DINGO FABRIC CHAINCODE GUIDE AND SUMMARY

v1.0

Author: Calum Oke
QUT TEAM: UNHANDLED EXCEPTIONS

Overview **2**

Development Stage **6**

Methods 6

initLedger 6

updateDateInService 7

changeComponent 7

changeComponentToDiffAsset 7

queryAllComps 8

queryAllAssets 8

queryAllObs 8

queryAllHashes 9

queryObj 9

exportData 9

Operation 10

Chaincode Installation 11

Chaincode Instantiation 11

Chaincode Invocation 12

Chaincode Querying 13

REST API and Client Side **14**

Endpoint Access and Actions 14

Methods 15

Errors Encountered **16**

Inputting Data to Ledger 16

Exporting Ledger Data 16

Fabric Software Development Kit for REST API 16

Potential Issues and Considerations **17**

Scaling 17

51% Attack 17

Possible Extensions **18**

Embed Into Current Trakka System 18

Design Improvement/Front-End Development 18

References **19**

Overview

Chaincode is the foundation of smart contracts which can be written in Golang (Go) or Node.js (javascript). It allows the user to perform functions via chaincode arguments and interact with the ledger/blockchain. The purpose of utilising chaincode in this project is to provide data integrity by posting data to the chaincode ledger and retrieving it back to an external source. The ledger emulates a database which can be accessed or manipulated by the chaincode using CRUD operations.

When a client wants to POST an 'Asset' to the ledger, it must be permitted through a channel and to a peer with the chaincode installed into it (within the channel). The client will send the data they want to go through the ledger via a REST endpoint. At this point depending on the Fabric network configuration, generally the client would be authenticated by the peer's IP address, username and password to pass the data through to the chaincode. Then the chaincode inserts the new Asset into the ledger. A very similar process is undergone for GET'ing the data. Except the chaincode will retrieve the data entry from the ledger and pass it through to the endpoint that displays all data.

Each channel has its own ledger, which is copied to each peer node that the chaincode is installed on. This makes it a decentralised and distributed, improving the data integrity and trust. If data is tampered with on one node to the ledger, then the rest of the nodes can be compared with on their ledgers and confirm the legitimate data. The orderer confirms that the data is valid with its authority using the certificate file, which completes non-repudiation.

