Chaincode for Go developers, Part 1: Writing Blockchain chaincode in Go for Hyperledger Fabric v0.6

How to write chaincode for a blockchain network

Varun Ojha

March 06, 2017
(First published March 01, 2017)

Learn how to develop chaincode using Golang for a blockchain network based on Hyperledger Fabric v0.6. This deep-dive tutorial covers the fundamentals, such as the APIs for interacting with the Fabric, as well as advanced topics like data modeling, access control, and events. Abundant sample code demonstrates a home loan and purchase contract process on blockchain.

In this tutorial, learn how to develop chaincode using Golang for a blockchain network based on **Hyperledger Fabric v0.6**. I cover the fundamentals, such as the role of chaincode and the APIs for interacting with the underlying Fabric, as well as advanced topics like data modeling, access control, and events. Abundant code examples demonstrate a home loan and purchase contract process on blockchain. (See "Downloadable resources" at the end of this tutorial to download the entire sample chaincode.)

This tutorial is the first of a series; follow-on tutorials will cover how to unit test your chaincode and develop client applications that can invoke your deployed chaincode.

What is chaincode?

Chaincode, also called the ***smart contract***, is essentially the business logic that governs how the different entities or parties in a blockchain network interact or transact with each other. Simply put, the chaincode is the encapsulation of business network transactions in code. Invocations of the chaincode result in sets and gets of the ledger or world state.

At the time of publishing this tutorial, Hyperledger supports writing chaincode in Golang or the Java™ language, which eventually runs inside a docker container. Because chaincode support for Java is still in beta, I'll focus on Go in this tutorial.

Setting up your development environment

Follow the steps in the IBM Bluemix documentation starting at "Setting up the development environment." When you reach the section titled "Set up your development pipeline," stop there; you are now ready to start developing chaincode in Go.

Chaincode structure

Let's take a close look at chaincode structure. As mentioned, the sample chaincode in Listing 1 and throughout this tutorial, as well as the architecture discussed, strictly conform to the v0.6 preview of the Hyperledger Fabric.

Line 4 of Listing 1 imports the shim package into your chaincode. The ***shim package*** provides APIs that let your chaincode interact with the underlying blockchain network to access state variables, transaction context, caller certificates and attributes, and to invoke other chaincodes, among other operations.

Listing 1. Sample chaincode

```
package main

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

type SampleChaincode struct {
}

func (t *SampleChaincode) Init(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 return nil, nil
}
```

```
func (t *SampleChaincode) Query(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 return nil, nil
}

func (t *SampleChaincode) Invoke(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 return nil, nil
}

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

}
```

Main function

The starting point for any Go program is the main function, and hence it's used for bootstrapping/ starting the chaincode. When the peer deploys its instance of the chaincode, the main function gets executed.

As shown in **line 2** of Listing 2, the shim.Start(new(SampleChaincode)) line starts the chaincode and registers it with the peer. You can verify this locally by running the code in your development environment, which will produce the following error: [shim] CRIT : peer.address not configured, can't connect to peer.

Listing 2. Main()

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

}
```

The SampleChaincode is the struct that is required to implement the shim.Chaincode interface, which has three methods — Init, Query, and Invoke — for it to be considered a valid Chaincode type by the shim package. Let's look at each of the three methods.

Init method

The ***Init method*** is called when the chaincode is first deployed onto the blockchain network and will be executed by each peer that deploys its own instance of the chaincode. This method can be used for any tasks related to initialization, bootstrapping, or setup.

Listing 3. Init()

```
func (t *SampleChaincode) Init(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 return nil, nil
}
```

Query method

The ***Query method*** is invoked whenever any read/get/query operation needs to be performed on the blockchain state. Depending upon the complexity of the chaincode, this method can hold your read/get/query logic, or it could be outsourced to separate methods that can be invoked from within the Query method.

The Query method is not intended to change the state of the underlying blockchain, and hence does not run within a transactional context. If you try to modify the state of the blockchain within the Query method, an error will complain about the lack of a transactional context. Also, because this method is only for reading the state of the blockchain, its invocations are not recorded on the blockchain.

Listing 4. Query()

