

Hyperledger Fabric Masterclass for Developers – Student Lab Guide

BLOCKCHAIN TRAINING ALLIANCE
BY KEN MYATT AND KRIS BENNETT

Table of Contents

Lab 1 - Creating the Chaincode Structure	3
Creating the Chaincode.....	3
Adding the Basics	5
Defining the Assets	8
Adding the Main Methods	10
Creating the <i>createCoin</i> Invoke Method.....	16
Creating the <i>transferCoin</i> Invoke Method	21
Creating the <i>revokeCoin</i> Invoke Method	26
Creating the <i>initLedger</i> Method.....	29
Lab 2 - Adding Rich Queries	31
Getting Started.....	31
Creating the <i>returnCoinBank</i> Method	35
Adding the <i>findByAttributes</i> and <i>showCoinsAbovePercentage</i> Methods.....	41
Lab 3 - Packaging and Deploying the Chaincode	42
Getting Started.....	42
Adding the <i>btacoincc</i> Container Definition.....	44
Restarting the Network.....	46
Install and Instantiate the Chaincode	48
Testing the Chaincode via the Command Line	51
Viewing World State Data in <i>Fauxton</i>	53
Lab 4 - Intro to Complex Queries using Composite Keys	56
Getting Started.....	56
Building the <i>Invoke</i> Function.....	59
Initializing the Chaincode.....	60
Querying by Composite Key.....	63
Building the New Chaincode	65
Installing the New Chaincode	66
Testing the New Chaincode	69
Lab 5 - Creating the Connection Profile	70
Getting Started.....	70
The <i>Client</i> Section	71
The <i>Channels</i> Section	72

The *Organizations* Section 73

The *Orderers* Section..... 74

The *Peers* Section..... 75

The *CertificateAuthorities* Section 76

Lab 1 - Creating the Chaincode Structure

In this lab you will create the main chaincode used throughout this course, *btacoin*.

Creating the Chaincode

Open a new terminal window and navigate to the *Desktop/fabric/cc* folder using the command below.

```
cd ~/Desktop/fabric/cc
```

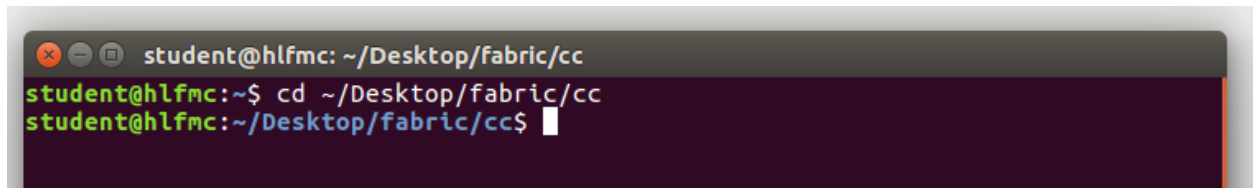
A terminal window with a dark background. The title bar shows a window icon, a close button, and the text 'student@hlfmc: ~/Desktop/fabric/cc'. The terminal shows the command 'cd ~/Desktop/fabric/cc' being entered and executed, resulting in a new prompt 'student@hlfmc:~/Desktop/fabric/cc\$'.

Figure 1 - Snippet 1.1

Using the commands below, create a new sub-folder within *cc* called *src*. Within *src* a subfolder called *btacoin* will be created.

```
mkdir src && cd src
```

```
mkdir btacoin && cd btacoin
```

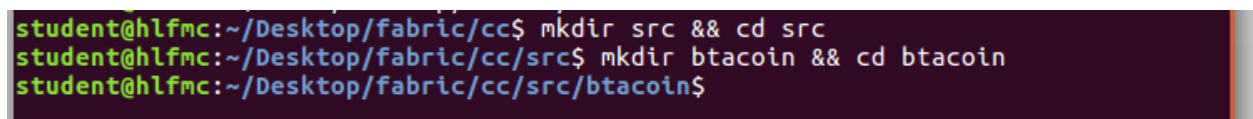
A terminal window with a dark background. The title bar shows a window icon, a close button, and the text 'student@hlfmc: ~/Desktop/fabric/cc/src/btacoin'. The terminal shows three lines of commands: 'mkdir src && cd src', 'mkdir btacoin && cd btacoin', and the final prompt 'student@hlfmc:~/Desktop/fabric/cc/src/btacoin\$'.

Figure 2 - Snippet 1.2

Use the *touch* command below to create a new empty file called *btacoin.go*.

```
touch btacoin.go
```

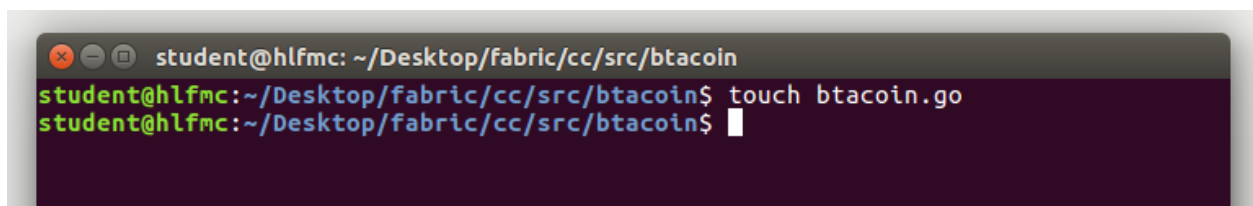
A terminal window with a dark background. The title bar shows a window icon, a close button, and the text 'student@hlfmc: ~/Desktop/fabric/cc/src/btacoin'. The terminal shows the command 'touch btacoin.go' being entered and executed, resulting in a new prompt 'student@hlfmc:~/Desktop/fabric/cc/src/btacoin\$'.

Figure 3 - Snippet 1.3

Open the newly created *btacoin.go* file in Visual Studio Code. Use the *files* browser to navigate to *Desktop/fabric/cc/src/btacoin*. Then right-click on the *btacoin.go* file and select *Visual Studio Code* from the *Open With* option on the dialogue box.

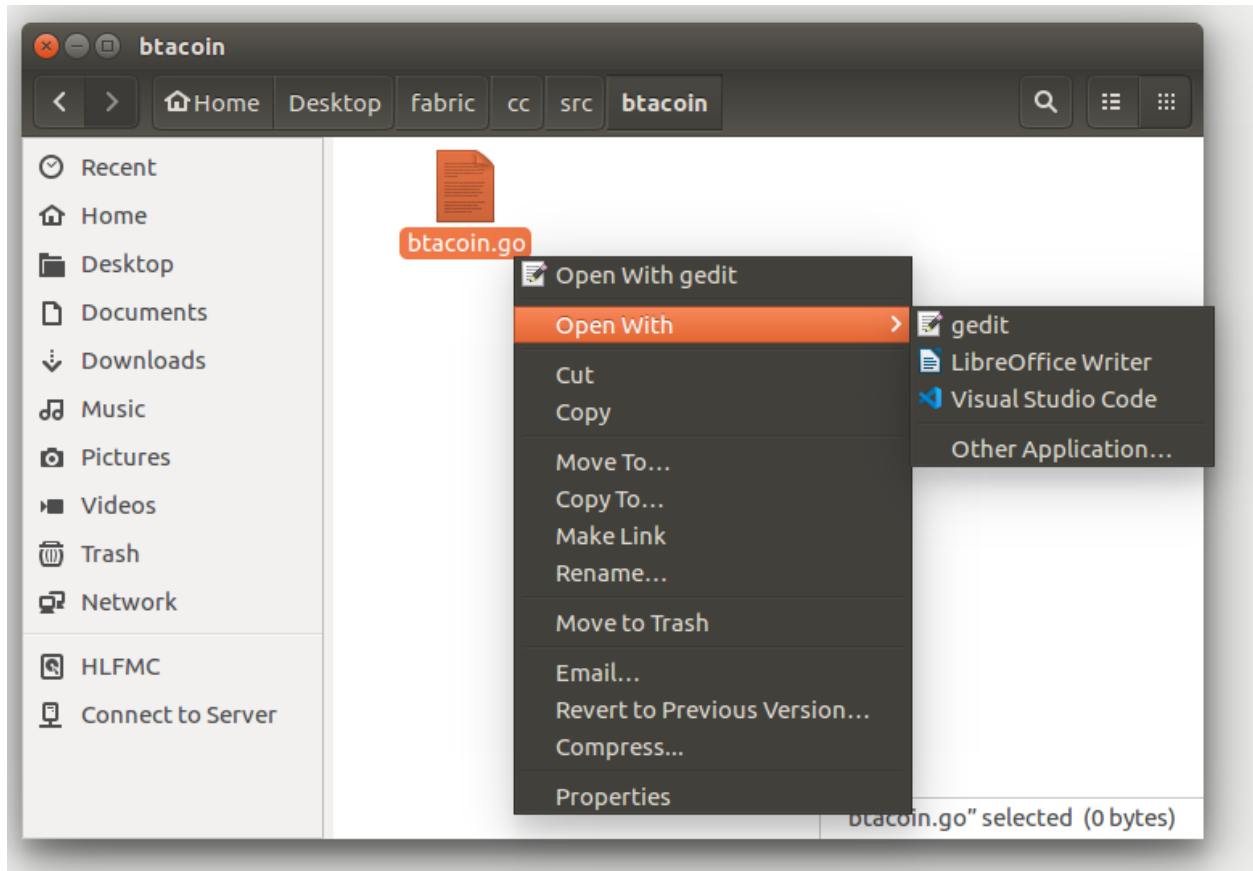


Figure 4 - Launching Visual Studio Code

Adding the Basics

Begin by adding the following code to add the traditional *main* package as well as importing some Golang default utility packages.

```
package main

import (
    "encoding/json"
    "fmt"
    "strconv"
    "bytes"
)
```

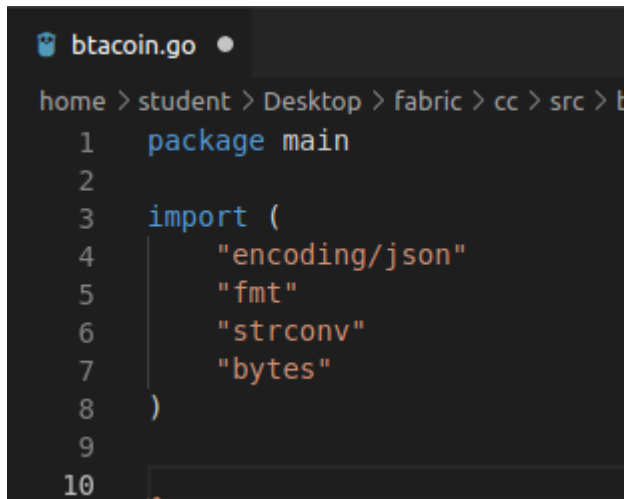


Figure 5 - Snippet 1.4

Packages for peer communication and the fabric-shim package will need to be imported. Add the following to the *import* statement.

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

A screenshot of a code editor with a dark background. The editor shows a Go file with the following code:

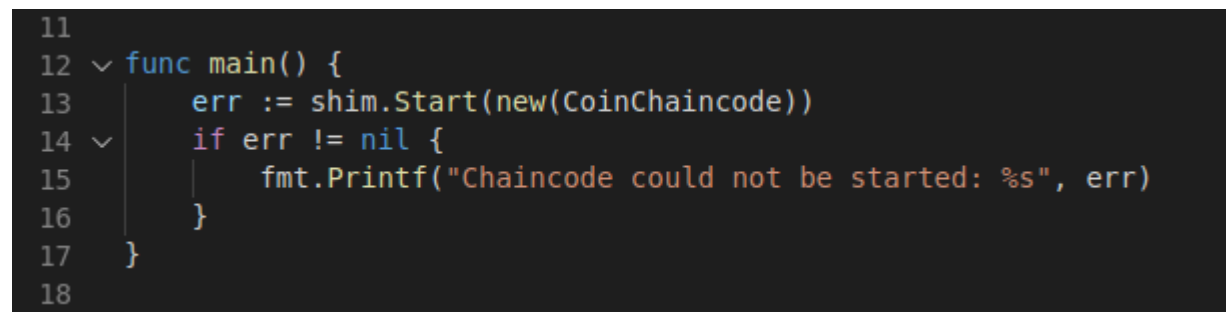
```
home > student > Desktop > fabric > cc > src > btacoin > 🐛 btacoin.go > ...  
1  package main  
2  
3  import (  
4      "encoding/json"  
5      "fmt"  
6      "strconv"  
7      "bytes"  
8      "github.com/hyperledger/fabric/core/chaincode/shim"  
9      peer "github.com/hyperledger/fabric/protos/peer"  
10 )  
11
```

The code is color-coded: `package` is blue, `main` is blue, `import` is blue, and the strings are orange. Line numbers 1 through 11 are on the left. The path at the top is `home > student > Desktop > fabric > cc > src > btacoin > 🐛 btacoin.go > ...`.

Figure 6 - Snippet 1.5

Now the *main* function will be created. This function will initiate the chaincode and start the Fabric Shim utility. Note that when initializing the Fabric Shim, a chaincode struct must be passed in as a parameter. This struct will be created in a later step - for now pass in a struct named *CoinChaincode*. This struct will possess the *SimpleChaincodeInterface*.

```
func main() {  
    err := shim.Start(new(CoinChaincode))  
    if err != nil {  
        fmt.Printf("Chaincode could not be started: %s", err)  
    }  
}
```



```
11  
12 ✓ func main() {  
13     err := shim.Start(new(CoinChaincode))  
14 ✓     if err != nil {  
15         fmt.Printf("Chaincode could not be started: %s", err)  
16     }  
17 }  
18
```

Figure 7 - Snippet 1.6

Defining the Assets

Start by defining the `SimpleChaincodeInterface` and `Chaincode` structs using the code below. These structs will be passed into all functions as well as initialized at the beginning of runtime. Place the code after the *import* statement but before the *main* function.

```
type CoinChaincode struct {  
  
}  
  
type btaCoin struct {  
    CoinID int `json:"coinID"`  
    FullName string `json:"fullName"`  
    CourseName string `json:"courseName"`  
    Score int `json:"score"`  
}
```

```
btacoin.go •
home > student > Desktop > fabric > cc > src > btacoin > btacoin.go > ...
1  package main
2
3  import (
4      "encoding/json"
5      "fmt"
6      "strconv"
7      "bytes"
8      "github.com/hyperledger/fabric/core/chaincode/shim"
9      peer "github.com/hyperledger/fabric/protos/peer"
10 )
11
12 type CoinChaincode struct {
13 }
14
15 type btaCoin struct {
16     CoinID int `json:"coinID"`
17     FullName string `json:"fullName"`
18     CourseName string `json:"courseName"`
19     Score int `json:"score"`
20 }
21
22 func main() {
23     err := shim.Start(new(CoinChaincode))
24     if err != nil {
25         fmt.Printf("Chaincode could not be started: %s", err)
26     }
27 }
28
29
```

Figure 8 - Snippet 1.7

Adding the Main Methods

Begin by defining the *init* method. Remember that for all methods a pointer to the *CoinChaincode* struct must be created and the chaincode stub (from the Fabric shim package) must be passed in as well.

```
func (Contract *CoinChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response {
}
}
```

```
21
22 func (Contract *CoinChaincode) Init(stub shim.ChaincodeStubInterface) peer.Response {
23 }
24
```

Figure 9 - Snippet 1.8

Use the code below to create the *Invoke* method.

```
func (Contract *CoinChaincode) Invoke(stub
shim.ChaincodeStubInterface) peer.Response {
}
}
```

```
24
25 func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
26 }
27
```

Figure 10 - Snippet 1.9

At this point logic will be added to the *Invoke* method so that it can determine how to route incoming requests. In order to facilitate this any input parameters passed into *Invoke* will need to be obtained. The *GetFunctionAndParameters* function will return the function name requested as well as any input parameters. Add the code below to the *Invoke* method.

```
fnType, args := stub.GetFunctionAndParameters()
```

```
25 func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
26     fnType, args := stub.GetFunctionAndParameters()
27 }
```

Figure 11 - Snippet 1.10

If no arguments were passed into the *Invoke* method an error should be raised. Use the code below to check if any arguments were received and raise an error if not.

```
        if len(args) == 0 {  
            return shim.Error("Please rerun w/ an approved method...  
Methods are: returnCoinBank \n,listCoinsByCourse \n,findByName  
\n,createCoin \n, transferCoin \n, or startLedger \n")  
        }  
  
24  
25 func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {  
26     fnType, args := stub.GetFunctionAndParameters()  
27  
28     if len(args) == 0 {  
29         return shim.Error("Please rerun w/ an approved method... Methods are: returnCoinBank \n,listCoinsByCourse \n,findB  
30     }  
31  
32 }  
33
```

Figure 12 - Snippet 1.11

Now code will be added to handle request routing. The *fnType* parameter will be validated against the list of methods to choose the correct one.

```
if fnType == "initLedger" {
    Contract.initLedger(stub)
} else if fnType == "transferCoin" {
    Contract.transferCoin(stub, args)
} else if fnType == "createCoin" {
    Contract.createCoin(stub, args)
} else if fnType == "revokeCoin" {
    Contract.revokeCoin(stub, args)
} else if fnType == "findByID" {
    Contract.findByID(stub, args)
} else if fnType == "returnCoinBank" {
    Contract.returnCoinBank(stub)
} else if fnType == "findByAttributes" {
    Contract.findByAttributes(stub, args)
} else if fnType == "showCoinsAbovePercentage" {
    Contract.showCoinsAbovePercentage(stub, args)
}
```

```
31
32 ✓   if fnType == "initLedger" {
33       |       Contract.initLedger(stub)
34 ✓   } else if fnType == "transferCoin" {
35       |       Contract.transferCoin(stub, args)
36 ✓   } else if fnType == "createCoin" {
37       |       Contract.createCoin(stub, args)
38 ✓   } else if fnType == "revokeCoin" {
39       |       Contract.revokeCoin(stub, args)
40 ✓   } else if fnType == "findByID" {
41       |       Contract.findByID(stub, args)
42 ✓   } else if fnType == "returnCoinBank" {
43       |       Contract.returnCoinBank(stub)
44 ✓   } else if fnType == "findByAttributes" {
45       |       Contract.findByAttributes(stub, args)
46 ✓   } else if fnType == "showCoinsAbovePercentage" {
47       |       Contract.showCoinsAbovePercentage(stub, args)
48       |   }
49
```