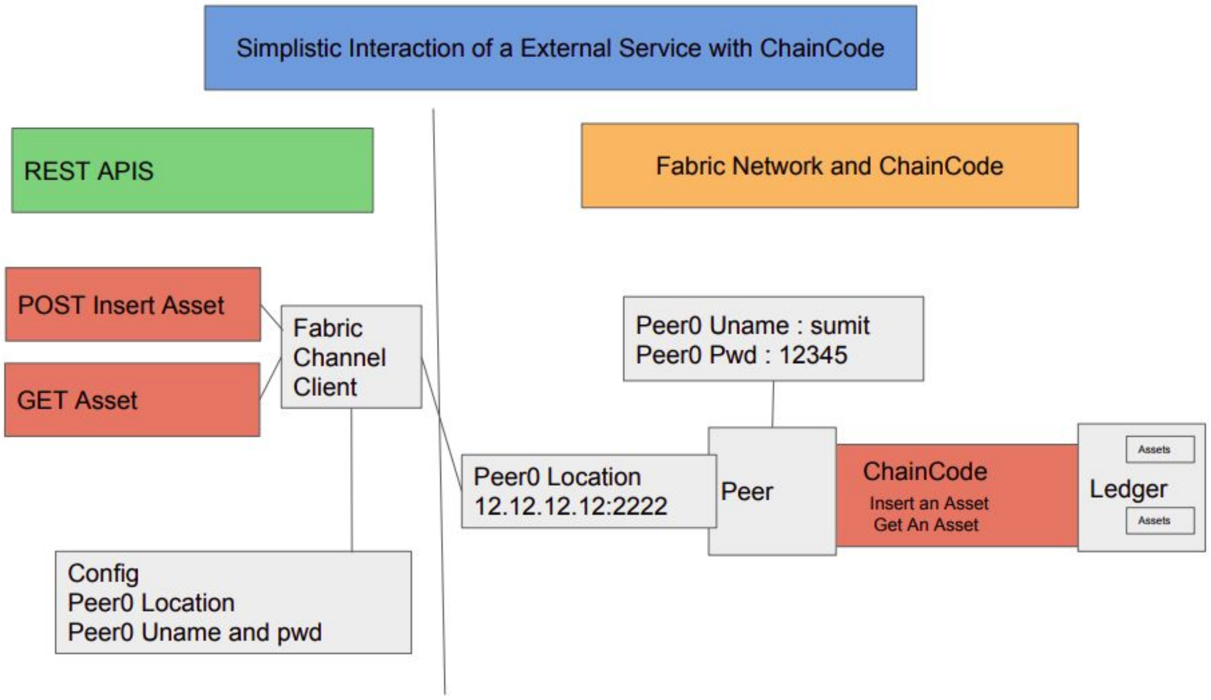

Figure 1.1: Diagram of data flow utilising Hyperledger Fabric networking system

The following diagram below depicts the difference in processing time between both REST API endpoints and the possible window-time for a man-in-the-middle (MITM) attack

