Mastering Hyperledger fabric

Master the art of Hyperledger Fabric on Kubernetes

(First Edition)

Author

Narendranath Reddy Thota

Blockchain Software Engineer

Consensys, Dubai

Book Title: Mastering Hyperledger fabric

Book Subtitle: Master the art of Hyperledger Fabric on docker, docker swarm and Kubernetes

Book Author: Narendranath Reddy Thota

Published by Narendranath Reddy

Niranjan Nagar Colony, kadapa, Andhrapradesh

This edition published April 2020

ASIN: B084KZP9M7

Copyright © Narendranath Reddy Thota 2020

Narendranath Reddy Thota asserts the moral right to be identified as the author of this work.

All rights reserved. No part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted, in any form, or by any means (electrical, mechanical, photocopying, recording or otherwise) without the prior written permission of the author or publisher. Any person who does any unauthorized act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

**Who this Book is for?**

This Book benefits Software Engineers who are ready to shift their focus to distributed technologies and Blockchain. This book provides a comprehensive view of Solution Architecture, so it will be easy for architects to architect their solution. CTOs around the world want to add Hyperledger fabric to their technology stack. Managers to cope up with the latest trend. Faculty Professors in order to get industry insights. Even Engineering Students who want to be ready with the latest technologies.

**Book Description**

Mastering Hyperledger Fabric is a craving topic for all Hyperledger Fabric Developers around the world. Hyperledger Fabric is an open-source project that helps organizations create and maintain permissioned distributed Blockchain consortiums. This book is for readers who are looking for Hyperledger offerings to build end-to-end projects with growing complexity and functionalities. This book will be a one-stop solution for all developers who want to build blockchain consortiums using Hyperledger Fabric. Topics include TLS, Unix sockets, caliper (Benchmark tool), raft consensus, advanced chaincode development, key collision and MVCC, chaincode access controls, chaincode encryption, node.js SDK, Golang SDK, docker daemon API, private data concepts, onboarding organizations using node.js SDK, deploy Hyperledger fabric using Kubernetes, deploy Hyperledger fabric using docker swarm, monitoring Hyperledger fabric, monitoring Kubernetes, monitoring docker swarm, logging Hyperledger fabric. After reading this book the reader will be able to set up Production grade Hyperledger fabric consortium using raft consensus mechanisms with monitoring using Prometheus and Grafana, even logging. This book explains so many key concepts of Hyperledger fabric including 2.0 and written with three years of Hyperledger fabric production experience.

**Preface**

The main goal of this book is to provide a one-stop solution for all implementations related to Hyperledger fabric projects. This book covered a variety of technologies such as Docker, docker swarm, Kubernetes, elk stack, Prometheus, Grafana, TLS, sockets, Raft, NodeJS, etc. and a variety of implementations such as caliper, logging, monitoring, and three diverse use cases. The content in this book helps developers to complete their solutions around Hyperledger Fabric. After reading this book readers will be able to develop production grade projects around Hyperledger fabric with different container orchestrator tools.

It is recommended that readers have a basic understanding of programming experience, docker, Node.js, and introduction to Hyperledger fabric, Kubernetes in order to benefit fully from this book.

**What this book covers**

Chapter1: Introduction to the Hyperledger Landscape

Chapter2: The Disruptive Potential of TLS

Chapter3: All about docker sockets

Chapter4: Installation Guide of Prerequisites

Chapter5: All about fabric CLI

Chapter6: All about SDK's (go lang and Node.js)

Chapter7: Advanced Chain code Development

Chapter8: End to End with Solo consensus using docker with one use case

Chapter9: End to End with Kafka consensus using docker swarm with one use case

Chapter10: End to End with Raft consensus using Kubernetes with one use case

Chapter11: Private Data Concepts, Consortium level ACL (Access Control Lists) and raft consensus mechanism

Chapter12: Setup and Benchmark Blockchain Consortium Using Caliper

Chapter13: Monitoring Consortium with Prometheus and Grafana

Chapter14: Logging Consortium with ELK Stack

Chapter15: Glimpse of Hyperledger fabric 2.0

Chapter16: Some Interesting tools

Table of Contents

[Welcome to the distributed world 12](#_Toc82356756)

[The Hyperledger Greenhouse 16](#_Toc82356757)

[Fabric Terminology - A quick review 17](#_Toc82356758)

[**Important Elements** 18](#_Toc82356759)

[**Consensus mechanism** 19](#_Toc82356760)

[**Transaction Flow** 19](#_Toc82356761)

[A glimpse of Container Orchestration 21](#_Toc82356762)

[Summary: 24](#_Toc82356763)

[Chapter2: The Disruptive Potential of TLS 25](#_Toc82356764)

[An introduction to TLS (One-Way) 25](#_Toc82356765)

[Bank - Passport Analogy (A Very Low Level of TLS) 28](#_Toc82356766)

[Double secure with Mutual SSL/TLS 29](#_Toc82356767)

[How does a cryptogen tool generate a bundle of certificates? 29](#_Toc82356768)

[**How can we trigger mutual TLS in fabric?** 30](#_Toc82356769)

[**Connecting to an orderer or peer with mutual TLS:** 31](#_Toc82356770)

[**Configuration Errors:** 31](#_Toc82356771)

[Summary: 31](#_Toc82356772)

[Chapter3: All about docker sockets 33](#_Toc82356773)

[An introduction to Docker daemon API 33](#_Toc82356774)

[Interact with docker daemon API via docker sock 33](#_Toc82356775)

[**List of Available Images**: 34](#_Toc82356776)

[**List of running containers**: 34](#_Toc82356777)

[**Create a tag to an image**: 35](#_Toc82356778)

[**Stop a running container**: 35](#_Toc82356779)

[**Pull an Image**: 35](#_Toc82356780)

[How Fabric peers benefit from using docker sock? 35](#_Toc82356781)

[Summary: 35](#_Toc82356782)

[Chapter4: Install Prerequisites 36](#_Toc82356783)

[Install and Setup Virtual Box 36](#_Toc82356784)

[Install dependencies 37](#_Toc82356785)

[**PlanA**: 38](#_Toc82356786)

[**PlanB**: 38](#_Toc82356787)

[Tips on Errors, Tricks and Best Practices 39](#_Toc82356788)

[Solution: 39](#_Toc82356789)

[Summary: 39](#_Toc82356790)

[Chapter5: All about fabric CLI 40](#_Toc82356791)

[CLI Basic Operations 40](#_Toc82356792)

[**Create Channel** 40](#_Toc82356793)

[**Join Channel** 41](#_Toc82356794)

[**Install Chaincode** 41](#_Toc82356795)

[**Instantiate Chaincode** 41](#_Toc82356796)

[**Invoke Chaincode** 42](#_Toc82356797)

[**Query Chaincode** 42](#_Toc82356798)

[CLI Advanced Operations 42](#_Toc82356799)

[**Retrieve present Configuration of a Channel** 42](#_Toc82356800)

[**Sign Config Transaction** 43](#_Toc82356801)

[**Update Channel** 43](#_Toc82356802)

[Register an Identity 43](#_Toc82356803)

[Enroll Identity 44](#_Toc82356804)

[Summary: 45](#_Toc82356805)

[Chapter6: All about SDK's (go lang and Node.js) 46](#_Toc82356806)

[The basic setup of Golang SDK 46](#_Toc82356807)

[Advanced Operations using Golang 49](#_Toc82356808)

[**Create Channel**: 49](#_Toc82356809)

[**Install Chaincode**: 50](#_Toc82356810)

[**Join Channel**: 51](#_Toc82356811)

[**Instantiate Chaincode**: 51](#_Toc82356812)

[**Invoke:** 52](#_Toc82356813)

[**Query transaction**: 53](#_Toc82356814)

[**Register Identity**: 53](#_Toc82356815)

[**Enroll Identity**: 54](#_Toc82356816)

[Basic Operations using Node.js 54](#_Toc82356817)

[**Setup Initialize**: 55](#_Toc82356818)

[**Create Channel**: 55](#_Toc82356819)

[**Join Channel**: 56](#_Toc82356820)

[**Install Chaincode**: 56](#_Toc82356821)

[**Instantiate Chaincode**: 57](#_Toc82356822)

[**Invoke Transaction**: 58](#_Toc82356823)

[**Query Chaincode**: 59](#_Toc82356824)

[Advanced Operations using Node.js 59](#_Toc82356825)

[**Register Identity**: 61](#_Toc82356826)

[**Enroll Identity**: 61](#_Toc82356827)

[**Re-Enroll**: 61](#_Toc82356828)

[**Revoke**: 62](#_Toc82356829)

[**Generate Certificate Revocation List**: 62](#_Toc82356830)

[**Block Event Listener**: 62](#_Toc82356831)

[**Chaincode Event Listener**: 63](#_Toc82356832)

[**Get Present Channel Config As JSON**: 63](#_Toc82356833)

[**UpdateChannel**: 64](#_Toc82356834)

[Tips on Errors: 67](#_Toc82356835)

[Summary: 69](#_Toc82356836)

[Chapter7: Advanced Chaincode Development 71](#_Toc82356837)

[key collision and MVCC 71](#_Toc82356838)

[Client Identity Chaincode Library 72](#_Toc82356839)

[**Based on MSP ID** 73](#_Toc82356840)

[Based On Attributes 73](#_Toc82356841)

[End-to-End Encryption 73](#_Toc82356842)

[**Encrypt**: 75](#_Toc82356843)

[**Decrypt**: 75](#_Toc82356844)

[CI/CD Installing chaincode From GitHub to the peers 75](#_Toc82356845)

[**STEP1**: Create and configure GitHub repo 76](#_Toc82356846)

[**STEP2**: 76](#_Toc82356847)

[**STEP3**: Proceed with webhook handler 77](#_Toc82356848)

[Summary: 78](#_Toc82356849)

[Chapter8: Running fabric consortium with Solo consensus using docker 79](#_Toc82356850)

[SubTitle: Building​ ​Invoice Financing mechanism with solo consensus 79](#_Toc82356851)

[Getting started: 81](#_Toc82356852)

[Define Business Network 82](#_Toc82356853)

[Design Network Topology 90](#_Toc82356854)

[Deploy Consortium: 90](#_Toc82356855)

[Building an application: 92](#_Toc82356856)

[Chain Code: 97](#_Toc82356857)

[Interact with fabric: 103](#_Toc82356858)

[Tips on Errors: 108](#_Toc82356859)

[Troubleshooting: 109](#_Toc82356860)

[Summary 110](#_Toc82356861)

[Chapter9: Running fabric Consortium with Kafka consensus using docker swarm 111](#_Toc82356862)

[Use Case in Detail: 111](#_Toc82356863)

[Technical Requirements: 112](#_Toc82356864)

[Project overview: 112](#_Toc82356865)

[Getting started: 113](#_Toc82356866)

[Define Business Network 114](#_Toc82356867)

[Design Network Topology 126](#_Toc82356868)

[Authentication Flow: 128](#_Toc82356869)

[Deploy Consortium: 128](#_Toc82356870)

[Monitoring Swarm Network: 130](#_Toc82356871)

[Building an application 133](#_Toc82356872)

[Chain Code: 139](#_Toc82356873)

[Interact with fabric: 146](#_Toc82356874)

[Tips on Errors: 151](#_Toc82356875)

[Error: connection issue 151](#_Toc82356876)

[Summary 152](#_Toc82356877)

[Chapter10: Running fabric Consortium with Raft consensus using Kubernetes 153](#_Toc82356878)

[Use Case in Detail: 153](#_Toc82356879)

[Technical Requirements 154](#_Toc82356880)

[Project overview: 155](#_Toc82356881)

[Getting started 155](#_Toc82356882)

[Define Business Network 156](#_Toc82356883)

[Design Network Topology 178](#_Toc82356884)

[Deploy Consortium: 179](#_Toc82356885)

[Monitoring Kubernetes: 181](#_Toc82356886)

[Monitoring Kubernetes using Weave Net: 182](#_Toc82356887)

[Building an application: 185](#_Toc82356888)

[Chain Code: 186](#_Toc82356889)

[Interact with fabric: 192](#_Toc82356890)

[Summary: 199](#_Toc82356891)

[Chapter11: Private Data Concept, Consortium level ACL (Access Control Lists) and Raft Consensus 201](#_Toc82356892)

[Private Data Concept: 202](#_Toc82356893)

[**Theory**: 202](#_Toc82356894)

[**Practical**: 204](#_Toc82356895)

[**Create a collection definition JSON file** 205](#_Toc82356896)

[Access Control Lists: 208](#_Toc82356897)

[Raft Consensus: 212](#_Toc82356898)

[**Hyperledger fabric Raft consensus**: 213](#_Toc82356899)

[Tips on Errors: 216](#_Toc82356900)

[Error: Failed disseminating 2 out of 2 private RWSets 216](#_Toc82356901)

[**Summary:** 217](#_Toc82356902)

[Chapter12: Setup and Benchmark Blockchain Consortium Using Caliper 218](#_Toc82356903)

[Caliper Supported blockchain solutions are the following: 218](#_Toc82356904)

[Caliper Supported performance metrics are the following: 218](#_Toc82356905)

[Install and Setup Caliper 218](#_Toc82356906)

[Benchmark configuration file 219](#_Toc82356907)

[Network configuration file 221](#_Toc82356908)

[Write A Sample Chaincode 229](#_Toc82356909)

[Benchmark scripts 231](#_Toc82356910)

[Run the total setup: 233](#_Toc82356911)

[Report Summary: 234](#_Toc82356912)

[**Summary:** 236](#_Toc82356913)

[Chapter13: Monitoring Consortium with Prometheus and Grafana 237](#_Toc82356914)

[Deploy Default Consortium: 239](#_Toc82356915)

[Summary: 243](#_Toc82356916)

[Chapter14: Logging Consortium with ELK Stack 244](#_Toc82356917)

[Setup ELK Stack: 246](#_Toc82356918)

[Deploy and Test: 249](#_Toc82356919)

[Shipping docker data into dockerized ELK stack: 252](#_Toc82356920)

[Summary: 256](#_Toc82356921)

[Chapter15: Glimpse of Hyperledger fabric 2.0 258](#_Toc82356922)

[Do we really need to discuss 2.0? 258](#_Toc82356923)

[Smart Contracts Decentralized Governance 259](#_Toc82356924)

[Private Data Enhancements 260](#_Toc82356925)

[External Chaincode Launcher 260](#_Toc82356926)

[Alpine-Based Docker Images 260](#_Toc82356927)

[Upgrading to Fabric v2.0 260](#_Toc82356928)

[Upgrade ordering nodes: 261](#_Toc82356929)

[Upgrade the peers: 261](#_Toc82356930)

[Summary: 262](#_Toc82356931)

[Chapter16: Some Interesting tools 263](#_Toc82356932)

[● Ledgerfsck - Tool to verify ledger integrity 263](#_Toc82356933)

[● Security Analyzer for Hyperledger Fabric Smart Contracts 263](#_Toc82356934)

[Ledgerfsck - Tool to verify ledger integrity 263](#_Toc82356935)

[Security Analyzer 264](#_Toc82356936)

[Summary: 266](#_Toc82356937)

Chapter1: Introduction to the Hyperledger Landscape

Hyperledger is undoubtedly a unique platform which allows enterprises to build private, permissioned, distributed ledger frameworks and their supporting tools. Using Hyperledger, developing blockchain frameworks in a disruptive way is an easy way for enterprises. Enterprise use cases require capabilities such as scalability, high throughput, built-in or inter-operable identity modules, and different levels of access controls for the parties involved in the consortiums. Design Access controls in such a way that it must allow regulators to access ledger data as read-only to ensure compliance. According to the business requirement, one can onboard any organization at any time, at the same time one can deboard at any time. Likewise, blockchain networks should also behave accordingly. We will definitely boost our knowledge by looking into industry-level ways of setting up the consortium and some innovative use cases in the coming chapters.

