**Chaincode for Developers**

**What is Chaincode?**

Chaincode is a program, written in [Go](https://golang.org/), [Node.js](https://nodejs.org/), or [Java](https://java.com/en/) that implements a prescribed interface. Chaincode runs in a separate process from the peer and initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it similar to a “smart contract”. A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a Query, which does not participate in state validation checks in subsequent commit phase.

In the following sections, we will explore chaincode through the eyes of an application developer. We’ll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API. If you are a network operator who is deploying a chaincode to running network, visit the [Deploying a smart contract to a channel](https://hyperledger-fabric.readthedocs.io/en/latest/deploy_chaincode.html) tutorial and the [Fabric chaincode lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode_lifecycle.html) concept topic.

This tutorial provides an overview of the low level APIs provided by the Fabric Chaincode Shim API. You can also use the higher level APIs provided by the Fabric Contract API. To learn more about developing smart contracts using the Fabric contract API, visit the [Smart Contract Processing](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html) topic.

**Chaincode API**

Every chaincode program must implement the Chaincode interface whose methods are called in response to received transactions. You can find the reference documentation of the Chaincode Shim API for different languages below:

* [Go](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#Chaincode)
* [Node.js](https://hyperledger.github.io/fabric-chaincode-node/master/api/fabric-shim.ChaincodeInterface.html)
* [Java](https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/shim/Chaincode.html)

In each language, the Invoke method is called by clients to submit transaction proposals. This method allows you to use the chaincode to read and write data on the channel ledger.

You also need to include an Init method in your chaincode that will serve as the initialization function. This function is required by the chaincode interface, but does not necessarily need to invoked by your applications. You can use the Fabric chaincode lifecycle process to specify whether the Init function must be called prior to Invokes. For more information, refer to the initialization parameter in the [Approving a chaincode definition](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode_lifecycle.html#step-three-approve-a-chaincode-definition-for-your-organization) step of the Fabric chaincode lifecycle documentation.

The other interface in the chaincode “shim” APIs is the ChaincodeStubInterface:

* [Go](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStubInterface)
* [Node.js](https://hyperledger.github.io/fabric-chaincode-node/master/api/fabric-shim.ChaincodeStub.html)
* [Java](https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/shim/ChaincodeStub.html)

which is used to access and modify the ledger, and to make invocations between chaincodes.

In this tutorial using Go chaincode, we will demonstrate the use of these APIs by implementing a simple chaincode application that manages simple “assets”.

**Simple Asset Chaincode**

Our application is a basic sample chaincode to create assets (key-value pairs) on the ledger.

**Choosing a Location for the Code**

If you haven’t been doing programming in Go, you may want to make sure that you have [Go](https://golang.org/) installed and your system properly configured. We assume you are using a version that supports modules.

Now, you will want to create a directory for your chaincode application.

To keep things simple, let’s use the following command:

mkdir sacc **&&** cd sacc

Now, let’s create the module and the source file that we’ll fill in with code:

go mod init sacc

touch sacc**.**go

**Housekeeping**

First, let’s start with some housekeeping. As with every chaincode, it implements the [Chaincode interface](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#Chaincode) in particular, Init and Invoke functions. So, let’s add the Go import statements for the necessary dependencies for our chaincode. We’ll import the chaincode shim package and the [peer protobuf package](https://godoc.org/github.com/hyperledger/fabric-protos-go/peer). Next, let’s add a struct SimpleAsset as a receiver for Chaincode shim functions.

package main

**import** (

"fmt"

"github.com/hyperledger/fabric-chaincode-go/shim"

"github.com/hyperledger/fabric-protos-go/peer"

)

**//** SimpleAsset implements a simple chaincode to manage an asset

type SimpleAsset struct {

}

**Initializing the Chaincode**

Next, we’ll implement the Init function.

**//** Init **is** called during chaincode instantiation to initialize any data**.**

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

}

**Note**

Note that chaincode upgrade also calls this function. When writing a chaincode that will upgrade an existing one, make sure to modify the Init function appropriately. In particular, provide an empty “Init” method if there’s no “migration” or nothing to be initialized as part of the upgrade.

Next, we’ll retrieve the arguments to the Init call using the [ChaincodeStubInterface.GetStringArgs](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "ChaincodeStub.GetStringArgs) function and check for validity. In our case, we are expecting a key-value pair.

**//** Init **is** called during chaincode instantiation to initialize any

**//** data**.** Note that chaincode upgrade also calls this function to reset

**//** **or** to migrate data, so be careful to avoid a scenario where you

**//** inadvertently clobber your ledger's data!

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Get the args **from** the transaction proposal

args :**=** stub**.**GetStringArgs()

**if** len(args) **!=** 2 {

**return** shim**.**Error("Incorrect arguments. Expecting a key and a value")

}

}

Next, now that we have established that the call is valid, we’ll store the initial state in the ledger. To do this, we will call [ChaincodeStubInterface.PutState](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "ChaincodeStub.PutState) with the key and value passed in as the arguments. Assuming all went well, return a peer.Response object that indicates the initialization was a success.

**//** Init **is** called during chaincode instantiation to initialize any

**//** data**.** Note that chaincode upgrade also calls this function to reset

**//** **or** to migrate data, so be careful to avoid a scenario where you

**//** inadvertently clobber your ledger's data!

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Get the args **from** the transaction proposal

args :**=** stub**.**GetStringArgs()

**if** len(args) **!=** 2 {

**return** shim**.**Error("Incorrect arguments. Expecting a key and a value")

}

**//** Set up any variables **or** assets here by calling stub**.**PutState()

**//** We store the key **and** the value on the ledger

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)

}

**Invoking the Chaincode**

First, let’s add the Invoke function’s signature.