```
func (t *SampleChaincode) Query(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 return nil, nil
}
```

Invoke method

The ***Invoke method*** is invoked whenever the state of the blockchain is to be modified. Simply put, all create, update, and delete operations should be encapsulated within the Invoke method. Because this method will modify the state of the blockchain, the blockchain Fabric code will automatically create a transaction context inside which this method will get executed. All invocations of this method are recorded on the blockchain as transactions, which ultimately get written into blocks.

Listing 5. Invoke()

```
func (t *SampleChaincode) Invoke(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 return nil, nil
}
```

Data models in chaincode

The Hyperledger ledger consists of two parts:

1. ***World state***, which is stored in a key value store. This key value store is powered by the RocksDB. This key value store takes in a byte array as the value, which can be used to store a serialized JSON structure. Essentially this key value store can be used to store any custom data model/schema required by your smart contract to function.
2. ***Blockchain***, which consists of a series of blocks each containing a number of transactions. Each block contains the hash of the world state and is also linked to the previous block. Blockchain is append-only.

Listing 6 shows how to create custom data models/schemas. It defines data models required for a home loan application. The primary model is called **LoanApplication**, which in turn has primitive and complex data types, namely Personal and Financial Info.

Since our key value store stores data as JSON, these data models would eventually need to be converted into a JSON string. The annotation for each field, for example, **json:"id"** acts like metadata for the marshal/unmarshal API, which will use these annotations to map each field with its corresponding json string equivalent representation.

Listing 6. Code to create custom data models/schemas

```
//custom data models
type PersonalInfo struct {
 Firstname string `json:"firstname"`
 Lastname  string `json:"lastname"`
 DOB       string `json:"DOB"`
 Email     string `json:"email"`
 Mobile    string `json:"mobile"`
}

type FinancialInfo struct {
 MonthlySalary      int `json:"monthlySalary"`
 MonthlyRent        int `json:"monthlyRent"`
 OtherExpenditure   int `json:"otherExpenditure"`
 MonthlyLoanPayment int `json:"monthlyLoanPayment"`
}

type LoanApplication struct {
 ID                    string        `json:"id"`
 PropertyId            string        `json:"propertyId"`
 LandId                string        `json:"landId"`
 PermitId              string        `json:"permitId"`
 BuyerId               string        `json:"buyerId"`
 SalesContractId       string        `json:"salesContractId"`
 PersonalInfo          PersonalInfo  `json:"personalInfo"`
 FinancialInfo         FinancialInfo `json:"financialInfo"`
 Status                string        `json:"status"`
 RequestedAmount       int           `json:"requestedAmount"`
 FairMarketValue       int           `json:"fairMarketValue"`
 ApprovedAmount        int           `json:"approvedAmount"`
 ReviewerId            string        `json:"reviewerId"`
 LastModifiedDate      string        `json:"lastModifiedDate"`
}
```

Storing and retrieving data

The code in Listing 7 and Listing 8 shows how to store and fetch data from the ledger.

Storing data into the ledger

The CreateLoanApplication method on **line 1** of Listing 7 takes in two arguments. The first argument is the ChaincodeStubInterface, which has useful APIs to interact with the blockchain ledger, transaction context, caller certificates, etc. The second argument is a string array that can be used by the invoker of the method to pass in required arguments.

Lines 2-8 handle logging and validation of input arguments.

On **line 9**, the loan application Id value, which would be used as the key to store the actual loan application object, is retrieved.

On **line 10**, the actual loan application content is retrieved in the form of a JSON string. For example,

‘{"propertyId":"prop1","landId":"land1","permitId":"permit1","buyerId":"vojha24","personalInfo":
{"firstname":"Varun","lastname":"Ojha","dob":"dob","email":"varun@gmail.com","mobile":"99999999"},"fina
{"monthlySalary":10000,"otherExpenditure":0,"monthlyRent":1000,"monthlyLoanPayment":1000},"status":"
2:30pm"}’

Line 12 is where the **stub.PutState** method is invoked to store the loan application id and the actual loan application JSON content as a key value pair into the blockchain ledger. Note that the value being stored in the key value store must always be a byte array. Hence the loan application JSON string is first converted to a byte array before storing it into the ledger.

Listing 7. Code to store data into the ledger