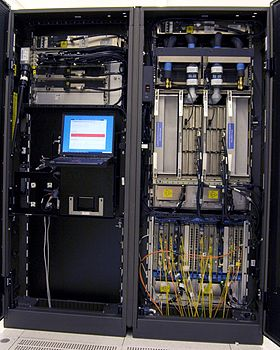
In this chapter, we will cover the following topics:

* Welcome to the distributed world
* The Hyperledger greenhouse
* Fabric Terminology - A Quick Review
* A glimpse of Container Orchestration

## **Welcome to the distributed world**

Over the past decades, ever-widening technological expansion is a nightmare because of many reasons. But we are in a high-tech era where we can observe lots of changes and many innovations happening in the computer science space. The world is amazing, ever-amazing so-called distributed systems are becoming very popular. It is a fact that distributed systems are always a huge pitfalls and landmines zone in the computer science subject. Wait, before thinking about distributed systems, take a long breath and look back to those days of computer science technologies and architectures.

We all know that when operating systems were invented, it was only supporting single- process computing resources. It's like a life cycle starting from booting up machines, loading a program (which is written on cards) and executing a program, after stipulated time it goes to shut downstate. Engineers were super smart and have been contributing to the innovation ecosystem which leads to the multitasking operating systems. Soon the world has completely adopted the multitasking paradigm for so many decades. This disruption is usually referred to as host-based Computers, later used to call as “mini-computers”. Discussion on Computer science History never completes without having a discussion on mainframes, Users run their programs from gigantic terminals inside terminal rooms which display generated output from the keyboard.

Source: wikipedia.org

Enough time spent in the past, let us come back to the present world and compare with present computer science technologies and architectures, we can easily identify a huge variation between the centralized systems so-called centralization and subsequently distributed systems. We have distributed - (computing, protocols, power, infrastructure, file systems, data stores, processing, messaging, applications, ledger, storage) and even code also yes, you heard correct we have distributed video coding, distributed source coding etc., the best example of distributed processing is LAN our local area network. We started with centralized systems and now we are in a distributed & decentralized era.

The world is amazing to us at every stage, every moment with amazing innovations. With the advance of personal computing resources and private peer to peer networking, typical computational capabilities were now equipped both at the client-side meaning that the user interface side, as well as the server-side. By this it gave rise to the traditional 'client-server' architecture, soon this architecture has been promoting the development of relational & non-relational database systems to store the data permanently on a disk. In the client-server architecture, if we want to persist data for a long time then we have to take the help of any databases such as MongoDB, CouchDB, etc. these databases present on one node, whenever you want to create/query data in it, you access to that node directly.

|  |  |
| --- | --- |
| |  | | --- | |  | |

Later this becomes mature enough to distribute a database instead of using the single point of failure databases. It means a lot to us; we can say data could replicate from one server to another server with direct peer to peer communication instead of the client-server communication. Let us put in simple terms, our best buddy distributed system, which is a collection of independent nodes, working restlessly to appear as a single coherent computer. These nodes will run concurrently with a strongly shared consistent state using distributed consensus protocols such as raft, Kafka, etc. So, what does that mean to the client? The client must be able to access his data whichever node he connects and the client should get the expected result. This makes the client is accessing the distributed systems, it means not a single node instead of multiple nodes. If he inserts a record into Node 1 or Node 3 or Node 4, later must be able to retrieve that record from any database node without having an error response.

**Note**: Nodes or computers or servers all are the same.

Let us go furthermore, the Internet with cloud computing giants like Amazon has opened doors for global access from a variety of computing devices such as mobile, laptop, etc., whereas mainframe has been giving services to large centralized corporations and governments. It is an open secret that 'cloud architecture' is decentralized in terms of hardware, but it has given rise to application-level centralization (e.g., giant Facebook network, Twitter, Google, YouTube, etc.). Web 3.0 is an innovative way of transforming centralized systems like storage engines, computing engines, processing engines, and hosting engines to decentralized engines and systems where free from third-party hosted servers, only your computer back in your home with underlying infrastructure will act as a node in a decentralized network, this gives huge transformation in the web 3.0 era".

So far, we have been discussing all distributed systems, but how far Decentralization is different from distributed systems? I can say that in distributed systems a user can control his data when privacy demands, means can control data replication but in decentralized systems, a user has no control over data by default it is replicated in all systems present in that network. That's what all public blockchains are designed, now a big question: **Do we really need to distribute a system?** Systems need to be in a distributed state by demand/need, not by fashion otherwise you will have to sacrifice on many things and pay a lot. Distributed systems open doors to scale horizontally which means, appending more computers to the existing network like appending blocks in a blockchain rather than upgrading the hardware of a single computer which is vertical scaling. Let us take an example: assume, we have a single database system which is connected to a Node.js server, suddenly more traffic appeared due to some typical reasons so, scaling horizontally means adding more servers with the help of load balancing mechanism but, nowadays cloud giants like Amazon have an inbuilt load balancer to handle load equally. We can write an Ansible script to scale that is another way. But some situations where we just need to improve our system capacity, this is what scaling vertically. It means adding more CPU power to the existing system without extending to multiple servers. Do distributed systems just give scaling? The answer is No, we have few benefits like consistency, Fault tolerance, and low latency, etc.

“If a system crashes in a distributed system but, urgently need to access the system will never be a nightmare precisely a collection of computers that never share common memory or common processor but communicate through messages or sockets over a communication network where each computer has its own memory or processor and importantly it has its own operating system. We can say that computers in a distributed network are semi-autonomous and loosely coupled to address a common problem.

Now will see how many types are present in the distributed system category:

* Data Store in a Distributed fashion like all cloud data centers
* Computing in a Distributed fashion like the golem project
* File Systems in a Distributed fashion like Napster project
* Messaging in a Distributed fashion like Kafka
* Application in a Distributed fashion
* Ledgers in a Distributed fashion and many more….

**Let us focus on the most favorite concept: Distributed Ledger**

The ledger that replicates itself across nodes is called a distributed ledger. In simple terms, it is an immutable way of storing data in an append-only fashion. Every node in the network will be waiting for a data change, once it happens, using remote procedure calls, nodes will be in sync, finally, all nodes in the distributed network will be having the same copy of data. The underlying technology of distributed ledger is Blockchain. The blockchain is a distributed ledger, transactions are present in the form of blocks in a giant chain. So, what are transactions here? These are normal interactions, exchange or transfer of something within the network. Transactions will present in the form of blocks with a strong timestamp and a number. One cannot be able to alter the data present in blocks without destroying the whole chain from the subsequent block. Merkel tree is a data structure using this one can audit at any time right from block 1 to the latest block.

Using blockchain we can have our physical agreement as a programmable contract. Distributed ledgers have existed since prior to Bitcoin, but blockchain grabs everyone's attention with game-changing technologies, including cryptography, Peer-to-Peer networks, time-stamp, and shared computational power, and consensus agreement.

Blockchain generally consists of three main components:

* Replicated way of the single state of the entire ledger
* Transactions that may or may not change the ledger state depends on success/failure.
* Consensus mechanism to make all participants with one agreement in what order, by the ledger to be present in the whole block-chain.

## **The Hyperledger Greenhouse**

Hyperledger is undoubtedly an awesome idea by The Linux Foundation and it is an open-source umbrella project which has been created with a clear vision to build cross-industry applications backed by blockchain technology, Hyperledger has many collaborators from the finance industry, banking industry, supply chain industry, manufacturing industry, and even from IoT based industries. Hyperledger provides us a way to develop core blockchain-backed frameworks and applications. Hyperledger is all about a house of enterprise-ready frameworks, communities of software developers, and researchers thereby building blockchain-focused frameworks, libraries, and tools. After Bitcoin we have two big giants in the blockchain space which are Hyperledger and Ethereum, let us look into what they are and how they differ, both are open-source having innovation hubs with attracting developers, innovators, researchers. Ethereum blockchain came into the world because of Vitalik Buterin, who was inspired by bitcoin-colored coins but Hyperledger was hosted by The Linux Foundation in December 2015 and got contributions and support from major corporations such as Intel, IBM, Monax, and others. The Ethereum project has been funded through initial coin offering and Thiel fellowship award prize money one lakh dollars. Vitalik was inspired by bitcoin-colored coins and introduced the concept of ICO through Ethereum to the entire world. By using ICO, crowdfunding is ever easy to get investments and simple to audit and it is available to everyone, which innovatively paved a way for all sorts of business. Awesome news rolled out in August 2019 that their product Pantheon built by Pegasys from consensys now part of Hyperledger named Besu. Ethereum is almost like B2C approach since, dealing with consumers, but Hyperledger is a typical B2B approach blockchain uses cases so-called private blockchains | consortium-based.

Six Powerful Open-Source Projects under Hyperledger umbrella:

* Project Fabric
* Project Sawtooth
* Project Burrow
* Project Indy
* Project Iroha
* Project Besu

A lot of projects lead to a lot of confusion, So The selection of the relevant project or framework out of the above six depends on the use case and requirement.

## **Fabric Terminology - A quick review**

The fabric is the first-ever permission distributed ledger framework which has more importance for privacy, which also supports chain codes or smart contracts like other frameworks that are authored in one of these languages Java, Go, and JavaScript, these are general-purpose languages and this framework is a Turing complete framework. Most of the enterprises have the skill set to write smart contracts, so no need for additional resources. The fabric allows participants to make transactions with other participants in a secure and transparent way because these participants themselves may not fully trust one another (they may be competitors in the same industry), governance is all about in such a typical scenario.

### **Important Elements**

**Channel is** the most important and innovative element in the fabric ecosystem because using channels, organizations can maintain multiple ledgers in the same consortium. Only channel members on which the transaction was performed can see the transaction details. fabric delegates the transactions at runtime to different ledgers based on the channel. The fabric has given more and more importance to privacy for transactions and introduced Zero-Knowledge proofs using Identity Mixer which enables anonymous authentication for client identities in their transactions. A channel needs to know how many orderers, chain codes, peers, are part of the channel So, we are installing chain code on peers and instantiate on a channel, joining peers to the channel.

**Some Points to Consider**:

* One can have multiple channels across a consortium.
* One can have Multiple chain codes across channels.
* One can make a channel policy and endorsement policy.
* One can use either level DB or CouchDB as a world state store.
* One can use private data collections which reduce no of channels
* Support for cryptogen or Fabric certificate authority or own certificate authority

In general Certificate Authority, issues & manages certificates for users. Hyperledger Fabric has CA which is the default certificate authority for Fabric but it is completely pluggable. CA can handle the requests for registration and enrollments of user identities.

**Peers**, basically organizations will host at least one peer to a channel and they maintain the copy of log which is called ledger of that channel. We have physically three different types of peers: committing, endorsing, and ordering peers. Yes, you heard correct, orderer also a peer, but logically there are a total of 5 peers present. The extra peers are anchor peers and leader peers. Endorsing peers are a special kind of committing peers. They do special things such as simulating a proposed transaction and producing an endorsement signature.

The **chain code** which wraps both the business logic and asset definitions. After many security steps, Successful Transaction invocations will result in modifications to the world state & Blockchain, for unsuccessful transactions world state is not altered but Blockchain has that transaction.

The **ordering service** is a set of orderers in a cluster that writes or orders transactions into a block and distributes it to all peers. **Peers** then append this block to their own ledger by having an update to the world state. The world state then reflects with the current latest ledger, whenever the user retrieves fabric make sure the user always gets the latest data from any peer. Basically, this data is stored in a database for efficient access. Currently, supported databases are Level DB this is inbuilt and CouchDB this is pluggable

### **Consensus mechanism**

The consensus is a process of coming to an agreement by all parties in a network. An agreement conclusion decides the next set of transactions to be added to the ledger. We have three distinct steps to reach consensus:

* Transaction endorsement a security check & policy agreement.
* Ordering transactions into blocks ready to commit by peers
* Validate endorsements against the current state and affix to a ledger.

Hyperledger Fabric allows users from organizations to define policies around the lifecycle of chaincode. These endorsement policies define which set of peers must agree to transactions before it can be added to the blocks.

### **Transaction Flow**

Transactions rolled out from client applications such as Nodejs or from CLI to endorsing peers.

Each endorsing peer from all organizations (Based on the policy) simulates the proposed transaction, without updating the ledger and returns endorsement with RW sets. The signed response of endorsing peer comes as a transaction which is called transaction endorsement which is a result of the simulated transaction. Endorsing peers must hold smart contracts in order to simulate the transaction proposals

An application collects all endorsed transactions from the endorsing peers and then submits to the orderer for further process.

|  |  |
| --- | --- |
| |  | | --- | |  | |

The ordering service which is a cluster of orderers collects endorsed transactions and RW sets from an application wrap into a block and delivers the block to all peers, then peers validate all transactions present in the block by checking to make sure that the RW sets still match the current world state to avoid tampering. The successful result leads to writing the block to the ledger and unsuccessful transactions still written to the ledger but the world state is not updated.

|  |  |
| --- | --- |
| |  | | --- | |  | |

**MSP** (Membership Service Provider)

The membership service provider, or MSP, is a trust factor in a network, which always defines the rules in which identities are being validated and then access to a network. The MSP manages trusted identities to authenticate clients who want to join the network. The MSP fully depends on a Certificate Authority, which is a pluggable interface that verifies and revokes user certificates. Fabric-CA is the default interface for MSP as I said earlier this is pluggable. Think of a big building which has many secret rooms. To enter into anyone room you need access, let say a building has 10 rooms, Alice has only access to 5 rooms, how we can make sure that Alice should not ether to a room which she is not allowed, we need some back- end mechanism to check whenever a person enters need to be validated by a person or by using a system. In fabric networks, MSP is such a type of mechanism. In this whole building out of 10, 5 rooms are finance, 5 rooms are the compliance department. We can have an ID to each department so that identities can be issued | tracked on behalf of ID. For peers, it will have a local MSP folder in that admin certificate, RCA certificate present. Whenever a peer receives a transaction, it will check the issuer of the user certificate. If it issued by its RCA

then it will check whether he is an admin or not by comparing the certificate with the admin certificate in his local MSP. By this way fabric MSP makes PKI level authentication & authorization. Hyperledger Fabric has channel MSP, Orderer MSP, Organization MSP, Peer MSP

A set of parameters need to be specified to allow for identity (certificate) validation and signature verification Example of Organization MSP:

* A list of RCA (X.509) certificates.
* A list of ICA (X.509) certificates.
* Org Admin certificate X.509.
* Revoked Certificates list.