Figure 13 - Snippet 1.12

If the function request gets routed successfully a *Success* status should be returned to the caller. Add the code below after the *if/else* if statement added in the previous step.

```
return shim.Success(nil)
```

```
24
25 func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
26     fnType, args := stub.GetFunctionAndParameters()
27
28     if len(args) == 0 {
29         return shim.Error("Please rerun w/ an approved method... Methods are: returnCoinBank")
30     }
31
32     if fnType == "initLedger" {
33         Contract.initLedger(stub)
34     } else if fnType == "transferCoin" {
35         Contract.transferCoin(stub, args)
36     } else if fnType == "createCoin" {
37         Contract.createCoin(stub, args)
38     } else if fnType == "revokeCoin" {
39         Contract.revokeCoin(stub, args)
40     } else if fnType == "findByID" {
41         Contract.findByID(stub, args)
42     } else if fnType == "returnCoinBank" {
43         Contract.returnCoinBank(stub)
44     } else if fnType == "findByAttributes" {
45         Contract.findByAttributes(stub, args)
46     } else if fnType == "showCoinsAbovePercentage" {
47         Contract.showCoinsAbovePercentage(stub, args)
48     }
49
50     return shim.Success(nil)
51 }
52
53
```

Figure 14 - Snippet 1.13

An error handler will be added at the bottom of the *if/else* statement. This error handler will execute if the *fnType* variable does not match any of the known function names. Add the code below to the end of the *if/else* statement.

```
else {  
    return shim.Error("Please rerun w/ an approved method...  
Methods are: returnCoinBank \n,listCoinsByCourse \n,findByName  
\n,createCoin \n, transferCoin \n, or startLedger \n")  
}  
  
24  
25 func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {  
26     fnType, args := stub.GetFunctionAndParameters()  
27  
28     if len(args) == 0 {  
29         return shim.Error("Please rerun w/ an approved method... Methods are: returnCoinBank \n,li  
30     }  
31  
32     if fnType == "initLedger" {  
33         Contract.initLedger(stub)  
34     } else if fnType == "transferCoin" {  
35         Contract.transferCoin(stub, args)  
36     } else if fnType == "createCoin" {  
37         Contract.createCoin(stub, args)  
38     } else if fnType == "revokeCoin" {  
39         Contract.revokeCoin(stub, args)  
40     } else if fnType == "findByID" {  
41         Contract.findByID(stub, args)  
42     } else if fnType == "returnCoinBank" {  
43         Contract.returnCoinBank(stub)  
44     } else if fnType == "findByAttributes" {  
45         Contract.findByAttributes(stub, args)  
46     } else if fnType == "showCoinsAbovePercentage" {  
47         Contract.showCoinsAbovePercentage(stub, args)  
48     } else {  
49         return shim.Error("Please rerun w/ an approved method... Methods are: returnCoinBank \n,li  
50     }  
51  
52     return shim.Success(nil)  
53  
54 }  
55
```

Figure 15 - Snippet 1.14

Creating the *createCoin* Invoke Method

Create the first of the known *Invoke* methods, *createCoin* using the code below.

```
func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response  
  
{  
  
}
```

```
62  
63 func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response  
64 {  
65 }  
66
```

Figure 16 - Snippet 1.15

In the new *createCoin* method input parameters will be parsed and stored using the *GetStringArgs* method. This method returns a string array, making it simpler to interact with supplied arguments. For debug purposes the input parameters will be logged to the console output.

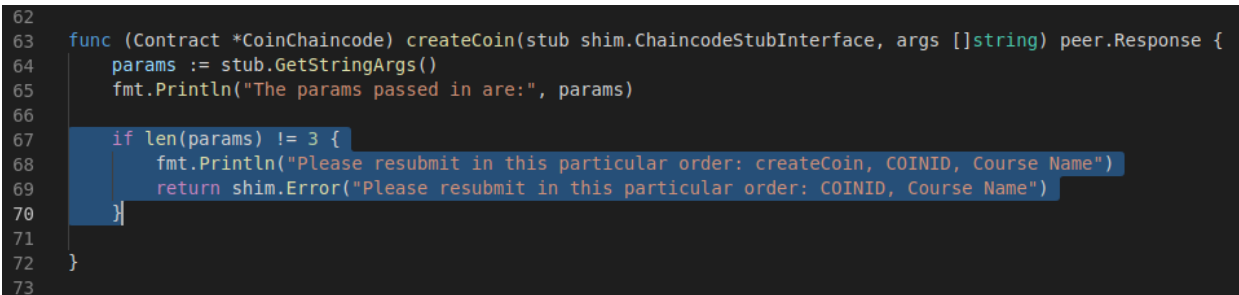
```
params := stub.GetStringArgs()  
  
fmt.Println("The params passed in are:", params)
```

```
62  
63 v func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
64     params := stub.GetStringArgs()  
65     fmt.Println("The params passed in are:", params)  
66 }  
67
```

Figure 17 - Snippet 1.16

Use the code below to ensure three input parameters are passed in (including the function name). If any other number of parameters is supplied an error will be raised.

```
        if len(params) != 3 {  
            fmt.Println("Please resubmit in this particular order:  
createCoin, COINID, Course Name")  
            return shim.Error("Please resubmit in this particular  
order: COINID, Course Name")  
        }  
    }
```

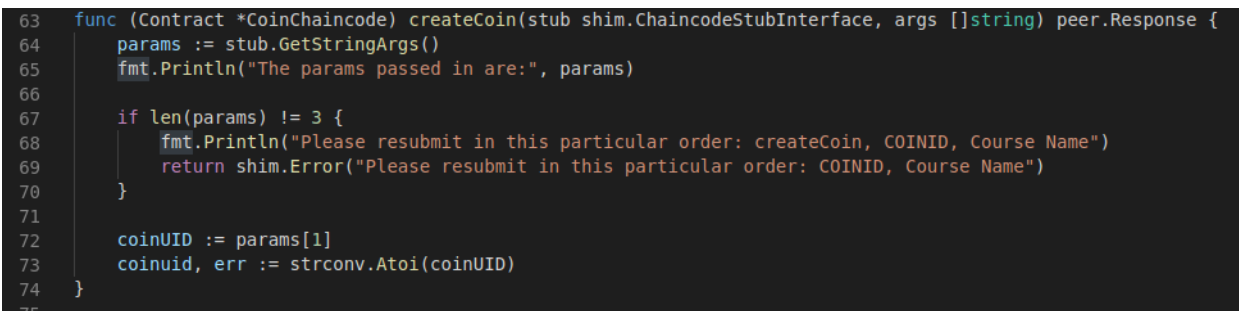


```
62 func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
63     params := stub.GetStringArgs()  
64     fmt.Println("The params passed in are:", params)  
65     if len(params) != 3 {  
66         fmt.Println("Please resubmit in this particular order: createCoin, COINID, Course Name")  
67         return shim.Error("Please resubmit in this particular order: COINID, Course Name")  
68     }  
69 }  
70  
71  
72  
73
```

Figure 18 - Snippet 1.17

Use the code below to assign the second parameter to a variable, allowing documents (or records) to be searched by ID later. Note that the value is converted to an integer data type in the second line of code.

```
coinUID := params[1]  
  
coinuid, err := strconv.Atoi(coinUID).
```



```
63 func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
64     params := stub.GetStringArgs()  
65     fmt.Println("The params passed in are:", params)  
66     if len(params) != 3 {  
67         fmt.Println("Please resubmit in this particular order: createCoin, COINID, Course Name")  
68         return shim.Error("Please resubmit in this particular order: COINID, Course Name")  
69     }  
70     coinUID := params[1]  
71     coinuid, err := strconv.Atoi(coinUID)  
72 }  
73  
74  
75
```

Figure 19 - Snippet 1.18

The code below will check to see if the String to Integer type conversion failed. If so an error will be raised.

```
        if err != nil {
            return shim.Error("Something went wrong!" + err.Error())
        }

63 func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {
64     params := stub.GetStringArgs()
65     fmt.Println("The params passed in are:", params)
66
67     if len(params) != 3 {
68         fmt.Println("Please resubmit in this particular order: createCoin, COINID, Course Name")
69         return shim.Error("Please resubmit in this particular order: COINID, Course Name")
70     }
71
72     coinUID := params[1]
73     coinuid, err := strconv.Atoi(coinUID)
74
75     if err != nil {
76         return shim.Error("Something went wrong!" + err.Error())
77     }
78
79 }
80
```

Figure 20 - Snippet 1.19

Next a new struct will be created. This struct will be based off of the *btacoin* struct created earlier. Note that the CoinID value will come from the client itself and will NOT be assigned within chaincode.

```
        var coinDetails = btaCoin{CoinID: coinuid, FullName:
"Unassigned", CourseName: params[2], Score: 0}

62
63 func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {
64     params := stub.GetStringArgs()
65     fmt.Println("The params passed in are:", params)
66
67     if len(params) != 3 {
68         fmt.Println("Please resubmit in this particular order: createCoin, COINID, Course Name")
69         return shim.Error("Please resubmit in this particular order: COINID, Course Name")
70     }
71
72     coinUID := params[1]
73     coinuid, err := strconv.Atoi(coinUID)
74
75     if err != nil {
76         return shim.Error("Something went wrong!" + err.Error())
77     }
78
79     var coinDetails = btaCoin{CoinID: coinuid, FullName: "Unassigned", CourseName: params[2], Score: 0}
80
81 }
```

Figure 21 - Snippet 1.20

Before the coin is committed to the *coinRepository* a check must be performed to ensure the ID assigned to it is not already in use. Use the code below to perform this check and raise an error if the ID is already in use.

```
    oldKeyVal, err := stub.GetState(coinUID)

    if oldKeyVal != nil {

        fmt.Println("There's a coin with the same ID here. Use a
different ID. Error Code: " + err.Error())

    }
```

```
63 ✓ func (Contract *CoinChaincode) createCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {
64     params := stub.GetStringArgs()
65     fmt.Println("The params passed in are:", params)
66
67 ✓     if len(params) != 3 {
68         fmt.Println("Please resubmit in this particular order: createCoin, COINID, Course Name")
69         return shim.Error("Please resubmit in this particular order: COINID, Course Name")
70     }
71
72     coinUID := params[1]
73     coinuid, err := strconv.Atoi(coinUID)
74
75 ✓     if err != nil {
76         return shim.Error("Something went wrong!" + err.Error())
77     }
78
79     var coinDetails = btaCoin{CoinID: coinuid, FullName: "Unassigned", CourseName: params[2], Score: 0}
80
81     oldKeyVal, err := stub.GetState(coinUID)
82 ✓     if oldKeyVal != nil {
83         fmt.Println("There's a coin with the same ID here. Use a different ID. Error Code: " + err.Error())
84     }
85
86 }
```

Figure 22 - Snippet 1.21

Use the code below to serialize the coin info and commit the state back to the registry.

```
newCoin, err := json.Marshal(coinDetails)

if err != nil {

    shim.Error("Something went wrong in serializing. Error: " +
err.Error())

} else {

    err = stub.PutState(args[0], newCoin)

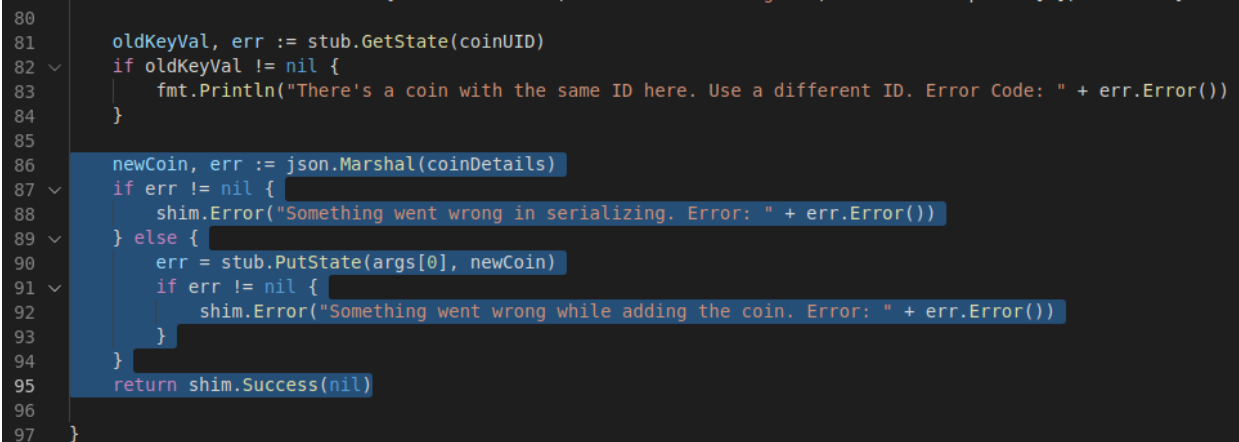
    if err != nil {

        shim.Error("Something went wrong while adding the
coin. Error: " + err.Error())

    }

}

return shim.Success(nil)
```



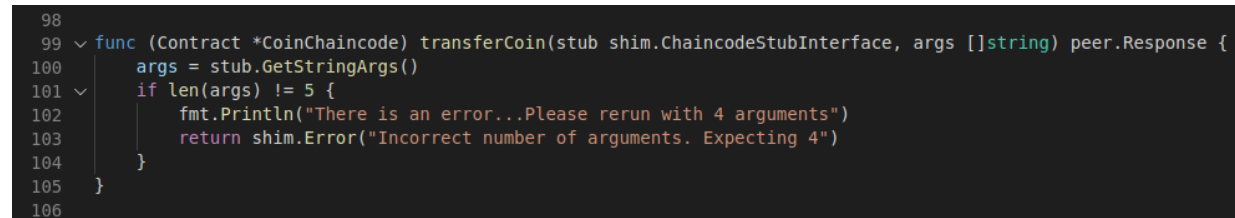
```
80 oldKeyVal, err := stub.GetState(coinUID)
81 if oldKeyVal != nil {
82     fmt.Println("There's a coin with the same ID here. Use a different ID. Error Code: " + err.Error())
83 }
84
85 newCoin, err := json.Marshal(coinDetails)
86 if err != nil {
87     shim.Error("Something went wrong in serializing. Error: " + err.Error())
88 } else {
89     err = stub.PutState(args[0], newCoin)
90     if err != nil {
91         shim.Error("Something went wrong while adding the coin. Error: " + err.Error())
92     }
93 }
94 return shim.Success(nil)
95
96
97 }
```

Figure 23 - Snippet 1.22

Creating the *transferCoin* Invoke Method

The Transfer Coin function will allow a coin to be transferred between the current owner (BTA) and a new owner (the student). In the code below, input arguments will be parsed and stored in a variable *args*. The code will also confirm the correct number of arguments have been supplied and raise an error if not.

```
func (Contract *CoinChaincode) transferCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
    args = stub.GetStringArgs()  
    if len(args) != 5 {  
        fmt.Println("There is an error...Please rerun with 4 arguments")  
        return shim.Error("Incorrect number of arguments. Expecting 4")  
    }  
}
```



```
98  
99 ~ func (Contract *CoinChaincode) transferCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
100     args = stub.GetStringArgs()  
101     if len(args) != 5 {  
102         fmt.Println("There is an error...Please rerun with 4 arguments")  
103         return shim.Error("Incorrect number of arguments. Expecting 4")  
104     }  
105 }  
106
```

Figure 24 - Snippet 1.23

Use the code below to assign input parameter values to variables.

```
coinUID := string(args[1])  
cName := string(args[2])  
pName := string(args[3])  
score := string(args[4])
```

```
98  
99 func (Contract *CoinChaincode) transferCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
100     args = stub.GetStringArgs()  
101     if len(args) != 5 {  
102         fmt.Println("There is an error...Please rerun with 4 arguments")  
103         return shim.Error("Incorrect number of arguments. Expecting 4")  
104     }  
105  
106     coinUID := string(args[1])  
107     cName := string(args[2])  
108     pName := string(args[3])  
109     score := string(args[4])  
110  
111 }  
112
```

Figure 25 - Snippet 1.24

Next, the World State database will be queried to ensure that a coin exists with the ID in question. The query response will be logged to the console output to make sure data is being returned. Additionally, an error handler will be created to raise any errors returned during the query.

```
    foundCoin, err := stub.GetState(coinUID)

    fmt.Println("Here's the response: ", foundCoin)

    if err != nil {

        shim.Error("Something went wrong! Error Status:" +
err.Error())

    }
```

```
98
99  ✓ func (Contract *CoinChaincode) transferCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {
100      args = stub.GetStringArgs()
101  ✓  if len(args) != 5 {
102      |     fmt.Println("There is an error...Please rerun with 4 arguments")
103      |     return shim.Error("Incorrect number of arguments. Expecting 4")
104      | }
105
106      coinUID := string(args[1])
107      cName := string(args[2])
108      pName := string(args[3])
109      score := string(args[4])
110
111      foundCoin, err := stub.GetState(coinUID)
112      fmt.Println("Here's the response: ", foundCoin)
113  ✓  if err != nil {
114      |     shim.Error("Something went wrong! Error Status:" + err.Error())
115      | }
116
117  }
118
```

