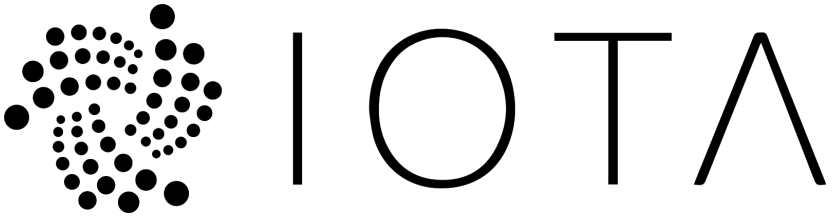


Blueprint – Hyperledger Fabric Integration

Version 1.0 - December 2nd, 2019 - Alexey Sobolev

Introduction

The IOTA Foundation (IF) is a non-profit organization supporting the development and adoption of the IOTA Tangle, a permissionless Distributed Ledger Technology (DLT), particularly suitable for creating trusted information and value sharing across multi-stakeholder ecosystems. IOTA technology is open source. IF's agile approach to solutions creation leverages on identifying industry problems with real stakeholders, building PoCs and collaboratively testing and validating or refining assumptions.

A blueprint is a document that explains how IOTA technology is used to solve real problems and to support well defined business needs. It provides guidelines for other to replicate and deploy the developed solution into similar system and to test different business models.

A blueprint is generally easy to read and understand, but nevertheless, it provides enough clarity on how easy it would be to integrate the described solution into other systems. At the same time, it contains links to more technical resources available to developers.

This blueprint describes integration between Hyperledger Fabric and IOTA Ecosystem, it is organized as follows:

- We first describe the problem worth solving, underpinning the technical as well as business challenges associated with it and the benefits derived from developing a novel solution.
- We then present a technical architecture showing how IOTA Tangle can be integrated with systems running on Hyperledger Fabric.
- We finally describe source code showing implementation patterns used and replicable for the integration of IOTA in a similar context.

Summary

As the Hyperledger Fabric project becomes increasingly popular for supply chain and asset tracking projects, we introduce the IOTA Connector which allows data to be mirrored into the Tangle, benefitting from all the features available, such as fee-less payments, encrypted transaction payload, and public/private message chains.

At this point we are considering the Hyperledger Fabric DLT - in which all data are initially stored and managed - as the primary source of truth.

Each smart contract (`chaincode`) execution can now trigger a request to the IOTA Tangle using the provided software. This process allows to store/update results of the Hyperledger Fabric smart contract execution on the Tangle and to perform payments between IOTA Wallet holders.

It is also possible to read data from the Tangle and trigger Hyperledger Fabric smart contract execution based on the fetched transaction data or payment confirmation.

The code samples provided below showcase how you can implement multiple scenarios of the IOTA integration in a common supply chain project based on Hyperledger Fabric.

Smart contracts in Hyperledger Fabric can be written in several programming languages. In this document we will review the IOTA connector for the most common one: Go.

We will not go into details of the Hyperledger Fabric architecture, setup, and configuration in this blog post. We assume that you are already familiar with a typical Hyperledger Fabric project and most likely have one such project running.

You can download the source code files for Hyperledger Fabric chaincode IOTA connector from this public GitHub repository.

Architecture

Hyperledger Fabric is a permissioned blockchain network that gets set by the organizations that intend to set up a consortium. The organizations that take part in building the Hyperledger Fabric network are called the “**members**”.

Each **member** organization in the blockchain network is responsible to set up their **peers** for participating in the network. All of these peers need are configured with appropriate cryptographic materials like Certificate Authority and other information.

Peers in the member organization receives transaction invocation requests from the clients inside the organization. A **client** can be any specific application/portal serving specific organization/business activities. The client application uses Hyperledger Fabric SDK or REST web service to interact with the Hyperledger Fabric network. Chaincode (similar to Ethereum Smart Contract) installed in peers causes to initiate transaction invocation request.

All the peers maintain their one ledger per channel that they are subscribed to. Hence Distributed Ledger Technology (DLT). But unlike Ethereum in Hyperledger Fabric blockchain network peers have different roles.

So not all peer nodes are the same. There are different types of peer nodes with different roles in the network:

- Endorser peer
- Anchor peer
- Orderer peer

Endorser peer

Peers can be marked as Endorser peer (i.e. endorsing peer). Upon receiving the “transaction invocation request” from the Client application the Endorser peer

- Validates the transaction. i.e. check certificate details and roles of the requester.
- Executes the Chaincode (i.e. Smart Contract) and simulates the outcome of the transaction. Does **not** update the ledger.

At the end of the above two tasks, the Endorser may approve or disapprove the transaction.

As only the Endorser node executes the Chaincode (Smart Contract) so there is no necessity to install Chaincode in each and every node of the network which increases the scalability of the network.

Anchor peer

Anchor peer or cluster of Anchor peers is configured at the time of Channel configuration. *In Hyperledger Fabric you can configure secret channels among peers and transactions among the peers of that channel are visible only to them.*

Anchor peer receives updates and broadcasts the updates to the other peers in the organization. Anchor peers are discoverable. Any peer marked as Anchor peer can be discovered by the Orderer peer or any other peer.