## **A glimpse of Container Orchestration**

In IT Engineering, packaging an application is ever easy, whether written in Java or Node.js or Golang Containers are a standard or professional way to package an entire application including its dependencies so that the application can be deployed to any server as a one goes and can move between servers and run without any dependency issues of infrastructure changes, version changes, etc. By the way, Containers work intelligently by isolating an application inside the container without disturbing the outside environment unless we mount volumes. So, everything outside the container can be on the host machine and will never be changed.

If one container or three or some extent we can manage their life cycle, but what about bundles of containers, managing is very difficult and definitely will be a nightmare after some extent. Thanks to the Container orchestration mechanism which is all about an automated way of managing life-cycles of containers. There are many orchestrators available in the market but the popular are Kubernetes by google and the simplest is swarm by docker. We have many advantages using orchestration tools:

* Easy way Provisioning and deployment of containers
* Full Redundancy and availability of containers made easy
* East to scale Deploying containers across servers
* Load balancing and horizontal scaling is easy
* Self - healing of containers is pretty awesome
* Achieve Service discovery very easily
* Storage orchestration made easy

Swarm orchestration tool is one of the family members from the docker itself and it starts in action when users start describing the application configuration in a YAML file which is called a compose file. These configuration files are answers for several questions from orchestration tools, like from where to get container images (public | private Docker Hub), mounting volumes, exposing ports, injecting environment variables, networking between containers, giving privileges.

In a nutshell, Swarm transforms several docker daemons together as a single virtual daemon with full sing ready to sail, in layman, it is a set of nodes with at least one master node powered by raft consensus algorithm and several worker nodes that can be virtual machines or physical machines. Manager nodes that are part of a swarm cluster that manages and scheduling tasks will be in a consistent state through raft as a distributed consensus algorithm. When there are several tools that are available to monitor swarm clusters, the popular one is swarm pit will guide how we can run swarm pit in one of the following chapters.

Kubernetes is a Google’s product that is made available to open source as production-ready only after long flashbacks of running massive workloads in google at a huge scale of production environments. Manager nodes that are part of the cluster that store configuration data, metadata, and state data into etcd, etcd implement raft distributed consensus because it is a distributed data store.

In the Kubernetes analogy, a pod is a main core of the system where our application runs in the form of containers, a pod can have more than one container. Deployment represents the set of pods and each deployment there is a replica set which maintains the state of the pod.

service represents the deployments. We will define all the configuration of our application details in the deployments at the container section. Service is the main entry for communication from outside of the cluster or even inside of the cluster.

This means suppose in a Raft consensus mechanism orderer1, orderer2, orderer3 needs to be in sync and communicate between these orderers happening through services. There are four types of services

* Cluster IP
* Load Balancer
* NodePort
* External Name

For example, if a peer in Hyperledger fabric wants to talk to orderer then orderer must expose a service so that a peer can talk to orderer through service and in the same way orderer can reply back to peer using peer service.

We can use PVC (persistent volume claims) for storage needs by the pod and storage provisioner take’s care where to store and it will be defined by the administrator but a default storage class should be defined for dynamic provisioning.

Any confidential files such as private keys to be deployed to the cluster then will use secrets, and normal files such as certificates, channel.tx, genesis.block etc, as configmaps. Kubernetes is not a complex system at all unless you're a beginner.

Alternatively, we can use Helm charts but fabric charts are maintained by third party clients, not by the fabric. Moreover, you will not get complete freedom. We can write our own chart for advanced users, otherwise strict to bare metal style. We will write all the necessary configurations from scratch.

|  |
| --- |
| * [https://helm.sh/docs/topics/charts/](https://helm.sh/docs/topics/charts/%20) * <https://hub.helm.sh/> |

|  |
| --- |
|  |

**Note**: Helm charts are coming with loaded default configuration such as environment variables, fabric entity image, volumes, etc. with an option to override defaults.

In a nutshell, use docker swarm for simple and easy applications and use Kubernetes for more complex applications because it is simply superb and powerful. One can use swarm or Kubernetes in production.

## **Summary:**

In this chapter, we have seen a really immense amount of information about distributed systems and its diverse types and explored the Hyperledger umbrella of projects. We also discussed the various functionalities and elements of Hyperledger Fabric, one of the most popular enterprise-grade permissioned blockchain. Finally, we end the chapter by discussing container orchestration as part of the Hyperledger fabric deployment.

In the next chapter, we will discuss the disruptive potential of the SSL/TLS in the field of cybersecurity and how it gives importance to the Hyperledger fabric.

# 

# Chapter2: The Disruptive Potential of TLS

Let me first clarify TLS vs SSL, these two are cryptography protocols, SSL is the old version and TLS is the new and latest version. We use TLS for authentication purposes and encryption purposes. Authentication and encryption may happen between client and server or between server and another server. The whole purpose of the TLS is to make sure the data transmitted by the client is not tampered by any middle man hacker and it reaches the destination server safely. TLS is more secure. We can use it in two ways. The most commonly used way is verified only by the client which means the client validates the authenticity of the server to make sure that it receives data from the designated server. It happens when a client sends a request to the server and the server shares its public certificate to the client. The other way is both client and server validate each other certificates, in this way both the client and server must share public certificates with each other. This is called mutual TLS which is highly secure. It starts with a handshake which means both client and server handshakes with hello messages

## **An introduction to TLS (One-Way)**

One-way SSL is the most common type of TLS used nowadays. The best example is https itself. Symmetric key encryption, asymmetric key encryption is the two cryptographic algorithms from cryptology which play a crucial role in the SSL/TLS. The symmetric key algorithm means encryption and decryption happens with the same key; the popular example is AES. The asymmetric key is a bit different. They need two keys; one is a public key and another key is a private key. Encryption happens with public key and decryption will happen only with a private key which is completely reverse functioning of digital certificates. Let's see the action below.

**STEP1**:

Server Administrators will have to request an order to get an SSL certificate from the major certificate providers. Some are VeriSign, DigiCert, GoDaddy, Comodo, etc. these are the official SSL certificate providers. Becoming a certificate provider is complex and needs a lot of investments so do not ever think about it. After the admin places an order, he will receive an SSL certificate and he will have to install it on the server and it will be ready to serve the requests coming from the clients such as browsers.

|  |  |
| --- | --- |
| |  | | --- | |  | |

**STEP2**:

When the user starts typing any domine in the browser search bar window it resolves the domine with actual server Public IP and it will route the traffic to the destination server with hello message, and server also reply with hello message plus server publicly verifiable SSL certificate

|  |  |
| --- | --- |
| |  | | --- | |  | |

**STEP3**:

All browsers in the world such as Google Chrome, Safari, Firefox, etc. will have a trusted certificate database whenever browser receives hello message and server certificate it first checks the authenticity of the certificate against its trusted database, at the initial,, it will definitely fail because the browser will have root CA certificate database not the SSL certificates database, so browser second time will search against the issuer, which means suppose browser database has DigiCert root certificate and [www.google.com](http://www.google.com) has been issued by DigiCert then this certificate will be accepted by the browser.

|  |
| --- |
|  |

**STEP4**:

As many people do not know or are confused about SSL whether it uses symmetric key encryption or asymmetric key encryption. It uses a hybrid mechanism which means both. First browser generates a symmetric key (a pair) and it encrypts with the server public key which will be available in the certificate and it sends to the server. On the other side the server receives the complete payload and it decrypts with his private key, now the server has a key and browser also has a key. From now onwards browsers and servers will use asymmetric keys for data exchange. So, the rule of thumb here is Asymmetric key encryption for authenticity and symmetric key encryption for data transfer.

|  |  |
| --- | --- |
| |  | | --- | |  | |

**Step5:** Once the server received a symmetric key through the Asymmetric key encryption. The data transfer from client to server and server to client happens through symmetric key encryption. Till the user closes the session.

|  |  |
| --- | --- |
| |  | | --- | |  | |

## **Bank - Passport Analogy (A Very Low Level of TLS)**

Let us deep dive complete TLS/SSL at a very low level. We will have a bank manager, a customer, and a courier boy with a total of three characters. The goal of the story is that the customer wants to send his documents to the bank manager in a secure and safe manner without tapering by the courier boy. Beware we will compare some parts of the story with real life TLS verbs.

**Part1**: Bank manager needs documents of a customer, so he will send a box to the customer through the courier. This box is a normal box which has two keys: one will be kept by the bank manager and another key will be sent along with the box in order for the customer to lock the box with the key. Up one received by the customer, the key would have been copied by the courier boy and if the customer sends his documents there are high chances it may be tampered/copied by the courier boy this is called plain http.

**Part2**: Bank manager instead of sending a normal box he will purchase a new magic box which will be closed with key1 and it will open only with key2. Key1 is a public key which we call a certificate. Now the bank manager sends the magic box along with a public certificate through courier and upon receiving the customer put his documents and lock it using a public certificate and he will send it back to the bank. But the courier boy still tampered it, how? Take a pause and think how is this possible?

**Answer**: Courier boy, upon receiving the courier he replaced the complete magic box and public certificate with his own box and key. Customer did not know about this and he locked it later. The courier boy copied the documents and he replaced it with a bank box and he locked it. So how can a customer know that a box is coming from a trusted person? This is why Certificate Authority comes into the picture. CA makes sure that the box/certificate coming from a right legitimate person and this process is called https.

## **Double secure with Mutual SSL/TLS**

We already know about one way SSL/TLS and mutual TLS is almost the same but a bit different and this difference makes use of a super secure mode. In the one-way SSL, only clients validate the certificate by checking against trusted root CA certificates. In the mutual TLS now, the server also will do the same which means the server validates the client certificate by checking against the trusted root CA certificates. The whole process makes sure that the client knows about the server and the server knows about the client and it is almost impossible to tamper the information by the intruder.

|  |  |
| --- | --- |
| |  | | --- | |  | |

## **How does a cryptogen tool generate a bundle of certificates?**

Cryptogen is a tool given to us through open-source community, complete source code of the cryptogen tool is developed using Golang and Golang has a rich library we can generate a root certificate and signed certificate using crypto/x.509, crypto/ECDSA, etc. libraries. So cryptogen uses crypto/x.509 library to generate the certificates, x.509 is a certificate standard. Cryptogen tool generates a self-signed root certificate and bundle of other signed certificates based on the input given by the user.

One of the popular Asymmetric key cryptography algorithms is ECDSA which stands for **Elliptic Curve Digital Signature Algorithm.** We use ECDSA for authentication purposes, of course there are other popular algorithms such as RSA, but fabric supports only ECDSA**.** Here in this context authentication refers to the process of verifying that a particular message has been signed with a public key, and which was created by the owner of a specific private key. The algorithms which are used for authentication purposes are collectively known as **digital signature algorithms.**

Certificate is just a high-level document and there are many ways to format a document if we do not agree with a particular format then it will be difficult for the browser to parse the certificate this is why ASN.1 data structure comes into the picture and it tells us how to format a document in object/certificate and to how to parse. Certificate and private keys are encoded using ASN.1, by the way, we have a Golang library for this encoding/asn1.

By combining all piece’s digital certificates also known as x.509 certificates (x.509 is a standard) are defined using ASN.1 and encoded in DER encoding rules.

### **How can we trigger mutual TLS in fabric?**

Enabling orderer mutual TLS authentication is by setting the appropriate environment variable to true.

**ORDERER\_GENERAL\_TLS\_CLIENTAUTHREQUIRED** environment variable to true.

Enabling peer mutual TLS authentication is by setting the appropriate environment variable to true.

**CORE\_PEER\_TLS\_CLIENTAUTHREQUIRED** environment variable to true.

Peer:

|  |
| --- |
|  |

Orderer:

|  |
| --- |
|  |

### **Connecting to an orderer or peer with mutual TLS:**

Once we have the user/client certificate and private key, we can assign the credentials to the client instance.

Assume that we have client certificate at /usr/certs/mutual/my-org1.crt

Assume that we have client private key at /usr/certs/mutual/my-org1.key

Assume that we have client CA TLS root cert at /usr/certs/mutual/my-org1.pem

1. let serverCert = fs.readFileSync(path.join(\_\_dirname, '/usr/certs/mutual/my-org1.pem'));
3. let clientKey = fs.readFileSync(path.join(\_\_dirname, '/usr/certs/mutual/my-org1.key'));
5. let clientCert = fs.readFileSync(path.join(\_\_dirname, '/usr/certs/mutual/my-org1.crt'));
7. client.setTlsClientCertAndKey(Buffer.from(clientCert).toString(), Buffer.from(clientKey).toString());
9. let orderer = client.newOrderer(
10. 'grpcs://localhost:7050', {
11. 'pem': Buffer.from(serverCert).toString()
12. });
14. let peer = client.newPeer(
15. 'grpcs://localhost:7051', {
16. 'pem': Buffer.from(serverCert).toString()
17. }
18. );

### **Configuration Errors:**

If we make any configuration errors then it will throw a handshake error.

E0923 16:30:14.963494564 31166 ssl\_transport\_security.cc:188] ssl\_info\_callback: error occurred.  
E0923 16:30:14.963567129 31166 ssl\_transport\_security.cc:989] Handshake failed with fatal error SSL\_ERROR\_SSL: error:14094412:SSL routines:ssl3\_read\_bytes:sslv3 alert bad certificate.  
E0923 16:30:15.964456710 31166 ssl\_transport\_security.cc:188] ssl\_info\_callback: error occurred

|  |
| --- |
|  |

## **Summary:**

In this chapter, we have seen the main difference between SSL and TLS and followed by exploring more about SSL towards securing securable things. At the introduction phase, we discussed one way of TLS which is the basic PKI and it has been in force for many years with WWW (world wide web). We also discussed the most advanced two-way TLS also called mutual TLS which is most secure.