Figure 26 - Snippet 1.25

Once the queried coin has been returned, it can be deserialized and put into the coin struct.

```
changedCoin := btaCoin{}  
  
json.Unmarshal(foundCoin, changedCoin)  
  
intCoinUID, err := strconv.Atoi(coinUID)
```

```
98  
99 func (Contract *CoinChaincode) transferCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
100     args = stub.GetStringArgs()  
101     if len(args) != 5 {  
102         fmt.Println("There is an error...Please rerun with 4 arguments")  
103         return shim.Error("Incorrect number of arguments. Expecting 4")  
104     }  
105  
106     coinUID := string(args[1])  
107     cName := string(args[2])  
108     pName := string(args[3])  
109     score := string(args[4])  
110  
111     foundCoin, err := stub.GetState(coinUID)  
112     fmt.Println("Here's the response: ", foundCoin)  
113     if err != nil {  
114         shim.Error("Something went wrong! Error Status:" + err.Error())  
115     }  
116  
117     changedCoin := btacoin{}  
118     json.Unmarshal(foundCoin, changedCoin)  
119     intCoinUID, err := strconv.Atoi(coinUID)  
120  
121 }  
122
```

Figure 27 - Snippet 1.26

Finally, variables with the input parameter values will be used to update the coin asset. Then the coin will be serialized and committed back to the registry. To complete the process, a *shim.Success* will be returned to the caller.

```
changedCoin.CoinID = intCoinUID

changedCoin.FullName = pName

changedCoin.CourseName = cName

changedCoin.Score, err = strconv.Atoi(score)

putChangedCoin, err := json.Marshal(changedCoin)

stub.PutState(coinUID, putChangedCoin)

return shim.Success(nil)
```

```
98
99 func (Contract *CoinChaincode) transferCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {
100     args = stub.GetStringArgs()
101     if len(args) != 5 {
102         fmt.Println("There is an error...Please rerun with 4 arguments")
103         return shim.Error("Incorrect number of arguments. Expecting 4")
104     }
105
106     coinUID := string(args[1])
107     cName := string(args[2])
108     pName := string(args[3])
109     score := string(args[4])
110
111     foundCoin, err := stub.GetState(coinUID)
112     fmt.Println("Here's the response: ", foundCoin)
113     if err != nil {
114         shim.Error("Something went wrong! Error Status:" + err.Error())
115     }
116
117     changedCoin := btacoin{}
118     json.Unmarshal(foundCoin, &changedCoin)
119     intCoinUID, err := strconv.Atoi(coinUID)
120
121     changedCoin.CoinID = intCoinUID
122     changedCoin.FullName = pName
123     changedCoin.CourseName = cName
124     changedCoin.Score, err = strconv.Atoi(score)
125
126     putChangedCoin, err := json.Marshal(changedCoin)
127     stub.PutState(coinUID, putChangedCoin)
128
129     return shim.Success(nil)
130 }
131
132
```

Figure 28 - Snippet 1.27

Creating the *revokeCoin* Invoke Method

Coins may need to be occasionally revoked. For example, if a student was caught cheating on an exam his or her coin would be revoked. The *revokeCoin* function will provide this functionality. Use the code below to create the outline of the *revokeCoin* function.

```
func (Contract *CoinChaincode) revokeCoin(stub  
shim.ChaincodeStubInterface, args []string) peer.Response {  
  
}
```

```
132  
133 func (Contract *CoinChaincode) revokeCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
134 }
```

Figure 29 - Snippet 1.28

The code below will perform several functions. First, the number of arguments will be checked and an error will be raised if the incorrect number of arguments has been passed. Then the ID and Course Name will be assigned to variables (*coinToRemove* and *cName*). Next, the code will check to make sure the coin exists and raise an error if not. Next, the key returned from the *GetState* call will be unmarshalled and an instance of *btacoin* will be created. The course name will be checked to ensure the correct coin is being revoked, then the coin will be removed from the registry using the *DelState* command. Finally, a success status will be returned to the caller.

```
    coinToRemoveargs := stub.GetStringArgs()

    if len(coinToRemoveargs) != 3 {

        fmt.Println("Args are not correct! Please provide two input
parameters!")

    }

    cName := coinToRemoveargs[3]
    coinToRemove := coinToRemoveargs[2]

    existingCoin, err := stub.GetState(coinToRemove)

    if err != nil {

        return shim.Error("Coin Does not exist! Error message: " +
err.Error())

    }

    btacoin := btaCoin{}
    json.Unmarshal(existingCoin, btacoin)

    if existingCoin != nil {

        if btacoin.CourseName == cName {

            stub.DelState(coinToRemove)

        }

    }

    return shim.Success(existingCoin)
```

```

132
133 func (Contract *CoinChaincode) revokeCoin(stub shim.ChaincodeStubInterface, args []string) peer.Response {
134     coinToRemoveargs := stub.GetStringArgs()
135     if len(coinToRemoveargs) != 3 {
136         fmt.Println("Args are not correct! Please provide two input parameters!")
137     }
138
139     cName := coinToRemoveargs[3]
140     coinToRemove := coinToRemoveargs[2]
141
142     existingCoin, err := stub.GetState(coinToRemove)
143     if err != nil {
144         return shim.Error("Coin Does not exist! Error message: " + err.Error())
145     }
146
147     btacoin := btaCoin{}
148     json.Unmarshal(existingCoin, btacoin)
149
150     if existingCoin != nil {
151         if btacoin.CourseName == cName {
152             stub.DelState(coinToRemove)
153         }
154     }
155
156     return shim.Success(existingCoin)
157 }

```

Figure 30 - Snippet 1.29

Creating the *initLedger* Method

The function below will be used to perform any initialization logic during instantiations or upgrades.

This code performs several tasks. First, an empty array is created to hold coins. Next, the array is marshalled into binary so it can be stored. Next, a single coin will be placed into the array to initialize it and a success status will be returned to the caller.

```
func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) peer.Response {  
    coinBank := []btaCoin{  
        btaCoin{CoinID: 1, FullName: "Genesis Coin", CourseName: "Architect", Score: 100},  
    }  
    coinBuffer, err := json.Marshal(coinBank)  
    if err != nil {  
        fmt.Println("Error Occurred: ", err.Error())  
    }  
  
    fmt.Println("Created and now preparing your Genesis Coin for commit: ", coinBank)  
  
    stub.PutState("1", coinBuffer)  
  
    return shim.Success(nil)  
}
```

```
161  
162 func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) peer.Response {  
163     coinBank := []btaCoin{  
164         btaCoin{CoinID: 1, FullName: "Genesis Coin", CourseName: "Architect", Score: 100},  
165     }  
166     coinBuffer, err := json.Marshal(coinBank)  
167     if err != nil {  
168         fmt.Println("Error Occurred: ", err.Error())  
169     }  
170  
171     fmt.Println("Created and now preparing your Genesis Coin for commit: ", coinBank)  
172  
173     stub.PutState("1", coinBuffer)  
174  
175     return shim.Success(nil)  
176 }
```

Figure 31 - Snippet 1.30

As a final step the *initLedger* function added in the previous step must be called from the default *Init* chaincode function. Add the following code to the *Init* function.

```
Contract.initLedger(stub)

return shim.Success(nil)
```

```
21
22 ✓ func (Contract *CoinChaincode) Init(stub shim.ChaincodeStubInterface) peer.Response {
23     Contract.initLedger(stub)
24     return shim.Success(nil)
25 }
26
```

Figure 32 - Snippet 1.31

Lab 2 - Adding Rich Queries

Getting Started

In this lab two query functions will be added to the *btacoin.go* chaincode. One query will return a coin based on its ID, the second query function will return all coins. Begin by opening the *btacoin.go* file for editing in Visual Studio Code. Start by creating the method definition for the *findByID* method using the code below.

```
func (Contract *CoinChaincode) findById(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
}
```

```
173  
174 func (Contract *CoinChaincode) findById(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
175 }
```

Figure 33 - Snippet 2.1

Add the following code to ensure the correct number of input parameters have been supplied.

```
params := stub.GetStringArgs()  
coinID := params[1]  
fmt.Println("Coin ID you turned in are: ", coinID)  
if len(params) != 2 {  
    fmt.Println("Please pass in the coinID")  
}
```

```
173  
174 ✓ func (Contract *CoinChaincode) findById(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
175     params := stub.GetStringArgs()  
176     coinID := params[1]  
177     fmt.Println("Coin ID you turned in are: ", coinID)  
178 ✓ if len(params) != 2 {  
179     fmt.Println("Please pass in the coinID")  
180 }  
181 }
```

Figure 34 - Snippet 2.2

Next, add the code below to make the call to the *GetState* function which will query the World State database.

```
    coinFound, err := stub.GetState(coinID)

    if err != nil {

        fmt.Println("Couldn't find coin! Please Try again",
err.Error())

    }
```

```
173
174 ✓ func (Contract *CoinChaincode) findByID(stub shim.ChaincodeStubInterface, args []string) peer.Response {
175     params := stub.GetStringArgs()
176     coinID := params[1]
177     fmt.Println("Coin ID you turned in are: ", coinID)
178 ✓     if len(params) != 2 {
179         fmt.Println("Please pass in the coinID")
180     }
181
182     coinFound, err := stub.GetState(coinID)
183 ✓     if err != nil {
184         fmt.Println("Couldn't find coin! Please Try again", err.Error())
185     }
186
187 }
```

Figure 35 - Snippet 2.3

Use the code below to check the returned results. If the returned results are empty an error will be raised.

```
        if len(coinFound) == 0 {  
            return shim.Error("Coin does not exist! Please Try again")  
        }  
  
        fmt.Println("Here is the coin result from your query:",  
coinFound)
```

```
173  
174 func (Contract *CoinChaincode) findByID(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
175     params := stub.GetStringArgs()  
176     coinID := params[1]  
177     fmt.Println("Coin ID you turned in are: ", coinID)  
178     if len(params) != 2 {  
179         fmt.Println("Please pass in the coinID")  
180     }  
181  
182     coinFound, err := stub.GetState(coinID)  
183     if err != nil {  
184         fmt.Println("Couldn't find coin! Please Try again", err.Error())  
185     }  
186  
187     if len(coinFound) == 0 {  
188         return shim.Error("Coin does not exist! Please Try again")  
189     }  
190     fmt.Println("Here is the coin result from your query:", coinFound)  
191  
192 }
```

Figure 36 - Snippet 2.4

Finally, use the code below to create a human-readable output to the console and return success to the caller.

```
        foundCoin := btaCoin{}

        json.Unmarshal(coinFound, &foundCoin)

        fmt.Println("Here is the human readable version: \n Course
Name:", foundCoin)

    return shim.Success(coinFound)
```

```
173
174 func (Contract *CoinChaincode) findByID(stub shim.ChaincodeStubInterface, args []string) peer.Response {
175     params := stub.GetStringArgs()
176     coinID := params[1]
177     fmt.Println("Coin ID you turned in are: ", coinID)
178     if len(params) != 2 {
179         fmt.Println("Please pass in the coinID")
180     }
181
182     coinFound, err := stub.GetState(coinID)
183     if err != nil {
184         fmt.Println("Couldn't find coin! Please Try again", err.Error())
185     }
186
187     if len(coinFound) == 0 {
188         return shim.Error("Coin does not exist! Please Try again")
189     }
190     fmt.Println("Here is the coin result from your query:", coinFound)
191
192     foundCoin := btaCoin{}
193     json.Unmarshal(coinFound, &foundCoin)
194     fmt.Println("Here is the human readable version: \n Course Name:", foundCoin)
195
196     return shim.Success(coinFound)
197
198 }
```

Figure 37 - Snippet 2.5

Creating the *returnCoinBank* Method

Now create the outline for the method that will return all coins, *returnCoinBank*. Use the code below to create the method outline.

```
func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {  
  
}
```

```
199  
200 func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {  
201 }
```

Figure 38 - Snippet 2.6

The *GetStateByRange* method will be used to return all coins. Note the use of the double-quotes in the *GetStateByRange* call to indicate all records should be returned.

```
keyIteratorInterface, err := stub.GetStateByRange("", "")  
  
if err != nil {  
    fmt.Println("Error when grabbing the Range:", err.Error())  
}
```

```
199  
200 func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {  
201     keyIteratorInterface, err := stub.GetStateByRange("", "")  
202     if err != nil {  
203         fmt.Println("Error when grabbing the Range:", err.Error())  
204     }  
205  
206     var courseCoins []string  
207  
208     for keyIteratorInterface.HasNext() {  
209         currentCoin, error := keyIteratorInterface.Next()  
210         if error != nil {  
211             return shim.Error(error.Error())  
212         }  
213  
214         fmt.Println("The current coin Values \n Key: ", currentCoin.GetKey(), "\n Value: ", currentCoin.GetValue())  
215  
216         key := string(currentCoin.GetKey())  
217         value := string(currentCoin.GetValue())  
218  
219         coinFoundObject := "{\"Key\":\"" + key + "\",\"token\":\"" + value + "\"}"  
220  
221         courseCoins = append(courseCoins, coinFoundObject)  
222     }  
223  
224 }
```

Figure 39 - Snippet 2.7

Since the transaction will be submitted to the peer over protobuf, it must be serialized. The easiest and fastest way to serialize data to bytes is through the JSON object. Start by initializing a string array to hold all coinResults (JSON Objects) using the code below.

```
var courseCoins []string
```

```
199
200 v func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {
201     keyIteratorInterface, err := stub.GetStateByRange("", "")
202 v     if err != nil {
203         fmt.Println("Error when grabbing the Range:", err.Error())
204     }
205
206     var courseCoins []string
207
208 }
```

Figure 40 - Snippet 2.8

The code below will create a *for* loop to loop through the returned results and add each to the *courseCoins* array. An error handler is included to log any errors during the process. Next, each key and its value will be assigned to a variable to it can be templated into a string object. Then the Key / Value variables can be written to the JSON object. The new string is then pushed into the array of string objects.

```
for keyIteratorInterface.HasNext() {
    currentCoin, error := keyIteratorInterface.Next()
    if error != nil {
        return shim.Error(error.Error())
    }

    fmt.Println("The current coin Values \n Key: ",
currentCoin.GetKey(), "\n Value: ", currentCoin.GetValue())

    key := string(currentCoin.GetKey())
    value := string(currentCoin.GetValue())

    coinFoundObject := "{\"Key\":\"" + key + "\",\"token\":\"" +
value + "\""

    courseCoins = append(courseCoins, coinFoundObject)
}
```

```

199
200 func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {
201     keyIteratorInterface, err := stub.GetStateByRange("", "")
202     if err != nil {
203         fmt.Println("Error when grabbing the Range:", err.Error())
204     }
205
206     var courseCoins []string
207
208     for keyIteratorInterface.HasNext() {
209         currentCoin, error := keyIteratorInterface.Next()
210         if error != nil {
211             return shim.Error(error.Error())
212         }
213
214         fmt.Println("The current coin Values \n Key: ", currentCoin.GetKey(), "\n Value: ", currentCoin.GetValue())
215
216         key := string(currentCoin.GetKey())
217         value := string(currentCoin.GetValue())
218
219         coinFoundObject := "{\"Key\":\"" + key + "\",\"token\":\"" + value + "\"}"
220
221         courseCoins = append(courseCoins, coinFoundObject)
222     }
223
224 }

```

Figure 41 - Snippet 2.9

Use the code below to log out the full list of coins.

```

    fmt.Println("Here is your full list of coins in your Coin Bank.
\n", courseCoins)

```

```

199
200 - func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {
201     keyIteratorInterface, err := stub.GetStateByRange("", "")
202     if err != nil {
203         fmt.Println("Error when grabbing the Range:", err.Error())
204     }
205
206     var courseCoins []string
207
208     for keyIteratorInterface.HasNext() {
209         currentCoin, error := keyIteratorInterface.Next()
210         if error != nil {
211             return shim.Error(error.Error())
212         }
213
214         fmt.Println("The current coin Values \n Key: ", currentCoin.GetKey(), "\n Value: ", currentCoin.GetValue())
215
216         key := string(currentCoin.GetKey())
217         value := string(currentCoin.GetValue())
218
219         coinFoundObject := "{\"Key\":\"" + key + "\",\"token\":\"" + value + "\"}"
220
221         courseCoins = append(courseCoins, coinFoundObject)
222     }
223
224     fmt.Println("Here is your full list of coins in your Coin Bank. \n", courseCoins)
225
226 }

```

Figure 42 - Snippet 2.10

Now the *coinBank* object must be encoded so results can be send back to the peer.

```
coursecoinsBytes := &bytes.Buffer{}  
gob.NewEncoder(coursecoinsBytes).Encode(courseCoins)
```

```
199  
200 v func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {  
201     keyIteratorInterface, err := stub.GetStateByRange("", "")  
202 v     if err != nil {  
203         fmt.Println("Error when grabbing the Range:", err.Error())  
204     }  
205  
206     var courseCoins []string  
207  
208 v     for keyIteratorInterface.HasNext() {  
209         currentCoin, error := keyIteratorInterface.Next()  
210 v         if error != nil {  
211             return shim.Error(error.Error())  
212         }  
213  
214         fmt.Println("The current coin Values \n Key: ", currentCoin.GetKey(), "\n Value: ", currentCoin.GetValue())  
215  
216         key := string(currentCoin.GetKey())  
217         value := string(currentCoin.GetValue())  
218  
219         coinFoundObject := "{\"Key\":\"" + key + "\",\"token\":\"" + value + "\"}"  
220  
221         courseCoins = append(courseCoins, coinFoundObject)  
222     }  
223  
224     fmt.Println("Here is your full list of coins in your Coin Bank. \n", courseCoins)  
225  
226     coursecoinsBytes := &bytes.Buffer{}  
227     gob.NewEncoder(coursecoinsBytes).Encode(courseCoins)  
228  
229 }  
230  
231
```