Orderer peer

Orderer peer is considered as the central communication channel for the Hyperledger Fabric network. Orderer peer/node is responsible for consistent Ledger state across the network. Orderer peer creates the block and delivers that to all the peers.

Orderer is built on top of a message-oriented architecture. There are two options are currently available to implement Orderer peer:

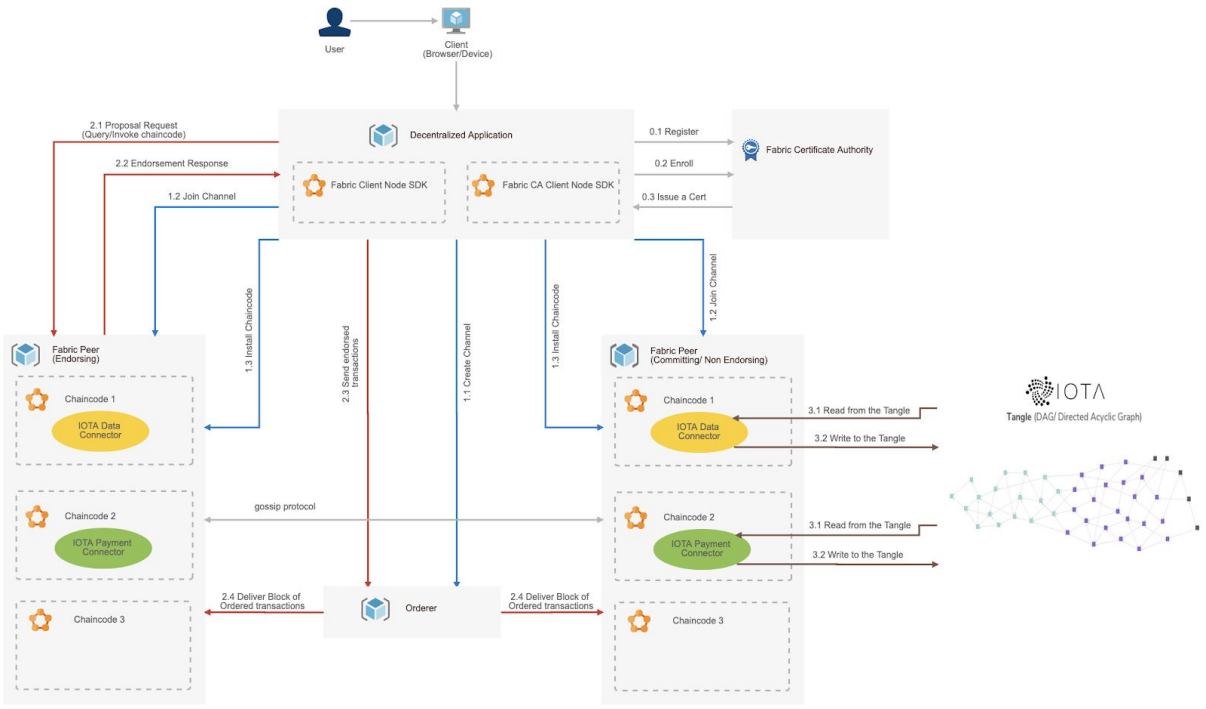
- Solo: Suitable for development. Single point of failure. Solo should not be used for the production-ready network.
- Kafka: Production-ready Hyperledger Fabric network uses Kafka as the Orderer implementation. Kafka is messaging software that has high throughput fault tolerant feature.

Workflow

1. A participant in the member Organization invokes a transaction request through the client application.
2. Client application broadcasts the transaction invocation request to the Endorser peer.
3. Endorser peer checks the Certificate details and others to validate the transaction. Then it executes the Chaincode (i.e. Smart Contract) and returns the Endorsement responses to the Client. Endorser peer sends transaction approval or rejection as part of the endorsement response.
4. The client now sends the approved transaction to the Orderer peer for this to be properly ordered and be included in a block.
5. Orderer node includes the transaction into a block and forwards the block to the Anchor nodes of different member Organizations of the Hyperledger Fabric network.
6. Anchor nodes then broadcast the block to the other peers inside their own organization. These individual peers then update their local ledger with the latest block. Thus all the network gets the ledger synced.

IOTA Connector can be integrated into existing chaincode files, and will be invoked at the moment where the original smart contract is executed.

In the diagram below you can find several IOTA connectors for data mirroring (digital twin) and payment support. In the next chapter we will explain how IOTA connector can be integrated into existing chaincode functions.



How to Integrate the IOTA Connectors

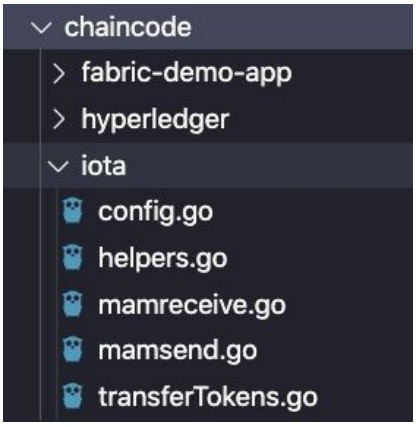
We will start with a simple chaincode file suitable for a supply chain project and will add a few IOTA connectors one by one with a short usage description.