In the next chapter, we will explore the most interesting topic which is docker sockets and how Hyperledger fabric peers benefit with docker sockets.

# Chapter3: All about docker sockets

Have you ever wondered to see this line of code in major applications including Portainer, Hyperledger fabric peer itself `unix:///host/var/run/docker.sock`, in this chapter we will cover what is all about docker.sock.

## **An introduction to Docker daemon AP**I

Docker daemon is nothing but a program running in the background on top of a Unix-like operating system, so now the question is who will start the daemon? Daemon will be started by the system utility and a user does not have a need to do this manually. Docker Daemon in simple words dockerd which is a persistent process that manages the life cycle of the containers. Docker provides us an exclusive REST API in order to interact with the docker daemon which is called Docker Engine API.

So far, I hope many of us know about Docker Engine API. Let us discuss something new or something hidden. The Docker daemon which is (dockerd) can listen for API requests through three dedicated different types of Sockets which are called Unix, fd and TCP. Docker Daemon gives us remote access through API by enabling the tcp socket. Coming to unix socket we generally call it as Unix domain socket which is also called as IPC socket which is by default created at `/var/run/docker.sock`. Finally using systemd socket activation (fd) we can communicate with docker API. Syntactically it is `dockerd -H fd:*//*`

## **Interact with docker daemon API via docker sock**

Basically, Unix Sockets utilize local file systems for communication purposes that is why it required docker.sock file. This communication mechanism makes Unix Sockets faster. There is a tight coupling here every time we will have to mount the host path of docker.sock location to the running container, if the running container wants to talk to the API of the Docker engine. We can conclude the whole thing as Docker Server uses a socket to listen to the REST API, and the clients use the socket to send API requests to the server. Let us see some live interactions using docker.sock

We will use curl in order to communicate, one important thing is we will have to enable a particular flag which is `*--unix-socket*`

Please Install jq tool before proceeding further for a better view in the JSON form.

Note: Hostname is a dummy since we are using IPC, you can use any name as hostname it will work as we expected.

### **List of Available Images**:

curl --unix-socket /var/run/docker.sock http://localhost/images/json | jq

[  
 {  
 "Containers": -1,  
 "Created": 1573781880,  
 "Id": "sha256:9756aed98c6bb0995634d276bbe2006590b231090b4d13e27694a886235ebe5c",  
 "Labels": {  
 "org.hyperledger.fabric.base.version": "0.4.18",  
 "org.hyperledger.fabric.version": "1.4.4" },  
 "ParentId": "",  
 "RepoDigests": [  
 "hyperledger/fabric-peer@sha256:92c2bef91e80f54f6d73a89b796eab1b616f372e2258431f17d50dd0c2ce316b" ],  
 "RepoTags": null,  
 "SharedSize": -1,  
 "Size": 127666443,  
 "VirtualSize": 127666443 }  
]

We can get complete events of docker API through `/events`. We can use --no-buffer in order to get real-time events.

curl --no-buffer --unix-socket /var/run/docker.sock http://localhost/events

After executing the above command, one can see all types of events happening in the daemon, such as creating an image of an event, deleting container an event, creating container an event, etc.

**Some useful commands are the following**:

### **List of running containers**:

curl --unix-socket /var/run/docker.sock http://localhost/containers/json | jq

### **Create a tag to an image**:

curl -i -X POST --unix-socket /var/run/docker.sock "http://localhost/images/a95fgf468dfd/tag?repo=redis&tag=latest"

### **Stop a running container**:

curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/ae66e9b84a26/stop | jq

### **Pull an Image**:

curl --unix-socket /var/run/docker.sock -X POST "http:/v1.24/images/create?fromImage=alpine"

## **How Fabric peers benefit from using docker sock?**

Chaincode installation in the fabric means installation followed by instantiation. Installation means installing the chaincode onto the peers, the compiled version will be deployed on to the peer. Instantiation means joining chaincode to the channel and peers can create a chaincode container on the fly using docker.sock and some supportive images such as ccenv (chaincode builder), baseos (chaincode runtime).

From the source code of Hyperledger fabric a package called core/container which handles all life cycle methods of a chaincode. Basically it uses the Golang version of docker client and using docker.sock it will be able to communicate to the docker daemon or docker-engine API.

## **Summary**:

This chapter started with an interesting question about Unix docker.sock. Answering to the question turns out into a big explanation about docker daemon API followed by exploiting different ways docker exposes its functionalities in order clients to interact with its API layer. We also covered some practical examples and provided some code snippets. Finally, we end the chapter by discussing how fabric peer is getting benefit with docker daemon.

In the next chapter, we will see the prerequisites and installation guide of prerequisites.

# Chapter4: Install Prerequisites

# 

If you haven’t already completed Prerequisites so, you may wish to check that you have all the prerequisites below installed on the servers on which you’ll be developing projects and/or operating Hyperledger Fabric. I would recommend you to go with Ubuntu to complete hands-on for all projects. If you are on windows or mac or ubuntu Make sure at least 24gb Ram and 8cores in the host machine as we deal with minimum 4 virtual machines.

* Oracle VM Virtual Box Ubuntu 16.04
* CURL
* NodeJS install with NVM
* Docker & Docker Compose
* Go Programming Language

## **Install and Setup Virtual Box**

For the installation of a virtual box, please refer to the detailed guide found on the below link

[*https://websiteforstudents.com/install-virtualbox-latest-on-ubuntu-16-04-lts-17-04-17-10/*](https://websiteforstudents.com/install-virtualbox-latest-on-ubuntu-16-04-lts-17-04-17-10/)

After you install the virtual box proceed for four virtual servers through installing Ubuntu server 16.04 link below and clone take the help of below screenshots.

|  |
| --- |
| *http://releases.ubuntu.com/16.04/ Download > "Ubuntu-16.04.5-server-amd64.iso"* |

Installing a ubuntu 16.04 OS is pretty straight forward, once we install it we can start the OS before we start the OS, we will have to make settings in the oracle virtual box. Screenshots are mentioned below. Change the network adaptor to the bridged adaptor.

|  |
| --- |
|  |

## **Install dependencie**s

There are some dependencies need to be installed in order to proceed to do hands-on with Hyperledger fabric

**Note**: The Internet is flooded with installation guides, so no point reinventing the wheel, instead we can make use of some good articles. We will have total two Plans.

Plan A) install each and every prerequisite manually by referencing some good tutorials

Plan B) There is an installation script available from Hyperledger fabric we can use that.

### **PlanA**:

**Docker and Compose**: A self-explanatory comprehensive document from the docker website itself to install docker, please find the link below in order to install docker and docker-compose.

[Get Docker Engine - Community for Ubuntu](https://docs.docker.com/install/linux/docker-ce/ubuntu/)

[Install Docker Compose](https://docs.docker.com/compose/install/)

**Minikube**: A detailed document to install minikube from Kubernetes website itself which is called Kubernetes docs

[Install Minikube](https://kubernetes.io/docs/tasks/tools/install-minikube/)

**Node.js**:

Installing node js using nvm is highly recommendable. Please find the installation guide from below link

[How to install Node.js on Ubuntu 16.04/18.04 using NVM (Node Version Manager)](https://hackernoon.com/how-to-install-node-js-on-ubuntu-16-04-18-04-using-nvm-node-version-manager-668a7166b854)

**Golang** :

One can find a tutorial for installing the Golang from below mentioned URL

[How To Install Go 1.13 on Ubuntu 18.04 & 16.04 LTS](https://tecadmin.net/install-go-on-ubuntu/)

### **PlanB**:

When Hyperledger Composer was active the community had released a prerequisites script. We can make use of that and which is still working. However, if there is a sudden installation halt or installation error, no need to worry/panic. one can easily solve such problems by looking into the error stack trace that contains detailed messages.

STEP1: curl -O https://hyperledger.github.io/composer/v0.19/prereqs-ubuntu.sh  
STEP2: chmod u+x prereqs-ubuntu.sh  
STEP3: ./prereqs-ubuntu.sh

**Note**: In the installation script provided above there is no room for installation instructions of minikube and Golang, one can install manually.

**Tip**: Installing Virtual Box is not a mandatory step. If you want to do all practical sessions in an isolated environment then some sort of virtualization or cloud-based virtual instance will be helpful. Generally, a normal laptop or desktop will be sufficient.

## **Tips on Errors, Tricks and Best Practices**

**Error: node-gyp errors**

gyp ERR! build error gyp ERR! stack Error: `make` failed with exit code: 2 gyp ERR! stack at ChildProcess.onExit (/usr/local/lib/node\_modules/npm/node\_modules/node-gyp/lib/build.js:270:23)  
gyp ERR! stack at emitTwo (events.js:87:13)  
gyp ERR! stack at ChildProcess.emit (events.js:172:7)  
gyp ERR! stack at Process.ChildProcess.\_handle.onexit (internal/child\_process.js:200:12)  
gyp ERR! System Darwin 14.5.0

## **Solution:**

Always install Nodejs by using NVM (Node Version Manager). Do not skip python installation. Always use the stable version of NodeJS

## **Summary**:

In this chapter, we have seen the complete list of dependencies and their installation guides. We have utilized fabric composer installation script to finish our installation of prerequisites and alternatively, we have seen individual installation instructions.

In the next chapter, we will discuss complete details and its functionalities of Hyperledger fabric CLI.

# Chapter5: All about fabric CLI

We all know about what the CLI abbreviation is and for sure here I am not going to tell what CLI is instead let us memorize what we can do with CLI generally. Whenever an open-source community or a specific vendor builds us a project then the first tool we can expect from the project is the CLI, it is useful for testing functionalities by the project maintainers or project consumers. Many of us think that CLI is for advanced users but that is not at all true and CLI is for all types of users. Hyperledger Fabric comes with both CLI and GUI (SDK + React), CLI is built with Golang. CLI consumes fewer resources when we compare with other modes. Installation for Hyperledger Fabric CLI is easy because of docker. Yes, CLI comes as a docker container.

**FYI**: Here in the Fabric context CLI is nothing but a peer and as we know Hyperledger fabric is strictly identity-based and for any operation from any entity requires identity and even for CLI. For this purpose, many of us just supply one of the peer identities to the CLI docker container.

After we successfully deployed the CLI docker container we can make use of exec. docker exec will execute commands in the running docker container. All we have to do is exec with interactive mode.

**Note**: docker exec only runs with the container’s primary process ID 1 (PID1). Command will run in the default directory. We can change to whichever directory we want.

docker exec -it {container\_id} bash

## **CLI Basic Operations**

We are here to learn about basic operations and commands of CLl, hence we can avoid long repeated theories so that our focus is only on command and operation

### **Create Channel**

Create Channel operation allow us to create a brand new channel by supplying channel.tx file

**Input**: channel.tx, tls certificate, channel Name, orderer address

**Command**:

peer channel create -o my-org-orderer:7050 -c myorgchannel -f /etc/hyperledger/artifacts/myorgchannel.tx --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

### **Join Channel**

The Join Channel operation allows us to onboard a set of peers to the newly created channel by supplying channel.block file. Channel.block needed to know the anchor peers

**Input**: channel.block, tls certificate

**Command**:

peer channel join -b myorgchannel.block --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

### **Install Chaincode**

Install Chaincode operation allows us to deploy chaincode to a set of peers. In order to successfully install chaincode we should mount the local path of the chaincode running container.

**Input**: chaincode\_name, chaincode\_version, chaincode\_language\_type, chaincode\_path, tls certificate

**Command**:

peer chaincode install -n {chaincode\_name} -v v0 -l Golang -p github.com/chaincode/mycc/go --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

**Refer mount instructions**:

- ./chaincode/src/github.com/:/opt/gopath/src/github.com/chaincode

### **Instantiate Chaincode**

Instantiate Chaincode operation allows us to onboard chaincode to the channel. We should instantiate chaincode only once per channel in order to complete the deployment of the chaincode.

**Input**:

chaincode\_name, channel\_name, chaincode\_args,orderer\_address, chaincode\_version, chaincode\_language\_type, tls certificate

**Command**:

peer chaincode instantiate -o my-org1-orderer:7050 -C myorgchannel -n {chaincode\_name} -l Golang -v v0 -c '{"Args":[""]}' -P "AND(my-org1.Peer, my-org2.Peer)" --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

Above command restricting endorsement only to Peer Roles. If we miss -P then a default policy will be applied automatically.

**A default policy**: Any member of the channel organizations can endorse. For example, a channel called mychannel with “my-org1” and “my-org2” would have a default endorsement policy of OR(my-org1.member', my-org1.member').

### **Invoke Chaincode**

Invoke Chaincode operation allows us to call one of the methods written in the chaincode.

**Input**: chaincode\_name, channel\_name, chaincode\_args, orderer\_address, tls certificate

**Command**:

peer chaincode invoke -o org1-orderer:7050 --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt -C public -n rate -c '{"Args":["createRate","001","CAR BUSINESS", "7Lakhs"]}'

### **Query Chaincode**

Query Chaincode operation allows us to query data present in the world state. Orderer address is not needed as query operation is fulfilled by the peer.

**Input**: chaincode\_name, channel\_name, chaincode\_args, tls certificate

**Command**:

peer chaincode query -C public -n rate -c '{"Args":["queryRate","001"]}' --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

## **CLI Advanced Operations**

As many of us are aware of only basic operations and commands, which we see at the basic operations section. If we observe carefully CLI is using a special image from the hyperledger fabric repository called hyperledger/fabric-tools. It is clear that CLI can access all the available tools in the fabric those are, cryptogen, configtxgen, configtxlator, protolator, idemixgen, peer, etc. These tools are needed for advanced operations.

### **Retrieve present Configuration of a Channel**

Get the present configuration of a channel where we can see the total number of organizations, total no of orderers, etc. what not? a complete blueprint of a channel just by fetching the latest configuration block.

**Input**: chaincode\_name, channel\_name, orderer\_address, tls certificate

**Command**:

peer channel fetch config config\_block.pb -o org1-orderer:7050 -c {channel\_name} --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

### **Sign Config Transaction**

Sign a config transaction operation by an organization admin which left us a signature which is needed by update channel operation.

**Input**: updated\_config\_as\_proto\_file, channel\_name, orderer\_address, tls certificate

**Command**:

peer channel update -f config\_update\_as\_envelope.pb -c myorgchannel -o myorg1-orderer:7050 --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

**Note**: We need to do this operation many times. How Many times it completely depends on the majority of the organizations. Ex: If a channel has three organizations then two organizations admin signatures are mandatory in order to update the channel

### **Update Channel**

Update Channel operation allows us to onboard or deboard organizations to a particular channel at any time.