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The 'set'

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

}

As with the Init function above, we need to extract the arguments from the ChaincodeStubInterface. The Invoke function’s arguments will be the name of the chaincode application function to invoke. In our case, our application will simply have two functions: set and get, that allow the value of an asset to be set or its current state to be retrieved. We first call [ChaincodeStubInterface.GetFunctionAndParameters](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "ChaincodeStub.GetFunctionAndParameters) to extract the function name and the parameters to that chaincode application function.

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The Set

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Extract the function **and** args **from** the transaction proposal

fn, args :**=** stub**.**GetFunctionAndParameters()

}

Next, we’ll validate the function name as being either set or get, and invoke those chaincode application functions, returning an appropriate response via the shim.Success or shim.Error functions that will serialize the response into a gRPC protobuf message.

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The Set

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Extract the function **and** args **from** the transaction proposal

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 the result **as** success payload

**return** shim**.**Success([]byte(result))

}

**Implementing the Chaincode Application**

As noted, our chaincode application implements two functions that can be invoked via the Invoke function. Let’s implement those functions now. Note that as we mentioned above, to access the ledger’s state, we will leverage the [ChaincodeStubInterface.PutState](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub.PutState) and [ChaincodeStubInterface.GetState](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub.GetState) functions of the chaincode shim API.

**//** Set stores the asset (both key **and** value) on the ledger**.** If the key exists,

**//** it will override the value **with** the new one

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

}

**//** Get returns the value of the specified asset key

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

}

**Pulling it All Together**

Finally, we need to add the main function, which will call the [shim.Start](https://godoc.org/github.com/hyperledger/fabric-chaincode-go/shim" \l "Start) function. Here’s the whole chaincode program source.

package main

**import** (

"fmt"

"github.com/hyperledger/fabric-chaincode-go/shim"

"github.com/hyperledger/fabric-protos-go/peer"

)

**//** SimpleAsset implements a simple chaincode to manage an asset

type SimpleAsset struct {

}

**//** Init **is** called during chaincode instantiation to initialize any

**//** data**.** Note that chaincode upgrade also calls this function to reset

**//** **or** to migrate data**.**

func (t **\***SimpleAsset) Init(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Get the args **from** the transaction proposal

args :**=** stub**.**GetStringArgs()

**if** len(args) **!=** 2 {

**return** shim**.**Error("Incorrect arguments. Expecting a key and a value")

}

**//** Set up any variables **or** assets here by calling stub**.**PutState()

**//** We store the key **and** the value on the ledger

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)

}

**//** Invoke **is** called per transaction on the chaincode**.** Each transaction **is**

**//** either a 'get' **or** a 'set' on the asset created by Init function**.** The Set

**//** method may create a new asset by specifying a new key**-**value pair**.**

func (t **\***SimpleAsset) Invoke(stub shim**.**ChaincodeStubInterface) peer**.**Response {

**//** Extract the function **and** args **from** the transaction proposal

fn, args :**=** stub**.**GetFunctionAndParameters()

var result string

var err error

**if** fn **==** "set" {

result, err **=** set(stub, args)

} **else** { **//** assume 'get' even **if** fn **is** nil

result, err **=** get(stub, args)

}

**if** err **!=** nil {

**return** shim**.**Error(err**.**Error())

}

**//** Return the result **as** success payload

**return** shim**.**Success([]byte(result))

}

**//** Set stores the asset (both key **and** value) on the ledger**.** If the key exists,

**//** it will override the value **with** the new one

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

}

**//** Get returns the value of the specified asset key

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

}

**//** main function starts up the chaincode **in** the container during instantiate

func main() {

**if** err :**=** shim**.**Start(new(SimpleAsset)); err **!=** nil {

fmt**.**Printf("Error starting SimpleAsset chaincode: %s", err)

}

}

**Chaincode access control**

Chaincode can utilize the client (submitter) certificate for access control decisions by calling the GetCreator() function. Additionally the Go shim provides extension APIs that extract client identity from the submitter’s certificate that can be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute.

For example an asset that is represented as a key/value may include the client’s identity as part of the value (for example as a JSON attribute indicating that asset owner), and only this client may be authorized to make updates to the key/value in the future. The client identity library extension APIs can be used within chaincode to retrieve this submitter information to make such access control decisions.

See the [client identity (CID) library documentation](https://github.com/hyperledger/fabric-chaincode-go/blob/master/pkg/cid/README.md) for more details.

To add the client identity shim extension to your chaincode as a dependency, see [Managing external dependencies for chaincode written in Go](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html#vendoring).

**Managing external dependencies for chaincode written in Go**

Your Go chaincode depends on Go packages (like the chaincode shim) that are not part of the standard library. The source to these packages must be included in your chaincode package when it is installed to a peer. If you have structured your chaincode as a module, the easiest way to do this is to “vendor” the dependencies with go mod vendor before packaging your chaincode.

go mod tidy

go mod vendor

This places the external dependencies for your chaincode into a local vendor directory.

Once dependencies are vendored in your chaincode directory, peer chaincode package and peer chaincode install operations will then include code associated with the dependencies into the chaincode package.