The plain Hyperledger Fabric chaincode written in Go has a namespace, which can be defined is the docker-compose.yml file under `volumes`:

```
volumes:
    - /var/run/:/host/var/run/
    - ./../chaincode/:/opt/gopath/src/github.com/
```

In our case, the chaincode folder is mapped to the `github.com` namespace. All external packages, including IOTA connector, need to be imported using the same namespace.

We start by adding the IOTA connector code into the `chaincode` folder.

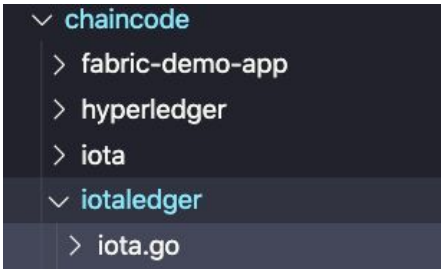


In addition, you will need to add 3 other complementary packages.
You can do this either by adding an instruction to the shell script that installs and starts the Hyperledger instance or by manually downloading and copying the projects into the `chaincode` folder.
To add instructions to the shell script, add the following 3 lines before the "`chaincode install`" command

```
docker-compose -f ./docker-compose.yml up -d cli

+ docker exec cli go get github.com/cespare/xxhash
+ docker exec cli go get github.com/pkg/errors
+ docker exec cli go get github.com/iotaledger/iota.go
+
docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=,
docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=,
```

This will ensure that required packages are downloaded and placed into the `chaincode` folder, mapped as `github.com`
Depending on your version of Go, the last command that downloads the iota.go library might fail. In this case, you can download it manually and place it into the `chaincode` folder.



Next, add import of the IOTA connector library to the list of imports of your `chaincode.go` file. Please note, `gibhub.com` namespace in from of the **`iota`** package name. If your organization uses a different namespace, please adjust accordingly.

```
  import (
        "encoding/json"

        "github.com/hyperledger/fabric/core/chaincode/shim"
        sc "github.com/hyperledger/fabric/protos/peer"
+       "github.com/iota"
  )
```

Digital Twin

Hyperledger chaincode contains a function called `initLedger`, where you can define an initial structure of the ledger and store data as key/value pairs. For complex types, you would typically use structures to describe object fields.

In our example, the initial ledger structure consists of a number of container definitions, where each container includes the Holder field among many others

```
containers := []Container{
        Container{Description: "Apples", Location: "67.0006, -70.5476", Timestamp: timestamp, Holder: "Producer"},
        Container{Description: "Oranges", Location: "91.2395, -49.4594", Timestamp: timestamp, Holder: "Freight Forwarder"},
        Container{Description: "Avocados", Location: "58.0148, 59.01391", Timestamp: timestamp, Holder: "Customs"},
```

Once the structure is defined, it is stored on a ledger one by one

```
i := 0
for i < len(containers) {
        containerAsBytes, _ := json.Marshal(containers[i])
        APIstub.PutState(strconv.Itoa(i+1), containerAsBytes)

        fmt.Println("New Asset", strconv.Itoa(i+1), containers[i])

        i = i + 1
}
```

This is one possible way to store data on a ledger. We do not claim to develop a best possible option.

We will add the IOTA connector here, to ensure that the digital twin is stored on the Tangle in MAM message streams. We will create an individual message stream for each container, where all changes to the container data, location, holder etc. will be tracked.

First, define the structure of the IOTA MAM stream. It should contain randomly generated Seed, MamState object, Root address to fetch data from, stream mode and sideKey (encryption key).

```
56  + type IotaPayload struct {
57  +       Seed        string `json:"seed"`
58  +       MamState    string `json:"mamState"`
59  +       Root        string `json:"root"`
60  +       Mode            string `json:"mode"`
61  +       SideKey     string `json:"sideKey"`
62    }
```

Then, define mode and encryption key for every asset, that will get a digital twin on the Tangle.
You can use one of the following modes: “public”, “private”, “restricted”.
SideKey (encryption key) is only required for the “restricted” mode. Otherwise, it can remain an empty string.
We provide two helper functions for the `sideKey`. Both functions are accessible from the `iota` namespace:

- `iota.GenerateRandomSeedString(length)` will generate a random string of the given length. It can be used as seed or encryption key.
- `iota.PadSideKey()` will automatically adjust the length of the short key to 81 characters.

If you do not want to define your own values for mode and sideKey, you can use default values `iota.MamMode` and `iota.MamSideKey`, which you can inspect and modify under `chaincode/iota/config.go`

After that, you can call a function iota.PublishAndReturnState(), which will publish the message payload as a new Mam channel. This function returns the mamState, root and seed values.
MamState and seed are needed for further appends to the same channel. Root value is used to read data from the channel.
These values need to be stored on the ledger and communicated to each peer of the organisation.
If you do not want to append new messages to the same channel in the future, you can call `iota.Publish()` function instead, which won't return the MamState.