**Input**: updated\_config\_as\_proto\_file, channel\_name, orderer\_address, tls certificate

**Command**:

peer channel update -f config\_update\_as\_envelope.pb -c {channel\_name} -o myorg1-orderer:7050 --tls --cafile /etc/hyperledger/crypto/peer/tls/ca.crt

**Note**: Generating the update channel payload.pb with detailed guided information is completely out of scope and there are many online articles there for that. The best article from IBM developer tutorials mentioned below.

<https://developer.ibm.com/tutorials/cl-add-an-organization-to-your-hyperledger-fabric-blockchain>

### Register an Identity

Create crypto materials such as SSL credentials for users, orderers, and even for peers by using fabric-CA instead of using cryptogen tools. This is the production recommended way of creating the crypto materials because cryptogen generated materials cannot be able to track their lifecycle using fabric-CA as they are all pre-generated and fabric-CA does not have a clue or persistence in CA database.

**Note**: In order to register a new identity, you must be fabric-CA admin so please make sure you have the fabric-CA credentials and TLS certificate if it is enabled with TLS

Below environment variables may be needed and be sure to pass those while interacting with the fabric-CA server

export FABRIC\_CA\_SERVER\_CA\_NAME=ca.{CA\_NAME}  
export FABRIC\_CA\_CLIENT\_HOME=$HOME/cas/$FABRIC\_CA\_SERVER\_CA\_NAME

export FABRIC\_CA\_CLIENT\_TLS\_CERTFILES=/etc/hyperledger/fabric-ca-server-config/ca.{COMPLETE\_NAME}.pem

**Input**: Name, Secret, ORG, tls Attributes\_if\_any, Country Code, Organization, Location

**Command**:

fabric-ca-client register -d --csr.cn $NAME --id.name $NAME --id.secret $GENERATE\_ANY\_SECRET --id.affiliation $ORG --id.attrs 'hf.Revoker=true,admin=true:ecert' --csr.names C={COUNTRY\_CODE},O=$ORG,L={LOCATION}

Please Refer for country code website <https://countrycode.org/>

**Note**: If you do not want to generate secret then please do not pass secret, as fabric-CA will generate one for us once it registers the identity successfully.

**Note**: Please be noted that in order to register an identity you must enroll the fabric-CA admin and your client will store the credentials in the following directory and these credentials are needed by the client for registering new identities.

$HOME/cas/$FABRIC\_CA\_SERVER\_CA\_NAME

### Enroll Identity

Once we have successfully registered an identity then it is time to enroll. Enroll commands give us the crypto materials like identity private key, PEM certificate, etc. In a typical organization, CA admin registers an identity and passes the secret to Hyperledger fabric developer, and the developer can get the crypto materials by enrolling the already registered identity.

fabric-ca-client enroll -d -u https://$NAME:$SECRET@$FABRIC\_CA\_SERVER\_CA\_NAME:7054 -M /tmp/tls --csr.hosts $NAME,20.46.147.51 --csr.names C={COUNTRY\_CODE},O=$ORG,L={LOCATION}

**Note**: we can pass SANS and IPs if any during the enrollment and this information will be embedded in the generated certificate.

## Summary:

In this chapter, we have started discussing CLI and slowly jumped into the fabric CLI and we have divided the Fabric CLI functionalities into basic and advanced. Both basic and advanced explanations are covered with commands with necessary examples.

In the next chapter, we will discuss complete details and its functionalities of Hyperledger fabric Golang SDK and NodeJS SDK.

# Chapter6: All about SDK's (go lang and Node.js)

We have seen a lot of open-source projects coming with rich SDKs in order to provide a rich implementation of the solutions with ongoing projects. Moving forward one should clearly understand the difference between an API and SDK. At the high-level API is simply an interface that allows us to interact with another piece of software and on the other hand, an SDK provides us a set of tools and libraries, rich documentation, code samples, etc. that allow developers to develop rich and enhanced software applications. However, Hyperledger fabric provides us SDKs in almost all popular languages but lets us restrict our focus to Golang and node js.

## **The basic setup of Golang SDK**

The entire project of Hyperledger fabric is built with Golang. In Fact, the tools we use cryptogen, configtxgen, and configtxlator are also built with Golang. Such a beautiful Golang is. The only sad part here is that there is not much active community support for developing Golang SDK for fabric. Alternatively, there is a big community for nodejs SDK. One can observe this easily. Simply go to their GitHub handlers and check for commits, releases, and versions.

Please complete the Golang installation before proceeding further. One can get help while installing the same in chapter 4. After the installation is done, import the SDK by using the command below.

|  |
| --- |
| go get github.com/hyperledger/fabric-sdk-go |

The Hyperledger fabric built-in go. We are now using Golang SDK and if we use chaincode also in go then this is the complete stack of Golang and we now have all the advantages of using Golang. Before proceeding further let us see a gift from ChainHeros. A project which is a sample implementation of the first app using Golang SDK. Take a look and we will explain many from this implementation.

<https://github.com/chainHero/heroes-service>

**Tip**: One can use an echo framework or simply can use a gorilla/mux HTTP router in order to minimize the complexity. One can use any boilerplate code to jump into the logic completely instead of wasting time setting up the boilerplate code.

There are two pieces that need to be completed before we actually write the operations which are related to the fabric. These pieces are 1) Setup Initialize and Setup MSPCLient

**Setup Initialize**:

Setup Initialize has two sub pieces 1) create SDK 2) create SDK client. Basically, these two operations mainly read the configuration file which we called as connection profile and check for any compile errors. Once the connection profile has been loaded successfully then we can start interacting using the client. Let us first create a struct and we will attach all necessary methods to that struct.

|  |
| --- |
| **type** FabricSetup struct {  ConfigFile string  ChannelID string  ChainCodeID string  initialized bool  ChannelConfig string  ChaincodeGoPath string  ChaincodePath string  OrgAdmin string  OrgName string  UserName string  ChaincodeVersion string  client \*channel.Client  admin \*resmgmt.Client  sdk \*fabsdk.FabricSDK  event \*event.Client } |

Now let us write the initialization method. If we observe carefully this method is again calling two other methods. Having struct with parameters and methods just completes our job easier. That is why people always love Golang.

func (setup \*FabricSetup) Initialize() error {  
 // Add parameters for the initialization  
 if setup.initialized {  
 return errors.New("sdk already initialized..")  
 }  
 setup.CreateSDK()  
 setup.SDKClient()  
 golog.Info("Initialization Successful..")  
 setup.initialized = true  
 return nil  
}

func (setup \*FabricSetup) CreateSDK() error {  
 // Initialize the SDK with the configuration file  
 sdk, err := fabsdk.New(config.FromFile(setup.ConfigFile))  
 if err != nil {  
 return errors.WithMessage(err, "failed to create SDK")  
 }  
 setup.sdk = sdk  
 golog.Info("SDK has been created successfully..")  
 return nil  
}

func (setup \*FabricSetup) SDKClient() error {  
 resourceManagerClientContext := setup.sdk.Context(  
 fabsdk.WithUser(setup.OrgAdmin),  
 fabsdk.WithOrg(setup.OrgName))  
 resMgmtClient, err := resmgmt.New(resourceManagerClientContext)  
 if err != nil {  
 return errors.WithMessage(err, "failed to create channel management client from Admin identity")  
 }  
 setup.admin = resMgmtClient golog.Info("Ressource management client created")  
 return nil  
}

**Note**:

The resource management client is responsible for managing channels (create/update channel).

Now we have the base almost ready and let us proceed to complete the piece of cake. From the main function, we will have to call the initialization method. As we already know, this method is the preload method.

fSetup := bcservices.FabricSetup{  
 //OrdererID: "rak-orderer-rak",  
 ChannelConfig: os.Getenv("GOPATH") + "/src/github.com/narendranath/gofabsdk/config/artifacts/mychannel.tx",  
 ChaincodeGoPath: os.Getenv("GOPATH"),  
 OrgAdmin: "Admin",  
 OrgName: "myorg",  
 ConfigFile: "config/config.json",  
 }

err := fSetup.Initialize()  
 if err != nil {  
 fmt.Printf("Unable to initialize the Fabric SDK: %v\n", err)  
 return }  
 // Close SDK  
 defer fSetup.CloseSDK()

**Note**: we will fill all the necessary struct fields only when we are required.

We have successfully initialized the SDK and the client. Before we start with the operations, we will have to do a last piece of cake, which is MSP client and signing identity. MSPClient allows us to retrieve user information from their identity, like its signing identity which we will need to create the channel and some of the admin related tasks.

mspclient "github.com/hyperledger/fabric-sdk-go/pkg/client/msp"

func (setup \*FabricSetup) MSPClient() (\*mspclient.Client, error) {  
 mspClient, err := mspclient.New(  
 setup.sdk.Context(),  
 mspclient.WithOrg(setup.OrgName),  
 )  
 if err != nil {  
 return mspClient, errors.WithMessage(err, "failed to create MSP client")  
 }  
 return mspClient, nil  
}

It is clear that the MSP client is org based. What it means is the loaded client instance is directly proportional to one particular organization.

We can clearly understand that MSPClient is needed in order to get the signing identity. There is a clear dependency here. We will call these methods from creating channels, installing and instantiating chaincodes, etc.

func (setup \*FabricSetup) GetSigningIdentity(mspClient \*mspclient.Client) (mspctx.SigningIdentity, error) {  
 adminIdentity, err := mspClient.GetSigningIdentity(setup.OrgAdmin)  
 if err != nil {  
 return nil, errors.WithMessage(err, "failed to get admin signing identity")  
 }  
 return adminIdentity, nil  
}

## **Advanced Operations using Golang**

### **Create Channel**:

Create Channel operation initiates with the supplied MSP client and a signing identity that we have done in the basic setup. Basically this allows us to create a brand new channel by supplying a channel.tx file to the channel config path, a channel ID, and an array of signing identities. Sometimes channel policies are very strict. Majority organization admin signatures are needed in order to create the channel (refer Chapter 11).

func (setup \*FabricSetup) CreateChannel() error {  
 mspClient, err := setup.MSPClient()  
 adminIdentity, err := setup.GetSigningIdentity(mspClient); if err != nil {  
 return err }  
 req := resmgmt.SaveChannelRequest{  
 ChannelID: setup.ChannelID,  
 ChannelConfigPath: setup.ChannelConfig,  
 SigningIdentities: []msp.SigningIdentity{adminIdentity},  
 }  
 // resmgmt.WithOrdererEndpoint(setup.OrdererID)  
 txID, err := setup.admin.SaveChannel(req)  
 if err != nil || txID.TransactionID == "" {  
 return errors.WithMessage(err, "failed to create channel")  
 }  
 golog.Info("Channel has been created successfully")  
 return nil  
}

**Note**: No need to supply the orderer endpoint, as it will be loaded from the supplied connection profile.

### **Install Chaincode**:

The main core part of the install chaincode operation is preparing the chaincode package. This will be completed by gopackager. Install Chaincode operation allows us to deploy chaincode to a set of peers. CCPkg will package the chaincode from the supplied chaincode path. Go SDK provides a chance to supply some specific options. Options like retry, selection of specific peers for installation, etc.

|  |
| --- |
| packager "github.com/hyperledger/fabric-sdk-go/pkg/fab/ccpackager/gopackager" |

func (setup \*FabricSetup) CCPkg(ChaincodePath string) (\*resource.CCPackage, error) {  
 ccPkg, err := packager.NewCCPackage(ChaincodePath, setup.ChaincodeGoPath)  
 if err != nil {  
 return nil, errors.WithMessage(err, "failed to create chaincode package")  
 }  
 golog.Info("ccPkg created")  
 return ccPkg, nil  
}

The complete install chaincode function as below.

func (setup \*FabricSetup) InstallCC(peers []string) error {  
 // Create the chaincode package that will be sent to the peers  
 ChainCodePath := "github.com/" + setup.ChainCodeID + "/go/"  
 ccPkg, err := setup.CCPkg(ChainCodePath)  
 // Install example cc to org peers  
 installCCReq := resmgmt.InstallCCRequest{  
 Name: setup.ChainCodeID,  
 Path: ChainCodePath,  
 Version: setup.ChaincodeVersion,  
 Package: ccPkg,  
 }  
 \_, err = setup.admin.InstallCC(installCCReq)  
 if err != nil {  
 return errors.WithMessage(err, "failed to install chaincode..")  
 }  
 golog.Info("Chaincode has been installed successfully..")  
 return nil  
}

We can supply comma-separated options to the installCC function and below listed some of the options.

resmgmt.WithRetry(  
 retry.DefaultResMgmtOpts,  
 )

WithTargetEndpoints is a Variadic function and it takes any arbitrary number of strings as arguments

resmgmt.WithTargetEndpoints(  
 peers...,  
 )

### **Join Channel**:

Join Channel operation allows us to onboard a set of peers to the newly created channel. By SDK gift here also we can supply the options.

func (setup \*FabricSetup) JoinChannel(peers []string) error {  
 If err := setup.admin.JoinChannel(setup.ChannelID); err != nil {  
 return errors.WithMessage(err, "failed to make admin join channel")  
 }  
 golog.Info("Peers joined channel successfully..")  
 return nil  
}

### **Instantiate Chaincode**:

As we all know that instantiate means onboarding the chaincode to the channel with a set of policies. We can write chaincode endorsement policies using the cauthdsl package. One thing we have to remember is that we can supply arguments only as a byte array. One should convert it into a byte array.

|  |
| --- |
| "github.com/hyperledger/fabric-sdk-go/third\_party/github.com/hyperledger/fabric/common/cauthdsl" |

func (setup \*FabricSetup) InstantiateCC(peers []string) error {  
 ccPolicy := cauthdsl.SignedByAnyMember([]string{"myorgMSP"})// Set up chaincode policy  
 resp, err := setup.admin.InstantiateCC(  
 setup.ChannelID,  
 resmgmt.InstantiateCCRequest{  
 Name: setup.ChainCodeID,  
 Path: setup.ChaincodeGoPath,  
 Version: setup.ChaincodeVersion,  
 Args: [][]byte{[]byte("")}, //No args aka empty args  
 Policy: ccPolicy,  
 },  
 )  
 if err != nil || resp.TransactionID == "" {  
 return errors.WithMessage(err, "failed to instantiate the chaincode")  
 }  
 golog.Info("Chaincode Instantiation Successful")  
 return nil  
}

### **Invoke:**

Invoke transactions will be created by importing the channel context. Channel context defines a particular channel a transaction to be sent. Later followed by creating a client for the event. In order to confirm whether the transaction has been created successfully or not.

