDB. For more information about Go's testing framework, please refer to the Go docs: https://golang.org/pkg/testing/

4.1. Testing Overview

a. Creating and running tests

Create a new file in the marbles folder in chaincodes called `marbles_test.go`:

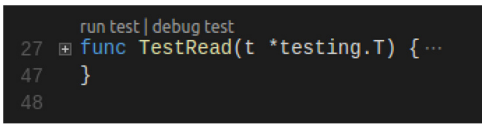




Tests Suites in Go consist at minimum of a file with the name `<file>_test.go` containing the tests for a given collection of code. To execute a test suit navigate into the directory of the test file and issue the following:

```
go test
```

In this situation, VS Code has a Go plugin that will let you run tests from within VS Code:



b. Code Structure

Test Suites in Go consist of a collection of functions of the form `func TestXxx(t *testing.T)` where `Xxx` is the name of the test. Use the following skeleton for your chaincode tests:

```
func TestXxx(t *testing.T) {
      marblesChaincode := new(Chaincode)
      stub := shim.NewMockStub("marbles", marblesChaincode)
}
```

Each test declares it's own instance of the `Chaincode` (`Chaincode` here being the name of the object in marbles.go, if it something different then change it accordingly) and World State (`MockStub`). The MockStub is an implementation of the shim that spoofs a World State for use in testing, eliminating the need to connect to a real fabric network.

Other types of test in Go exist too, but only TestXxx functions will be covered here. Please refer to the Go docs for more information.

Something which may also be useful to include in the above Mock stub is the `MockInit` function:

```
stub.MockInit("init", [][]byte{})
```

As the name suggests, this is used to call the Init function of chaincode, however given in this instance the `Init` function is empty there is no point in doing this. If `Init` is used, it is recommended practice to have a test which specifically examines if `Init` has worked. You would write this test in the same manner that the other tests covered in this section would be.

4.2. Testing read

Add the following test:

```
package main

import (
	"encoding/json"
	"fmt"
	"testing"

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

func TestRead(t *testing.T) {
	// Init
	marblesChaincode := new(Chaincode)
	stub := shim.NewMockStub("marbles", marblesChaincode)

	// Put a test marble in the world state
	testMarble := Marble{ObjectType: "Marble", Color: "green", Size: 19, Owner:
"Matthew"}
	marbleAsBytes, err := json.Marshal(testMarble)
	if err != nil {
	  fmt.Println("Marshalling the testMarble failed", err)
	  t.FailNow()
	}
	stub.MockTransactionStart("1")
	stub.PutState("MARBLE1", marbleAsBytes)
	stub.MockTransactionEnd("1")

	// Read out the marble
```

```
	res := stub.MockInvoke("marbles", [][]byte{[]byte("read"),
[]byte("MARBLE1")})
	if res.Status != shim.OK {
	 fmt.Println("read failed", string(res.Message))
	 t.FailNow()
	}
	checkRes(t, res,
`{"objectType":"Marble","color":"green","size":19,"owner":"Matthew"}`)
}
```

This creates a Marble object and places it directly into to the World State. Read is then called to fetch this value and it's return string is compared to an expected output.

Unlike when using the `peer` client application, when writing tests `MockInvoke` covers both invokes and queries, reflecting the structure of the code. The syntax for `MockInvoke` is as follows: `MockInvoke(<chaincode id declared in NewMockStub>, <function name and arguments>).`

The 'function name and arguments' is inputted as a multidimensional `byte` array. Under the covers these are the data types used to exchange information, because we do not have a peer to swap information between `byte` and `string` format we must do it ourselves.

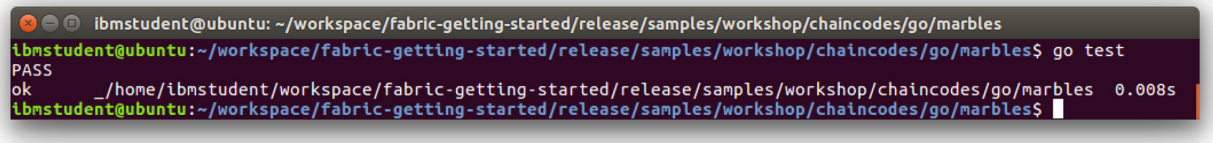
You will also notice that `PutState` is surrounded by `MockTransactionStart` and `MockTransactionEnd`. Functions like PutState are ordinarily part of transaction functions – given in this context we are calling it outside of a transaction (in a test) we have to make a dummy transaction for it to run inside.