`iota.PublishAndReturnState()` requires the following parameters:

- Message payload (string)
- Existing Mam stream flag (bool)
- Existing seed value (string)
- Existing MamState value (string)
- Mode (string)
- sideKey (string)

If you create a new Mam stream, set values for existing Mam stream, seed and mamState to `false, "", ""`

Once the message was published, it is time to persist values on the ledger. Create a new object of type IotaPayload and put it on the ledger, similarly as you created records for container assets.
Please note that we recommend to add a prefix like "IOTA_" in front of the asset ID.
`APIstub.PutState("IOTA_" + strconv.Itoa(i+1), iotaPayloadAsBytes)`

```
        i := 0
        for i < len(containers) {
                containerAsBytes, _ := json.Marshal(containers[i])
                APIstub.PutState(strconv.Itoa(i+1), containerAsBytes)

-               fmt.Println("New Asset", strconv.Itoa(i+1), containers[i])
+               // Define own values for IOTA MAM message mode and MAM message encryption key
+               // If not set, default values from iota/config.go file will be used
+               mode := iota.MamMode
+               sideKey := iota.PadSideKey(iota.MamSideKey) // iota.PadSideKey(iota.GenerateRandomSeedString(50))
+
+               mamState, root, seed := iota.PublishAndReturnState(string(containerAsBytes), false, "", "", mode, sideKey)
+               iotaPayload := IotaPayload{Seed: seed, MamState: mamState, Root: root, Mode: mode, SideKey: sideKey}
+               iotaPayloadAsBytes, _ := json.Marshal(iotaPayload)
+               APIstub.PutState("IOTA_" + strconv.Itoa(i+1), iotaPayloadAsBytes)
+
+               fmt.Println("New Asset", strconv.Itoa(i+1), containers[i], root, mode, sideKey)

                i = i + 1
        }
```

If your smart contract contains a function to add new records to the ledger, please update this function by adding the IOTA connector code to it.

Please note that the ID of the new asset is used for the new IOTA object
`APIstub.PutState("IOTA_" + args[0], iotaPayloadAsBytes)`

```
-       fmt.Println("New Asset", args[0], container)
+       // Define own values for IOTA MAM message mode and MAM message encryption key
+       // If not set, default values from iota/config.go file will be used
+       mode := iota.MamMode
+       sideKey := iota.PadSideKey(iota.MamSideKey) // iota.PadSideKey(iota.GenerateRandomSeedString(50))
+
+       mamState, root, seed := iota.PublishAndReturnState(string(containerAsBytes), false, "", "", mode, sideKey)
+       iotaPayload := IotaPayload{Seed: seed, MamState: mamState, Root: root, Mode: mode, SideKey: sideKey}
+       iotaPayloadAsBytes, _ := json.Marshal(iotaPayload)
+       APIstub.PutState("IOTA_" + args[0], iotaPayloadAsBytes)
+
+       fmt.Println("New Asset", args[0], container, root, mode, sideKey)
```

Similarly, if your smart contract contains a function to modify existing records, please update this function by adding the IOTA connector code to it as well.
Please note that in this case, we are appending the modified asset data to the existing Mam stream of this asset. Therefore we retrieve the `iotaPayload` for specific assets from the ledger, and communicate this information to the `iota.PublishAndReturnState()` function, along with the existing Mam stream flag set to **`true`**.
Also note, at this point, the mode and encryption key of the existing Mam stream can not be changed. Please use existing values stored on the ledger.

Once the new message was added to the existing stream, the new MamState should replace the previous state on the ledger. All other values should remain the same as before.

```
        container.Holder = args[1]

        timestamp := strconv.FormatInt(time.Now().UnixNano() / 1000000, 10)
        container.Timestamp = timestamp

        containerAsBytes, _ = json.Marshal(container)
        err := APIstub.PutState(args[0], containerAsBytes)
        if err != nil {
                return shim.Error(fmt.Sprintf("Failed to change container holder: %s", args[0]))
        }

+       iotaPayloadAsBytes, _ := APIstub.GetState("IOTA_" + args[0])
+       if iotaPayloadAsBytes == nil {
+               return shim.Error("Could not locate IOTA state object")
+       }
+       iotaPayload := IotaPayload{}
+       json.Unmarshal(iotaPayloadAsBytes, &iotaPayload)
+
+       mamState, _, _ := iota.PublishAndReturnState(string(containerAsBytes), true, iotaPayload.Seed, iotaPayload.MamState, iotaPayload.Mode, iotaPayload.SideKey)
+       iotaPayloadNew := IotaPayload{Seed: iotaPayload.Seed, MamState: mamState, Root: iotaPayload.Root, Mode: iotaPayload.Mode, SideKey: iotaPayload.SideKey}
+       iotaPayloadNewAsBytes, _ := json.Marshal(iotaPayloadNew)
+       APIstub.PutState("IOTA_" + args[0], iotaPayloadNewAsBytes)
```

If your smart contract contains a function to query a specific record from the ledger, you might want to also fetch and return the state stored on the Tangle. To perform the query from the Tangle, you can use the following function available from the IOTA connector code:
`iota.Fetch(root, mode, sideKey)`

Please note that the ID of the asset is used to retrieve the corresponding IOTA object from the ledger.

