DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.

CoScienze
Associazione

DESCRIZIONE CHAINCODE GO

Descrivere cosa realizza il seguente chaincode in Go: //traccia 13/01/2021

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

type HeroesServiceChaincode struct {
}
func (t *HeroesServiceChaincode) Init(stub shim.ChaincodeStubInterface)
pb.Response {
    function, args := stub.GetFunctionAndParameters()
    if function != "init" {
        return shim.Error("Unknown function call")
    }
    err := stub.PutState("hello", []byte("world"))
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(nil)
}

func (t *HeroesServiceChaincode)Invoke(stub shim.ChaincodeStubInterface)
pb.Response {
    function, args := stub.GetFunctionAndParameters()
    if function != "invoke" {
        return shim.Error("Unknown function call")
    }
    if len(args) < 1 {
        return shim.Error("The number of arguments is insufficient.")
    }
    if args[0] == "query" {
        return t.query(stub, args)
    }
    if args[0] == "invoke" {
        return t.invoke(stub, args)
    }
    return shim.Error("Unknown action, check the first argument")
}

func (t *HeroesServiceChaincode) query(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
    fmt.Println("########### HeroesServiceChaincode query ###########")
    if len(args) < 2 {
        return shim.Error("The number of arguments is insufficient.")
    }
    if args[1] == "hello" {
        state, err := stub.GetState("hello")
        if err != nil {
            return shim.Error("Failed to get state of hello")
        }
        return shim.Success(state)
    }
    return shim.Error("Unknown query action, check the second argument.")
}

func (t *HeroesServiceChaincode) invoke(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
    if len(args) < 2 {
```

```
        return shim.Error("The number of arguments is insufficient.")
    }
    if args[1] == "hello" && len(args) == 3 {
        err := stub.PutState("hello", []byte(args[2]))
        if err != nil {
            return shim.Error("Failed to update state of hello")
        }
        err = stub.SetEvent("eventInvoke", []byte{})
        if err != nil {
            return shim.Error(err.Error())
        }
        return shim.Success(nil)
    }
    return shim.Error("Unknown invoke action, check the second argument.")
}

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

RISPOSTA

Dopo aver importato le varie librerie necessarie al contratto viene definita la struttura HeroesService chaincode del contratto.

La funzione init inizializza la blockchain inserendo elementi in essa e ritornando oggetti di tipo peer. Response.

poi vengono passati la funzione e i parametri richiesti con GetFunctionAndParameters() e fa un controllo per verificare se function è diverso da init, in tal caso stampa un messaggio di errore.

Viene fatta una Putstate per aggiornare la blockchain e quindi inserire la chiave e l'oggetto json ovvero un array di byte e poi viene fatto il controllo sull'errore

La funzione invoke è una funzione di invocazione, ovvero specifica le azioni che vengono eseguite dal contratto e anche in tal caso a function e args viene passato la funzione e i parametri con la funzione getFunctionAndParameters() e viene fatto un controllo per verificare la funzione invoke e in caso sia diverso stampa un messaggio di errore. Successivamente viene fatto un controllo sul numero di argomenti e subito dopo il controllo sulla funzione query e invoke, in tal caso passando la funzione e i parametri stub e args altrimenti ritorna un errore.

La funzione query ha l'interfaccia chaincodestubinterface ed essenzialmente permette al peer di accedere in modo indiretto alla blockchain. Viene preso il primo argomento passato, vengono fatti diversi controlli e restituisce le query fatta altrimenti da errore.

La funzione invoke essenzialmente consiste nel verificare se la lunghezza dei parametri è minore di 2 in tal caso rimanda a un messaggio di errore. Poi viene controllato che il parametro in posizione 2 e il numero di argomenti sia uguale a 3, in tal caso viene fatto un assegnamento dello state, cioè dello stato passando la chiave "hello" e l'oggetto json nella blockchain contenuta nel parametro in terza posizione e Notifica agli ascoltatori che un evento "eventInvoke" è stato eseguito. Infine c'è il controllo sull'errore per verificare la correttezza della procedura.

In main invece con start viene deployato il contratto e verificato se andato a buon fine.