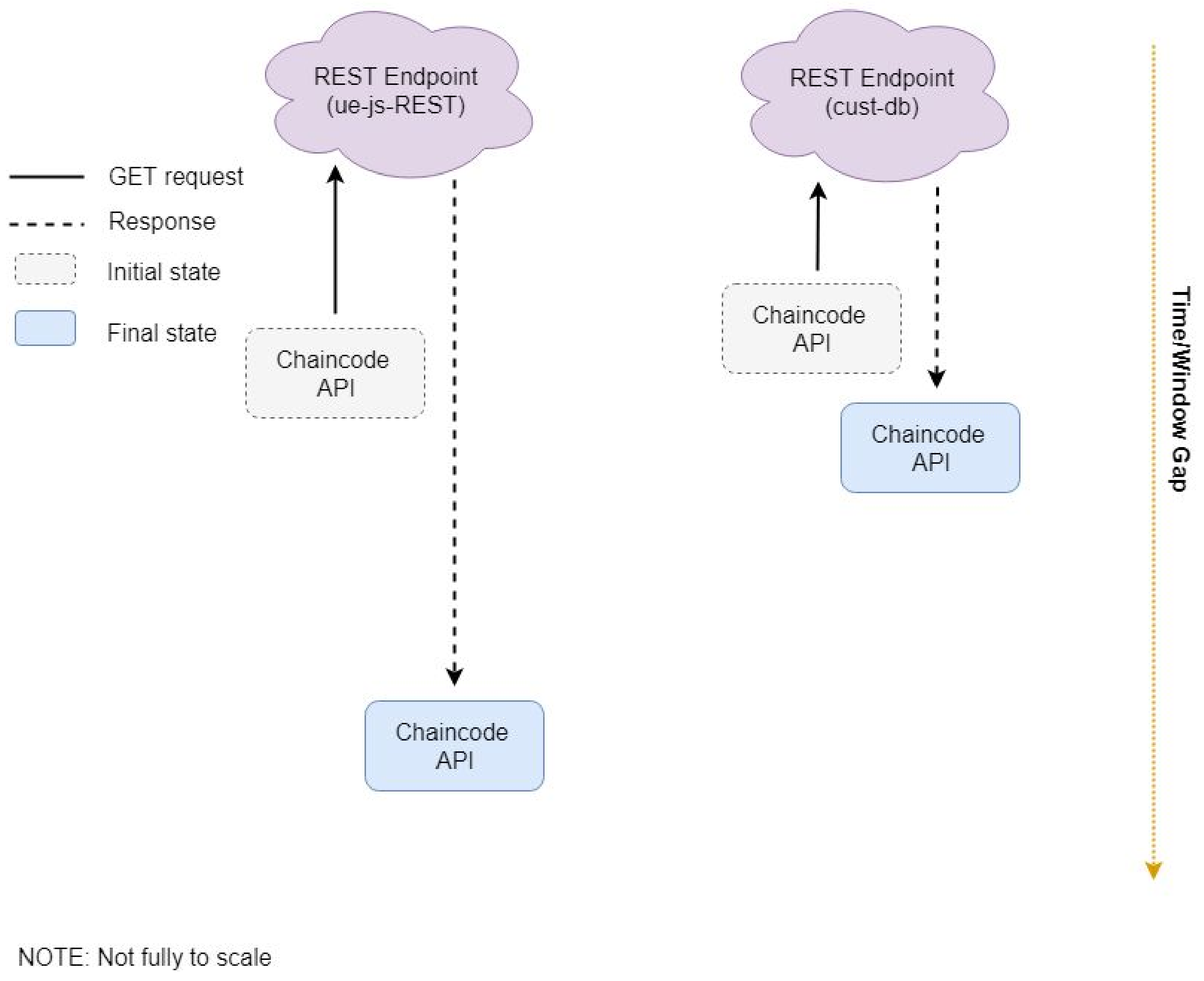

Figure 1.2: Diagram of requesting time length based on load

As the data load increases on both endpoints, the proportion of extraction time from the ue-js-REST compared to cust-db largely increases, leaving a way more timeframe of vulnerability. Due to this fact, it was much more ideal that the hashes were created before pulling them into the ledger, reducing latency by a high amount.

[**NOTE:** ue-js-REST is now mock-trakka due to change in Node host servers]

The end configuration of the project was most definitely not the ideal solution for how the project should have been laid out. By description the data flow of the servers might be confusing to understand, so the following diagram depicts a visual representation of how the architecture is setup