`iotaPayloadAsBytes, _ := APIstub.GetState("IOTA_" + args[0])`

```
+       iotaPayloadAsBytes, _ := APIstub.GetState("IOTA_" + args[0])
+       if iotaPayloadAsBytes == nil {
+               return shim.Error("Could not locate IOTA state object")
+       }
+       iotaPayload := IotaPayload{}
+       json.Unmarshal(iotaPayloadAsBytes, &iotaPayload)
+
+       // IOTA MAM stream values
+       messages := iota.Fetch(iotaPayload.Root, iotaPayload.Mode, iotaPayload.SideKey)
+
        out := map[string]interface{}{}
        out["container"] = container
+       out["messages"] = strings.Join(messages, ", ")
```

Fetched messages are returned as an array of strings. If you want to join them together into one string and add to the output object, you will also need to import the `"strings"` Go package.

In addition, if you want to output the MamState values, to be able to perform other actions on the UI, like confirm and compare data from the ledger with data stored on the Tangle, you can add the MamState object to the output

```
+       mamstate := map[string]interface{}{}
+       mamstate["root"] = iotaPayload.Root
+       mamstate["sideKey"] = iotaPayload.SideKey
        out := map[string]interface{}{}
        out["container"] = container
+       out["mamstate"] = mamstate

        result, _ := json.Marshal(out)
```

Sending payments

Supply chain projects might benefit from the built-in payment solution, where payments for certain services and goods can be sent between supply chain participants. One possible use case could be when retailers or end consumers send payments to the producers, logistics and fulfillment providers for the ordered assets.

Since none of the Hyperledger projects support cryptocurrency or any other type of payments, IOTA connector can be used to perform feeless payments between participants at the moment where a smart contract confirms successful transaction.
To send payment using the IOTA wallet, you will need to store `wallet` seed and `keyIndex` on the ledger. `Seed` is used to initiate a transaction, and `keyIndex` is specific for IOTA implementation and represents the index of the current wallet address, which holds the tokens. After every outgoing payment transaction, the remaining tokens are transferred to the next address in order to prevent double-spending. The index of the new address (called `remainderAddress`) should be stored on the ledger and used for the next outgoing payment. Incoming payments do not trigger address or index change.

In the example, we will maintain only one wallet for outgoing payments. This wallet will be assigned to the retailer, who is the end consumer of the asset in this supply chain project. The payment will be sent to the previous asset holder each time the holder is changed, which indicates asset movement towards the end consumer. In other words, once a producer prepare container for shipment and transfer it over to a freight forwarder, the retailer will pay to the producer in IOTA Tokens. Then, once freight forwarder delivers the container to the next destination, the retailer will transfer IOTA tokens to pay for this service.

All possible participants in this sample of a supply chain project are defined upon initialization of the ledger. For simplicity, we assume that there is only one participant with the role of “Producer”, “Shipper” and so on.

```
type Participant struct {
	Role string `json:"role"`
	Description string `json:"description"`
}

func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface) sc.Response {
	participants := []Participant{
		Participant{Role: "Producer", Description: "Farmer / Goods producer"},
		Participant{Role: "Freight Forwarder", Description: "Logistics"},
		Participant{Role: "Customs", Description: ""},
		Participant{Role: "Shipper", Description: ""},
		Participant{Role: "Distributor", Description: "Fruits distributor"},
		Participant{Role: "Retailer", Description: "Large grocery store"},
	}

	for i := range participants {
		participantAsBytes, _ := json.Marshal(participants[i])
		APIstub.PutState(participants[i].Role, participantAsBytes)
	}
```

We will start with the definition of the structure for the wallet object

```
+ type IotaWallet struct {
+       Seed        string `json:"seed"`
+       Address     string `json:"address"`
+       KeyIndex    uint64 `json:"keyIndex"`
+ }
+
```

This structure contains `seed` and `keyIndex` as described above. In addition it also contains the actual address where Tokens are currently stored. You can perform balance check to ensure sufficient balance of the wallet. Enter your wallet address on [this page] to check the current balance.

Next, we will extend the existing Participant structure by adding the IotaWallet part into it

```
  type Participant struct {
        Role string `json:"role"`
        Description string `json:"description"`
+       IotaWallet
  }
```

And then we will generate a new empty wallet and add wallet information to every participant record.
To generate a new wallet, you can use a function from the IOTA connector:
`walletAddress, walletSeed := iota.CreateWallet()`

```
    for i := range participants {
+           walletAddress, walletSeed := iota.CreateWallet()
+           participants[i].Seed = walletSeed
+           participants[i].Address = walletAddress
+           participants[i].KeyIndex = 0
            participantAsBytes, _ := json.Marshal(participants[i])
            APIstub.PutState(participants[i].Role, participantAsBytes)
    }
```

Since we generate a new wallet for every record, we set the `keyIndex` value to 0. If you are about to use existing wallets, please adjust `keyIndex` values accordingly.
The generated wallets are empty and currently can only receive IOTA tokens.