Descrivere cosa realizza il seguente chaincode in Go: //traccia 27/01/2021

```
package main
import (
```

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

type SimpleChaincode struct {
}

func (t *SimpleChaincode)Init(stub shim.ChaincodeStubInterface) pb.Response {
            fmt.Println("ex02 Init")
            _, args := stub.GetFunctionAndParameters()
            var A, B string
            var Aval, Bval int
            var err error
            if len(args) != 4 {
                        return shim.Error("Incorrect number of arguments.
Expecting 4")
            }
            A = args[0]
            Aval, err = strconv.Atoi(args[1])
            if err != nil {
                        return shim.Error("Expecting integer value for asset
holding")
            }
            B = args[2]
            Bval, err = strconv.Atoi(args[3])
            if err != nil {
                        return shim.Error("Expecting integer value for asset
holding")
            }
            fmt.Printf("Aval = %d, Bval = %d\
", Aval, Bval)

            err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
            if err != nil {
                        return shim.Error(err.Error())
            }

            err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
            if err != nil {
                        return shim.Error(err.Error())
            }

            return shim.Success(nil)
}

func (t *SimpleChaincode)Invoke(stub shim.ChaincodeStubInterface) pb.Response {
            fmt.Println("ex02 Invoke")
            function, args := stub.GetFunctionAndParameters()
            if function == "invoke" {
                        return t.invoke(stub, args)
            } else if function == "delete" {
                        return t.delete(stub, args)
            } else if function == "query" {
                        return t.query(stub, args)
            }

            return shim.Error("Invalid invoke function name. Expecting
\\"invoke\\" \\"delete\\" \\"query\\"")
}
 func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
```

```
            var A, B string
            var Aval, Bval int
            var X int
            var err error
             if len(args) != 3 {
                        return shim.Error("Incorrect number of arguments.
Expecting 3")
            }

            A = args[0]
            B = args[1]
             Avalbytes, err := stub.GetState(A)
            if err != nil {
                        return shim.Error("Failed to get state")
            }
            if Avalbytes == nil {
                        return shim.Error("Entity not found")
            }
            Aval, _ = strconv.Atoi(string(Avalbytes))
             Bvalbytes, err := stub.GetState(B)
            if err != nil {
                        return shim.Error("Failed to get state")
            }
            if Bvalbytes == nil {
                        return shim.Error("Entity not found")
            }
            Bval, _ = strconv.Atoi(string(Bvalbytes))
             X, err = strconv.Atoi(args[2])
            if err != nil {
                        return shim.Error("Invalid transaction amount, expecting
a integer value")
            }
            Aval = Aval - X
            Bval = Bval + X
            fmt.Printf("Aval = %d, Bval = %d\
", Aval, Bval)
             err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
            if err != nil {
                        return shim.Error(err.Error())
            }
             err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
            if err != nil {
                        return shim.Error(err.Error())
            }
             return shim.Success(nil)
}
        func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
            if len(args) != 1 {
                 return shim.Error("Incorrect number of arguments. Expecting 1")
            }
            A := args[0]
            err := stub.DelState(A)
            if err != nil {
                        return shim.Error("Failed to delete state")
            }
             return shim.Success(nil)
}

func (t *SimpleChaincode)query(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
            var A string
            var err error
```

```
             if len(args) != 1 {
                        return shim.Error("Incorrect number of arguments.
Expecting name of the person to query")
             }
             A = args[0]
             Avalbytes, err := stub.GetState(A)
             if err != nil {
                        jsonResp := "{\\"Error\\":\\"Failed to get state for " +
A + "\\"}"
                        return shim.Error(jsonResp)
             }
              if Avalbytes == nil {
                        jsonResp := "{\\"Error\\":\\"Nil amount for " + A +
"\\"}"
                        return shim.Error(jsonResp)
             }
              jsonResp := "{\\"Name\\":\\"" + A + "\\",\\"Amount\\":\\"" +
string(Avalbytes) + "\\"}"
             fmt.Printf("Query Response:%s\", jsonResp)
             return shim.Success(Avalbytes)
}

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

RISPOSTA

Dopo aver importato le varie librerie necessarie al contratto viene definita la struttura Simplechaincode del contratto.

- La funzione init inizializza la blockchain inserendo elementi in essa e ritornando oggetti di tipo peer. Response. Dopo aver stampato un messaggio e passato a _ e args la funzione e i parametri richiesti con GetFunctionAndParameters() fa un controllo per verificare se il numero di parametri è diverso da 4, ritornando un messaggio di errore in tal caso. Dopo aver passato ad A il primo argomento viene fatta una conversione del secondo argomento e passato ad Aval viene verificato dal nil, lo stesso viene fatto per B a cui viene passato il terzo argomento e Bval la conversione nel quarto. Fatto questo viene fatta una Putstate per aggiornare la blockchain e quindi inserire la chiave e l'oggetto json ovvero un array di byte. Se err è diverso da nil stampa errore altrimenti ritorna success.
- La funzione invoke è una funzione di invocazione, cioè specifica le azioni che vengono eseguite dal contratto, anche in tal caso a function e args viene passato la funzione e i parametri con la funzione getFunctionAndParameters() e viene fatto un controllo per verificare la funzione invoke, delete, query, in tal caso passando la funzione e i parametri stub e args altrimenti ritorna un errore.
- La funzione invoke essenzialmente consiste nel verificare se la lunghezza dei parametri è diversa da 3 in tal caso rimanda a un messaggio di errore. Viene fatto un assegnamento dello state, cioè lo stato. Vengono assegnati i parametri alle variabili. Fatto questo effettua la putstate per inserire la chiave e oggetto json nella blockchain e ulteriori controlli per verificare la correttezza della procedura.
- La funzione delete permette di verificare se il numero di argomenti è diverso da uno, viene assegnato il primo argomento e viene rimosso lo state, in tal caso viene fatto un controllo.
- La funzione query ha l'interfaccia chaincodestubinterface ed essenzialmente permette al peer di accedere in modo indiretto alla blockchain. Viene preso il primo argomento passato, viene preso il primo argomento passato, viene fatto diversi controlli e restituisce le query fatta altrimenti errore.
- In main invece con start viene deployato il contratto e verificato se andato a buon fine.