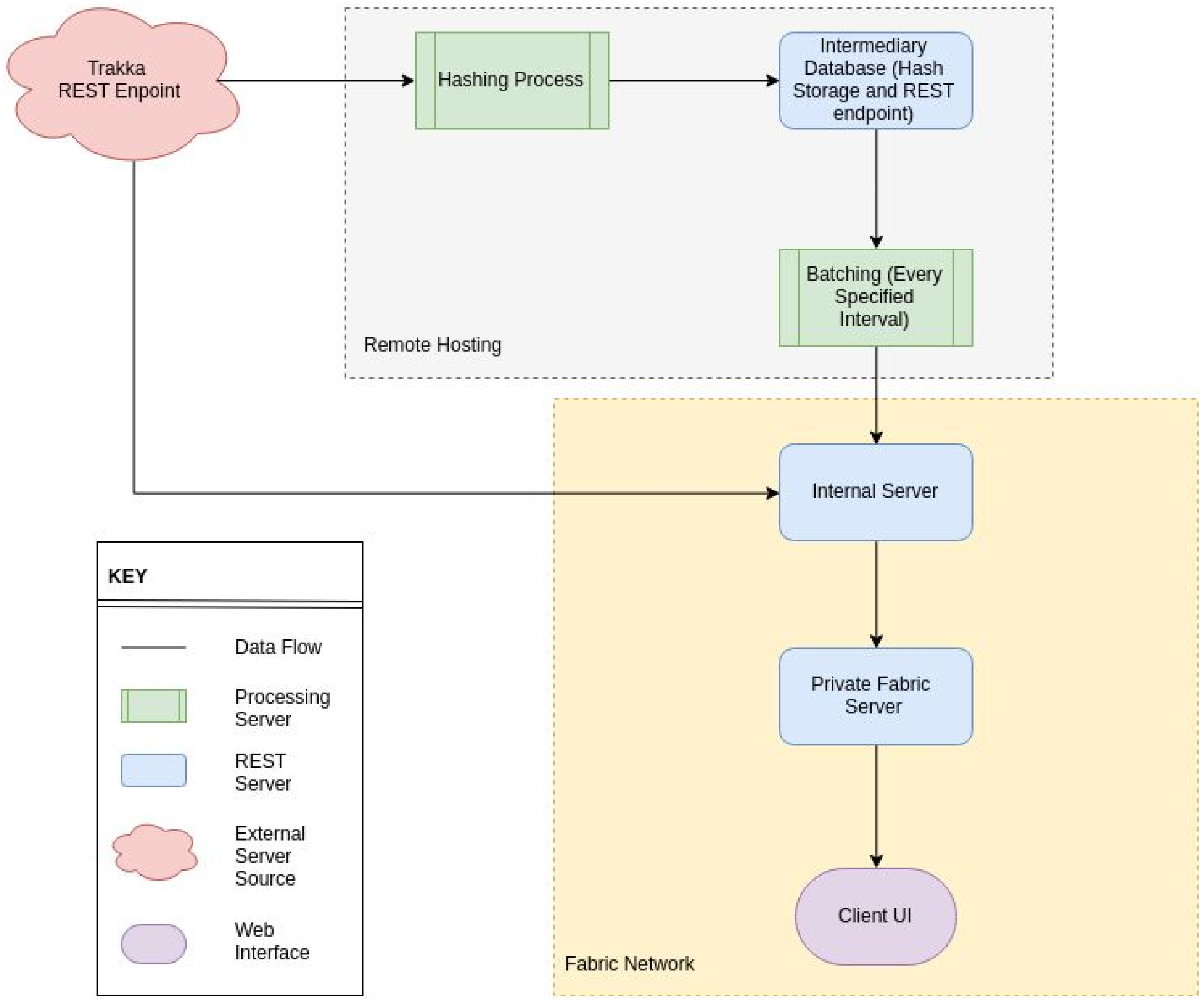

Figure 1.3: Diagram of data flow through servers

In a ideal solution, the Internal Server (Golang web_app) could have more routes that GETs and POSTs data to and from the ledger, which then eliminates the unnecessary Fabric Server. However, it was difficult to find an appropriate MongoDB plug-in in Go, so the Fabric Node server was created. The client UI could even access the chaincode routes directly, and make that more complex. This is all depending on how you would want to configure the system, so it is fairly subjective.

Development Stage

The development starts with building the Fabric network and chaincode functionality to make sure chaincode operations are completed semantically correct. Each chaincode function is what would be executed when the user accesses a 'route' which has a function assigned to it in the REST web application.

Using fabric-shim version (branch) *release-1.2*

Methods

Refer to the GoDocs for all method summaries and descriptions call **chaincode_documentation.html**

This will mainly focus on the chaincode command structure to execute these functions.
To view it so that clickable links work, run the following command

```
godoc -http=:6060
```

Then go to the link to view all methods:

http://localhost:6060/pkg/fab_dingo/?m=all

If you do not have Golang installed, visit the website for instructions (it's a bit complex). Otherwise open the file through a text editor to view the source code.

initLedger

Retrieves all the data from the external source and writes them to the ledger. This would be used by the REST API to POST and GET data from the ledger within chaincode. In CLI however, the test command is

```
peer chaincode invoke -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["initLedger"]}'
```

updateDateInService

The date in service represent’s when an Asset was last serviced. The data should be written in the format: YYYY-MM-DDThh:mm:ss where T is the literal value that denotes time. The first argument takes an Asset’s Key where N is the ID number and the second takes a string in date formatting

```
peer chaincode invoke -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c
'{"Args":["updateDateInService","ASSET<N>","<NEW_DATE>"]}'
```

changeComponent

The first argument takes the Component Key where N is the ID number and the second argument requires a string which would be the component product (e.g. Petrol Tank)

```
peer chaincode invoke -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c
'{"Args":["changeComponent","COMP<N>","<COMP_NAME>"]}'
```

changeComponentToDiffAsset

The first argument takes an Component Key where N is the Key number, and the following argument takes an AssetID where the AssetID refers to the Asset object that Component belongs to

```
peer chaincode invoke -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c
'{"Args":["changeComponentToDiffAsset","COMP<N>","<AssetID>"]}'
```

queryAllComps

Retrieves all Component objects which contain their Key value, and the object itself under the Record field

```
peer chaincode query -o orderer.dingo.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["queryAllComps"]}'
```

queryAllAssets

Retrieves all Asset objects which contain their Key value, and the object itself under the Record field

```
peer chaincode query -o orderer.dingo.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["queryAllAssets"]}'
```

queryAllObs

Retrieves all Observation objects which contain their Key value, and the object itself under the Record field

```
peer chaincode query -o orderer.dingo.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["queryAllObs"]}'
```

queryAllHashes

Retrieves all Hashes objects which contain their Key value, and the object itself under the Record field

```
peer chaincode query -o orderer.dingo.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["queryAllHashes"]}'
```

queryObj

Retrieves a single JSON object based on the Key provided as the argument (e.g. ASSET1)

```
peer chaincode query -o orderer.dingo.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["queryObj","<KEY>"]}'
```

exportData

POSTS the Hashes in the Ledger data in all it latest states to the private Fabric server (http://fab-priv-srv.glitch.me/hashes).