Figure 43 - Snippet 2.11

To complete the function, return a success status to the caller.

```
return shim.Success(coursecoinsBytes.Bytes())
```

```
199
200 func (Contract *CoinChaincode) returnCoinBank(stub shim.ChaincodeStubInterface) peer.Response {
201     keyIteratorInterface, err := stub.GetStateByRange("", "")
202     if err != nil {
203         fmt.Println("Error when grabbing the Range:", err.Error())
204     }
205
206     var courseCoins []string
207
208     for keyIteratorInterface.HasNext() {
209         currentCoin, error := keyIteratorInterface.Next()
210         if error != nil {
211             return shim.Error(error.Error())
212         }
213
214         fmt.Println("The current coin Values \n Key: ", currentCoin.GetKey(), "\n Value: ", currentCoin.GetValue())
215
216         key := string(currentCoin.GetKey())
217         value := string(currentCoin.GetValue())
218
219         coinFoundObject := "{\"Key\":\"" + key + "\",\"token\":\"" + value + "\"}"
220
221         courseCoins = append(courseCoins, coinFoundObject)
222     }
223
224     fmt.Println("Here is your full list of coins in your Coin Bank. \n", courseCoins)
225
226     coursecoinsBytes := &bytes.Buffer{}
227     gob.NewEncoder(coursecoinsBytes).Encode(courseCoins)
228
229     return shim.Success(coursecoinsBytes.Bytes())
230 }
231
232
```

Figure 44 - Snippet 2.12

To support the encoder a package will need to be added to the package imports list. Please add the following code to the *import* section at the top of the *btacoin.go* file.

```
"encoding/gob"
```

```
2
3  ✓ import (
4      "encoding/json"
5      "fmt"
6      "strconv"
7      "bytes"
8      "github.com/hyperledger/fabric/core/chaincode/shim"
9      peer "github.com/hyperledger/fabric/protos/peer"
10     "encoding/gob"
11 )
12
```

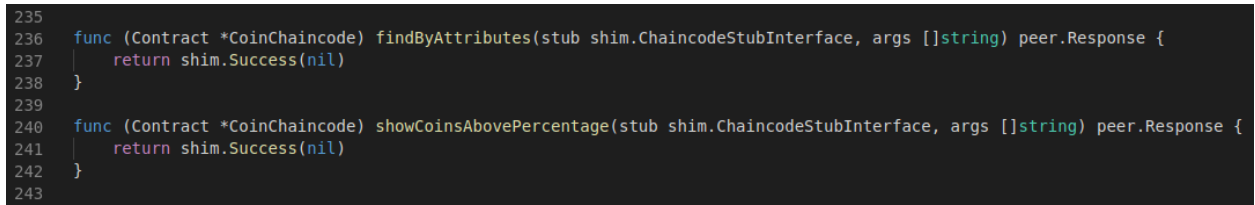
Figure 45 - Snippet 2.13

Adding the `findByAttributes` and `showCoinsAbovePercentage` Methods

Finally, add method outlines for the *findByAttributes* and *showCoinsAbovePercentage* methods. Additional logic will be added to these methods in a later lab.

```
func (Contract *CoinChaincode) findByAttributes(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
    return shim.Success(nil)  
}
```

```
func (Contract *CoinChaincode) showCoinsAbovePercentage(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
    return shim.Success(nil)  
}
```



```
235  
236 func (Contract *CoinChaincode) findByAttributes(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
237 | return shim.Success(nil)  
238 | }  
239  
240 func (Contract *CoinChaincode) showCoinsAbovePercentage(stub shim.ChaincodeStubInterface, args []string) peer.Response {  
241 | return shim.Success(nil)  
242 | }  
243  
244
```

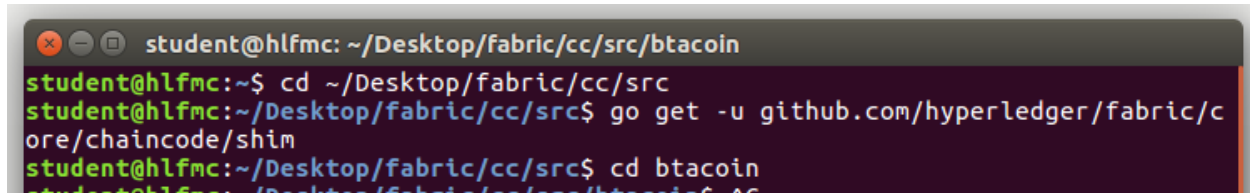
Figure 46 - Snippet 2.14

Lab 3 - Packaging and Deploying the Chaincode

Getting Started

In this lab you will deploy the chaincode written over the last two labs into a separate Docker container for execution. Begin by opening a terminal window and running the following commands.

```
cd ~/Desktop/fabric/cc/src
go get -u github.com/hyperledger/fabric/core/chaincode/shim
cd btacoin
```

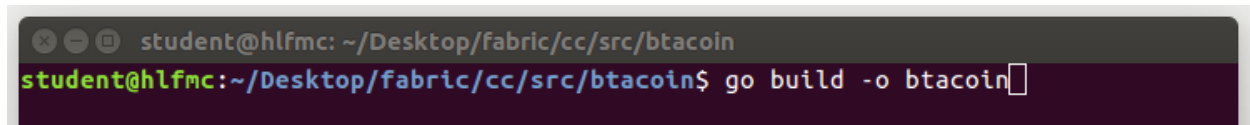
A terminal window with a dark background and light-colored text. The title bar shows 'student@hlfmc: ~/Desktop/fabric/cc/src/btacoin'. The terminal content shows three lines of commands and their prompts: 'student@hlfmc:~\$ cd ~/Desktop/fabric/cc/src', 'student@hlfmc:~/Desktop/fabric/cc/src\$ go get -u github.com/hyperledger/fabric/core/chaincode/shim', and 'student@hlfmc:~/Desktop/fabric/cc/src\$ cd btacoin'.

```
student@hlfmc: ~/Desktop/fabric/cc/src/btacoin
student@hlfmc:~$ cd ~/Desktop/fabric/cc/src
student@hlfmc:~/Desktop/fabric/cc/src$ go get -u github.com/hyperledger/fabric/c
ore/chaincode/shim
student@hlfmc:~/Desktop/fabric/cc/src$ cd btacoin
student@hlfmc:~/Desktop/fabric/cc/src/btacoin$
```

Figure 47 - Snippet 3.1

Compile the chaincode using the command below.

```
go build -o btacoin
```

A terminal window with a dark background and light-colored text. The title bar shows 'student@hlfmc: ~/Desktop/fabric/cc/src/btacoin'. The terminal content shows a single line of command and its prompt: 'student@hlfmc:~/Desktop/fabric/cc/src/btacoin\$ go build -o btacoin'.

```
student@hlfmc:~/Desktop/fabric/cc/src/btacoin$ go build -o btacoin
```

Figure 48 - Snippet 3.2

Open the *docker-compose.yml* file from the *Desktop/fabric/network* folder for editing in Visual Studio Code. Locate the container definition for *Andy.BTA.btacoin.com* and update the *command* line to the following.

```
command: peer node start --peer-chaincodedev=true
```

```
43
44 Andy.BTA.btacoin.com:
45   container_name: Andy.BTA.btacoin.com
46   image: hyperledger/fabric-peer
47   environment:
48     - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_btacoin
49     - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
50     - FABRIC_LOGGING_SPEC=debug
51     - CORE_PEER_ID=Andy.BTA.btacoin.com
52     - CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051
53     - CORE_PEER_LOCALMSPID=BTAMSP
54     - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
55     - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
56     - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=WorldStateCouchAndy:5984
57     - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=Andy
58     - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=SimplePassword
59   command: peer node start --peer-chaincodedev=true
60   volumes:
61     - /var/run:/host/var/run/
62     - ./crypto-config/peerOrganizations/BTA.btacoin.com/peers/Andy.BTA.btacoin.com/msp:/etc/hyperledger/msp/peer
63     - ./crypto-config/peerOrganizations/BTA.btacoin.com/users:/etc/hyperledger/msp/users
64     - ./config:/etc/hyperledger/configtx
65     - ../../cc:/etc/hyperledger/chaincode
66   ports:
67     - 7051:7051
68     - 7053:7053
69   networks:
70     - btacoin
71   depends_on:
72     - Devorderer.btacoin.com
73     - WorldStateCouchAndy
74
```

Figure 49 - Snippet 3.3

Adding the *btacoincc* Container Definition

At the bottom of the *docker-compose.yml* file add the following container definition. This container will be used for the newly created chaincode to be executed in.

```
btacoincc:
  container_name: btacoincc
  image: hyperledger/fabric-ccenv
  tty: true
  environment:
    - GOPATH=/opt/gopath/src
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - FABRIC_LOGGING_SPEC=debug
    - CORE_PEER_ID=btacoincc
    - CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051
    - CORE_PEER_LOCALMSPID=BTAMSP
    -
    CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
    peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.btacoin.
    com/msp
  working_dir: /opt/gopath/src/btacoin
  command: /bin/bash -c 'sleep 6000000'
  volumes:
    - /var/run:/host/var/run/
    - ./msp:/etc/hyperledger/msp
    - ../cc/src:/opt/gopath/src/
  depends_on:
    - Devorderer.btacoin.com
    - Andy.BTA.btacoin.com
  networks:
    - btacoin
```

```

250
251 ✓ btacoincc:
252   container_name: btacoincc
253   image: hyperledger/fabric-ccenv
254   tty: true
255 ✓   environment:
256     - GOPATH=/opt/gopath/src
257     - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
258     - FABRIC_LOGGING_SPEC=debug
259     - CORE_PEER_ID=btacoincc
260     - CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051
261     - CORE_PEER_LOCALMSPID=BTAMSP
262     - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/BTA.btacoin.com/
263   working_dir: /opt/gopath/src/btacoin
264   command: /bin/bash -c 'sleep 6000000'
265 ✓   volumes:
266     - /var/run:/host/var/run/
267     - ./msp:/etc/hyperledger/msp
268     - ../cc/src:/opt/gopath/src/
269 ✓   depends_on:
270     - Devorderer.btacoin.com
271     - Andy.BTA.btacoin.com
272 ✓   networks:
273     - btacoin

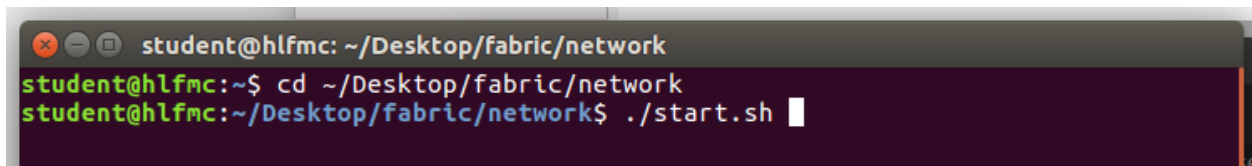
```

Figure 50 - Snippet 3.4

Restarting the Network

In order for the changes to our network to take effect, the network needs to be restarted. This can be accomplished using the *start.sh* script located in the *Desktop/fabric/network* folder. Use the following commands to run the *start.sh* script.

```
cd ~/Desktop/fabric/network  
./start.sh
```

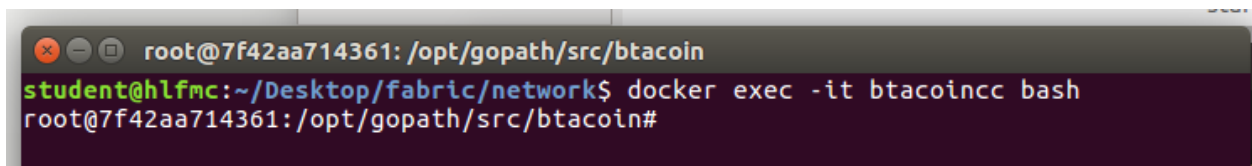
A terminal window with a dark background. The title bar shows 'student@hlfmc: ~/Desktop/fabric/network'. The prompt is 'student@hlfmc:~\$'. The first command entered is 'cd ~/Desktop/fabric/network', and the second is './start.sh'. The cursor is at the end of the second command.

```
student@hlfmc: ~/Desktop/fabric/network  
student@hlfmc:~$ cd ~/Desktop/fabric/network  
student@hlfmc:~/Desktop/fabric/network$ ./start.sh
```

Figure 51 - Snippet 3.5

Once all the network containers are up and running a connection can be made to the chaincode container. Once inside the chaincode container the new chaincode can be started. Connect to the chaincode container using the command below.

```
docker exec -it btacoincc bash
```

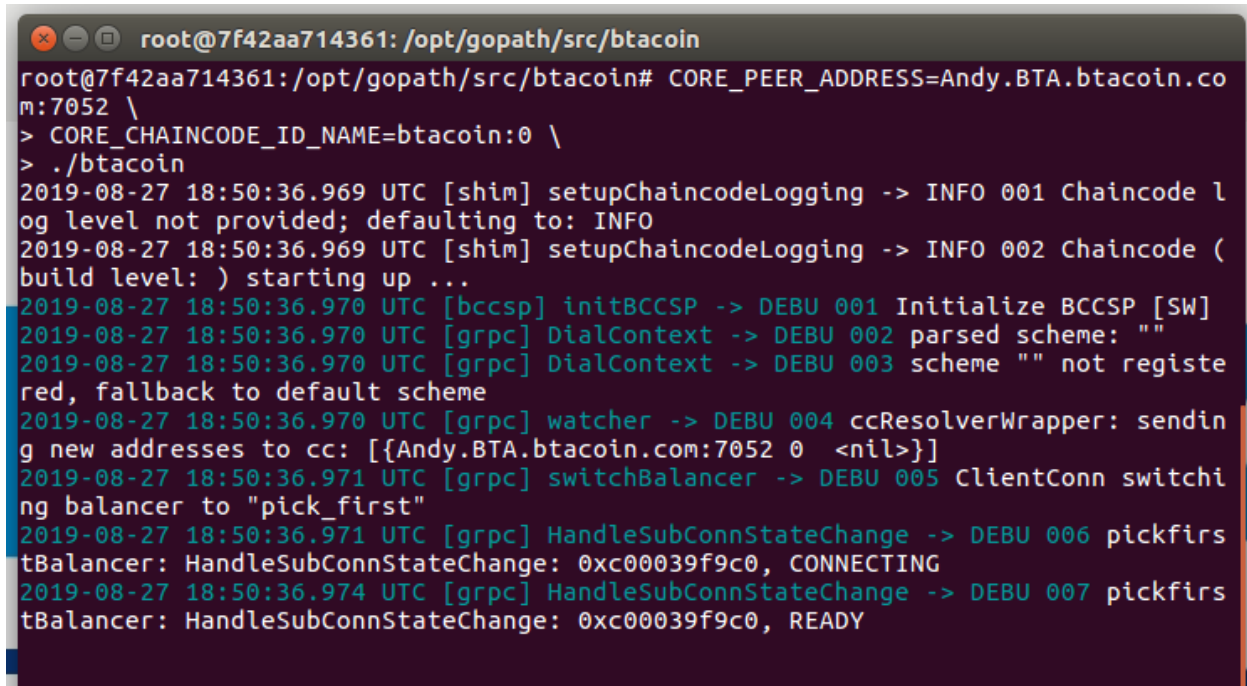
A terminal window with a dark background. The title bar shows 'root@7f42aa714361: /opt/gopath/src/btacoins'. The prompt is 'student@hlfmc:~/Desktop/fabric/network\$'. The command entered is 'docker exec -it btacoincc bash'. The prompt changes to 'root@7f42aa714361: /opt/gopath/src/btacoins#'.

```
root@7f42aa714361: /opt/gopath/src/btacoins  
student@hlfmc:~/Desktop/fabric/network$ docker exec -it btacoincc bash  
root@7f42aa714361: /opt/gopath/src/btacoins#
```

Figure 52 - Snippet 3.6

Use the commands below to start the *btacoin* Go chaincode file.

```
CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7052 \  
CORE_CHAINCODE_ID_NAME=btacoin:0 \  
./btacoin
```

A terminal window with a dark background and light-colored text. The window title is 'root@7f42aa714361: /opt/gopath/src/btacoin'. The prompt is 'root@7f42aa714361: /opt/gopath/src/btacoin#'. The user enters the command 'CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7052 \> CORE_CHAINCODE_ID_NAME=btacoin:0 \> ./btacoin'. The output shows a series of log messages with timestamps and log levels. The messages indicate the setup of chaincode logging, initialization of BCCSP, parsing of the scheme, and the start of the balancer. The balancer starts in 'CONNECTING' state and then transitions to 'READY' state.

```
root@7f42aa714361: /opt/gopath/src/btacoin  
root@7f42aa714361: /opt/gopath/src/btacoin# CORE_PEER_ADDRESS=Andy.BTA.btacoin.co  
m:7052 \  
> CORE_CHAINCODE_ID_NAME=btacoin:0 \  
> ./btacoin  
2019-08-27 18:50:36.969 UTC [shim] setupChaincodeLogging -> INFO 001 Chaincode l  
og level not provided; defaulting to: INFO  
2019-08-27 18:50:36.969 UTC [shim] setupChaincodeLogging -> INFO 002 Chaincode (  
build level: ) starting up ...  
2019-08-27 18:50:36.970 UTC [bccsp] initBCCSP -> DEBU 001 Initialize BCCSP [SW]  
2019-08-27 18:50:36.970 UTC [grpc] DialContext -> DEBU 002 parsed scheme: ""  
2019-08-27 18:50:36.970 UTC [grpc] DialContext -> DEBU 003 scheme "" not registe  
red, fallback to default scheme  
2019-08-27 18:50:36.970 UTC [grpc] watcher -> DEBU 004 ccResolverWrapper: sendin  
g new addresses to cc: [{Andy.BTA.btacoin.com:7052 0 <nil>}]  
2019-08-27 18:50:36.971 UTC [grpc] switchBalancer -> DEBU 005 ClientConn switchi  
ng balancer to "pick_first"  
2019-08-27 18:50:36.971 UTC [grpc] HandleSubConnStateChange -> DEBU 006 pickfirs  
tBalancer: HandleSubConnStateChange: 0xc00039f9c0, CONNECTING  
2019-08-27 18:50:36.974 UTC [grpc] HandleSubConnStateChange -> DEBU 007 pickfirs  
tBalancer: HandleSubConnStateChange: 0xc00039f9c0, READY
```

