**Writing Your First Chaincode**

**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 asset-transfer chaincode sample walkthrough, and the purpose of each method in the Fabric Contract 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 high 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.

**Fabric Contract API**

The fabric-contract-api provides the contract interface, a high level API for application developers to implement Smart Contracts. Within Hyperledger Fabric, Smart Contracts are also known as Chaincode. Working with this API provides a high level entry point to writing business logic. Documentation of the Fabric Contract API for different languages can be found at the links below:

* [Go](https://godoc.org/github.com/hyperledger/fabric-contract-api-go/contractapi)
* [Node.js](https://hyperledger.github.io/fabric-chaincode-node/master/api/)
* [Java](https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/contract/package-summary.html)

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

**Asset Transfer Chaincode**

Our application is a basic sample chaincode to initialize a ledger with assets, create, read, update, and delete assets, check to see if an asset exists, and transfer assets from one owner to another.

**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:

**//** atcc **is** shorthand **for** asset transfer chaincode

mkdir atcc **&&** cd atcc

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

go mod init atcc

touch atcc**.**go

**Housekeeping**

First, let’s start with some housekeeping. As with every chaincode, it implements the [fabric-contract-api interface](https://godoc.org/github.com/hyperledger/fabric-contract-api-go/contractapi), so let’s add the Go import statements for the necessary dependencies for our chaincode. We’ll import the fabric contract api package and define our SmartContract.

package main

**import** (

"fmt"

"log"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

**//** SmartContract provides functions **for** managing an Asset

type SmartContract struct {

contractapi**.**Contract

}

Next, let’s add a struct Asset as a receiver for Chaincode functions.

// Asset describes basic details of what makes up a simple asset

type Asset struct {

ID string `json:"ID"`

Color string `json:"color"`

Size int `json:"size"`

Owner string `json:"owner"`

AppraisedValue int `json:"appraisedValue"`

}

**Initializing the Chaincode**

Next, we’ll implement the Init function to populate the ledger with some initial data.

**//** InitLedger adds a base set of assets to the ledger

func (s **\***SmartContract) InitLedger(ctx contractapi**.**TransactionContextInterface) error {

assets :**=** []Asset{

{ID: "asset1", Color: "blue", Size: 5, Owner: "Tomoko", AppraisedValue: 300},

{ID: "asset2", Color: "red", Size: 5, Owner: "Brad", AppraisedValue: 400},

{ID: "asset3", Color: "green", Size: 10, Owner: "Jin Soo", AppraisedValue: 500},

{ID: "asset4", Color: "yellow", Size: 10, Owner: "Max", AppraisedValue: 600},

{ID: "asset5", Color: "black", Size: 15, Owner: "Adriana", AppraisedValue: 700},

{ID: "asset6", Color: "white", Size: 15, Owner: "Michel", AppraisedValue: 800},

}

**for** \_, asset :**=** range assets {

assetJSON, err :**=** json**.**Marshal(asset)

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

**return** err

}

err **=** ctx**.**GetStub()**.**PutState(asset**.**ID, assetJSON)

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

**return** fmt**.**Errorf("failed to put to world state. %v", err)

}

}

**return** nil

}

**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 write a function to create an asset on the ledger that does not yet exist. When writing chaincode, it is a good idea to check for the existence of something on the ledger prior to taking an action on it, as is demonstrated in the CreateAsset function below.

**//** CreateAsset issues a new asset to the world state **with** given details**.**

func (s **\***SmartContract) CreateAsset(ctx contractapi**.**TransactionContextInterface, id string, color string, size int, owner string, appraisedValue int) error {

exists, err :**=** s**.**AssetExists(ctx, id)

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

**return** err

}

**if** exists {

**return** fmt**.**Errorf("the asset %s already exists", id)

}

asset :**=** Asset{

ID: id,

Color: color,

Size: size,

Owner: owner,

AppraisedValue: appraisedValue,

}

assetJSON, err :**=** json**.**Marshal(asset)

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

**return** err

}

**return** ctx**.**GetStub()**.**PutState(id, assetJSON)

}

Now that we have populated the ledger with some initial assets and created an asset, let’s write a function ReadAsset that allows us to read an asset from the ledger.