NOTE: This is not the proper way of doing this. The REST API is designed to POST and GET ledger data via proper procedure. This is purely for testing in development mode

```
peer chaincode invoke -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["exportData"]}'
```

Operation

Once the chaincode has been implemented and built, it can now be interacted with via the CLI Docker container (refer to Dingo Network configuration guide to see how to add chaincode to the container volume). Make sure the Dingo channels have been created by running (This is assuming you're in the directory **fabric/dingobc/**). The **s** flag specifies which database to use for storing state (so it does not store in memory).

```
./dingofabric.sh up -s couchdb
```

If you have run that before without resetting channels after any changes, run the command below to reset the channels again. If you have changed anything in the chaincode and wnat to perform the new changes, then run the following command after the first line. After that, run the above command again.

```
../resetchannel.sh
# Only run below if chaincode changes were made
../remove_all_images.sh
```

Next, check to see if the 'cli' and peer node containers are present by running

```
sudo docker ps -a
```

If the container is not present, then reset and remove all containers and images again. Next we need to enter the CLI container, run the following command to execute the container and use the bash

```
sudo docker exec -it cli /bin/bash
```

Once inside the container, the user can perform two main peer chaincode operations

- peer chaincode **install***
- peer chaincode **instantiate***
- peer chaincode **invoke**
- peer chaincode **query**

***NOTE: install** and **instantiate** are called in **dingofabric.sh**, so the you only have to use the **invoke** and **query** commands, however we will go over all of them anyway.

While in the CLI container, you should open another terminal to view the debugging of the chaincode operations for the peer it is installed on. In this case, the following command enters the peer chaincode container and views all logs that are happening

```
sudo docker logs -f dev-peer0.client1.dingo.com-dcc-1.0
```

Where the flag (alternatively written as **--follow**) runs the logging in real-time.

Chaincode Installation

Firstly, the chaincode must be installed from the executable in order to create the chaincode interaction. The following command installs the chaincode to the fabric network. We will break down each flag and arguments

```
peer chaincode install -n dcc -v 1.0 -p
github.com/chaincode/dingo/go/fab-dingo/
```

- **-n**: specifies the smart contract/chaincode name, in this case dcc (Dingo Chaincode) is the argument
- **-v**: the arbitrary versioning, in this case 1.0 is used
- **-p**: the path to the directory that the executable and go file is located in (this directory is in the cli container)

Chaincode Instantiation

Next the installed chaincode must be instantiated to be able to continue any interaction, which calls the **Init** inbuilt function declared in the chaincode. Use the following command to do so

```
peer chaincode instantiate -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n dcc -l
${LANGUAGE} -v 1.0 -c '{"Args":["init"]}' -P "AND ('Client${ORG}MSP.peer')"
```

- **-o**: designates the orderer of the network
- **--tls**: enables TLS protocol on the packets (use **true** if environment variable does not exist)
- **--cafile**: add path to the orderer's certificate file (use **/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cert.pem** if environment variable does not exist)
- **-C**: the channel name the chaincode is instantiated in (use **dingochannel1** if environment variable does not exist)
- **-n**: the name of the chaincode

- **-l**: the language of the chaincode (use **golang** if environment variable does not exist)
- **-v**: versioning of chaincode
- **-c**: any arguments that could specified in the Init parameters (alternatively nothing could have been added)
- **-P**: provides the client to be able to use the chaincode in that channel along with a boolean where AND means they need all clients, and OR allows one or more specified clients

Chaincode Invocation

After the instanciaton is successful, we can now interact with the chaincode's functions via chaincode invoking. This calls the other inbuilt function **Invoke** where you can provide a valid function name and appropriate parameters (if any) and manipulate data. However the first function that should be called is **initLedger** to initiate the ledger with all relevant data. The following command below initiates the ledger, which pulls data from the private Fabric server and creates a ledger with all pulled data

```
peer chaincode invoke -o orderer.dingo.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["initLedger"]}'
```

It recommended by IBM and Hyperledger to create the ledger in a custom function that's NOT the **Init**, as it is good practise. From here you should get a "Status 200 OK" message with a payload saying initialisation is successful. Now that all your assets, components, hashes and observations have been completed, you can now perform other invokable functions such as transfering component ownership to a different asset, or updating the service date and etc.