```
func CreateLoanApplication(stub shim.ChaincodeStubInterface, args []string) ([]byte, error) {
 fmt.Println("Entering CreateLoanApplication")

 if len(args) < 2 {
  fmt.Println("Invalid number of args")
  return nil, errors.New("Expected at least two arguments for loan application creation")
 }

 var loanApplicationId = args[0]
 var loanApplicationInput = args[1]

 err := stub.PutState(loanApplicationId, []byte(loanApplicationInput))
 if err != nil {
  fmt.Println("Could not save loan application to ledger", err)
  return nil, err
 }

 fmt.Println("Successfully saved loan application")
 return nil, nil
}
```

Fetching data from the ledger

in Listing 8, the **GetLoanApplication** method on **line 1** takes in two arguments. The first argument is the **ChaincodeStubInterface**, which has useful APIs to interact with the blockchain ledger, transaction context, caller certificates, etc. The second argument is a string array that can be used by the invoker of the method to pass in required arguments.

Lines 2-7 handle logging and validation of input arguments.

On **line 9**, the loan application Id value, which would be used as the key to retrieve the actual loan application object from the ledger, is retrieved.

Line 10 is where the **stub.GetState** method is invoked to retrieve the loan application JSON content in the form of a byte array by passing in the **loanApplicationId** key. Note that the value being stored in the key value store must always be a byte array. Hence the loan application JSON string is first converted to a byte array before storing it into the ledger.

Listing 8. Code to fetch data from the ledger

```
func GetLoanApplication(stub shim.ChaincodeStubInterface, args []string) ([]byte, error) {
 fmt.Println("Entering GetLoanApplication")

 if len(args) < 1 {
  fmt.Println("Invalid number of arguments")
  return nil, errors.New("Missing loan application ID")
 }

 var loanApplicationId = args[0]
 bytes, err := stub.GetState(loanApplicationId)
 if err != nil {
  fmt.Println("Could not fetch loan application with id "+loanApplicationId+" from ledger", err)
  return nil, err
 }
 return bytes, nil
}
```

Note: There is also a way to deal with data using traditional relational data models. For example, **ChaincodeStubInterface** has a way to create tables and deal with rows and columns. But this is only a logical abstraction, and the data will be stored as a key value pair in RocksDB. At the time of this writing, version 1.0 of Hyperledger Fabric is under development and has deprecated the table data structure. Hence this tutorial doesn't discuss it in order to minimize the changes a chaincode developer would need to make from v0.6 to v1.0.

Marshal and unmarshal golang structs to JSON strings

As demonstrated in Listing 7 and Listing 8, the stub.PutState and stub.GetState methods deal only with byte arrays. So it is important to be able to convert regular struct objects being used in the chaincode to JSON strings and vice versa.

Listing 9 shows how to marshal a struct into a JSON string byte array that can then be stored in the ledger. **Line 2** creates an instance of the **PersonalInfo** object. **Line 3** uses the **json** package to marshal the object into a JSON string and return the byte array for the same.

The json package can be imported by including **"encoding/json"** in the import block at the top. This byte array can then be stored in the ledger using the **stub.PutState** method demonstrated in Listing 7.

Listing 9. Code to marshal a struct into a JSON string byte array

```
var personalInfo PersonalInfo
 personalInfo = PersonalInfo{"Varun", "Ojha", "dob", "varun@gmail.com", "9999999999"}
 bytes, err := json.Marshal (&personalInfo)
 if err != nil {
        fmt.Println("Could not marshal personal info object", err)
  return nil, err
 }
 err = stub.PutState("key", bytes)
```

Listing 10 shows how to unmarshal a struct from a byte array into a populated struct. **Line 1** fetches the **PersonalInfo** JSON string bytes from the ledger using the associated key. **Line 3** unmarshals the bytes retrieved in line 1 into the PersonalInfo object referenced by the variable personalInfo.

Now you can access and modify the personalInfo object using the dot notation as shown in **line 4**.

Listing 10. Code to unmarshal a struct from a byte array into a populated struct