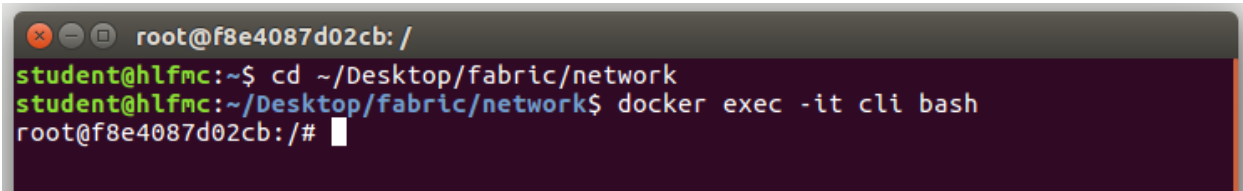
Figure 53 - Snippet 3.7

Install and Instantiate the Chaincode

Leave the terminal window open and open a second terminal window. From the second terminal window connect to the *cli* container using the commands below.

```
cd ~/Desktop/fabric/network
```

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

A terminal window with a dark background and light-colored text. The prompt is 'root@f8e4087d02cb: /'. The user enters 'student@hlfmc:~\$ cd ~/Desktop/fabric/network'. The prompt changes to 'student@hlfmc:~/Desktop/fabric/network\$'. The user enters 'student@hlfmc:~/Desktop/fabric/network\$ docker exec -it cli bash'. The prompt changes to 'root@f8e4087d02cb:/#'.

```
root@f8e4087d02cb: /  
student@hlfmc:~$ cd ~/Desktop/fabric/network  
student@hlfmc:~/Desktop/fabric/network$ docker exec -it cli bash  
root@f8e4087d02cb:/#
```

Figure 54 - Snippet 3.8

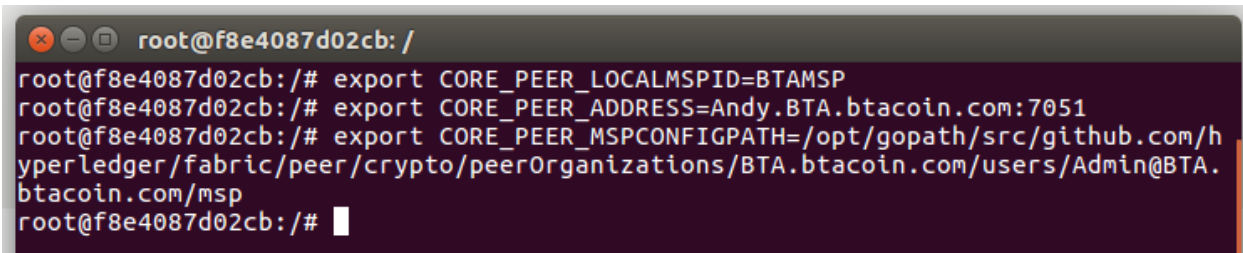
Run the following commands to ensure environment variables are set correctly.

```
export CORE_PEER_LOCALMSPID=BTAMSP
```

```
export CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051
```

```
export
```

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/  
peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.btacoin.  
com/msp
```

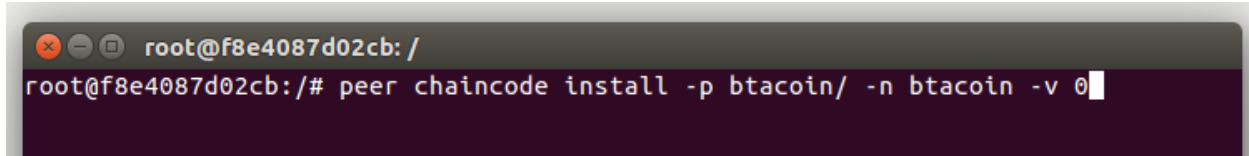
A terminal window with a dark background and light-colored text. The prompt is 'root@f8e4087d02cb: /'. The user enters 'root@f8e4087d02cb:/# export CORE_PEER_LOCALMSPID=BTAMSP'. The prompt changes to 'root@f8e4087d02cb:/#'. The user enters 'root@f8e4087d02cb:/# export CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051'. The prompt changes to 'root@f8e4087d02cb:/#'. The user enters 'root@f8e4087d02cb:/# export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.btacoin.com/msp'. The prompt changes to 'root@f8e4087d02cb:/#'.

```
root@f8e4087d02cb: /  
root@f8e4087d02cb:/# export CORE_PEER_LOCALMSPID=BTAMSP  
root@f8e4087d02cb:/# export CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051  
root@f8e4087d02cb:/# export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/h  
yperledger/fabric/peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.  
btacoin.com/msp  
root@f8e4087d02cb:/#
```

Figure 55 - Snippet 3.9

Now the *peer chaincode* command can be used to install the chaincode on the peer. Once installed the chaincode must be instantiated before it can be executed. Use the command below to install the chaincode on the peer. Note the use of the *-n* flag to indicate the chaincode name as well as the *-v* flag used to indicate the chaincode version.

```
peer chaincode install -p btacoin/ -n btacoin -v 0
```



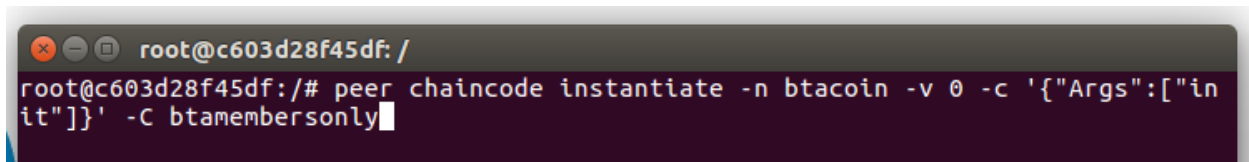
```
root@f8e4087d02cb: /  
root@f8e4087d02cb:/# peer chaincode install -p btacoin/ -n btacoin -v 0
```

Figure 56 - Snippet 3.10

Next, instantiate the chaincode on the *btamembersonly* channel and the *coinexchangechannel* using the commands below.

```
peer chaincode instantiate -n btacoin -v 0 -c '{"Args":["init"]}' -C  
btamembersonly
```

```
peer chaincode instantiate -n btacoin -v 0 -c '{"Args":["init"]}' -C  
coinexchangechannel
```

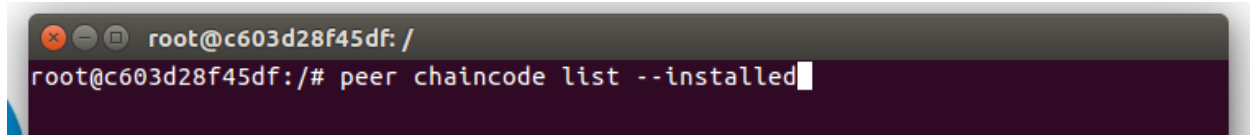


```
root@c603d28f45df: /  
root@c603d28f45df:/# peer chaincode instantiate -n btacoin -v 0 -c '{"Args":["in  
it"]}' -C btamembersonly
```

Figure 57 - Snippet 3.11

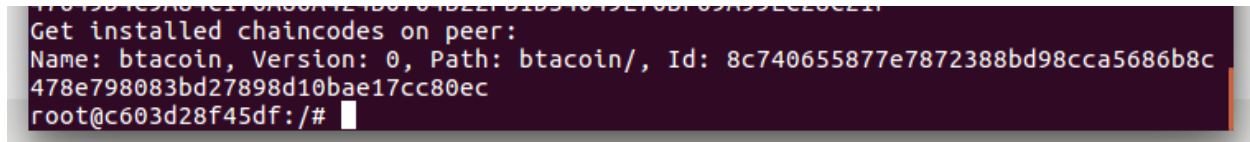
To verify the chaincode has been properly installed and instantiated the *peer chaincode list* command can be used. Use the command below to check the installed chaincodes.

```
peer chaincode list --installed
```



```
root@c603d28f45df: /  
root@c603d28f45df:/# peer chaincode list --installed
```

Figure 58 - Snippet 3.12



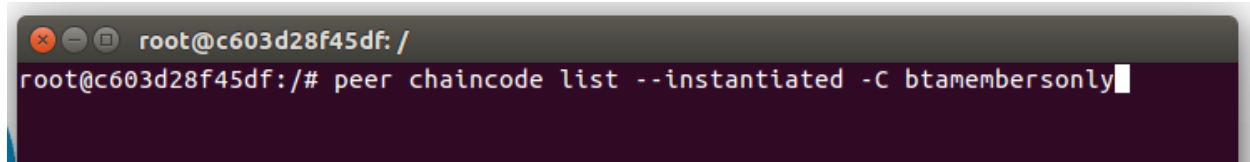
```
Get installed chaincodes on peer:  
Name: btacoin, Version: 0, Path: btacoin/, Id: 8c740655877e7872388bd98cca5686b8c  
478e798083bd27898d10bae17cc80ec  
root@c603d28f45df:/#
```

Figure 59 - The list of installed chaincodes.

Now verify the chaincode was instantiated on both the *btamembersonly* and *coinexchangechannel* channels using the commands below.

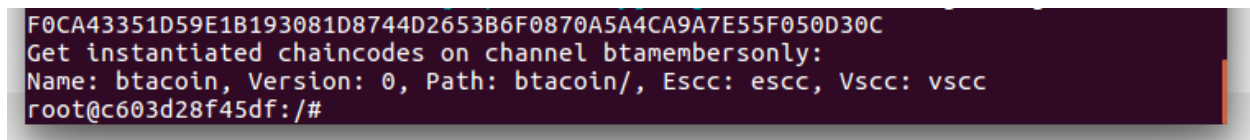
```
peer chaincode list --instantiated -C btamembersonly
```

```
peer chaincode list --instantiated -C coinexchangechannel
```



```
root@c603d28f45df: /  
root@c603d28f45df:/# peer chaincode list --instantiated -C btamembersonly
```

Figure 60 - Snippet 3.13



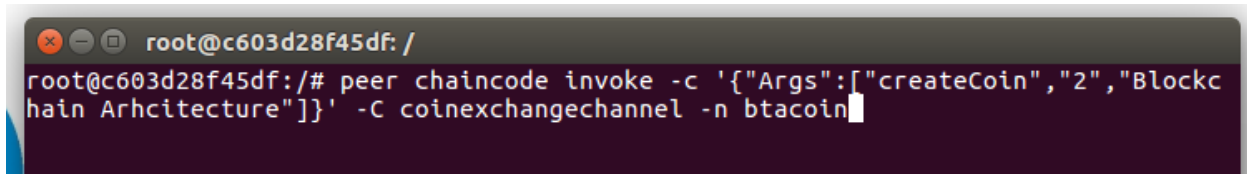
```
F0CA43351D59E1B193081D8744D2653B6F0870A5A4CA9A7E55F050D30C  
Get instantiated chaincodes on channel btamembersonly:  
Name: btacoin, Version: 0, Path: btacoin/, Escc: escc, Vscc: vscc  
root@c603d28f45df:/#
```

Figure 61 - The list of instantiated chaincodes

Testing the Chaincode via the Command Line

In this section the chaincode will be tested using the command line interface. Begin by creating a coin to be assigned later. Use the commands below in the second terminal window.

```
peer chaincode invoke -c '{"Args":["createCoin","2","Blockchain Architecture"]}' -C coinexchangechannel -n btacoin  
  
peer chaincode invoke -c '{"Args":["createCoin","3","Blockchain Architecture"]}' -C coinexchangechannel -n btacoin
```

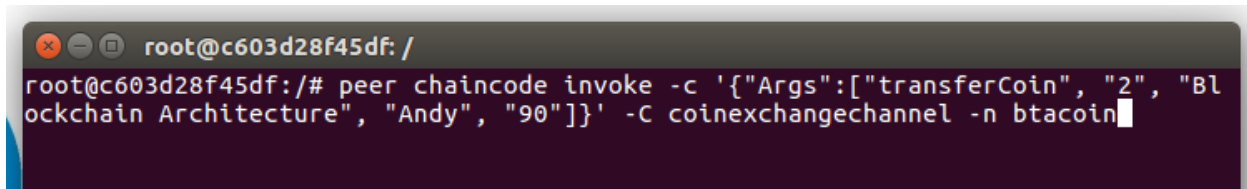
A terminal window with a dark background and light text. The prompt is 'root@c603d28f45df: /'. The command entered is 'peer chaincode invoke -c '{"Args":["createCoin","2","Blockchain Architecture"]}' -C coinexchangechannel -n btacoin'. The cursor is at the end of the command.

```
root@c603d28f45df: /  
root@c603d28f45df:/# peer chaincode invoke -c '{"Args":["createCoin","2","Blockchain Architecture"]}' -C coinexchangechannel -n btacoin
```

Figure 62 - Snippet 3.14

Assume that the student *Andy* has completed the Blockchain Architecture course and should now be awarded a coin. Use the command below to initiate a transaction which will assign ownership of coin 2 to *Andy*.

```
peer chaincode invoke -c '{"Args":["transferCoin", "2", "Blockchain Architecture", "Andy", "90"]}' -C coinexchangechannel -n btacoin
```

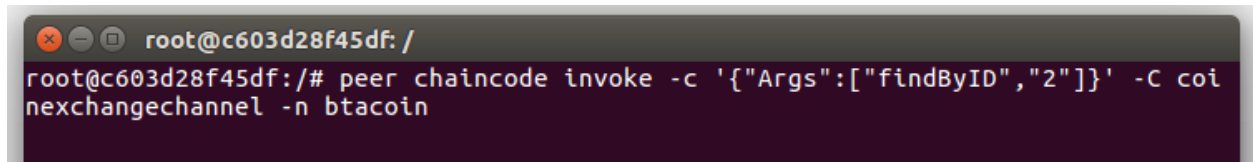
A terminal window with a dark background and light text. The prompt is 'root@c603d28f45df: /'. The command entered is 'peer chaincode invoke -c '{"Args":["transferCoin", "2", "Blockchain Architecture", "Andy", "90"]}' -C coinexchangechannel -n btacoin'. The cursor is at the end of the command.

```
root@c603d28f45df: /  
root@c603d28f45df:/# peer chaincode invoke -c '{"Args":["transferCoin", "2", "Blockchain Architecture", "Andy", "90"]}' -C coinexchangechannel -n btacoin
```

Figure 63 - Snippet 3.15

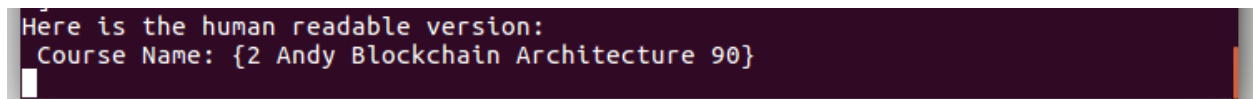
To see whether the assignment to Andy was successful the *findById* function from the chaincode can be called. Enter the command below in the second terminal window. Look in the first terminal window for results.

```
peer chaincode invoke -c '{"Args":["findById","2"]}' -C coi  
nexchangechannel -n btacoin
```

A terminal window with a dark background. The prompt is 'root@c603d28f45df: /'. The command entered is 'peer chaincode invoke -c '{"Args":["findById","2"]}' -C coinexchangechannel -n btacoin'.

```
root@c603d28f45df: /  
root@c603d28f45df:/# peer chaincode invoke -c '{"Args":["findById","2"]}' -C coi  
nexchangechannel -n btacoin
```

Figure 64 - Snippet 3.16

A terminal window showing the output of the command. It says 'Here is the human readable version:' followed by 'Course Name: {2 Andy Blockchain Architecture 90}' on the next line.