func (setup \*FabricSetup) Invoke(args []string) (string, error) {  
 clientContext := setup.sdk.ChannelContext(setup.ChannelID, fabsdk.WithUser(setup.OrgAdmin))  
 If client, err := channel.New(clientContext); err != nil {  
 golog.Error("failed to create new channel client")  
 }  
 setup.client = client golog.Info("Channel client has been created")  
 // Creation of the client which will enable access to our channel events  
 setup.event, err = event.New(clientContext)  
 if err != nil {  
 fmt.Printf("failed to create new channel client for event")  
 }  
  
 golog.Info("Event client created")  
 eventID := "eventInvoke" // Prepare arguments  
 reg, notifier, err := setup.event.RegisterChaincodeEvent(setup.ChainCodeID, eventID)  
 if err != nil {  
 return "", err  
 }  
 defer setup.event.Unregister(reg)  
 // Create a request (proposal) and send it  
 response, err := setup.client.Execute(  
 channel.Request{  
 ChaincodeID: setup.ChainCodeID,  
 Fcn: args[0],  
 Args: [][]byte{args...},  
 },  
 )  
 if err != nil {  
 return "", fmt.Errorf("failed to Invoke: %v", err)  
 }  
 // Wait for the result of the submission  
 select {  
 case ccEvent := <-notifier:  
 fmt.Printf("Received CC event: %v\n", ccEvent) //Here we will know success|Failure  
 case <-time.After(time.Second \* 20):  
 return "", fmt.Errorf("did NOT receive CC event for eventId(%s)", eventID)  
 }  
  
 return string(response.TransactionID), nil  
}

### **Query transaction**:

Query Chaincode operation allows us to query data present in the world state

func (setup \*FabricSetup) Query(args []string) (string, error) {  
 clientContext := setup.sdk.ChannelContext(setup.ChannelID, fabsdk.WithUser(setup.OrgAdmin))  
 client, err := channel.New(clientContext)  
 setup.client = client If response, err := setup.client.Query(  
 channel.Request{  
 ChaincodeID: setup.ChainCodeID,  
 Fcn: args[0],  
 Args: [][]byte{[]byte(args[1])},  
 },  
 ); err != nil {  
 return "", fmt.Errorf("failed to query: %v", err)  
 }  
 return string(response.Payload), nil  
}

### **Register Identity**:

As we all know, the fabric is a permissioned based consortium and we need a certificate-based identity. The certificate creation process includes two steps. First, it starts with registering the identity and followed by enrolling. Registration allows us to provide some attributes which will be used as attribute-based access control at the chaincode level.

func (setup \*FabricSetup) RegisterIdentity() error {  
 mspClient, err := setup.MSPClient()  
 if err != nil {  
 return err }  
 req := &msp.RegistrationRequest{  
 Name: "Admin@myorg",  
 Type: "user",  
 MaxEnrollments: 1,  
 Affiliation: "myorg",  
 Attributes: []msp.Attribute{  
 {  
 Name: "admin",  
 Value: "true",  
 ECert: true,  
 },  
 {  
 Name: "hf.Revoker",  
 Value: "true",  
 },  
 },  
 CAName: "myorg-rca",  
 Secret: "Admin@myorg",  
 }  
 \_, err = mspClient.Register(req)  
 if err != nil {  
 return errors.WithMessage(err, "failed to register identity")  
 }  
 log.Info("Identity has been registered successfully..")  
 return nil  
}

### **Enroll Identity**:

After registering the identity. Enrollment is a must in order to get the certificates.

func (setup \*FabricSetup) EnrollIdentity() error {  
 If mspClient, err := setup.MSPClient(); err != nil {  
 return err }  
 If err = mspClient.Enroll("Admin@myorg",  
 msp.WithSecret("Admin@myorg"),  
 msp.WithProfile("tls"),  
 ); err != nil {  
 return errors.WithMessage(err, "failed to register identity")  
 }  
 log.Info("Identity has been registered successfully")  
 return nil  
}

## **Basic Operations using Node.js**

Hyperledger Fabric is built-in Golang but it provides SDKs in almost all popular languages. There is a big community from all over the world for nodejs out of all languages. We can see nodejs SDK releases are almost parallel with the Hyperledger fabric releases. SDK provides us with a set of packages in order to fulfill our requirements. SDK is completely modular and enables us to write clean code with more implementations of key functions. Installation of the nodejs SDK is initiated by adding the fabric-client and fabric-ca-client details to the dependencies section in the package JSON file.

|  |
| --- |
| "fabric-ca-client": "~1.4.4", "fabric-client": "~1.4.4" |

Please refer to the below-mentioned project from Hyperledger fabric. We will explain from this repo

<https://github.com/hyperledger/fabric-samples/tree/release/balance-transfer>

**Tip**: One can use an express framework or simply can use any HTTP router in order to minimize the complexity. One can use any boilerplate code to jump into the logic completely instead of wasting time setting up the boilerplate code.

Unlike Golang SDK will have to just write a utility function in order to load the client for a particular organization. Loading clients includes the loading of the connection profile. The connection profile will be in YAML or JSON format only. We will follow the normal style instead of a gateway pattern as it hides all the functionalities and just provides one-line code for any operation which is not good for learners or beginners.

### **Setup Initialize**:

Setup Initialize is the first operation before initiating actual operations of the fabric. These operations construct clients from the supplied connection profile.

const hfc = require('fabric-client');  
 let config = '-connection-profile-path';  
 client = hfc.loadFromConfig(hfc.getConfigSetting('network' + config));  
 await client.initCredentialStores();  
 return client;

Implementation of all below snippets can be found in one of the chapters Git:

<https://github.com/narendranathreddythota/masteringhyperledgerfabric/tree/master/chapter-8/portals/opendata/app>

### **Create Channel**:

Starting from the create channel actual operations related to the fabric begins. Channel creation needs a channel.tx binary file as it contains channel configuration.

var envelope = fs.readFileSync(path.join(\_\_dirname,"blockchain/crypto/" + this.channelName + ".tx"));  
 var channelConfig = this.client.extractChannelConfig(envelope);  
 let signature = this.client.signChannelConfig(channelConfig);  
  
 let request = {  
 config: channelConfig,  
 signatures: [signature],  
 name: this.channelName,  
 txId: this.client.newTransactionID(true) // get an admin based transactionID  
 };  
 // send to orderer  
 var response = await this.client.createChannel(request)

### **Join Channel**:

The first thing needed in the join channel operation is genesis block. We can fetch the same using a channel client. Once we get the org client, we can get the channel client as well by simply calling getChannel method. Unlike creating channels, joining channels is a bit tricky. Since joining channels is joining (n) No peers at the same time so we need the status of each peer before we conclude the overall success result.

NewOrderer Syntax:

Public newOrderer(url: string, opts?: Client.ConnectionOpts): Client.Orderer;

const orderer = client.newOrderer('grpcs://localhost:7050',  
 {  
 pem: Buffer.from(data).toString(),  
 'ssl-target-name-override': orderer.org1.example.com'  
 }  
);

const channel = client.newChannel(channel\_name);  
 channel.addOrderer(orderer);

let genesis\_block = await channel.getGenesisBlock(request);  
 let join\_request = {  
 targets: peers,   
 txId: this.client.newTransactionID(true),  
 block: genesis\_block  
 };  
let peers\_results = this.channel.joinChannel(join\_request, 30000);

for (let i in peers\_results) {  
 let peer\_result = peers\_results[i];  
 if (peer\_result.response && peer\_result.response.status == 200) {  
 console.log('Successfully joined peer to the channel');   
 } else {  
 console.log('Failed to join');   
 }  
 }

### **Install Chaincode**:

Installing chaincode on peers involves two operations. The install operation on all peers is as usual and the other one is handling the chaincode source files. Remember that just chaincode installation is not enough and there is a need to initiate instantiation operation.

We must set the GOPATH env variable. In order, the peers are able to identify the chaincode location.

process.env.GOPATH = hfc.getConfigSetting('CC\_SRC\_PATH');  
 var request = {  
 chaincodePath: chaincodePath,  
 chaincodeId: this.chaincodeName,  
 chaincodeVersion: this.chaincodeVersion,  
 chaincodeType: this.chaincodeType };  
let results = await this.client.installChaincode(request);  
var proposalResponses = results[0];

Now, let's check the results of each peer, status of install chaincode on peer

for (var i in proposalResponses) {  
 let one\_good = false;  
 if (proposalResponses && proposalResponses[i].response && proposalResponses[i].response.status === 200) {  
 one\_good = true;  
 logger.info('install proposal was good');  
 } else {  
 stackError.push(proposalResponses[i].peer.name + ": " + proposalResponses[i] +" ");  
 }  
 all\_good = all\_good & one\_good;  
 }

**Ex**: See chaincode full path vs 'CC\_SRC\_PATH' **vs** chaincodePath

/Volumes/book/blockchain/chaincode/src/github.com/sample/go/sample.go 'CC\_SRC\_PATH' = /Volumes/book/blockchain/chaincode chaincodePath = github.com/sample/go

### **Instantiate Chaincode**:

Instantiate chaincode is joining a chaincode into the channel. There are a lot to submit while along with instantiate proposals. Endorsement policies, collection configs, etc. Instantiate operation is a bit complex for fabric. First it compiles the chaincode, only if successful it will talk to docker daemon and create a docker image followed by creating a chaincode container. Please refer to the docker sockets section for a detailed explanation.

var tx\_id = this.client.newTransactionID(true);  
var deployId = tx\_id.getTransactionID();  
 const collectionsConfigPath = path.join(\_\_dirname, '../privateData/privateColl.json');  
 var request = {  
 chaincodeId: this.chaincodeName,  
 chaincodeType: this.chaincodeType,  
 chaincodeVersion: this.chaincodeVersion,  
 args: args,  
 txId: tx\_id,  
 'collections-config': collectionsConfigPath };  
let results = await this.channel.sendInstantiateProposal(request, 6000000);  
var proposalResponses = results[0];

We will have to see all responses. As it will create chaincode containers for more than one peer. Once all results are good then we will have to send a transaction to the channel again with all proposal responses.

for (var i in proposalResponses) {  
 let one\_good = false;  
 if (proposalResponses && proposalResponses[i].response && proposalResponses[i].response.status === 200) {  
 one\_good = true;  
 logger.info('instantiate proposal was good');  
 } else {  
 logger.error('instantiate proposal was bad');  
 }  
 all\_good = all\_good & one\_good;  
 }

var orderer\_request = {  
 txId: tx\_id,  
 proposalResponses: proposalResponses,  
 proposal: proposal };  
var result = this.channel.sendTransaction(orderer\_request);

### **Invoke Transaction**:

Now it is time to push our data to the channel. This is called an invoke transaction. We will not simply fire and forget. We will send a transaction and will see the status of the transaction by subscribing to an event.

var tx\_id = this.client.newTransactionID();  
 var request = {  
 chaincodeId: chaincodeName,  
 fcn: fcn,  
 args: args,  
 chainId: this.channelName,  
 txId: tx\_id };  
 let results = await channel.sendTransactionProposal(request, 60000);  
 var proposalResponses = results[0];

As a client sends the transaction to the different endorsement peers and we need a loop in order to see the result coming from different peers.

for (var i in proposalResponses) {  
 if (proposalResponses && proposalResponses[i].response && proposalResponses[i].response.status === 200) {  
 logger.info('invoke chaincode proposal was good');  
 } else {  
 logger.error('invoke chaincode proposal was bad');  
 }  
 }

var orderer\_request = {  
 txId: tx\_id,  
 proposalResponses: proposalResponses,  
 proposal: proposal };  
var sendPromise = this.channel.sendTransaction(orderer\_request);

### **Query Chaincode**:

There are a few presents who are opposite to those who push the data. The nonother than a few queryingdata from the world state through a peer query command.

var tx\_id = this.client.newTransactionID();  
var request = {  
 chaincodeId: chaincodeName,  
 fcn: fcn,  
 args: args,  
 txId: tx\_id };  
let response\_payloads = await this.channel.queryByChaincode(request, 6000);

for (let i in response\_payloads) {  
 if (response\_payloads[i].toString('utf8') == '') {  
 console.log("peer" + i + " " + "response\_payloads is null")  
 } else if (response\_payloads[i] instanceof Error) {  
 console.log("peer" + i + " " + response\_payloads[i])  
 } else {  
 console.log("Peer" + i + ": " + response\_payloads[i].toString('utf8'));  
 }  
 }

The above snippet is the looping responses coming from different peers. In order to aggregate the result.