```
piBytes, err := stub.GetState("la1")
 var personalInfo PersonalInfo
 err = json.Unmarshal(piBytes, &personalInfo)
 fmt.Println(personalInfo.Firstname)
```

Implementing access control and permissions

One of the primary differences between Hyperledger and other blockchain fabrics is that it has a secure and permissioned ledger, which is suitable for implementing enterprise-grade solutions.

Membership services

The Membership services component in the Hyperledger plays a pivotal role in enabling security and permissions within the blockchain network. It's responsible for issuing Enrollment and Transaction certificates to users in response to registration and enrollment.

- **Enrollment certificate**: The certificate authority in the membership services will issue an enrollment certificate to a user that wants to transact on blockchain as a proof of identity.
- **Transaction certificate**: The transaction certificate is supposed to be used as a one-time token that is passed along each invocation request of the chaincode by the invoker/invoking application. The transaction certificates are mathematically derived from the parent enrollment certificate, and hence a theoretically unlimited number of transaction certificates can be generated from the parent enrollment certificate. The transaction certificate can be used to sign and encrypt the transaction data when it is stored in the blockchain. This ensures that only the user with the transaction certificate or a regulator/auditor etc. who has the parent certificate can actually view the contents of the transaction once they have been written.
- **Attributes**: Each transaction certificate can hold a number of user-defined attributes. These can be mentioned as a part of the registration request to the certificate authority from the client application. Follow-on tutorials in this series will deal with this in detail from the perspective of developing the client application.

Listing 11 shows how the attributes can be retrieved from the transaction certificate of the caller. As mentioned earlier, the user or the client application will need to pass in the user’s certificate as a part of each chaincode invocation request so as to authenticate with the target peer on the blockchain network. The HFC SDK takes care of passing the certificate as a part of the request automatically.

The Invoke function at **line 11** of Listing 11 has been modified to check for the input function name and delegate the call to the appropriate handler function. In addition, the Invoke function also validates the access and role of the caller who has sent the **CreateLoanApplication** invocation request. The Invoke function calls the custom **GetCertAttribute** function to retrieve a particular attribute from the transaction certificate of the caller.

The **GetCertAttribute** function fetches the attribute value by passing in the attribute name on **line 3**.

Line 15 checks whether the caller has the role of a Bank Admin and can invoke the **CreateLoanApplication** function. If the caller does not have the required role, an appropriate error is returned. In this way, attribute-based access control can be implemented in the chaincode.

The **ChaincodeStubInterface** has some utility functions that deal with attributes, such as **ReadCertAttribute**, **VerifyAttribute**, and **VerifyAttributes**. All of these methods rely on the *github.com/hyperledger/fabric/core/chaincode/shim/crypto/attr* package to create and use the AttributeHandlerImpl that handles attributes.

In the version of Hyperledger Fabric that is currently under development (v1.0), these utility functions have been removed from the ChaincodeStubInterface. Hence in v1.0, the chaincode developer would need to use the AttributeHandlerImpl directly to work with attributes.

Listing 11. Retrieving attributes from the transaction certificate of the caller

```
func GetCertAttribute(stub shim.ChaincodeStubInterface, attributeName string) (string, error) {
 fmt.Println("Entering GetCertAttribute")
 attr, err := stub.ReadCertAttribute(attributeName)
 if err != nil {
  return "", errors.New("Couldn't get attribute " + attributeName + ". Error: " + err.Error())
 }
 attrString := string(attr)
 return attrString, nil
}

func (t *SampleChaincode) Invoke(stub shim.ChaincodeStubInterface, function string, args []string) ([]byte,
 error) {
 if function == "CreateLoanApplication" {
  username, _ := GetCertAttribute(stub, "username")
  role, _ := GetCertAttribute(stub, "role")
  if role == "Bank_Home_Loan_Admin" {
   return CreateLoanApplication(stub, args)
  } else {
   return nil, errors.New(username + " with role " + role + " does not have access to create a loan
 application")
  }