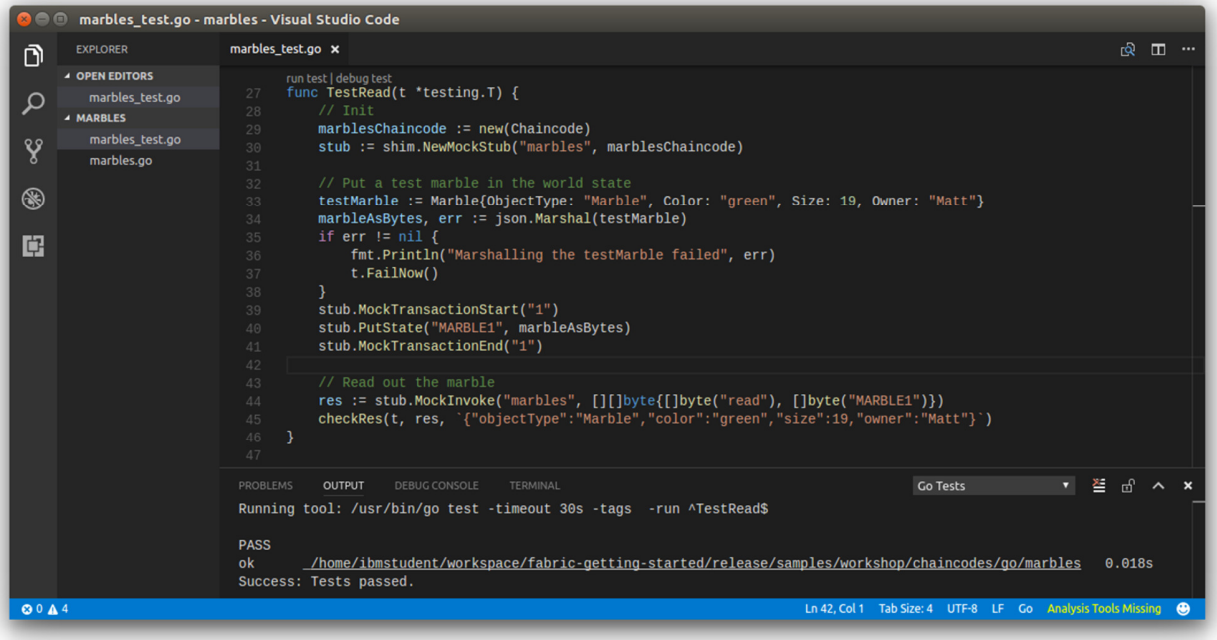
Finally, the result is inspected to ensure that the invocation was successful and no errors were thrown.

As you can see, this comparison is done by a function called `checkRes`. Add the following:

```
func checkRes(t *testing.T, res pb.Response, expected string) {
	if res.Status != shim.OK {
	 fmt.Println("read() failed", string(res.Message))
	 t.FailNow()
	}
	if res.Payload == nil {
	 fmt.Println("read() failed to get value")
	 t.FailNow()
	}
	if string(res.Payload) != expected {
	 fmt.Println("read()'s value was not as expected")
	 t.FailNow()
	}
}
```

Run the test by navigating to the `marbles` directory and issuing `go test` or by selecting `run test` above the test's function name in VS Code:





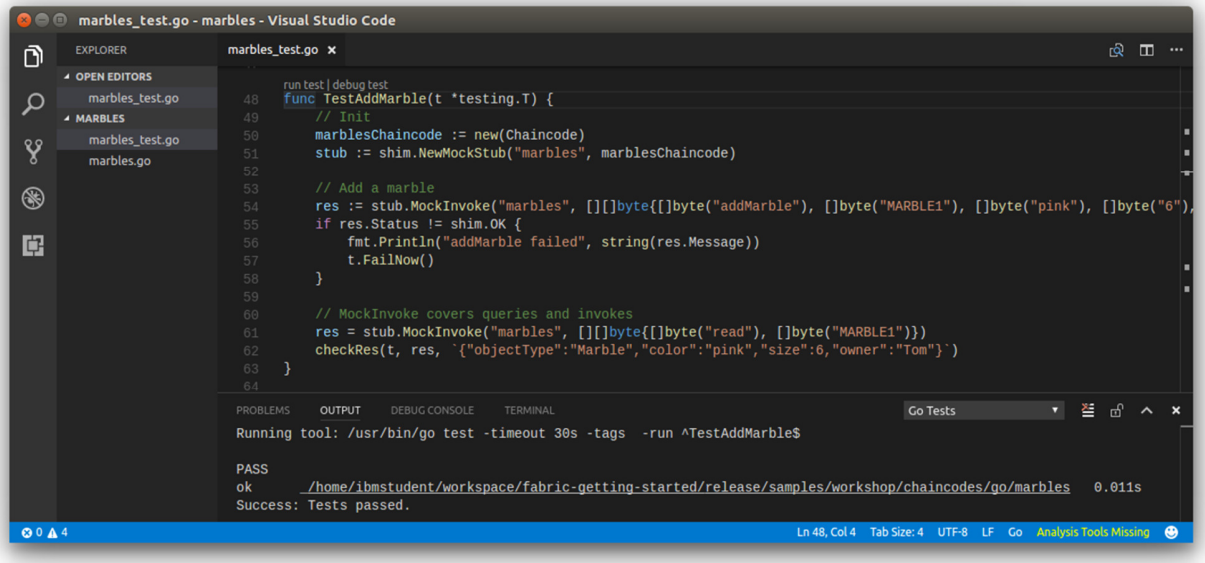
4.3. Testing add Marble

Add the following test:

```
func TestAddMarble(t *testing.T) {
	// Init
	marblesChaincode := new(Chaincode)
	stub := shim.NewMockStub("marbles", marblesChaincode)

	// Add a marble
	res := stub.MockInvoke("marbles", [][]byte{[]byte("addMarble"),
[]byte("MARBLE1"), []byte("pink"), []byte("6"), []byte("Tom")})
	if res.Status != shim.OK {
	  fmt.Println("addMarble failed", string(res.Message))
	  t.FailNow()
	}

	// MockInvoke covers queries and invokes
	res = stub.MockInvoke("marbles", [][]byte{[]byte("read"), []byte("MARBLE1")})
	checkRes(t, res,
`{"objectType":"Marble","color":"pink","size":6,"owner":"Tom"}`)
```

```
}
```