**//** ReadAsset returns the asset stored **in** the world state **with** given id**.**

func (s **\***SmartContract) ReadAsset(ctx contractapi**.**TransactionContextInterface, id string) (**\***Asset, error) {

assetJSON, err :**=** ctx**.**GetStub()**.**GetState(id)

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

**return** nil, fmt**.**Errorf("failed to read from world state: %v", err)

}

**if** assetJSON **==** nil {

**return** nil, fmt**.**Errorf("the asset %s does not exist", id)

}

var asset Asset

err **=** json**.**Unmarshal(assetJSON, **&**asset)

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

**return** nil, err

}

**return** **&**asset, nil

}

Now that we have assets on our ledger we can interact with, let’s write a chaincode function UpdateAsset that allows us to update attributes of the asset that we are allowed to change.

// UpdateAsset updates an existing asset in the world state with provided parameters.

func (s \*SmartContract) UpdateAsset(ctx contractapi.TransactionContextInterface, id string, color string, size int, owner string, appraisedValue int) error {

exists, err := s.AssetExists(ctx, id)

if err != nil {

return err

}

if !exists {

return fmt.Errorf("the asset %s does not exist", id)

}

// overwriting original asset with new asset

asset := Asset{

ID: id,

Color: color,

Size: size,

Owner: owner,

AppraisedValue: appraisedValue,

}

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

return ctx.GetStub().PutState(id, assetJSON)

}

There may be cases where we need the ability to delete an asset from the ledger, so let’s write a DeleteAsset function to handle that requirement.

// DeleteAsset deletes an given asset from the world state.

func (s \*SmartContract) DeleteAsset(ctx contractapi.TransactionContextInterface, id string) error {

exists, err := s.AssetExists(ctx, id)

if err != nil {

return err

}

if !exists {

return fmt.Errorf("the asset %s does not exist", id)

}

return ctx.GetStub().DelState(id)

}

We said earlier that is was a good idea to check to see if an asset exists before taking an action on it, so let’s write a function called AssetExists to implement that requirement.

**//** AssetExists returns true when asset **with** given ID exists **in** world state

func (s **\***SmartContract) AssetExists(ctx contractapi**.**TransactionContextInterface, id string) (bool, error) {

assetJSON, err :**=** ctx**.**GetStub()**.**GetState(id)

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

**return** false, fmt**.**Errorf("failed to read from world state: %v", err)

}

**return** assetJSON **!=** nil, nil

}

Next, we’ll write a function we’ll call TransferAsset that enables the transfer of an asset from one owner to another.

**//** TransferAsset updates the owner field of asset **with** given id **in** world state**.**

func (s **\***SmartContract) TransferAsset(ctx contractapi**.**TransactionContextInterface, id string, newOwner string) error {

asset, err :**=** s**.**ReadAsset(ctx, id)

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

**return** err

}

asset**.**Owner **=** newOwner

assetJSON, err :**=** json**.**Marshal(asset)

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

**return** err

}

**return** ctx**.**GetStub()**.**PutState(id, assetJSON)

}

Let’s write a function we’ll call GetAllAssets that enables the querying of the ledger to return all of the assets on the ledger.

**//** GetAllAssets returns all assets found **in** world state

func (s **\***SmartContract) GetAllAssets(ctx contractapi**.**TransactionContextInterface) ([]**\***Asset, error) {

**//** range query **with** empty string **for** startKey **and** endKey does an

**//** open**-**ended query of all assets **in** the chaincode namespace**.**

resultsIterator, err :**=** ctx**.**GetStub()**.**GetStateByRange("", "")

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

**return** nil, err

}

defer resultsIterator**.**Close()

var assets []**\***Asset

**for** resultsIterator**.**HasNext() {

queryResponse, err :**=** resultsIterator**.**Next()

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

**return** nil, err

}

var asset Asset

err **=** json**.**Unmarshal(queryResponse**.**Value, **&**asset)

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

**return** nil, err

}

assets **=** append(assets, **&**asset)

}

**return** assets, nil

}

**Pulling it All Together**

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

package main

import (

"encoding/json"

"fmt"

"log"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

// SmartContract provides functions for managing an Asset

type SmartContract struct {

contractapi.Contract

}

// Asset describes basic details of what makes up a simple asset

type Asset struct {

ID string `json:"ID"`

Color string `json:"color"`

Size int `json:"size"`

Owner string `json:"owner"`

AppraisedValue int `json:"appraisedValue"`

}

// InitLedger adds a base set of assets to the ledger

func (s \*SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {

assets := []Asset{

{ID: "asset1", Color: "blue", Size: 5, Owner: "Tomoko", AppraisedValue: 300},

{ID: "asset2", Color: "red", Size: 5, Owner: "Brad", AppraisedValue: 400},

{ID: "asset3", Color: "green", Size: 10, Owner: "Jin Soo", AppraisedValue: 500},

{ID: "asset4", Color: "yellow", Size: 10, Owner: "Max", AppraisedValue: 600},

{ID: "asset5", Color: "black", Size: 15, Owner: "Adriana", AppraisedValue: 700},

{ID: "asset6", Color: "white", Size: 15, Owner: "Michel", AppraisedValue: 800},

}

for \_, asset := range assets {

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

err = ctx.GetStub().PutState(asset.ID, assetJSON)

if err != nil {

return fmt.Errorf("failed to put to world state. %v", err)