In order to send tokens, you need to maintain at least one wallet funded with IOTA tokens. Wallet data of this wallet should be stored on the ledger.
You can provide wallet data upon ledger initialization, or you can modify values in the configuration file under `chaincode/iota/config.go`

chaincode/iota/config.go

```
@@ -0,0 +1,20 @@
+ package iota
+
+ // IOTA Wallet
+ const DefaultWalletSeed = "RTZKOKTX9WMASJMXG9SAIOZGLSA9UBOTG9LQ"
+ const DefaultWalletKeyIndex = 3
```

To store wallet on the ledger, please update the `initLedger()` function by adding the following code. You can replace values for `iota.DefaultWalletSeed` and `iota.DefaultWalletKeyIndex` with respective values of your wallet.

```
+       iotaWallet := IotaWallet{Seed: iota.DefaultWalletSeed, KeyIndex: iota.DefaultWalletKeyIndex, Address: ""}
+       iotaWalletAsBytes, _ := json.Marshal(iotaWallet)
+       APIstub.PutState("IOTA_WALLET", iotaWalletAsBytes)
+
```

Once the wallets are configured, we can add functionality to perform payments. This consists of 3 simple steps:

1. Identify function in your smart contract which should trigger payment.
2. Identify the payment recipient. Retrieve wallet address of the recipient.
3. Identify the payment sender. Retrieve wallet seed and `keyIndex` of the sender. Perform token transfer, then update `keyIndex` to the new value and store it on the ledger.

In our example, we will perform payments once the asset holder was changed. So, the function that triggers payments called `changeContainerHolder()` Payment recipient is the previous container holder. So, we need to preserve the holder value before changing, to be able to retrieve the corresponding wallet data from the ledger.
On the screenshot below you see the required updates to the function. Container data is retrieved based on the provided ID. Before the holder is reassigned, we store the original holder value. Later we query the ledger in order to get the wallet address of the original container holder.

```
        containerAsBytes, _ := APIstub.GetState(args[0])
        if containerAsBytes == nil {
                return shim.Error("Could not locate container")
        }
        container := Container{}

        json.Unmarshal(containerAsBytes, &container)
+       previousContainerHolder := container.Holder
        container.Holder = args[1]

+       // make payment to the participant
+       participantAsBytes, _ := APIstub.GetState(previousContainerHolder)
+       if participantAsBytes == nil {
+               return shim.Error("Could not locate participant")
+       }
+       participant := Participant{}
+       json.Unmarshal(participantAsBytes, &participant)
```

For the step 3 we will request the IOTA wallet data of the retailer. Then we will trigger the following function and submit `seed` and `keyIndex` values of the sender and `address` value of the recipient.

`iota.TransferTokens(seed, keyIndex, address)`

Once this is done, we just need to update `keyIndex` to the new value and store it on the ledger.

```
+       iotaWalletAsBytes, _ := APIstub.GetState("IOTA_WALLET")
+       if iotaWalletAsBytes == nil {
+               return shim.Error("Could not locate wallet data")
+       }
+       iotaWallet := IotaWallet{}
+       json.Unmarshal(iotaWalletAsBytes, &iotaWallet)
+
+       newKeyIndex := iota.TransferTokens(iotaWallet.Seed, iotaWallet.KeyIndex, participant.Address)
+       iotaWallet.KeyIndex = newKeyIndex
+       iotaWalletAsBytes, _ = json.Marshal(iotaWallet)
+       err = APIstub.PutState("IOTA_WALLET", iotaWalletAsBytes)
```

Token transfer usually requires a few seconds in order to be confirmed. Please do not attempt to trigger multiple transfers from the same wallet within a very short timeframe (less than 10 seconds), as it will result into “Invalid balance” error, and wallet keyIndex should be reset to previous value manually.

As always, you can check the status of the token transfer on this page by entering the wallet address.

API Reference

Exported functions

iota.GenerateRandomSeedString()

The GenerateRandomSeedString() method generates a random string of the given length using the following alphabet: "ABCDEFGHIJKLMNOPQRSTUVWXYZ9".

Syntax

```
iota.GenerateRandomSeedString(length)
```

Parameters

- length ***int***
  Desired length of the seed string.

Return values

- seed ***string***
  Randomly generated string of the given length.

Example usage

```
sideKey := iota.GenerateRandomSeedString(50)
```

iota.PadSideKey()

The PadSideKey() method extends short strings with "9" up until the length of 81.

Syntax

```
iota.PadSideKey()
```

Parameters

- shortString ***string***
  Short string that needs to be extended.

Return values

- extendedString ***string***
  String extended with "9" with the length of 81.

Example usage

```
sideKey := iota.PadSideKey(shortKey)
```

iota.Publish()

The Publish() method publishes a given message into a new or existing MAM channel with the given mode and encryption key. Channel state can be reconstructed from the returned transmitter and seed values.

Syntax

```
iota.Publish(message, transmitter, mode, sideKey)
```

Parameters

- message ***string***
  Message to be published on the Tangle.
- transmitter ****mam.Transmitter* | *nil***
  Transmitter object, will be created if **nil** is provided.
- mode ***string***
  MAM channel mode. Possible values: "public", "private", "restricted".
- sideKey ***string***
  Encryption key for MAM message. Can be an empty string.

