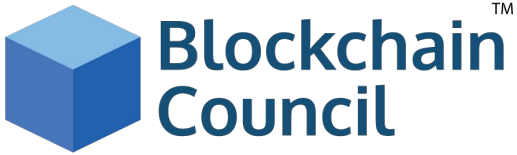

Certified Hyperledger Expert

Structure of Chaincode

Structure of Chaincode

The three main functions that makes up the structure of Chaincode are

- Init
- Invoke
- Query

All three functions have the same prototype; they take in a 'stub', which read from and write to the ledger, a function name, and an array of strings. The main difference between the functions is when they are called.

Shim provides APIs for the chaincode to access its state variables, transaction context and call other chaincodes. Shim contains the generic means for chaincode and validating peer to communicate with each other.

Init

Init is called when you first deploy your chaincode.

Init function is used to do any initialization your chaincode needs. For example, we use Init to configure the initial state of a single key/value pair on the ledger.

Init function takes up the key and value for chaincode as arguments where first argument is taken as the key and the other arguments as values. Let’s see an example for Init with “Hello World”

Init Example

```
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface, function
string, args []string) ([]byte, error) {
    if len(args) != 1 {
        return nil, errors.New("Incorrect number of arguments. Expecting 1")
    }

    err := stub.PutState("hello_world", []byte(args[0]))
    if err != nil {
        return nil, err
    }

    return nil, nil
}
```

Invoke

Invoke is called when you want to call chaincode functions to do real work.

Invocations will be captured as a transactions, which get grouped into blocks on the chain. When you need to update the ledger, you will do so by invoking your chaincode.

Invoke receives a function and an array of arguments. Based on what function was passed in through the function parameter in the invoke request, Invoke will either call a helper function or return an error.

Let’s see an example where invoke function calls a write function over the ledger.

Invoke Example

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface, function
string, args []string) ([]byte, error) {
    fmt.Println("invoke is running " + function)

    // Handle different functions
    if function == "init" {
        return t.Init(stub, "init", args)
    } else if function == "write" {
        return t.write(stub, args)
    }
    fmt.Println("invoke did not find func: " + function)

    return nil, errors.New("Received unknown function invocation: " +
function)
}
```

Invoke Example

```
func (t *SimpleChaincode) write(stub shim.ChaincodeStubInterface, args
[]string) ([]byte, error) {
    var key, value string
    var err error
    fmt.Println("running write()")

    if len(args) != 2 {
        return nil, errors.New("Incorrect number of arguments. Expecting 2.
name of the key and value to set")
    }
```

Invoke Example

```
    key = args[0]                                   //rename for fun
    value = args[1]
    err = stub.PutState(key, []byte(value))  //write the variable into the
chaincode state
    if err != nil {
        return nil, err
    }
    return nil, nil
}
```

Query

Query is called whenever you query your chaincode's state.

Queries do not result in blocks being added to the chain, and you cannot use functions like PutState inside of Query or any helper functions it calls. You will use Query to read the value of your chaincode state's key/value pairs.

Let’s see an example of query function that calls a read function to read the data from the Ledger.

This read function shown here is using GetState. While PutState allows you to set a key/value pair, GetState lets you read the value for a previously written key. A single argument used by this function which takes in the key for the value that should be retrieved. Next, this function returns the value as an array of bytes back to Query, who in turn sends it back to the REST handler.

Query Example

```
func (t *SimpleChaincode) Query(stub shim.ChaincodeStubInterface, function
string, args []string) ([]byte, error) {
    fmt.Println("query is running " + function)

    // Handle different functions
    if function == "read" {                                  //read a variable
        return t.read(stub, args)
    }
    fmt.Println("query did not find func: " + function)

    return nil, errors.New("Received unknown function query: " + function)
}
```

Query Example

```
func (t *SimpleChaincode) read(stub shim.ChaincodeStubInterface, args
[]string) ([]byte, error) {
    var key, jsonResp string
    var err error

    if len(args) != 1 {
        return nil, errors.New("Incorrect number of arguments. Expecting name
of the key to query")
    }
```

Query Example

```
    key = args[0]
    valAsbytes, err := stub.GetState(key)
    if err != nil {
        jsonResp = "{\"Error\":\"Failed to get state for " + key + "\"}"
        return nil, errors.New(jsonResp)
    }

    return valAsbytes, nil
}
```

Finally - Calling the Chaincode

In the end, you need to create a short main function that will execute when each peer deploys their instance of the chaincode. It just calls shim.Start(), which sets up the communication between chaincode and the peer that deployed it. You don't need to add any code for this function. Function is described like this:

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

THANK YOU!

Any questions?

You can mail us at

hello@blockchain-council.org