First, `MockInvoke` is called supplying the `addMarble` function as an argument. The response from this is checked to ensure there was not an error. Because we know from the previous test that we can check values from reading them we can use `checkRes` to do this. Using `GetState` and unmarshalling the code is also a perfectly valid way of doing this.
Check that the function works using the same methods as in 4.2:



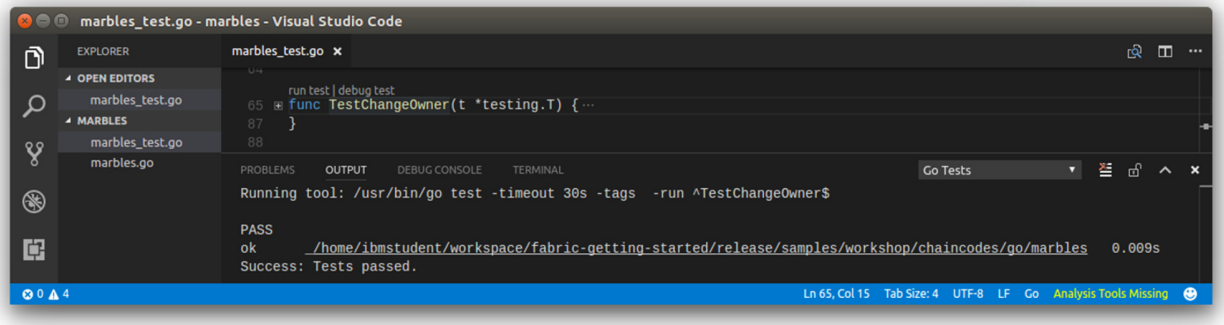
4.4. Testing changeOwner

Add the following test:

```
func TestChangeOwner(t *testing.T) {
	// Init
	marblesChaincode := new(Chaincode)
	stub := shim.NewMockStub("marbles", marblesChaincode)

	// Add a marble
	res := stub.MockInvoke("marbles", [][]byte{[]byte("addMarble"),
[]byte("MARBLE1"), []byte("pink"), []byte("6"), []byte("Tom")})
	if res.Status != shim.OK {
	  fmt.Println("addMarble failed", string(res.Message))
	  t.FailNow()
	}

	// Change it's owner
	res = stub.MockInvoke("marbles", [][]byte{[]byte("changeOwner"),
[]byte("MARBLE1"), []byte("Matthew")})
	if res.Status != shim.OK {
	  fmt.Println("changeOwner failed", string(res.Message))
	  t.FailNow()
	}
```

```
	// MockInvoke covers queries and invokes
	res = stub.MockInvoke("marbles", [][]byte{[]byte("read"), []byte("MARBLE1")})
	checkRes(t, res,
`{"objectType":"Marble","color":"pink","size":6,"owner":"Matthew"}`)
}
```

Testing `changeOwner` builds on what we learned in the previous tests. Multiple `MockInvokes` are used to add a new `Marble` and then change it's owner. Once again we will be checking the end state is correct by having the `MockStub` issue a `read` query to output the value.



4.5. Testing delete

Add the final test:

```
func TestDelete(t *testing.T) {
	// Init
	marblesChaincode := new(Chaincode)
	stub := shim.NewMockStub("marbles", marblesChaincode)

	// Add a marble
	res := stub.MockInvoke("marbles", [][]byte{[]byte("addMarble"),
[]byte("MARBLE1"), []byte("pink"), []byte("6"), []byte("Tom")})
	if res.Status != shim.OK {
	  fmt.Println("addMarble failed", string(res.Message))
	  t.FailNow()
	}

	res = stub.MockInvoke("marbles", [][]byte{[]byte("delete"),
[]byte("MARBLE1")})
	if res.Status != shim.OK {
	  fmt.Println("delete failed", string(res.Message))
	  t.FailNow()
	}

	// MockInvoke covers queries and invokes
	res = stub.MockInvoke("marbles", [][]byte{[]byte("read"), []byte("MARBLE1")})
	if res.Status != shim.OK {
	  fmt.Println("read() failed", string(res.Message))
	  t.FailNow()
```

```
        }
        if res.Payload != nil {
          fmt.Println("read() returned a value", res.Payload)
          t.FailNow()
        }
}
```

Again, this test builds on what we have seen in previous tests. The main distinction is that we are not using `checkRes` to validate this test as we *want* it to return nothing. As such we check the result was OK and then that it's return value was `nil`.

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