Return values

- transmitter ****mam.Transmitter***
  Transmitter object, holds the MAM channel state for eventual publishing of the further messages.
- seed ***string***
  Randomly generated seed string of the MAM channel.
- root ***string***
  Root address of the transaction. Used to fetch MAM message.

Example usage

```
transmitter, seed, root := iota.Publish(message, transmitter, mode, sideKey)
channelStateObject := transmitter.Channel()
channelState := iota.MamStateToString(channelStateObject)
```

iota.PublishAndReturnState()

The PublishAndReturnState() method publishes a given message into a new or existing MAM channel with the given mode and encryption key. Channel state is returned to simplify publishing of further messages.

Syntax

iota.PublishAndReturnState(*message, useTransmitter, seed, mamState, mode, sideKey*)

Parameters

- message ***string***
  Message to be published on the Tangle.
- useTransmitter ***bool***
  Flag indicating that existing channel should be used to publish a nem message into. In this case the Transmitter object will be reconstructed from the provided seed and mamState string values. If **false**, new MAM channel will be created.
- seed ***string***
  Seed string of the MAM channel. Empty string if **useTransmitter** flag is **false**.
- mamState ***string***
  Stringified MAM channel state. Empty string if **useTransmitter** flag is **false**.
- mode ***string***
  MAM channel mode. Possible values: "public", "private", "restricted".
- sideKey ***string***
  Encryption key for MAM message. Can be an empty string.

Return values

- mamState ***string***
  Stringified MAM channel state for eventual further messages.
- root ***string***
  Root address of the transaction. Used to fetch MAM message.
- seed ***string***
  Seed string (randomly generated for a new channel) of the MAM channel.

Example usage

New channel

```
mamState, root, seed := iota.PublishAndReturnState(string(payloadAsBytes),
false, "", "", mode, sideKey)

iotaPayload := IotaPayload{Seed: seed, MamState: mamState, Root: root, Mode:
mode, SideKey: sideKey}
iotaPayloadAsBytes, _ := json.Marshal(iotaPayload)
APIstub.PutState("IOTA", iotaPayloadAsBytes)
```

Appending to an existing channel

```
iotaPayloadAsBytes, _ := APIstub.GetState("IOTA")
iotaPayload := IotaPayload{}
json.Unmarshal(iotaPayloadAsBytes, &iotaPayload)

mamState, _, _ := iota.PublishAndReturnState(string(payloadAsBytes), true,
iotaPayload.Seed, iotaPayload.MamState, iotaPayload.Mode,
iotaPayload.SideKey)

iotaPayloadNew := IotaPayload{Seed: iotaPayload.Seed, MamState: mamState,
Root: iotaPayload.Root, Mode: iotaPayload.Mode, SideKey:
iotaPayload.SideKey}
iotaPayloadNewAsBytes, _ := json.Marshal(iotaPayloadNew)
APIstub.PutState("IOTA", iotaPayloadNewAsBytes)
```

iota.Fetch()

The Fetch() method retrieves messages from the MAM channel. Messages of a restricted channel will be decrypted using provided decryption key.

Syntax

```
iota.Fetch(root, mode, sideKey)
```

Parameters

- root ***string***
  Root address of the initial MAM channel transaction.
- mode ***string***
  MAM channel mode. Possible values: "public", "private", "restricted".
- sideKey ***string***
  Decryption key for MAM messages. Can be an empty string.

Return values

- messages **[]*string***
  Array of decoded and decrypted messages from the MAM channel.

Example usage

```
iotaPayloadAsBytes, _ := APIstub.GetState("IOTA")
iotaPayload := IotaPayload{}
json.Unmarshal(iotaPayloadAsBytes, &iotaPayload)

messages := iota.Fetch(iotaPayload.Root, iotaPayload.Mode,
iotaPayload.SideKey)
```

iota.CreateWallet()

The CreateWallet() method generates and returns a new seed and address of an empty IOTA wallet.

Syntax

```
iota.CreateWallet()
```

Return values

- walletAddress ***string***
  Root address of the wallet. Used to receive IOTA tokens and balance check.
- seed ***string***
  Seed string (randomly generated) of the wallet. Used to send IOTA tokens.

Example usage

```
walletAddress, walletSeed := iota.CreateWallet()
```

iota.TransferTokens()

The TransferTokens() method transfers IOTA tokens from the existing wallet to a provided address. Transferred amount can be changed in **iota.DefaultAmount**

Syntax

```
iota.TransferTokens(seed, keyIndex, recipientAddress)
```

Parameters

- seed ***string***
  Seed string of the own wallet.
- keyIndex ***uint64***
  Index of the current unspent address. Starts with 0 for a new wallet, increases by 1 after each outgoing transaction.
- recipientAddress ***string***
  Wallet address of the recipient.

Return values

- keyIndex ***uint64***
  Index of the next unspent address.