NOTE: These flags with arguments must be called with every invoke function for permissions to execute the chaincode.

Chaincode Querying

To perform searches that doesn’t change the state of any objects, it is best to use the **query** command instead of invoking. Below is an example of how to do a query of all assets

```
peer chaincode query -o orderer.dingo.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizati
ons/dingo.com/orderers/orderer.dingo.com/msp/tlscacerts/tlsca.dingo.com-cer
t.pem -C dingochannel1 -n dcc -c '{"Args":["queryAllAssets"]}'
```

This command gets all the assets that are in the ledger AFTER the **initLedger** function is called.

REST API and Client Side

The REST API endpoints is how the client performs these chaincode operations in the business logic layer. Each endpoint will have a route function which performs some chaincode operation based on the type of request method (e.g. GET, POST, PUT and etc.).

As explained previously, the GET request will be used to retrieve data from the ledger based on the endpoints url (e.g. "/assets" which will get all Assets in the ledger). The POST request will insert new data into the ledger, usually using the URL that gets all the data of that type. PUT request will also be used to update a data field in the object. DELETE would also be used remove unused or unwanted data that is no longer being utilised for some benefit. Restrictions should strongly be considered on write actions, as you would not want data integrity compromised from minor configuration errors.

Endpoint Access and Actions

Below depicts how a client would interact with the ledger in our current system with the REST API endpoints (or would if completed)

Request Method	**Route**	**Chaincode Function**	**Function Description**
GET	/assets	queryAllAssets	Retrieves all Asset objects and displays them in JSON
POST	/assets	N/a	Inserts a single Asset object to the ledger
GET	/assets/{AssetID}	queryObj {Asset_Key}	Retrieves the Asset object that contains the inputted Asset ID
PUT	/assets/{AssetID}/ {DateInService}	updateDateInService	Updates the Date In Service field for the Asset Id specifed with the new Date in the second argument
GET	/components	queryAllComps	Retrieves all Component objects and displays them in JSON
POST	/components	N/a	Inserts a single Component object to the ledge
GET	/components/ {ComponentID}	queryObj {Component_Key}	Retrieves the Component object that contains the inputted Component ID

PUT	/components/{ComponentID}/{Component}	changeComponent	Updates the component's component product for the Component ID specified with the component in the second argument
PUT	/components/{ComponentID}/{AssetID}	changeComponentToDiffAsset	Updates the Component's Asset ID to another (existing Asset) Asset to simulate a component swap in Assets
GET	/observations	queryAllObs	Retrieves all Observation objects and displays them in JSON
POST	/observations	N/a	Inserts a single Observation object to the ledger
GET	/observations/{ObservationID}	queryObj {Observation_Key}	Retrieves the Observation object that contains the inputted Observation ID
GET	/hashes	queryAllHashes	Retrieves all Hash objects and displays them in JSON
POST	/hashes	N/a	Inserts a single Hash object to the ledger
GET	/hashes/{HashID}	queryObj {Hash_Key}	Retrieves the Hash object that contains the inputted Hash ID

Methods

Like methods for the chaincode, when running the godoc for the web_app package, visit the link below to view the API documentation in full with functional links:

http://localhost:6060/pkg/web_app/?m=all

Else open the **web_app_documentation.html** to view it locally, and open the source file in a text editor to view the source code.

Errors Encountered

Inputting Data to Ledger

When specifying “using couchdb” (-s couchdb), initLedger would not pull data from the API endpoint and put into the couchdb0 container. This was due to the incompatible JSON data fields (“_id” and “__v”), which couchdb cannot use as it has it’s own “_id” and “_rev” fields. Fixed this by not including those two fields in the struct types

Exporting Ledger Data

Invokable Chaincode function “exportData” wouldn’’t POST the data to the private Fabric Server endpoints for client side to retrieve. This was because the Hash objects were being sent over as bytes and not encoded in JSON format properly. This issue ended being solved by ‘marshalling’ the Hash JSON object into bytes, and POSTing that over to the private Fabric Server.