```
Here is the human readable version:  
Course Name: {2 Andy Blockchain Architecture 90}
```

Figure 65 - The results returned by the *findById* function

Viewing World State Data in *Fauxton*

The *Fauxton* utility can also be used to view records in the World State database. Open a browser in your lab environment and navigate to http://localhost:5984/_utils. Login to the Fauxton utility using a username of *Andy* and a password of *SimplePassword*.

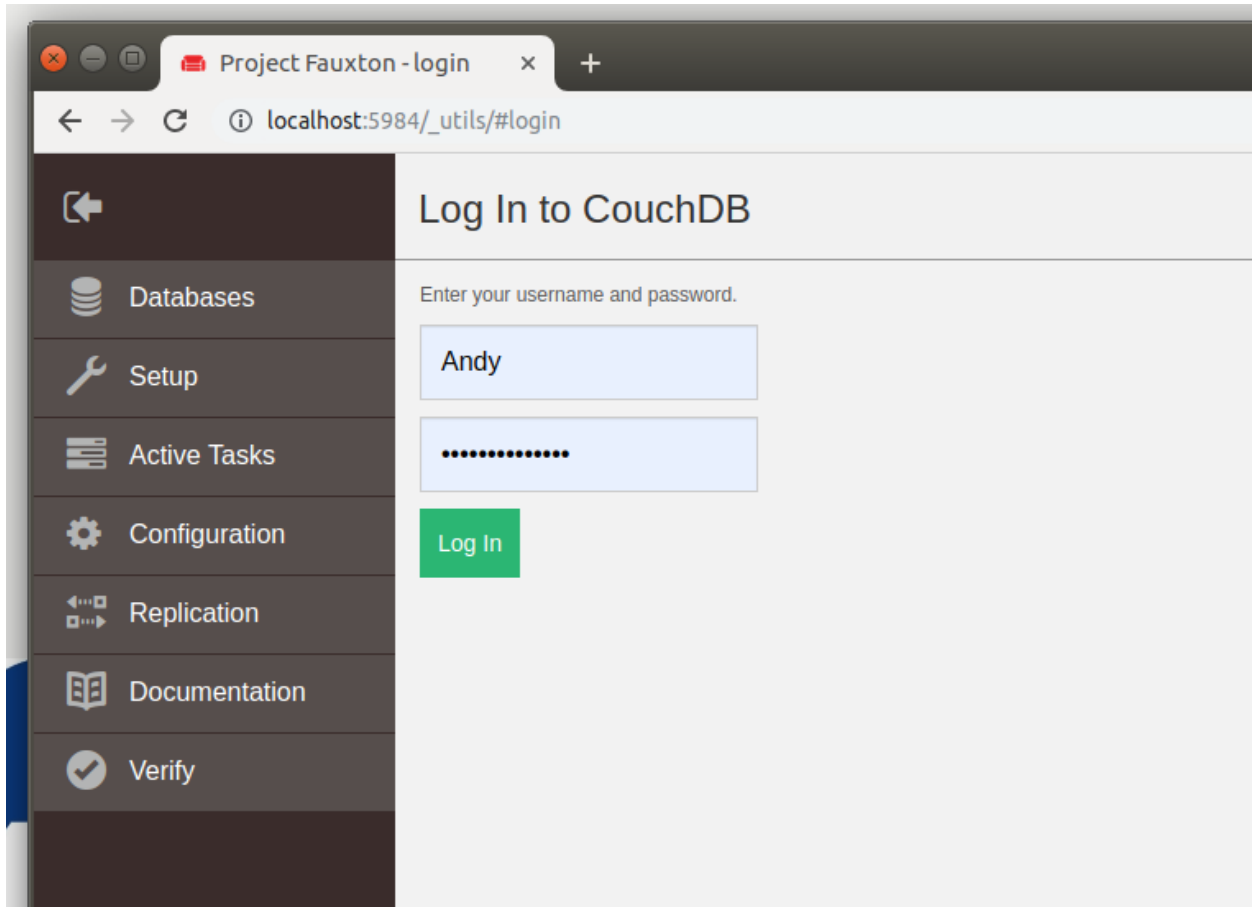


Figure 66 - Logging into Fauxton

Once logged in, click on the *coinexchangechannel_btacoin* listing.

Databases			Databases
Name	Size	# of Docs	
<i>_replicator</i>	2.3 KB	1	
<i>_users</i>	2.3 KB	1	
<i>btamembersonly_</i>	3.7 KB	2	
<i>btamembersonly_btacoin</i>	0.6 KB	1	
<i>btamembersonly_lsc</i>	0.7 KB	1	
<i>coinexchangechannel_</i>	5.5 KB	2	
<i>coinexchangechannel_btacoin</i>	1.5 KB	3	
<i>coinexchangechannel_lsc</i>	0.7 KB	1	

Figure 67 - The available databases

Notice all the created coins on the channel. Click on the coin with an ID of 2.

TableMetadata{ } JSON

Create Document




id		key	value
<input type="checkbox"/>	 1	1	{ "rev": "1-6d8086ce34c7603daa4ca01eb8..." }
<input type="checkbox"/>	 2	2	{ "rev": "2-e9a849eab15fcd4fd5505dd3a02..." }
<input type="checkbox"/>	 3	3	{ "rev": "1-1a408210bd5eb1e7c962cde8b3..." }

Figure 68 - Records / documents in the current database

Notice the coin has been assigned to *Andy* for the *Blockchain Architecture* course.

coinexchangechannel_btacoin > 2

☒ Save Changes Cancel

```
1 {  
2   "_id": "2",  
3   "_rev": "2-e9a849eab15fcd4fdf5505dd3a026939",  
4   "coinID": 2,  
5   "courseName": "Blockchain Architecture",  
6   "fullName": "Andy",  
7   "score": 90,  
8   "~version": "\u0000CgMBBAA=",  
9 }
```

Figure 69 - Record / document detail

Lab 4 - Intro to Complex Queries using Composite Keys

Getting Started

In this lab a new chaincode will be created. This new chaincode will be used to manage Composite Key queries. In this contract, the goal is create functionality to support querying on the two unique attributes of a coin - the full name and the course name.

Begin by opening a command prompt and issuing the following commands to create a *compositebtacoin* subfolder an empty Go code file.

```
cd ~/Desktop/fabric/cc/src/  
mkdir compositebtacoin && cd compositebtacoin  
touch compositebtacoin.go
```

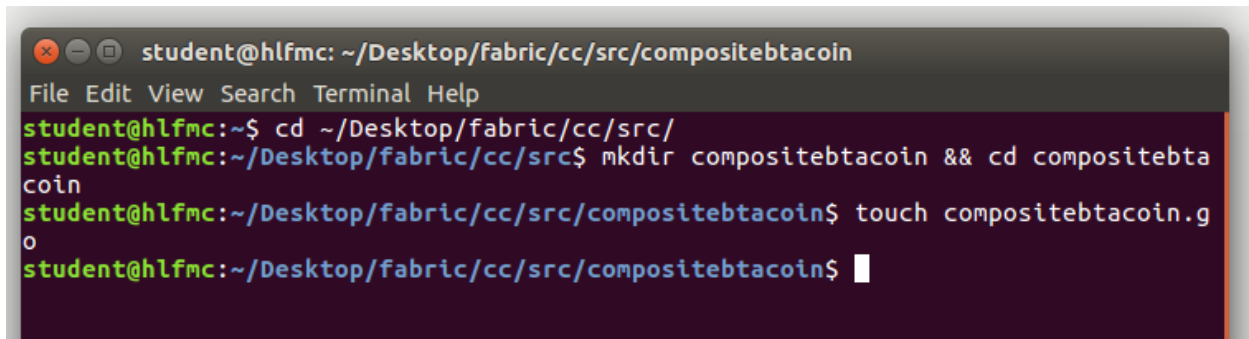
A terminal window titled 'student@hlfmc: ~/Desktop/fabric/cc/src/compositebtacoin' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:
student@hlfmc:~\$ cd ~/Desktop/fabric/cc/src/
student@hlfmc:~/Desktop/fabric/cc/src\$ mkdir compositebtacoin && cd compositebtacoin
student@hlfmc:~/Desktop/fabric/cc/src/compositebtacoin\$ touch compositebtacoin.go
student@hlfmc:~/Desktop/fabric/cc/src/compositebtacoin\$

Figure 70 - Snippet 4.1

Open the newly created *compositebtacoin.go* file in Visual Studio Code. Insert the code below. Note the similarities between this code and the code in the original *btacoin.go* chaincode.

```
package main

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

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

//CoinChaincode is the smart contract interface we initialize
type CoinChaincode struct {
}

// btaCoin is the type asset we are tracking
type btaCoin struct {
    CoinID int `json:"coinID"`
    FullName string `json:"fullName"`
    CourseName string `json:"courseName"`
    Score int `json:"score"`
}

// Init is The initialization function
func (Contract *CoinChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response {
    // Throw away func Name and get Parameters
    Contract.initLedger(stub)
    return shim.Success(nil)
}
```

```

func main() {
    err := shim.Start(new(CoinChaincode))

    if err != nil {
        fmt.Printf("Chaincode couldnt start:" + err.Error())
    }
}

```

```

compositebtacoin.go x btacoin.go
home > student > Desktop > Fabric > cc > src > compositebtacoin > compositebtacoin.go
1  package main
2
3  import (
4      "encoding/json"
5      "fmt"
6      "strconv"
7      "github.com/hyperledger/fabric/core/chaincode/shim"
8      peer "github.com/hyperledger/fabric/protos/peer"
9  )
10
11 //CoinChaincode is the smart contract interface we initialize
12 type CoinChaincode struct {
13 }
14
15 // btaCoin is the type asset we are tracking
16 type btaCoin struct {
17     CoinID int `json:"coinID"`
18     FullName string `json:"fullName"`
19     CourseName string `json:"courseName"`
20     Score int `json:"score"`
21 }
22 // Init is The initialization function
23 func (Contract *CoinChaincode) Init(stub shim.ChaincodeStubInterface) peer.Response {
24     // Throw away func Name and get Parameters
25     Contract.initLedger(stub)
26     return shim.Success(nil)
27 }
28
29 func main() {
30     err := shim.Start(new(CoinChaincode))
31     if err != nil {
32         fmt.Printf("Chaincode couldnt start:" + err.Error())
33     }
34 }
35

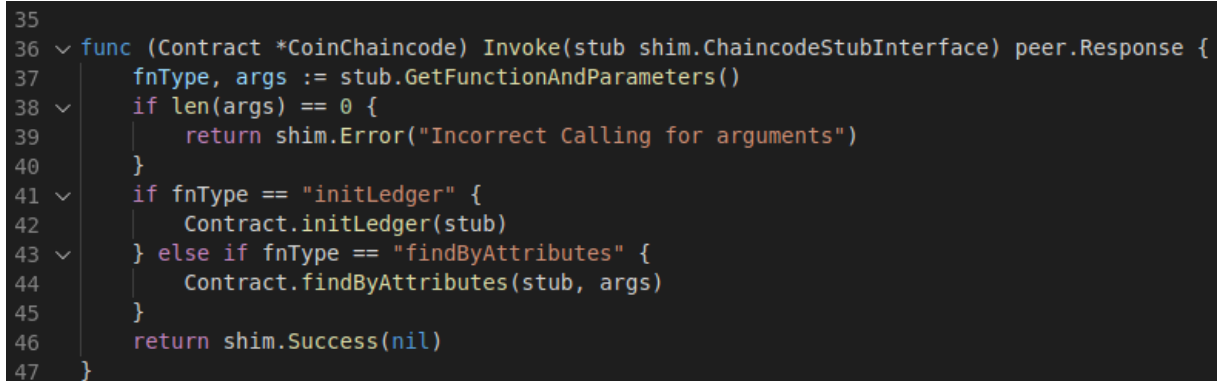
```

Figure 71 - Snippet 4.2

Building the *Invoke* Function

The *Invoke* function will be created next using the code below. Note the simpler implementation of *Invoke* - only two functions are being called, one to initialize the ledger and the second to return a coin based on its attributes.

```
func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {  
    fnType, args := stub.GetFunctionAndParameters()  
    if len(args) == 0 {  
        return shim.Error("Incorrect Calling for arguments")  
    }  
    if fnType == "initLedger" {  
        Contract.initLedger(stub)  
    } else if fnType == "findByAttributes" {  
        Contract.findByAttributes(stub, args)  
    }  
    return shim.Success(nil)  
}
```



```
35  
36 v func (Contract *CoinChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {  
37     fnType, args := stub.GetFunctionAndParameters()  
38 v     if len(args) == 0 {  
39         return shim.Error("Incorrect Calling for arguments")  
40     }  
41 v     if fnType == "initLedger" {  
42         Contract.initLedger(stub)  
43 v     } else if fnType == "findByAttributes" {  
44         Contract.findByAttributes(stub, args)  
45     }  
46     return shim.Success(nil)  
47 }
```

Figure 72 - Snippet 4.3

Initializing the Chaincode

Use the code below to create a new function for ledger initialization. This function will be filled out in the following steps.

```
func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) {  
  
}
```

```
48  
49 func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) {  
50 }
```

Figure 73 - Snippet 4.4

Add the following code to the *initLedger* function. In this step the composite key (full name + course name) is being set in the database for later use.

```
stub.PutState("FullName~CourseName", []byte{0x00})
```

```
49 ✓ func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) {  
50     stub.PutState("FullName~CourseName", []byte{0x00})  
51 }
```

Figure 74 - Snippet 4.5

In order to test the functionality being developed in this exercise, some sample data will be required. The code below will create five different students. First, the names of the students must be defined.

```
owners := []string{"GenesisCoin", "John", "Denny", "Ron", "Kris"}
```

```
48  
49 ✓ func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) {  
50     stub.PutState("FullName~CourseName", []byte{0x00})  
51     owners := []string{"GenesisCoin", "John", "Denny", "Ron", "Kris"}  
52 }
```

Figure 75 - Snippet 4.6

Use the following code to create a coin for each of the five students. Inside the *for* loop a counter *coinID* will be managed, a variable *currCoin* will be used for the coin currently being created, the *currCoin* will be marshalled, any errors will be caught, and coin will be added to the World State database.

```
    for i, owner := range owners {  
        coinID := i + 1  
  
        currCoin := btaCoin{CoinID: coinID, FullName: owner,  
CourseName: "Architect", Score: 100}  
  
        coinBuffer, err := json.Marshal(currCoin)  
        if err != nil {  
            fmt.Println("Something went wrong!")  
        }  
  
        fmt.Println("Created and now preparing your coin  
#" + strconv.Itoa(coinID), " for commit: ")  
  
        compKey, err :=  
stub.CreateCompositeKey("FullName~CourseName",  
[]string{currCoin.FullName, currCoin.CourseName})  
  
        if err != nil {  
            fmt.Println("Something went wrong!")  
        }  
  
        stub.PutState(compKey, coinBuffer)  
    }
```

```

48
49 func (Contract *CoinChaincode) initLedger(stub shim.ChaincodeStubInterface) {
50     stub.PutState("FullName~CourseName", []byte{0x00})
51     owners := []string{"GenesisCoin", "John", "Denny", "Ron", "Kris"}
52
53     for i, owner := range owners {
54         coinID := i + 1
55         currCoin := btaCoin{CoinID: coinID, FullName: owner, CourseName: "Architect", Score: 100}
56
57         coinBuffer, err := json.Marshal(currCoin)
58         if err != nil {
59             fmt.Println("Something went wrong!")
60         }
61
62         fmt.Println("Created and now preparing your coin #" + strconv.Itoa(coinID), " for commit: ")
63
64         compKey, err := stub.CreateCompositeKey("FullName~CourseName", []string{currCoin.FullName, currCoin.CourseName})
65
66         if err != nil {
67             fmt.Println("Something went wrong!")
68         }
69
70         stub.PutState(compKey, coinBuffer)
71     }
72 }

```

Figure 76 - Snippet 4.7

Querying by Composite Key

In this step a new function *findByAttributes* will be created. This function will search for a coin based on its composite key. Use the code below to create the *findByAttributes* function.

```
func (Contract *CoinChaincode) findByAttributes(stub shim.ChaincodeStubInterface, args []string) peer.Response {

    // The function will begin by displaying a status message.
    fmt.Println("Creating the composite key to query with...")

    // Next, the composite key to search the database for will be created.
    compositeKey, err :=
    stub.CreateCompositeKey("FullName~CourseName", args)

    // Any returned errors will be handled.
    if err != nil {
        return shim.Error("Something happened wrong: " +
err.Error())
    }

    // Print another status message.
    fmt.Println("Initializing query using Composite Key")

    // Use the getState call to query using the composite key. Handle any return errors.
    CoinFoundBytes, err := stub.GetState(compositeKey)
    if err != nil {
        return shim.Error("Could not find Coin! Error message: " +
err.Error())
    }

    // Print status information to the console.
    fmt.Println("Here is the returned coin info! ", CoinFoundBytes)
```



```

// Return success status to caller.
return shim.Success(CoinFoundBytes)
}

73
74 func (Contract *CoinChaincode) findByAttributes(stub shim.ChaincodeStubInterface, args []string) peer.Response {
75     // The function will begin by displaying a status message.
76     fmt.Println("Creating the composite key to query with...")
77
78     // Next, the composite key to search the database for will be created.
79     compositeKey, err := stub.CreateCompositeKey("FullName-CourseName", args)
80
81     // Any returned errors will be handled.
82     if err != nil {
83         return shim.Error("Something happened wrong: " + err.Error())
84     }
85
86     // Print another status message.
87     fmt.Println("Initializing query using Composite Key")
88
89     // Use the getState call to query using the composite key. Handle any return errors.
90     CoinFoundBytes, err := stub.GetState(compositeKey)
91     if err != nil {
92         return shim.Error("Could not find Coin! Error message: " + err.Error())
93     }
94
95     // Print status information to the console.
96     fmt.Println("Here is the returned coin info! ", CoinFoundBytes)
97
98     // Return success status to caller.
99     return shim.Success(CoinFoundBytes)
100 }

```

Figure 77 - Snippet 4.8

Building the New Chaincode

Ensure all changes are saved to the new chaincode. Open a terminal window and navigate to the *compositebtacoin* source code folder using the command below.

```
cd ~/Desktop/fabric/cc/src/compositebtacoin
```

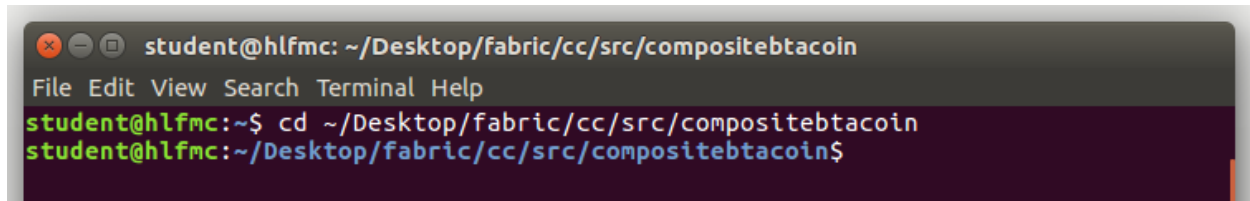
A terminal window titled 'student@hlfmc: ~/Desktop/fabric/cc/src/compositebtacoin'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'student@hlfmc:~\$'. The command 'cd ~/Desktop/fabric/cc/src/compositebtacoin' has been entered and executed. The prompt is now 'student@hlfmc:~/Desktop/fabric/cc/src/compositebtacoin\$'.

Figure 78 - Snippet 4.9

Use the following command to compile the new chaincode.