}

}

return nil

}

// CreateAsset issues a new asset to the world state with given details.

func (s \*SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id string, color string, size int, owner string, appraisedValue int) error {

exists, err := s.AssetExists(ctx, id)

if err != nil {

return err

}

if exists {

return fmt.Errorf("the asset %s already exists", id)

}

asset := Asset{

ID: id,

Color: color,

Size: size,

Owner: owner,

AppraisedValue: appraisedValue,

}

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

return ctx.GetStub().PutState(id, assetJSON)

}

// ReadAsset returns the asset stored in the world state with given id.

func (s \*SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface, id string) (\*Asset, error) {

assetJSON, err := ctx.GetStub().GetState(id)

if err != nil {

return nil, fmt.Errorf("failed to read from world state: %v", err)

}

if assetJSON == nil {

return nil, fmt.Errorf("the asset %s does not exist", id)

}

var asset Asset

err = json.Unmarshal(assetJSON, &asset)

if err != nil {

return nil, err

}

return &asset, nil

}

// UpdateAsset updates an existing asset in the world state with provided parameters.

func (s \*SmartContract) UpdateAsset(ctx contractapi.TransactionContextInterface, id string, color string, size int, owner string, appraisedValue int) error {

exists, err := s.AssetExists(ctx, id)

if err != nil {

return err

}

if !exists {

return fmt.Errorf("the asset %s does not exist", id)

}

// overwriting original asset with new asset

asset := Asset{

ID: id,

Color: color,

Size: size,

Owner: owner,

AppraisedValue: appraisedValue,

}

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

return ctx.GetStub().PutState(id, assetJSON)

}

// DeleteAsset deletes an given asset from the world state.

func (s \*SmartContract) DeleteAsset(ctx contractapi.TransactionContextInterface, id string) error {

exists, err := s.AssetExists(ctx, id)

if err != nil {

return err

}

if !exists {

return fmt.Errorf("the asset %s does not exist", id)

}

return ctx.GetStub().DelState(id)

}

// AssetExists returns true when asset with given ID exists in world state

func (s \*SmartContract) AssetExists(ctx contractapi.TransactionContextInterface, id string) (bool, error) {

assetJSON, err := ctx.GetStub().GetState(id)

if err != nil {

return false, fmt.Errorf("failed to read from world state: %v", err)

}

return assetJSON != nil, nil

}

// TransferAsset updates the owner field of asset with given id in world state.

func (s \*SmartContract) TransferAsset(ctx contractapi.TransactionContextInterface, id string, newOwner string) error {

asset, err := s.ReadAsset(ctx, id)

if err != nil {

return err

}

asset.Owner = newOwner

assetJSON, err := json.Marshal(asset)

if err != nil {

return err

}

return ctx.GetStub().PutState(id, assetJSON)

}

// GetAllAssets returns all assets found in world state

func (s \*SmartContract) GetAllAssets(ctx contractapi.TransactionContextInterface) ([]\*Asset, error) {

// range query with empty string for startKey and endKey does an

// open-ended query of all assets in the chaincode namespace.

resultsIterator, err := ctx.GetStub().GetStateByRange("", "")

if err != nil {

return nil, err

}

defer resultsIterator.Close()

var assets []\*Asset

for resultsIterator.HasNext() {

queryResponse, err := resultsIterator.Next()

if err != nil {

return nil, err

}

var asset Asset

err = json.Unmarshal(queryResponse.Value, &asset)

if err != nil {

return nil, err

}

assets = append(assets, &asset)

}

return assets, nil

}

func main() {

assetChaincode, err := contractapi.NewChaincode(&SmartContract{})

if err != nil {

log.Panicf("Error creating asset-transfer-basic chaincode: %v", err)

}

if err := assetChaincode.Start(); err != nil {

log.Panicf("Error starting asset-transfer-basic chaincode: %v", err)

}

}

**Chaincode access control**

Chaincode can utilize the client (submitter) certificate for access control decisions by calling the GetClientIdentity() function. Additionally the Fabric Contract API 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.

**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.