The private Server would then decode it back into the JSON objects. This is only for development purposes, this would not be how it would properly work through clients. Alternatively, the database can manually be exported and sent to the host OS from the container, and sent over to the client side for the client side to read the data from the DB file directly (Not ideal solution at all)

Fabric Software Development Kit for REST API

After thorough researching and trialing with the REST API related packages, unfortunately a REST API could not be constructed to invoke the chaincode. The package layout is very complex and not well documented at all, making it super difficult to understand, due to being such a new technology. The incomplete web application (*web_app*) simulates what the chaincode should complete with the ledger data.

The three packages below is what was understood to be able to create the REST API successfully, however if continuing this project, more should be considered:

- *fabric-sdk-go/pkg/fabsdk*
- *fabric-sdk-go/pkg/client/channel*
- *fabric-sdk-go/pkg/client/ledger*

Refer to the GoDocs to see the route and functions

Potential Issues and Considerations

Scaling

Scaling (Scaling up or scaling out), could potentially be a huge issue on a large business scale basis. If a large amount of clients are requesting data from the REST API, with large amounts of data load to be retrieved, the server is at risk of crashing and down time.

To assist with this, it might involve using micro services to help data load volume being distributed between multiple servers or utilising cloud services (e.g. AWS Load Balancing/Autoscaling groups)

51% Attack

As known, the method of decentralised and distribution of data in this blockchain platform is the amount of peers in a channel, so if a node is cut off, the data can still be retrieved and checked against the other peers. However a problem arises when an attacker (or group of attackers) control 51% or more of the peer nodes within a channel or over multiple channels. From here, attackers can spoof the data for the majority of the nodes, which makes the other nodes with legitimate data be considered 'invalid' or tampered data. This type of attack can be spawned by other vulnerabilites that are exploited and acted upon such as ARP poisoning, uploading reverse shells, DDoS'ing nodes and etc.

Checking the system for these potential vulnerabilities and always keeping on top of antivirus-ware updates should prevent these action from happening. It always helps to know what potential threats are coming out daily and how to prevent the threat manually.

Possible Extensions

Embed Into Current Trakka System

Once the mock Fabric networking system and REST API is fully functioning with testing, then the project could be incorporated into the workplace's system with the Trakka data being inputted into the Ledger. Everyone working there would be a peer and channels can be created with different clients to allow for certain data to be visible to particular client(s).

Additionally with selling, this would increase the probability of a customer buying a component/part as the data can be confirmed to be completely legitimate due to immutability, similarly to Carfax in America. This would also add extra profit to the company as well for each sale.

Design Improvement/Front-End Development

There is currently a lot of opportunity in front-end development with blockchaining, due to the nature of blockchains lacking in UI/visual aspect. Specifically for this project, there is not much visualisation that was implemented when completing the hash checking.

Some suggestions to improve the client side would be to add dynamic feedback while the hash comparison is happening (e.g. progress bar with a spinning mouse pointer) and improving the overall UI. Thorough UX testing could be done with a large number of clients to determine the best way of navigating and performing the hash comparisons.

References

Fabric-shim Repository:
https://github.com/hyperledger/fabric

Tutorial on chaincode construction and interaction:
https://www.youtube.com/watch?v=wKTSteRGVFs

Chaincode Documentation:
https://hyperledger-fabric.readthedocs.io/en/release-1.2/chaincode.html

GoDoc/API Documentation for shim package:
https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim

CouchDB importing/exporting:
http://guide.couchdb.org/draft/security.html
http://www.greenacorn-websolutions.com/couchdb/export-import-a-database-with-couchdb.php

REST API construction:
https://esumit.blog/2018/10/23/how-to-connect-to-hyperledger-fabric-networks-peer-smart-code-from-external-service/
https://github.com/hyperledger/fabric-sdk-go

Fabric Network Documentation:
https://docs.google.com/document/d/1yDa-kYaN_vKn-krejsLvgr5SMHWDIT0MAAtz6-w3YtU/edit

Docker Guide:
https://docs.google.com/document/d/1Du5Us5knoIQxbKL-0Cs5MEXmoj6S66agF70vM3RX2gg/edit

A very special thanks to:
Mark Coleman - Project Offering, Supervising and assisting with theory over the course of the project
Sumit Arora - Strongly assisting with technical aspects of Hyperledger Fabric (chaincode, network configuration, REST API)