```
go build -o compositebtacoin
```

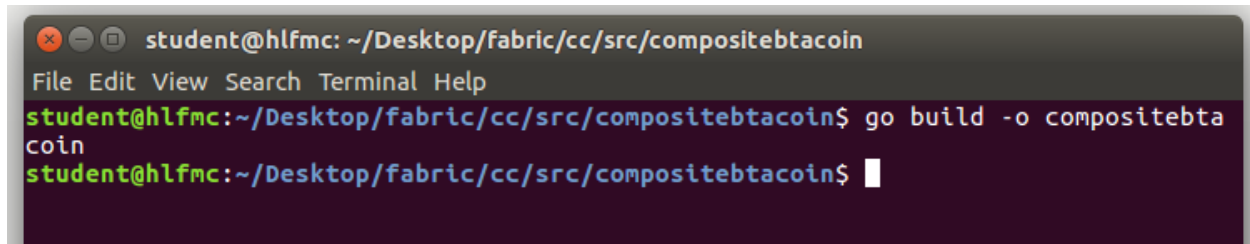
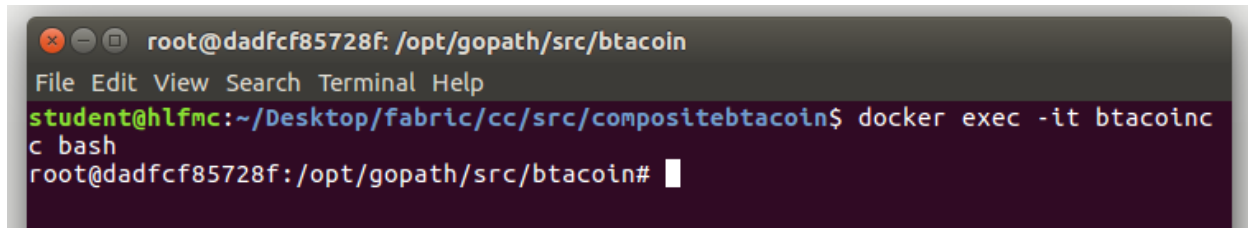
A terminal window titled 'student@hlfmc: ~/Desktop/fabric/cc/src/compositebtacoin'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'student@hlfmc:~/Desktop/fabric/cc/src/compositebtacoin\$'. The command 'go build -o compositebtacoin' has been entered and executed. The prompt is now 'student@hlfmc:~/Desktop/fabric/cc/src/compositebtacoin\$'.

Figure 79 - Snippet 4.10

Installing the New Chaincode

Now enter the container using the command below in preparation to start the chaincode.

```
docker exec -it btacoincc bash
```

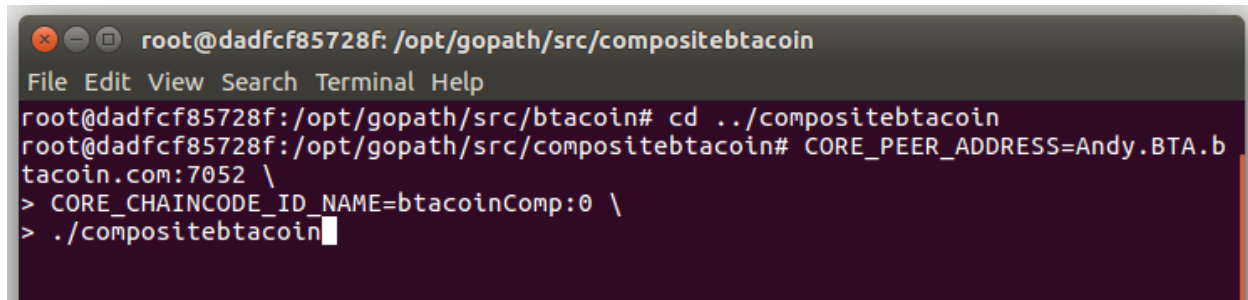


```
root@dadfcf85728f: /opt/gopath/src/btacoins
File Edit View Search Terminal Help
student@hlfmc:~/Desktop/fabric/cc/src/compositebtacoins$ docker exec -it btacoincc bash
root@dadfcf85728f: /opt/gopath/src/btacoins#
```

Figure 80 - Snippet 4.11

Start the chaincode using the commands below.

```
cd ../compositebtacoins
CORE_PEER_ADDRESS=Andy.BTA.btacoins.com:7052 \
CORE_CHAINCODE_ID_NAME=btacoinsComp:0 \
./compositebtacoins
```



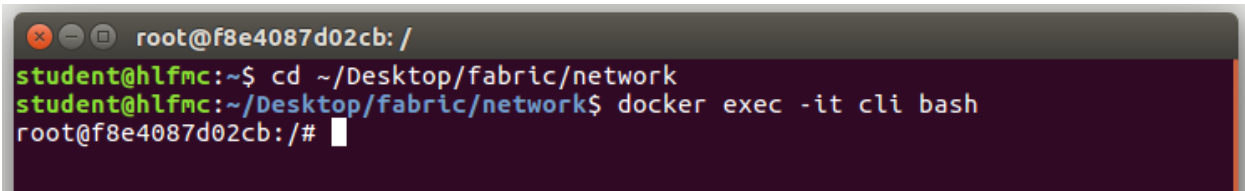
```
root@dadfcf85728f: /opt/gopath/src/compositebtacoins
File Edit View Search Terminal Help
root@dadfcf85728f: /opt/gopath/src/btacoins# cd ../compositebtacoins
root@dadfcf85728f: /opt/gopath/src/compositebtacoins# CORE_PEER_ADDRESS=Andy.BTA.btacoins.com:7052 \
> CORE_CHAINCODE_ID_NAME=btacoinsComp:0 \
> ./compositebtacoins
```

Figure 81 - Snippet 4.12

Leave the terminal window open and open a second terminal window. From the second terminal window connect to the *cli* container using the commands below.

```
cd ~/Desktop/fabric/network
```

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

A terminal window with a dark background and light text. The prompt is 'root@f8e4087d02cb: /'. The user enters 'student@hlfmc:~\$ cd ~/Desktop/fabric/network'. The prompt changes to 'student@hlfmc:~/Desktop/fabric/network\$'. The user enters 'docker exec -it cli bash'. The prompt changes to 'root@f8e4087d02cb:/#' with a cursor at the end.

```
root@f8e4087d02cb: /  
student@hlfmc:~$ cd ~/Desktop/fabric/network  
student@hlfmc:~/Desktop/fabric/network$ docker exec -it cli bash  
root@f8e4087d02cb:/#
```

Figure 82 - Snippet 4.13

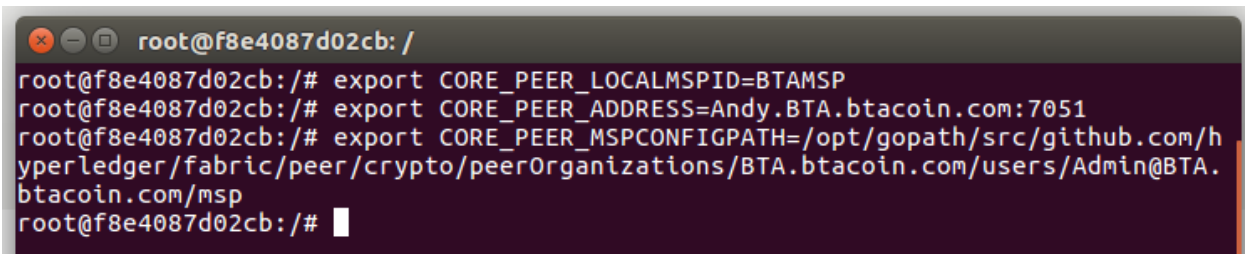
Run the following commands to ensure environment variables are set correctly.

```
export CORE_PEER_LOCALMSPID=BTAMSP
```

```
export CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051
```

```
export
```

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/  
peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.  
com/msp
```

A terminal window with a dark background and light text. The prompt is 'root@f8e4087d02cb: /'. The user enters 'export CORE_PEER_LOCALMSPID=BTAMSP'. The prompt changes to 'root@f8e4087d02cb:/#'. The user enters 'export CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051'. The prompt changes to 'root@f8e4087d02cb:/#'. The user enters 'export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.btacoin.com/msp'. The prompt changes to 'root@f8e4087d02cb:/#' with a cursor at the end.

```
root@f8e4087d02cb: /  
root@f8e4087d02cb:/# export CORE_PEER_LOCALMSPID=BTAMSP  
root@f8e4087d02cb:/# export CORE_PEER_ADDRESS=Andy.BTA.btacoin.com:7051  
root@f8e4087d02cb:/# export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/h  
yperledger/fabric/peer/crypto/peerOrganizations/BTA.btacoin.com/users/Admin@BTA.  
btacoin.com/msp  
root@f8e4087d02cb:/#
```

Figure 83 - Snippet 4.14

Now the *peer chaincode* command can be used to install the chaincode on the peer. Once installed the chaincode must be instantiated before it can be executed. Use the command below to install the chaincode on the peer. Note the use of the *-n* flag to indicate the chaincode name as well as the *-v* flag used to indicate the chaincode version.

```
peer chaincode install -p compositebtacoin/ -n btacoinComp -v 0
```

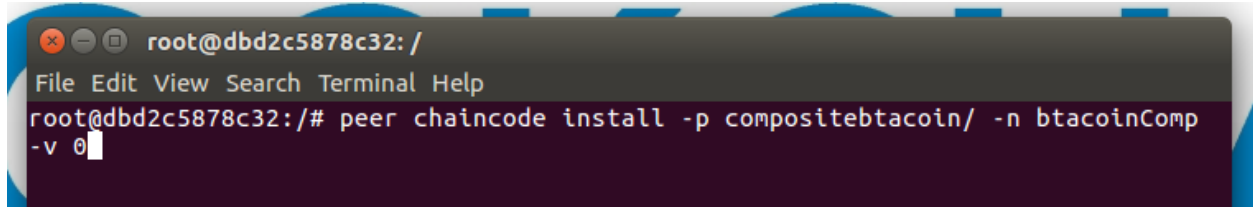
A terminal window with a dark background and light text. The prompt is 'root@dbd2c5878c32: /'. The command 'peer chaincode install -p compositebtacoin/ -n btacoinComp -v 0' is entered and executed. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

Figure 84 - Snippet 4.15

Next, instantiate the chaincode on the *coinexchangechannel* using the command below.

```
peer chaincode instantiate -n btacoinComp -v 0 -c '{"Args":["init"]}'  
-C coinexchangechannel
```

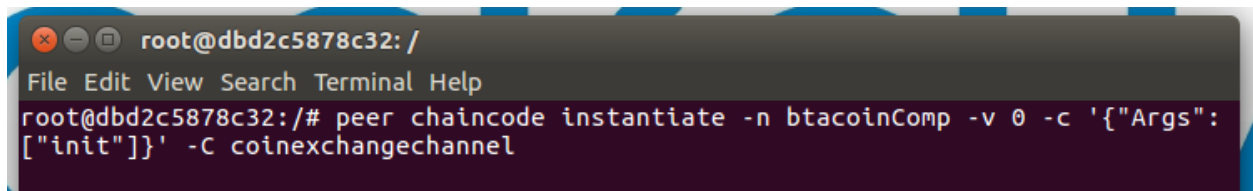
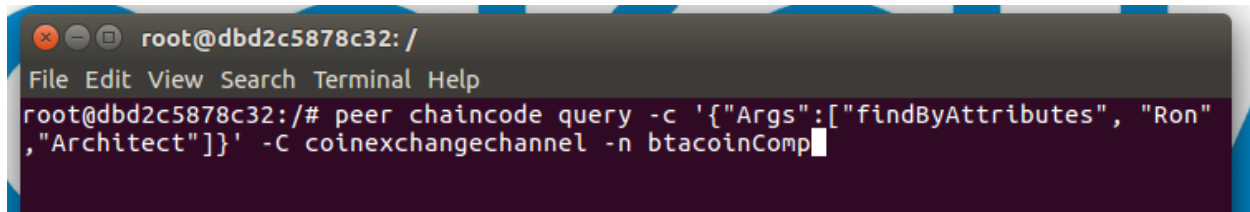
A terminal window with a dark background and light text. The prompt is 'root@dbd2c5878c32: /'. The command 'peer chaincode instantiate -n btacoinComp -v 0 -c '{"Args":["init"]}' -C coinexchangechannel' is entered and executed. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

Figure 85 - Snippet 4.16

Testing the New Chaincode

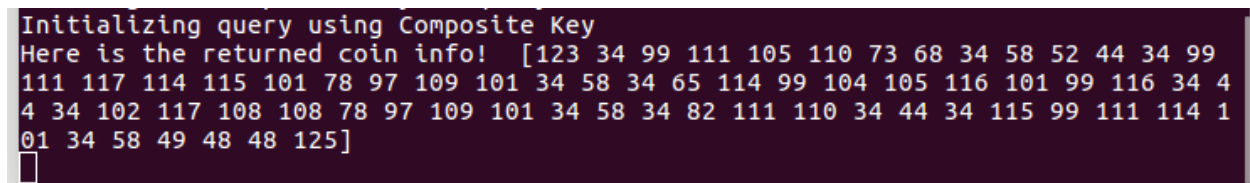
Ensure the chaincode is working properly by running the following query. Note the results returned in the first terminal window.

```
peer chaincode query -c '{"Args":["findByAttributes",  
"Ron","Architect"]}' -C coinexchangechannel -n btacoinComp
```

A terminal window with a dark background and light text. The prompt is 'root@dbd2c5878c32: /'. The command 'peer chaincode query -c '{"Args":["findByAttributes", "Ron", "Architect"]}' -C coinexchangechannel -n btacoinComp' is entered and the cursor is at the end of the line.

```
root@dbd2c5878c32: /  
File Edit View Search Terminal Help  
root@dbd2c5878c32:/# peer chaincode query -c '{"Args":["findByAttributes", "Ron",  
"Architect"]}' -C coinexchangechannel -n btacoinComp
```

Figure 86 - Snippet 4.17

A terminal window showing the output of the query. The text is: 'Initializing query using Composite Key', 'Here is the returned coin info!', followed by a long list of numbers in square brackets: '[123 34 99 111 105 110 73 68 34 58 52 44 34 99 111 117 114 115 101 78 97 109 101 34 58 34 65 114 99 104 105 116 101 99 116 34 4 4 34 102 117 108 108 78 97 109 101 34 58 34 82 111 110 34 44 34 115 99 111 114 1 01 34 58 49 48 48 125]'.

```
Initializing query using Composite Key  
Here is the returned coin info! [123 34 99 111 105 110 73 68 34 58 52 44 34 99  
111 117 114 115 101 78 97 109 101 34 58 34 65 114 99 104 105 116 101 99 116 34 4  
4 34 102 117 108 108 78 97 109 101 34 58 34 82 111 110 34 44 34 115 99 111 114 1  
01 34 58 49 48 48 125]  
█
```

Figure 87 - The query result

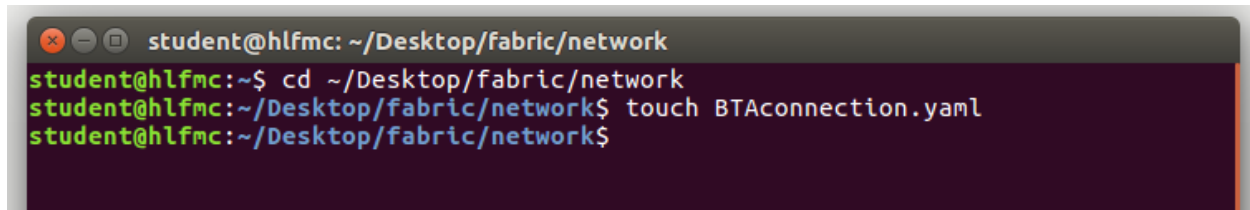
Lab 5 - Creating the Connection Profile

Getting Started

In this lab you will walk through a simple but integral series of configuration steps which allow a client to gather network information. In Fabric, this functionality is enabled by a connection profile. This connection profile will be imported and parsed as an object that our gateway will be able to use to gather information about the network. The connection profile will be referenced in the background by clients to receive the dynamic information.