## Advanced Operations using Node.js

Let us see some interesting stuff on the advanced side of node SDK. For CA admin related operations such as register, enroll, re-enroll, get revocation list, revoke a certificate. For these operations, we need a CA admin instance object. For the latest gateway, all these are hidden and just fulfilled by a function called getPresentIdentity. Let us see something really under the hood. All we need to do is build CA admin instances and complete admin related operations.

const Fabric\_Client = require('fabric-client');  
const fabric\_client = new Fabric\_Client();  
 var store\_path = path.join(\_\_dirname, '../bin'); //Pointing to a dummy dir  
 let state\_store = await Fabric\_Client.newDefaultKeyValueStore({  
 path: store\_path  
 });  
 fabric\_client.setStateStore(state\_store);  
 try {  
 let adminInstance = await fabric\_client.createUser({  
 username: username,  
 mspid: mspId,  
 cryptoContent: {  
 privateKeyPEM: enrollment.key, // ca private key  
 signedCertPEM: enrollment.certificate //ca admin certificate  
 },  
 skipPersistence: true //false create admin cereds in the dir  
 });  
 } catch (err) {  
 throw new Error ( +error.stack ? error.stack : error );  
 }

Mentioned sample CA admin key and CA admin certificate below for your reference.

|  |
| --- |
| {  "key": "-----BEGIN PRIVATE KEY-----\r\nMIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgypR4JnSNQ85yMM6F\r\nUkLeUHL5RJAEqsYrZjqYHYH7y3OhRANCAAREOTS76C+AVjKtPWvc4kX4NtC3txJO\r\nNWKdizZFTdVPIUhljGjf7QF5WlbA3G/cINucx5tKIgB42v7cwxaDF8FX\r\n-----END PRIVATE KEY-----\r\n", "certificate": "-----BEGIN CERTIFICATE-----\nMIICCTCCAa+gAwIBAgIUQfjPcE/5veGog/rWQVtPLoH/cdEwCgYIKoZIzj0EAwIw\nYjELMAkGA1UEBhMCVVMxFzAVBgNVBAgTDk5vcnRoIENhcm9saW5hMRQwEgYDVQQK\nEwtIeXBlcmxlZGdlcjEPMA0GA1UECxMGY2xpZW50MRMwEQYDVQQDEwptYWVyc2st\ncmNhMB4XDTE4MTIyNDA4MzQwMFoXDTE5MTIyNDA4MzkwMFowJjEPMA0GA1UECxMG\nY2xpZW50MRMwEQYDVQQDEwptYWVyc2staWNhMFkwEwYHKoZIzj0CAQYIKoZIzj0D\nAQcDQgAERDk0u+gvgFYyrT1r3OJF+DbQt7cSTjVinYs2RU3VTyFIZYxo3+0BeVpW\nwNxv3CDbnMebSiIAeNr+3MMWgxfBV6N/MH0wDgYDVR0PAQH/BAQDAgOoMB0GA1Ud\nJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjAMBgNVHRMBAf8EAjAAMB0GA1UdDgQW\nBBRLecG5ov7yDFNg/P9ngZGGt4je5DAfBgNVHSMEGDAWgBSjYA5K/VFHVmhrcE1n\noU13qY+pCTAKBggqhkjOPQQDAgNIADBFAiEA26ELkJhIgzdxpUMrNAi7YDj8qgou\nJDfs1MFLt9q+h9gCIFkbQH6+eSz1fJW18TiOXOrOHDBog6g7oTK2F0J4X/sa\n-----END CERTIFICATE-----\n", } |

Once we have the admin instance let us proceed with the other operations. Before proceeding further, a small operation is still pending. Create a CA service by supplying the ca endpoint.

const fabricCAServices = require("fabric-ca-client");  
 const caService = new fabricCAServices(  
 caEndPoint, // https://localhost:7054  
 caName //myorg-ca  
 );

### **Register Identity**:

The creation of an identity with a certificate is a bit complex process and the process starts by registering the identity, followed by enrolling the identity in order to get the certificates.

const identityService = caService.newIdentityService();  
 let registerObject = {  
 enrollmentID: "orderer1-myorg",  
 enrollmentSecret: "your secret (optional)" type:"orderer",  
 Affiliation: "myorg",  
 maxEnrollments: maxEnrollments || 1,  
 attrs: [{"name":"hf.Registrar.Roles", "value":"orderer, user"}],  
 Caname: "myorg-ca" };

const response = await identityService.create(  
 registerObject,  
 adminInstance //check above for implementation  
 );

### **Enroll Identity**:

Once registration is completed, we can proceed for enrollment. The output will be a private key and certificate. This also called as E-certs.

let req = {  
 enrollmentID: "orderer1-myorg",  
 enrollmentSecret: "your secret (optional)",  
 profile: "tls" };  
 try {  
 enrollment = await caService.enroll(req);  
 return enrollment;  
 } catch (err) {  
 throw new Error ( +error.stack ? error.stack : error );  
 }

### **Re-Enroll**:

In any case, your certificate has expired or has been compromised. We can apply for enrollment of the new certificate for the same user. No need for a CA admin instance for re-enrollment operation. Instead, we need to supply the current user private key and certificate to our instance builder.

try { //if any attributes  
 enrollment = await caService.reenroll(currentUserInstance, attr\_reqs); // check the above code in order to get current user instances by supplying key and cert.  
 return enrollment;  
 } catch (err) {  
 throw new Error ( +error.stack ? error.stack : error );  
 }

### **Revoke**:

Sometimes users do malicious activity in the consortium. If an admin of an organization suspects some user is doing a malicious activity, then he can revoke the traitor credentials. Remember as an organization admin you need to update the certificate revocation list to all MSPs once revocation of a particular certificate is being done.

var req = { enrollmentID: id\_to\_be\_revoked };  
 return await caService.revoke(req, adminInstance);//check above for implementation  
 } catch (error) {  
 throw new Error ( +error.stack ? error.stack : error );  
 }

### **Generate Certificate Revocation List**:

Once the organization admin’s revoke set of user certificates. It is the time to update all MSPs with updated certificate revocation lists. One can pass revokedBefore variable any date of user choice.

try {  
 var base64Result = await caService.generateCRL(  
 {  
 revokedBefore: new Date()  
 },  
 adminInstance //check above for implementation  
 );  
 } catch (err) {  
 throw new Error ( +error.stack ? error.stack : error );  
 }

### **Block Event Listener**:

Whenever there is a new block generated in a particular channel. Block Listener will listen for us and catch the latest block. We need a channel instance. Please see previous examples where we can see the implementation of channel instances.

const peer = client.newPeer( // Please check previous examples for client instance  
 'grpcs://localhost:7051',  
 {  
 pem: Buffer.from(data).toString(),  
 'ssl-target-name-override': 'peer0.org1.example.com'  
 }  
);

var channel\_event\_hub = channel.newChannelEventHub(peer);  
 channel\_event\_hub.connect(true);  
 channel\_event\_hub.registerBlockEvent(function(response\_payload, err) {  
 if (err) {  
 logger.error('[channelEventHub] Failed to receive the block event ::' + error);  
 return "[channelEventHub] Error while getting the latest block";  
 } else {  
 // response\_payload do whatever we want with the block.   
 }  
 } // registerBlockEvent No error so else  
 }); // responsePayload END

### **Chaincode Event Listener**:

Slice difference in order to listen to chaincode events. One can observe changes only after the channel\_event\_hub instance.

channel\_event\_hub.registerChaincodeEvent("chaincode\_name", '^{pattern}\*',  
 async (event, block\_num, txnid, status) => {  
 var payload = event.payload.toString('utf8');  
 // whichever data we registered to send in the event. That is payload  
 }, (error) => {  
 logger.error('\nFailed to receive the chaincode event' + error);  
 });

In the chaincode make changes as below mentioned:

|  |
| --- |
| APIstub := shim.ChaincodeStubInterface APIstub.SetEvent("record\_created", dataAsBytes) //record\_created is the pattern |

### **Get Present Channel Config As JSON**:

Want to see the present channel configuration as a JSON file then please follow the below-listed steps. We need an HTTP agent (Axios, request, superagent, etc.). We need a channel instance in order to proceed further with the operation. please see below examples in order to get the channel instance.

|  |
| --- |
| > CD fabric-samples/bin/  > ./configtxlator start   const agent = require('superagent-promise')(require('superagent'), Promise); const config\_envelope = await channel.getChannelConfig(); var original\_config\_proto = config\_envelope.config.toBuffer();   Let url = 'http://127.0.0.1:7059' +':7059/protolator/decode/common.Config' let response = await agent.post(url, original\_config\_proto).buffer(); var original\_config\_json = JSON.parse(response.text.toString()); |

### **UpdateChannel**:

Here, we come to the most craving topic: Updating existing channels with one or more organizations. Hyperledger Fabric is a consortium based Blockchain network, it means a group of organizations who have a common activity or common goals together maintain a Blockchain network or consortium in a private permissioned way. So, if any organizations invite new members to the consortium, it should get the majority of organizations' approval.

By default, channel policy in the Hyperledger fabric is the majority of signatures needed. We can modify the channel policy according to business use cases. The majority means if a consortium has 6 organizations, then anyone organization invites new members to the consortium, then it must get approval from the other three organizations 4 out of 6 is the majority in this scenario. What does it mean 4 organization admin signatures are needed? Channel policy mentioned below.

Channel: &ChannelDefaults Policies:  
 # Who may invoke the 'Deliver' API  
 Readers:  
 Type: ImplicitMeta  
 Rule: "ANY Readers"  
 # Who may invoke the 'Broadcast' API  
 Writers:  
 Type: ImplicitMeta  
 Rule: "ANY Writers"  
 # By default, who may modify elements at this config level  
 Admins:  
 Type: ImplicitMeta  
 Rule: "MAJORITY Admins"

**Tip**: Do not follow the patterns of fabric samples in production. If we read all the comments carefully, they advised us not to use it in production. Do not keep a separate organization just for ordering services, this becomes a complete centralized system.

Before proceeding with the action of updating the channel. Let us understand some hidden concepts. Unlike us, the data type is not JSON. A new data type called Protobuf is being used in Hyperledger fabric. We will be dealing with only one tool to complete the onboarding functionalities which is called Configtxlator. This tool is built-in Golang and exposes all functionalities such as encode from JSON to Protobuf and decode from ProtoBuf to JSON.

Following Reasons are behind Fabric to use ProtoBuf instead of JSON

|  |
| --- |
| - Data is fully Typed  - Data is fully compressed (less CPU Usage)  - Schema(message) is needed to generate code and read the code  - Documentation can be embedded in the schema  - Data can be read across any language  - Schema can evolve any time in a safe manner  - faster than XML  - code is generated for you automatically  - Google invented protobuf, they use 48000 protobuf messages & 12000.proto files  - Lots of RPC frameworks, including grpc use protocol buffers to exchange data |

**STEP1**: Start the Configtxlator tool

|  |
| --- |
| ➜ fabric-samples/bin git:(master) ✗ ./configtxlator start 2020-01-21 13:53:43.774 +04 [configtxlator] startServer -> INFO 001 Serving HTTP requests on [::]:7059 |

**Step2**: Let us get the present channel configuration from the channel

//Please see previous snippets to create channel instance

|  |
| --- |
| const config\_envelope = await channel.getChannelConfig(); |

**FYI**: Above snippet asks peers to get the present configuration block of a particular channel. config\_envelope is a javascript object and in order to get a JSON object, we will need configtxlator tool. Configtxlator is working with bytes so will have to convert to buffer.

const agent = require('superagent-promise')(require('superagent'), Promise);

var original\_config\_proto = config\_envelope.config.toBuffer();

let response = await agent.post('http://localhost:7059/protolator/decode/common.Config',  
 original\_config\_proto)  
 .buffer();

var original\_config\_json = response.text.toString();  
var updated\_config\_json = JSON.parse(original\_config\_json);

updated\_config\_json is the JSON format of channel configuration

**STEP3**: Get New Organization details (Policies, certificates, etc.)

|  |
| --- |
| configtxgen -printOrg ${ORG\_NAME} > $ORG\_NAME.json |

**STEP4**: Now append the new organization JSON to existing orgs in the payload

|  |
| --- |
| updated\_config\_json.channel\_group.groups.Application.groups["new-org-name"] = neworg\_config\_json |

Now it is the time to encode back to bytes from JSON using configtxlator tool

var bytesData = await agent.post('http://localhost:7059/protolator/encode/common.Config', updated\_config\_json)  
 .buffer();  
 var updated\_config\_binary = bytesData.body;

Let us memorize what we have done so far:

* Get Present channel config as a javascript object
* Using configtxlator tool convert into JSON
* Generate new organization config as JSON using cryptogen
* Append new organization JSON to the original channel configuration
* Convert the updated channel configuration into a binary format using confitxlator tool

**STEP5**: One Final step is remaining before we update the channel. We will have to get the delta between the original config and update config using configtxlator tool.

const formData =

{  
 channel: channelName,  
 original: {  
 value: original\_config\_proto,  
 options: {  
 filename: 'original.proto',  
 contentType: 'application/octet-stream' }  
 },  
 updated: {  
 value: updated\_config\_binary,  
 options: {  
 filename: 'updated.proto',  
 contentType: 'application/octet-stream' }  
 }  
 };

var delta\_change =

await agent.post({  
 url: 'http://localhost:7059/configtxlator/compute/update-from-configs',  
 encoding: null,  
 headers: {  
 accept: '/',  
 expect: '100-continue' },  
 formData: formData

})

|  |
| --- |
| const finalData = Buffer.from(delta\_change, 'binary'); |

We have got the payload to send to update the channel. According to the channel policy, the majority of the signatures are needed in order to update the channel. Let us go with a happy path that the consortium has only one organization and we are onboarding a second organization.

Var signatures = [client.signChannelConfig(finalData)]   
 request = {  
 config: finalData,  
 signatures: signatures,  
 name: channel\_name,  
 orderer: orderer,  
 txId: client.newTransactionID()  
 };  
 // this will send the update request to the orderer  
 result = await client.updateChannel(request);

That's it. Hola, we have successfully updated the channel using nodejs SDK.

**BonusConcepts**: Everyone is curious to know/see what is inside the mychannel.tx or genesis.block. If you are the person then follow my instructions in order to see what is inside.

STEP1: CD fabric-samples/bin/./configtxlator start

STEP2: GOTO genesis.block dir STEP3:

curl -X POST --data-binary @genesis.block http://127.0.0.1:7059/protolator/decode/common.Block > genesis.json

STEP4: GOTO channel.tx dir

STEP5: curl -X POST --data-binary @mychannel.tx http://127.0.0.1:7059/protolator/decode/common.Envelope > mychannel.json

|  |
| --- |
| Git Repo: https://github.com/narendranathreddythota/masteringhyperledgerfabric/chapter-6 |

## **Tips on Errors:**

**Error**: Policy not satisfied error:

{ status: 'BAD\_REQUEST',  
info: 'error authorizing update: error validating DeltaSet: policy for [Group] /Channel/Orderer not satisfied: Failed to reach implicit threshold of 2 sub-policies, required 1 remaining' }  
[2018-10-24 12:25:04.411] [DEBUG] Update-Channel -- response3 ::{"status":"BAD\_REQUEST","info":"error authorizing update: error validating DeltaSet: policy for [Group] /Channel/Orderer not satisfied: Failed to reach implicit threshold of 2 sub-policies, required 1 remaining"}

**Reason**: While updating the channel make sure you get a majority of signatures from all organizations.

The Detailed steps involved in the update channel are:

|  |
| --- |
|  |

## **Summary:**

In this chapter, we have covered both Golang and NodeJS Hyperledger Fabric SDK’s. Starting from, setting up SDK to interacting with the Fabric consortium through we have covered all aspects and it’s all functionalities including basic and advanced. We have also covered the most craving topic of updating channels programmatically through SDK.

In the next chapter, we will cover the advanced way of coding Hyperledger Fabric chaincodes.

# 

# 

# Chapter7: Advanced Chaincode Development

## **key collision and MVCC**

Key collision and MVCC is an interesting and most wanted topic in the distributed systems. We have many questions in Stack Overflow for this particular topic and still many of us commit many mistakes in this section. Generally, a key collision occurs when submitting multiple transactions at the same time by different parallel clients which aim to update the same key/pair and double-spending attacks occur when a client aims to utilize a particular resource(coin/token) twice instead of once by exploiting flaws in the system. Luckily, Fabric has an MVCC control check at the time of committing, and a double-spending problem/attack will be a nightmare.

Basically, key collision is prevented by the MVCC (Multi-version concurrency check). Initially, MVCC aims to avoid double-spending problems but when parallel clients run intentionally trying to update/change the same key/value pair because of MVCC all transactions marked as conflicts will be rejected by the committing peers. So, developers must use some logic here in order to avoid such issues. Let us see how MVCC works.