Example usage

```
iotaWalletAsBytes, _ := APIstub.GetState("IOTA_WALLET")
iotaWallet := IotaWallet{}
json.Unmarshal(iotaWalletAsBytes, &iotaWallet)

newKeyIndex := iota.TransferTokens(iotaWallet.Seed, iotaWallet.KeyIndex,
participant.Address)

iotaWallet.KeyIndex = newKeyIndex
iotaWalletAsBytes, _ = json.Marshal(iotaWallet)
err = APIstub.PutState("IOTA_WALLET", iotaWalletAsBytes)
```

Exported constants

All values can be changed in `iota/config.go` file

iota.MamMode
Mode of the MAM channel. Possible values: "public", "private", "restricted".
Default value: "public"

iota.MamSideKey
Encryption key for MAM message. Used to encrypt messages in "restricted" mode. Can be an empty string.
Default value: ""

iota.DefaultWalletSeed
Seed string of the default wallet to transfer IOTA tokens from.

iota.DefaultWalletKeyIndex
Index of the current unspent address of the default wallet to transfer IOTA tokens from.

iota.DefaultAmount
Amount that will be sent by default to any recipient.

Non exported constants

All values can be changed in `iota/config.go` file

endpoint

IOTA provider URL. All IOTA-related requests will be directed to this endpoint. Can point to a Mainnet node, Devnet node or a private Tangle node.

mwm

Difficulty of Proof-of-Work required to attach transaction to the Tangle.
Minimum value on mainnet & spamnet is `14`, `9` on Devnet and private nets.
Default value: 9

depth

Depth or how far to go for tip selection entry point. How many milestones back to start the random walk from.
Default value: 3

transactionTag

Default tag added to every data transaction (MAM channel message).
Default value: "HYPERLEDGER"

Testing and Deployment

Setting up a Demo Hyperledger Fabric Application

The demo project used to showcase the IOTA Connector integration is open-source and can be found in this GitHub repository.

This application demonstrates the creation and transfer of container shipments between actors leveraging Hyperledger Fabric in the supply chain. In this demo app we will set up the minimum number of nodes required to develop chaincode. It has a single peer and a single organization.

This code is based on code written by the Hyperledger Fabric community. Source code can be found here: (https://github.com/hyperledger/fabric-samples).

Starting the Application

1. Install the required libraries from the package.json file
   ```
   yarn
   ```

2. Set GOPATH variable
   ```
   export GOPATH=$GOPATH:~/go:~/go/src
   ```

3. Install Go dependencies
   ```
   go get github.com/cespare/xxhash
   go get github.com/pkg/errors
   go get github.com/iotaledger/iota.go/address
   go get github.com/iotaledger/iota.go/api
   go get github.com/iotaledger/iota.go/bundle
   go get github.com/iotaledger/iota.go/consts
   go get github.com/iotaledger/iota.go/mam/v1
   go get github.com/iotaledger/iota.go/pow
   go get github.com/iotaledger/iota.go/trinary
   ```

4. Start the Hyperledger Fabric network
   ```
   ./startFabric.sh
   ```

5. Register the Admin and User components of our network
   ```
   node registerAdmin.js
   node registerUser.js
   ```

6. Start the client application
   ```
   yarn dev
   ```

7. Load the client by opening localhost:3000 in any browser window of your choice, and you should see the user interface for our simple application at this URL.

Testing

You can test the chaincode by invoking any of its functions from the inside the Docker container or from the outside of it.

To invoke the chaincode function from the outside, run the following command and provide parameters to the function as an array of strings:

```
docker exec cli peer chaincode invoke -C mychannel -n
fabric-demo-app -c '{"Args":["changeContainerHolder", "1",
"Thomas"]}'
```

To invoke the chaincode inside the Docker container, do the following:

1. Run
   `docker exec -it cli bash`

2. Call a chaincode function
   `peer chaincode invoke -C mychannel -n fabric-demo-app -c '{"Args":["changeContainerHolder", "1", "Thomas"]}'`

Logging

The logs for chaincodes are in their respective Docker containers. To inspect logs for a chaincode called fabric-demo-app at version 1.0 on peer0 of org and see the output of any fmt.Print*s, run the following command:

```
docker logs -f dev-peer0.org1.example.com-fabric-demo-app-1.0
```

Updating the chaincode

1. Stop chaincode Docker container
2. Remove chaincode Docker container
3. Remove chaincode Docker container image
4. Run `./startFabric.sh` do create and deploy new chaincode

Cleanup

1. Stop existing HL Fabric Instance

```
cd basic-network
./stop.sh
./teardown.sh
```

2. Remove any pre-existing containers and images, as it may conflict with commands in this project

```
docker image rm -f XXXXXXXXXXXX
docker rm -f $(docker ps -aq)
```

3. Remove key store contents

```
cd ~ && rm -rf .hfc-key-store/
```

Troubleshooting

In case of permission error while running `./startFabric.sh` execute the following:

```
chmod a+x startFabric.sh
```

Glossary Of Terms

- MAM - Masked Authentication Messaging is a second layer data communication protocol which adds functionality to emit and access an encrypted data stream, like RSS, over the Tangle https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e

Resources

- The source code of the **IOTA Connector** can be downloaded from this open-source GitHub repository.
- The supplementary iota.go library, that need to be located within the same chaincode folder, can be downloaded from this open-source GitHub repository.
- The demo project used to showcase the IOTA Connector integration is open-source and can be found in this GitHub repository.