Begin by opening a terminal window and running the commands below.

```
cd ~/Desktop/fabric/network  
touch BTAconnection.yaml
```

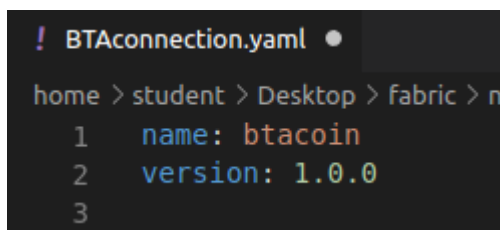


```
student@hlfmtc: ~/Desktop/fabric/network  
student@hlfmtc:~$ cd ~/Desktop/fabric/network  
student@hlfmtc:~/Desktop/fabric/network$ touch BTAconnection.yaml  
student@hlfmtc:~/Desktop/fabric/network$
```

Figure 88 - Snippet 5.1

Open up the newly created *BTAconnection.yaml* file for editing in Visual Studio Code. Add the following code.

```
name: btacoin  
version: 1.0.0
```



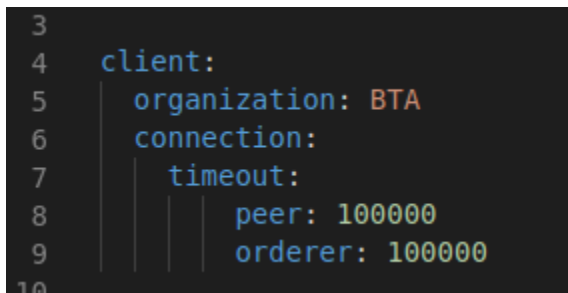
```
! BTAconnection.yaml •  
home > student > Desktop > fabric > n  
1 name: btacoin  
2 version: 1.0.0  
3
```

Figure 89 - Snippet 5.2

The *Client* Section

Next, define the parameters for the calling client application. The Org and timeout limits will be specified.

```
client:
  organization: BTA
  connection:
    timeout:
      peer: 100000
      orderer: 100000
```



```
3
4  client:
5    organization: BTA
6    connection:
7      timeout:
8        peer: 100000
9        orderer: 100000
10
```

Figure 90 - Snippet 5.3

The *Channels* Section

With the code below, all channels will be defined along with the nodes that belong to them.

```
channels:
  btamembersonly:
    orderers:
      - Devorderer.btacoin.com
    peers:
      Andy.BTA.btacoin.com: []
  coinexchangechannel:
    orderers:
      - Devorderer.btacoin.com
    peers:
      Andy.BTA.btacoin.com: []
      Ken.courseParticipants.btacoin.com: []
```



```
10
11 channels:
12   btamembersonly:
13     orderers:
14       - Devorderer.btacoin.com
15     peers:
16       Andy.BTA.btacoin.com: []
17   coinexchangechannel:
18     orderers:
19       - Devorderer.btacoin.com
20     peers:
21       Andy.BTA.btacoin.com: []
22       Ken.courseParticipants.btacoin.com: []
23
```

Figure 91 - Snippet 5.4

The *Organizations* Section

The code below provides a breakdown of each organization and their details.

```
organizations:
```

```
  BTA:
```

```
    mspid: BTAMSP
```

```
    peers:
```

```
      - Andy.BTA.btacoin.com
```

```
  certificateAuthorities:
```

```
    - btaCA.btacoin.com
```

```
  courseParticipants:
```

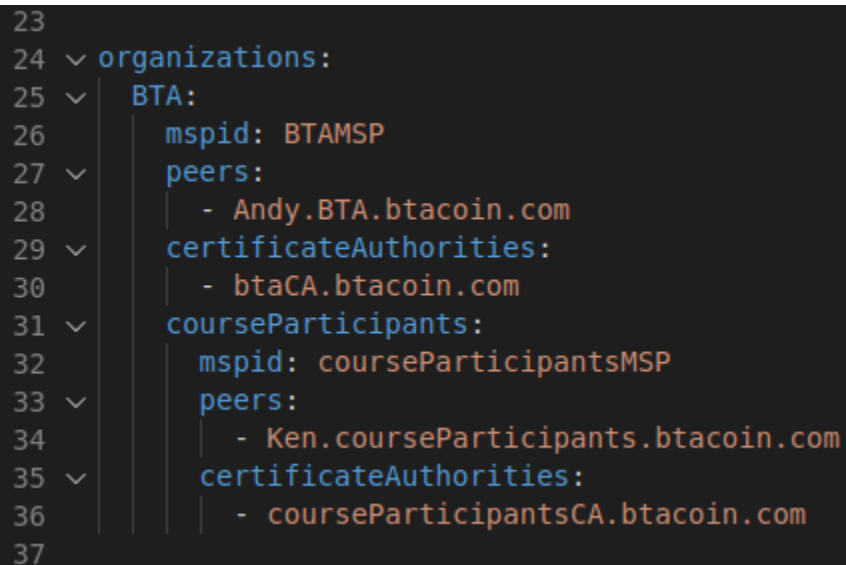
```
    mspid: courseParticipantsMSP
```

```
    peers:
```

```
      - Ken.courseParticipants.btacoin.com
```

```
  certificateAuthorities:
```

```
    - courseParticipantsCA.btacoin.com
```



```
23
24  v organizations:
25  v    BTA:
26      mspid: BTAMSP
27  v    peers:
28      - Andy.BTA.btacoin.com
29  v    certificateAuthorities:
30      - btaCA.btacoin.com
31  v    courseParticipants:
32      mspid: courseParticipantsMSP
33  v    peers:
34      - Ken.courseParticipants.btacoin.com
35  v    certificateAuthorities:
36      - courseParticipantsCA.btacoin.com
37
```

Figure 92 - Snippet 5.5

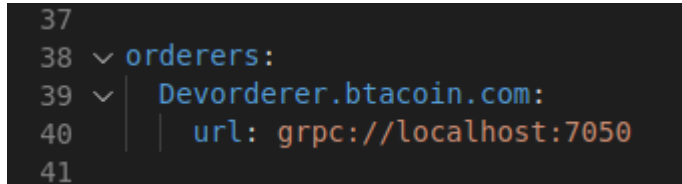
The *Orderers* Section

The orderers and their grpcs addresses are defined using the code below.

orderers:

Devorderer.btacoin.com:

url: grpc://localhost:7050



```
37
38 ∨ orderers:
39 ∨ | Devorderer.btacoin.com:
40 | | url: grpc://localhost:7050
41
```

Figure 93 - Snippet 5.6

The *Peers* Section

In the *peers* section a listing of all the peers, their URLs, and event addresses are defined.

peers:

Andy.BTA.btacoin.com:

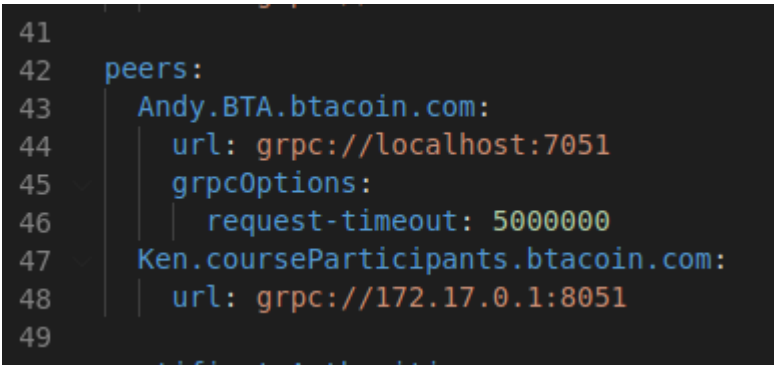
url: grpc://localhost:7051

grpcOptions:

request-timeout: 5000000

Ken.courseParticipants.btacoin.com:

url: grpc://172.17.0.1:8051



```
41
42 peers:
43   Andy.BTA.btacoin.com:
44     url: grpc://localhost:7051
45     grpcOptions:
46       request-timeout: 5000000
47   Ken.courseParticipants.btacoin.com:
48     url: grpc://172.17.0.1:8051
49
```

Figure 94 - Snippet 5.7

The *CertificateAuthorities* Section

Finally, all network CAs and their associated details must be listed.

```
certificateAuthorities:
```

```
  btaCA.btacoin.com:
```

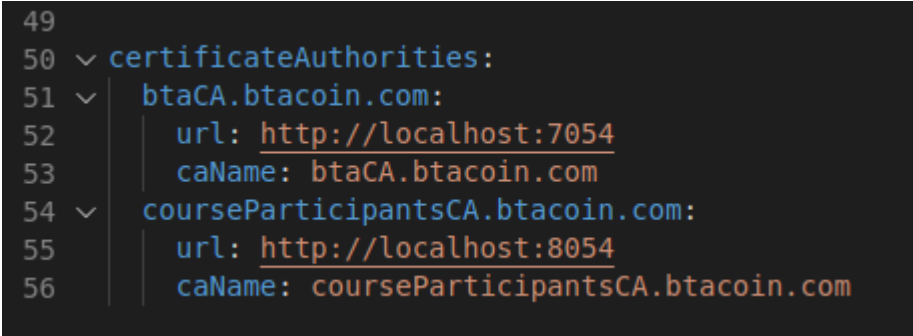
```
    url: http://localhost:7054
```

```
    caName: btaCA.btacoin.com
```

```
  courseParticipantsCA.btacoin.com:
```

```
    url: http://localhost:8054
```

```
    caName: courseParticipantsCA.btacoin.com
```



```
49
50  ✓ certificateAuthorities:
51  ✓   btaCA.btacoin.com:
52      url: http://localhost:7054
53      caName: btaCA.btacoin.com
54  ✓   courseParticipantsCA.btacoin.com:
55      url: http://localhost:8054
56      caName: courseParticipantsCA.btacoin.com
```

Figure 95 - Snippet 5.8

Lab 6: Client Application Creation & Identity

Create two files. One for us to use to set up and create transaction proposals in and the other to handle all of our identity items in.

```
mkdir client && cd client
mkdir src && cd src
mkdir fabricNetwork && cd fabricNetwork
mkdir wallet
touch identity.js
```

As with all node projects, we must initialize our package.json file using **init** command. (You can follow the below values and skip through the rest).

```
cd ..
npm init
```

```
[package name: (client)
|version: (1.0.0)
|description: For the btacoin network
|entry point: (identity.js) invoke.js
|
|_
```

Now we must save our dependencies to our project folder.

```
npm install fabric-network --save
npm install fabric-ca-client --save
```

Please open the **identity.js** file in your code editor. The first thing we must do is require in our necessary modules. We will first require in a couple packages on the fabric-network module as well.

```
const { FileSystemWallet, Gateway, x509WalletMixin } =
require('fabric-network');
```

```
const FabricCA = require('fabric-ca-client');
```

We will be using a few standard node packages such as Filesystem (to interact with our Wallet) and Path.

```
const fs = require('fs');  
const yaml = require('js-yaml')  
const path = require('path');
```

The basis of how this will operate is we create a new instance of our wallet, make sure there are no identity conflicts, instantiate our gateway.

Next let's setup the pathing for us to access our cryptographic assets.

```
const BTAconnectionProfile = path.join(__dirname,  
'../../../../../network/', 'BTAconnection.yaml');
```

Now we can load in that profile using the fs module, and load it into the file.

```
fs.readFileSync(BTAconnectionProfile, 'utf-8')
```

Now we can wrap the above statement in a safeLoad function, that handles correctly importing the YAML file and a parsing it as an object.

```
const connection =  
yaml.safeLoad(fs.readFileSync(BTAconnectionProfile, 'utf-  
8'));
```

Let's add the following line at the bottom

```
console.log(connection);
```

This is so we can see information about how the connection profile is used and referenced throughout our client. Now we can let's run the file from the command line. In your command prompt, run:

```
node identity.js
```

Now that you've seen the connection profile object, we can remove the `console.log` line and move on.

Last for pathing, we can setup the Path we want for our Wallet to hold our keys in.

```
const fswPath = path.join(process.cwd(), "/wallet");
```

Bootstrapping the Administrators

Please create the function `Adminwallet()` so we can setup the creation of Admin Credentials.

```
async function AdminWallet() {  
}
```

Inside of this asynchronous function, let's include a try/catch block. Your function should look like this:

```
async function AdminWallet() {  
  try{  
    } catch(error) {
```



```

    }
}

```

Now that we have our Pathing for where our wallet should live set up and confirmed it is correct, inside of the AdminWallet function, our first line can be to create a new instance of our FileSystemWallet, we can shorten it up to an acronym of fsw (using our wallet path.

```

const fsw = new FileSystemWallet(fswPath)
console.log('New wallet instance created at: ', fsw)

```

Now that we have our wallet created and set up, we can begin with the Certificate Authority setup for enrollment of Admin. First we will reference our connection profile to:

```

const AddressForCA =
connection.certificateAuthorities['btaCA.btacoin.com'].url;

```

Next, we will create and instantiate a new instance of the Fabric CA Client and connection to the BTACA.

```

const BTACA = new FabricCA(AddressForCA)

```

Now that we've created and instantiated a new CA Client Connection, we can move to performing an initial enrollment of the admin. We can use a method on the ca client called enroll, which takes the enrollment and secret.

```

const crypto = await BTACA.enroll({enrollmentID: 'btaCA',
enrollmentSecret: 'SimplePassword' });

```

Now that we have the crypto assets generated and returned to us, we can finally use all of those to create a wallet identity object.

```
const walletObject =  
X509WalletMixin.createIdentity('BTAMSP',  
crypto.certificate, crypto.key.toBytes());
```

Moving on, we can finally import the identity we just enrolled and created into our wallet.

```
await fsw.import("BTAAdmin", walletObject);
```

Now that we've finally added our Administrator to our wallet. Let's list out all our identities in the wallet to see it. (Obviously, this isn't dynamic because it's only calling the first identity, but since our Admin is the only one, it's ok for now)

```
walletIdentity = wallet.list()[0];
```

```
console.log("Here is the Admin identity:  
${walletIdentity}")
```

```
0b065e2fd5c53313a73530743d30d752cd1178108691217bac4bab73e50af7c6-priv BTAAdmin  
0b065e2fd5c53313a73530743d30d752cd1178108691217bac4bab73e50af7c6-pub
```

Now that we've accomplished the core of our AdminWallet() function, let's do some error handling. Just at the top of our function we can run a check to see if we already have the admin already created.

```
const check = await wallet.exists('admin');  
  
if (check) {  
    console.log("Checked For the admin and found  
one...Exiting");  
    return  
}
```

And we can head to our catch block at the bottom and inside of it add:

```
console.log("Something went wrong!", error)
```

Enrolling the Identity for our Client

Next we will be creating the `EnrollClient()` function which uses our admin to register and enroll the client with the CA. This function will be run behind our `AdminWallet()` Command.

Let's begin by defining the function outline with the try catch block inside like earlier.

```
async function EnrollClient() {  
  try{  
    } catch(error) {  
    }  
}
```

Inside of the try block please add a check that looks to make sure the admin identity still exists, and if it doesn't, then we will run the `AdminWallet()` function.

```
const check = await wallet.exists('BTAadmin')
```

```
If (check==true{  
  console.log("No administrative identity  
found...running AdminWallet() ")  
}
```

Now the next steps will look a bit familiar to the AdminWallet function definition before. We will create a new instance of the CA client and connect to it then register/enroll our client and store it locally in our wallet.

First we can handle the Certificate Authority Setup.

```
const AddressForCA =  
connection.certificateAuthorities[ 'btaCA.btacoin.com' ].url;
```

(Since we have used this AddressForCA variable twice already, a good is moving outside of the function and moving it toward the top amongst the other variables so both our AdminWallet and EnrollClient functions can access it.)

Now we will instantiate that Fabric CA.

```
const BTACA = new FabricCA(AddressForCA)
```

Next, we can begin the creation of our identity. We will first use the register method on the CA in order to register the identity with the CA.

```
const registration = await BTACA.register(affiliation:  
'BTA', enrollmentID: 'user1', role: 'client'),  
'BTAadmin');
```

Now that we have the registration information, we can enroll and create crypto assets.

```
const ClientCrypto = await BTACA.enroll({enrollmentID:  
'user1', enrollmentSecret: registration});
```

Last, we can create and store the identity in our wallet.

```
const ClientID = X509WalletMixin.createIdentity('BTAMSP',
ClientCrypto.certificate, ClientCrypto.key.toBytes());
```

```
wallet.import('user1', ClientID)
```

Now that we are finished with that, we can fill in our catch block with an error message.

```
    console.error("Unable to regisgter "user1", here's
why\n + error);
```

So that we can automatically call these functions, at the bottom of the file, please add

```
AdminWallet()
EnrollClient()
```

And that's it! We can test it by saving your work, returning to the command prompt and running

```
node identity.js
```

Let's confirm some changes occurred in our wallet.

```
ls wallet
```



Lab 7: Client Application: Transactions & Queries

```
cd FABRIC_HOME/client/src/fabricNetwork
touch invoke.js
```

In your favorite code editor, please open the invoke.js file. Similar to the identity.js file we will begin by importing our modules, and the setting up the pathing.

```
const {
  FileSystemWallet,
  Gateway,
} = require("fabric-network");
const fs = require("fs");
const path = require("path");
const yaml = require('js-yaml')
```

Now we can set the path to the filesystem wallet.

```
const fswPath = path.resolve(__dirname, "../wallet");
```

We will import and setup our connection profile as we've done before.

```
const BTAconnectionProfile = path.join(__dirname,
  '../.../network/', 'BTAconnection.yaml');
```

```
const connection =  
yaml.safeLoad(fs.readFileSync(BTAconnectionProfile, 'utf-  
8'));
```

Similar to how we've been doing everything in our client, we will define an invoke function that manages the setup and invocation of a smart contract all in one sequential function.

```
async function chaincodeOperations() {  
    try{  
  
        } catch (error){  
            console.error("Invocation could not be  
completed... ", error)  
        }  
  
    }  
  
    chaincodeOperations();
```

From this point on, we will add all of our functions inside of this.

First, we must create a new instance of the FileSystemWallet passing in our path to the wallet.

```
var usrwallet = new FileSystemWallet(fswPath);
```

Gateway

Assuming the identity check passed successfully, we can move on to creating a connection between the client, using the gateway class.

```
const gwy = new gateway();
```

Gateway has a method that allows us to connect to a peer from the next to gather network information. All we have to do is specify the connection profile we'd like to use, and the configuration options for the connection.

```
await gwy.connect(BTAconnectionProfile,{ wallet:
usrwallet, identity: 'BTAadmin'});
```

In order to access our smart contract we must first grab information about the specific channel it's deployed on.

```
const coinexchange = await
gwy.getNetwork('coinexchangechannel');
```

Getting the contract

Now that we've setup our gateway, we can call for an instance of our smart contract(btacoin).

```
const btacoin = await
coinexchange.getContract("btacoin");
```

Now that our smart contract is setup we can setup a handler to check what the value of our function name is. If its "invoke" then we will call our Transaction Submission handler. For everything else it will default to query.

```
if (txType == invoke) {

} else {
```



```
}
```

Transaction Submission

For exemplary purposes, we will hardcode the submission type to be for coin creation only. We should put these lines inside of the invoke code block (of the if statement)

```
await btacoin.submitTransaction('createCoin', '7',  
'Architect');
```

If you look at the Command Line window that is running and logging out your chaincode operations, you should see a new coin created with the value of 7.

Querying

As we are aware from the lectures, we can query using a simple command as well. Now you can make the call to query passing along the function name and parameter and store the result in a variable.

```
let answer = await  
contract.evaluateTransaction("findByID", 1);
```

```
console.log( "The response is: ", answer)
```