 }
 return nil, nil
}
```

Creating and emitting custom events

The Hyperledger contains an event framework that can be used to publish/subscribe predefined or custom events. Custom events can be created in the chaincode and emitted at your discretion. For example, an event could be generated whenever there is any change to the state of the blockchain. These events can be subscribed to and consumed by the client application by registering an event adapter with the event hub on blockchain. Follow-on tutorials in this series will show in detail how the client application can subscribe to and consume events produced via the chaincode using the HFC SDK.

In addition to custom events, some of the predefined internal events that are part of the Hyperledger are:

- Block events
- Chaincode events
- Rejection events
- Register events

The code in Listing 12 shows how to create and publish a custom event.

Lines 1-4 define a custom event object containing type and description fields.

The **CreateLoanApplication** function starts on **line 6** and has been modified to include the event creation on successful creation of loan application.

Line 23 creates the instance of the **customEvent** object and fills in appropriate event details.

Line 24 marshals the event object to a JSON string byte array as explained earlier.

Line 28 sets the custom event. The **stub.SetEvent** method takes two arguments: the event **name** and **payload**. The client application can subscribe to the same event name/topic to receive events as and when they are generated by the chaincode.

Listing 12. Creating and publishing a custom event

```
type customEvent struct {
 Type        string `json:"type"`
 Description string `json:"description"`
}

func CreateLoanApplication(stub shim.ChaincodeStubInterface, args []string) ([]byte, error) {
 fmt.Println("Entering CreateLoanApplication")

 if len(args) < 2 {
  fmt.Println("Invalid number of args")
  return nil, errors.New("Expected at least two arguments for loan application creation")
 }

 var loanApplicationId = args[0]
 var loanApplicationInput = args[1]

 err := stub.PutState(loanApplicationId, []byte(loanApplicationInput))
 if err != nil {
  fmt.Println("Could not save loan application to ledger", err)
  return nil, err
 }

 var event = customEvent{"createLoanApplication", "Successfully created loan application with ID " +
 loanApplicationId}
    eventBytes, err := json.Marshal(&event)
    if err != nil {
            return nil, err
    }
 err = stub.SetEvent("evtSender", eventBytes)
 if err != nil {
  fmt.Println("Could not set event for loan application creation", err)
 }
```

```
 fmt.Println("Successfully saved loan application")
 return nil, nil

}
```

Handling logging

Logging can be handled in the chaincode either by using the standard **fmt** package and print statements or by using the **ChaincodeLogger** type in the **shim** package.

The ChaincodeLogger supports the following log levels:

- CRITICAL
- ERROR
- WARNING
- NOTICE
- INFO
- DEBUG

You can set the logging level in three ways:

- **shim.SetChaincodeLoggingLevel()**: This method will pick up the logging level specified in the core.yaml file in the CORE_LOGGING_CHAINCODE set. The core.yaml file contains all the configuration information required to set up and deploy the blockchain network.
- **shim.SetLoggingLevel(level LoggingLevel)**: This method will set the logging level at the shim level.
- **ChaincodeLogger.SetLevel(level LoggingLevel)**: This method will set the logging level at the individual logger instance level.

Listing 13 shows how to create, configure, and use the ChaincodeLogger.

Listing 13. Creating, configuring, and using the ChaincodeLogger

```
func SampleLogging() {
  //Different Logging Levels
  criticalLevel, _ := shim.LogLevel("CRITICAL")
  errorLevel, _ := shim.LogLevel("ERROR")
  warningLevel, _ := shim.LogLevel("WARNING")
  noticeLevel, _ := shim.LogLevel("NOTICE")
  infoLevel, _ := shim.LogLevel("INFO")
  debugLevel, _ := shim.LogLevel("DEBUG")