Descrivere cosa realizza il seguente chaincode in Go:

```
package main
  import (
  " fmt "
  " github.com/hyperledger/fabric/core/chaincode/shim "
  " github.com/hyperledger/fabric/protos/peer "
)
type SampleChaincode struct {
}

func (t * SampleChaincode )Init(stub shim.ChaincodeStubInterface ) peer.Response
{
  args := stub.GetStringArgs()
  if len (args) != 2 {
    return shim.Error ("Incorrect arguments. Expecting a key and a value")
  }
  err := stub.PutState(args[0], [] byte (args[1]))
  if err != nil {
    return shim.Error (fmt.Sprintf("Failed to create asset: %s", args [0]))
  }
    return shim.Success(nil)
  }

func (t * SampleChaincode)Invoke(stub shim.ChaincodeStubInterface )peer.Response
{
  fn , args := stub.GetFunctionAndParameters ()
  var result string
  var err error
  if fn == "set" {
    result, err = set(stub, args )
  } else {
    result, err = get(stub, args )
  }
  if err != nil {
    return shim.Error(err.Error())
  }
  return shim.Success([] byte (result))
}

func set (stub shim.ChaincodeStubInterface, args[] string )( string, error ) {
  if len (args) != 2 {
    return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
  }
  err := stub.PutState(args[0], [] byte (args[1]))
  if err != nil {
    return "", fmt.Errorf("Failed to set asset : %s", args[0])
  }
  return args[1], nil
}

func get(stub shim.ChaincodeStubInterface, args[] string )( string, error ){
  if len(args) != 1 {
    return "", fmt.Errorf("Incorrect arguments.Expecting a key")
  }
  value, err := stub.GetState(args[0])
  if err != nil {
    return"", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
  }
  if value == nil{
    return "", fmt.Errorf("Asset not found : %s", args [0])
```

```
  }
  return string(value), nil
}

func main () {
  err := shim.Start(new(SampleChaincode ))
  if err != nil {
    fmt.Println("Could not start SampleChaincode")
  } else {
    fmt.Println("SampleChaincode successfully started")
  }
}
```

RISPOSTA

Dopo aver importato le librerie viene inizializzata una struttura di tipo SimpleChaincode nel modo seguente: Init, Invoke, e ChaincodeStubInterface. Per poter accedere ad Init ed Invoke è necessario fare uso dell'interfaccia ChaincodeStubInterface. La funzione init è una funzione di inizializzazione della blockchain e restituisce oggetti di tipo peer.Responce.

Poi vengono passati gli argomenti di tipo stringa, ovvero la key e un valore e viene effettuato un controllo sul numero di argomenti, che se diverso da 2 da errore. Poi viene aggiornato lo stato della blockchain passando la key e l'oggetto json rappresentato dall'array di byte e successivamente viene fatto un controllo sull'errore dove se diverso da nil da errore, altrimenti success.

- La funzione invoke è una funzione di invocazione e viene eseguita in risposta a colui che deploya il contratto, cioè specifica le azioni che vengono eseguite dal contratto, poi abbiamo l'assegnazione delle funzioni e dei parametri attraverso getFunctionAndParameters. Si fa il controllo per vedere se la funzione è una "set" o una "get". Dopo i controlli, viene restituito "success", con il risultato in bytes, se non ci sono stati errori, "Error" altrimenti.
- Nella funzione set si fa il controllo sul numero di parametri passati, se diverso da 2 restituisce errore, altrimenti aggiorna lo stato della blockchain e viene fatto un controllo sull'errore che se diverso da nil restituisce errore, altrimenti restituisce il secondo argomento e nil.
- nella funzione get si fa il controllo sul numero di parametri passati, se diverso da 1 restituisce errore, altrimenti effettua la lettura dello stato della blockchain con getState. Viene effettuato il controllo sull'errore che se diverso da nil restituisce errore, poi viene fatto il controllo sul valore che se uguale a nil restituisce errore, altrimenti restituisce il valore e nil.
- In main invece con start viene deployato il contratto e verificato se andato a buon fine.