  //Logging level at the shim level
  shim.SetLoggingLevel(infoLevel)

  //Create a logger instance
  myLogger := shim.NewLogger("SampleChaincodeLogger")

  //Set logging level on logger instance
  myLogger.SetLevel(infoLevel)

  //Check logging level
  fmt.Println(myLogger.IsEnabledFor(infoLevel))
```

```
  //Log statements
  myLogger.Info("Info Message")
  myLogger.Critical("Critical Message")
  myLogger.Warning("Warning Message")
  myLogger.Error("Error Message")
  myLogger.Notice("Notice Message")
  myLogger.Debug("Debug Message")

 }
```

FAQs and best practices

While developing blockchain applications with clients, I often answer these questions.

How do I store files (images, audio, video, PDF, etc.) in blockchain?

Both of these approaches work in the current Hyperledger Fabric (v0.6):

- Store all files/objects as base64-encoded strings. The client application would convert the file/object into a base64-encoded string and send it as an input parameter to a chaincode function. The chaincode in turn can store it as a byte array in the key/value store.
- Store the actual file/object contents outside of blockchain; for example, in IBM Bluemix Object Storage service. Store only the link/reference/ID of the file/object on blockchain along with the hash of the file/object. Storing the hash ensures that any tampering with the file/object outside of blockchain can be detected by concerned parties/entities.

How can I avoid disclosing private business logic/contract details to all peers in the network?

This question came up in a supply chain scenario where an end consumer of the blockchain solution was not comfortable sharing private business logic/contract information (such as different negotiated rates with different vendors) in the smart contract visible to all peers. In v0.6, this situation can be solved using external system integration.

The solution: The business logic/rules/contract that the peer wants to keep private can be run as a set of business rules in an external application like a service. The chaincode itself has the ability to make outbound calls. So the chaincode could make REST API calls, for example, to the business rules/logic service and fetch the results, thus keeping the logic hidden from the actual chaincode.

It is possible to integrate with systems external to blockchain from within the chaincode. For example, the chaincode can be used to talk to external databases, APIs, etc. But it is important to make sure that interaction with these systems does not make the chaincode non-deterministic.

Constraints and concerns

- The business rules service can be modified without knowledge of the remaining peers, since it is running outside of blockchain. Depending on the type of business interaction among the different participants in the blockchain network, this could lead to trust issues.

- The business rules service must be available to all peers in the blockchain network who will run the smart contract/chaincode.
- The solution can lead to chaincode becoming non-deterministic. **Chaincode MUST be deterministic.** In a nutshell, if the same function in the chaincode is invoked with the same parameters by multiple parties, the results should be the same. For example, if you use a timestamp or a counter in your chaincode functions that are tied to the response, invocations of the chaincode by multiple peers will lead to different results. This will lead to an inconsistent ledger state among the different peers in the blockchain network.
  Remember that each invocation of the chaincode causes all the peers that are part of the consensus network to invoke the chaincode on their local copies of the ledger.

Note: In v1.0 of the Hyperledger Fabric that is currently under development, this problem has been solved organically via changes to the architecture itself.

Conclusion

This tutorial began with the fundamentals of chaincode and then dove into the building blocks and different APIs available for performing important tasks in the chaincode, such as access control, data modeling, and event management. Download the consolidated code snippets in the SampleChaincode.go file.

Part 2 of this series covers test-driven development of chaincode. The final part will show how to develop Node.js client applications that can talk to the blockchain network and complete the development story.

Downloadable resources

Description	Name	Size
Sample chaincode in Go	chaincode_sample.go.zip	1.6KB

Related topics

- Part 2 of this series
- Blockchain Developer Center
- Hyperledger Fabric
- Hyperledger community
- Hyperledger discussion channels
- IBM Blockchain 101: Quick-start guide for developers
- IBM Blockchain service on Bluemix (free)
- IBM Blockchain courses for developers (free)
- IBM Blockchain videos on developerWorks TV