Suppose we have a token transfer chaincode with a logic coded in a way that it will first verify that the initiator has a sufficient balance of tokens and then proceed with the transaction. The story starts here as When Bob has 50 tokens and he puts a request to transfer tokens to Alice. All is good because chaincode checks for sufficient balance and it will allow. Just imagine that Bob is trying to take advantage of the nondeterministic time to complete a transaction and try to cheat by initiating two transactions 50 tokens to Alice and 50 tokens to Linda at the same time while having only 50 tokens. Chaincode will allow both transactions because at that time 50 bob has tokens. Such a type of problem is not limited to Hyperledger fabric alone. It is a common problem for databases and we call it a double-spending problem. Wah! There is a solution that is called MVCC, a well-accepted solution introduced to stop double-spending problems. So, by default MVCC is turned on in HLF and we need not worry about it.

Due to endorsing peers, we can easily identify the double-spending problem. When Bob initiates the first token transfer transaction to Alice, endorsing peers do simulation and record the key/value pair data, here Ex: {token value= 50} and the same thing happens for the second immediate transaction to Linda {token value= 50}. When SDK submits the read/write sets and actual payload to the orderer and orderer orders a block to peers. Committing peer first check for VSCC and then MVCC. For the first transaction bob => Alice initial token value at the time of simulation is 50 and ledger token value also 50 so committing peers will commit this to the ledger and now token value=0. For the second transaction to Linda’s initial token value of bob at the time of simulation is 50 but ledger token value of bob is 0 and committing peer marks as conflicts and it will be rejected and appropriate status will be informed to SDK.

According to this approach among all transactions in a block that is aimed to modify/update the same key, only one transaction will succeed. Assume that we have an airline ticket and token issue system with Hyperledger fabric where we have a global variable of token balance and it will be updated frequently when a user purchases an airline ticket. According to this architecture, only one transaction will be successful at a time due to double spending prevention (MVCC). For example, 10 tickets per second then 9 out of 10 tickets will become a failed state and it will be a very bad design. The developer has to code carefully.

Batching is a good technique as many chaincode developers have been using this technique for a while. Design chaincode in such a way that it should handle an array of records and process them correctly. In this approach, simulation results will be the latest value from all other transactions and committing peers to commit all transactions from a block easily as there is no change of value unless intentionally changed. But the better advice is one should not deal with such a situation inside the chaincode.

## **Client Identity Chaincode Library**

We are in the information age where we have witnessed a lot of innovation and advanced technologies with advanced security and cybersecurity also improved a lot. But still, there are security breaches happening here and there. One should understand the importance of access control mechanisms in order to avoid unauthorized access. There are several access controls available in the fabric. The Client Identity library which we called CID, which is focused on access control at the chaincode level. Using CID one can make access control decisions at the chaincode level based on the organization’s client identity.

We all know that fabric has given many options to write chaincode in many languages. But let us restrict one particular language in order to get the complete flavor of a particular topic and in order to avoid confusion. Let us go with Golang. Before proceeding further import the library first.

import "github.com/hyperledger/fabric/core/chaincode/lib/cid"

There are many ways to leverage this technique:

### **Based on MSP ID**

We can restrict access to the restricted data for a particular MSPID. So that we can block/unblock all identities which are issued earlier from a particular MSPID

|  |
| --- |
| mspID, err := cid.GetMSPID(myStub) |

### Based On Attributes

While registering an identity, CA admin can inject some attributes. So that enrollment cert will have attributes present. However, we can write anything as an attribute based on the requirement. Attributes are simply a key/value pair. We can assert an attribute value to fasten access control.

|  |
| --- |
| value, ok, err := cid.GetAttributeValue(myStub, "authName") err = cid.AssertAttributeValue(APIstub, "authName", "abc") |

Ex: department=admin. Here when chaincode gets an attribute as a department with the value of admin then it understands that there is a client who is from the admin department. So based on this further process will happen. Attributes are for better classification in the chaincode.

## **End-to-End Encryption**

Encryption makes our lives easier in this information age by hiding the actual information aka encrypting data. Once the data is encrypted only the authorized client can be able to decrypt the data. This type of technique comes under cryptography and of course, there are many different types available to complete the encryption term. The popular one is symmetric-key cryptography. Where the same key is used to encrypt and decrypt the data. The other one is Asymmetric key cryptography. Where different keys are used to encrypt and decrypt the data. Generally, all enterprises take care of security measures right from setting up the consortium to the data they push to one of the channels, but what if there is a need to have encryption mechanisms at the chaincode level instead of having encryption off the chain? Then this topic is for you

At a high level what we will do is utilizing the packages provided by the fabric for advanced chain coding purposes which contains an implementation of cryptographic standards and algorithms on-demand using bccsp which is called blockchain cryptographic service provider. There are two resources we will be dealing with throughout the section which are bccsp and entity both are packages provided by fabric.

BCCSP: A package from a fabric called BlockChainCryptographicServiceProvider

Entity: A package from fabric contains collective verbs with holding BCCSP instance such as signer, verifier, encryptor, decryptor

Have a look at https://github.com/hyperledger/fabric/blob/master/bccsp/bccsp.go

From the above package, we can see there are two options.

1) with symmetric key encryption (AES256EncrypterEntity)

2) Asymmetric key encryption (ECDSAEncrypterEntity).

For the time being, we shall go with option one Advanced encryption standard. Let us see the function signature and learn a little bit. In Golang return, type do exist next to the input parameters.

NewAES256EncrypterEntity(ID string, b bccsp.BCCSP, key, IV []byte) (\*BCCSPEncrypterEntity, error)

(ID string, b bccsp.BCCSP, key, IV []byte) < Input parameters  
\*BCCSPEncrypterEntity, error) < Output parameters

Let us see each parameter one by one in detail:

**(ID string)**:

The ID type is a string. If we have a requirement to have many such then we need a way to track all of them using IDs. So, for the time being, we can leave it as "ID-1"

**(b bccsp.BCCSP)**: It needs non-ephemeral (long-term) BCCSP and we can get the same from the factory package by issuing the following command. factory.GetDefault()

**Note**: Before calling factory.GetDefault(). we must call factory.InitFactories(nil)

**(key)**: AES\_KEY in the form of byte array for encryption

**(IV []byte)**: This is completely optional, one can ignore

Once we fill the above func with all necessary parameters then it will look like below

Import (  
"github.com/hyperledger/fabric/core/chaincode/shim/ext/entities"  
)  
factory.InitFactories(nil)  
If ent, err := entities.NewAES256EncrypterEntity("ID-1", factory.GetDefault(), encKey, iv); err != nil { return err }

The returned ent object contains two methods which are called Encrypt and Decrypt

### **Encrypt**:

|  |
| --- |
| If ciphertext, err := ent.Encrypt(value\_to\_be\_encrypted) ; err != nil { return err } |

### **Decrypt**:

|  |
| --- |
| If ciphertext, err := ent.Decrypt(ciphertext) ; err != nil { return err } |

Thus, the base encryption and decryption methods are successfully coded and are ready to integrate with our chain codes in order to implement encryption/decryption services at the chaincode level.

## **CI/CD Installing chaincode From GitHub to the peers**

CI/CD with chaincode deployment in the Hyperledger fabric? Yes, you heard correct. We will install chaincode to the peers whenever there is a completed PR aka pull request from chaincode Github repo. It looks simply but seems a bit complex when we think about how to implement this solution. We will solve this puzzle by using GitHub webhook handler and shelljs.

**High-Level Explanation**:

For the first, we will have to make a sample chaincode to be uploaded to the git repo. Next, we will have to register a webhook by giving our server URL and a secret key. Once we provide what is needed, then we will have to proceed to clone the repo in the server and then write a listener using nodejs. Here comes the magic, whenever a new closed PR is created then our nodejs listener gets that event and shelljs will pull the latest code and SDK peer install will install a new version followed by sending an upgrade proposal.

### **STEP1**: Create and configure GitHub repo

All you have to do is create a GitHub account followed by creating repo and then upload your chaincode. Once the prerequisites are done then clone the repo to the local server at the HLF nodejs SDK implementation project location. Better create a directory called chaincode and clone it there. Now Configure webhook by adding the node js server URL at the new webhook registration section. Go to the repository and click the setting tab and click on the Webhooks button. On the top right side of the page, there will be a button called “Add Webhook” click this.

|  |
| --- |
|  |

### **STEP2**:

Create a bash script file by following the below instructions and fill it with the below-mentioned snippet. This bash script will basically fetch the latest code from the git repo.

mkdir scripts CD scripts touch or nano pullcode.sh

Cat > pullcode.sh #Add below script

> #!/bin/bash  
 echo "Deploying stage your\_repo"  
 cd chaincode \  
 && git pull origin master \  
 && echo "your\_repo pulled the latest code successfully"

Give the appropriate execution permission:

chmod +x pullcode.sh

### **STEP3**: Proceed with webhook handler

Webhook basically is a callback from GitHub. In order to sync our local repo with the remote repo without manually updating the local repo then we have to subscribe to GitHub Webhook API which basically means updating some URLs to GitHub accounts. GitHub is using SHA1 HMAC signature in order to verify that it is sending the JSON object to the correct party. While we receive a notification from git we will have a signature hash in the X-Hub-Signature value of the header. There is a package called webhook handler which will be doing all the necessary configurations on our behalf so just relax by importing the package. One last piece in the cake is executing a bash file in order to pull the latest code from git. Again, shelljs will do it for us, just relax again.

const createHandler = require('github-webhook-handler');  
const shell = require('shelljs');

var handler = createHandler({ path: '/webhook', secret: MY\_SECRET })  
MY\_SECRET = "The one we just updated in the github webhook section"

handler.on('pull\_request', function (event) {  
 const repository = event.payload.repository.name;  
 const action = event.payload.action;  
  
 console.log('Received a Pull Request for %s to %s', repository, action);  
 if (repository === REPO\_NAME && action === 'closed') {  
 shell.cd('..');  
 shell.exec('scripts/pullcode');  
 }  
//Call a function to install chaincode to the peers  
});  
handler.on('error', function (err) {  
 console.error('Error has occurred:', err.message)  
})

**Note**: Once the latest code is pulled successfully. Using Node.js SDK, we can call install chaincode function and install the new version with a new version followed by an upgrade proposal.

[https://github.com/narendranathreddythota/masteringhyperledgerfabric](https://github.com/narendranathreddythota/masteringhyperledgerfabric/tree/master/chapter-7)

## **Summary:**

In this chapter, we completely focused on the chaincode development with advanced topics such as client identity, accessing user details during the endorsement phase. Encryption/Decryption at the chaincode level. We have also discussed key collisions and MVCC.

In the next chapter, we will learn End-to-end (design, development, and deployment) of fabric consortium with solo consensus using docker technology.

# Chapter8: Running fabric consortium with Solo consensus using docker

# **SubTitle**: Building​ ​Invoice Financing mechanism with solo consensus

The first-ever consensus mechanism from fabric is solo consensus, which means ordering service comes with a single order which is ideally perfect to make POC in a better & easy way in order to set up, deploy and test. Since the network will be dealing with fewer stakeholders’ minimum effort is needed. Generally, ​Invoice financing is financing unpaid invoices which is similar to short-term borrowing from one or group of lenders to business customers. We have many problems in the present way of financing invoices these are Invoices could be inflated, fake, or invoice would have been financed twice which leads to double financing to the same invoice. Blockchain single source of truth which can mitigate all fraud happening in the invoice financing sector.

In this chapter, we will build a consortium consisting of

* Two organizations,
* One channel,
* One orderer,
* One peer per organization and
* One invoice smart contract

**Use Case in Detail**:

We will not cover the complete invoice financing flows. As our scope is limited here. Let us consider two participants in a consortium 1) Opendata 2) factoring company. Open Data organization is the entry point where one can submit the details of the unpaid invoices. Factoring companies (limited to one in our case) in the consortium assesses client business and percentage of risk and then finally propose bids to the client as quotes. Clients can accept anyone’s highest bid and proceed further. For our use case client submit an invoice and only one factoring company submits a bid and the client accepts the bid. But can be expanded as a pool of borrowers from Opendata org and investor organizations can fulfill their needs without involving any finance as a central authority. The platform would open for all investors, which gives a tight competition between themselves which leads to a reasonable deal with the borrower.

**Technical Requirements**:

Technical requirements analysis always begins with a business use case and type of environment. Our main focus will be on the ubuntu 16.04 but, the selection of instances is completely on users' choice. A user can go with a local virtual machine or Aws Ec2 instance or Google Container Engine.

In a nutshell, A virtual instance from a local environment or cloud-based environment requires at least 4gb RAM and 40 GB ROM. Not very surprisingly, most of us are using virtual machines by using an Oracle virtual box on a computer or laptop. This is more than enough! I already developed this project for your reference. Feel free to have a look at any time. I would strongly recommend you to do this project by yourself and use the below-mentioned GitHub repository as a reference.

<https://github.com/narendranathreddythota/masteringhyperledgerfabric/tree/master/chapter-8>

**Beyond Financing:**

Basically, at the moment we have two different variants in the Invoice Financing sector they are invoice factoring and invoice discounting. Both are different in control and visibility but the same in concept.

Suppose, a big reputed company (CryptoBNK) has outsourced some work to a consultancy company (CryptoCS). Suppose, the allotted time period is 3 months. The CryptoCS would need to deliver the completed work to CryptoBNK, once CryptoBNK receives work from CryptoCS then, CryptoCS can raise an invoice to CryptoBNK. Not very surprisingly either profit/loss is an ultimate output In business. Sometimes even giant companies such as CryptoBNK can face huge losses, generally at that time getting invoices paid takes time but CryptoCS day to day life never changes due to CryptoBNK business. So, CryptoBNK has to pay the invoice amount else CryptoCS will impose huge interest. So, in a typical situation, Factoring is a short-term financing method where anyone can effectively ‘sell’ outstanding invoices to third-party commercial finance for quick relief. Result: CryptoCS will get money from a factoring instead of CryptoBNK.

Discounting is another concept where CryptoBNK can borrow short-term loans from finance and repay CryptoCS instead of a third party. The finance would get more profit from the interest rate it charges on the loan and monthly fee.

**Project overview:**

The chapter contains very small and easy concepts, as we go step by step further chapters, we will explore more and more concepts from fabric. This is mainly to build confidence and learning curve realization. Once the problem description is clearly articulated, the first step in the project is the design and define a business consortium. Next, we will be dealing with the deployment of our network. After deployment, our main focus will be on-chain code development. Finally, we will deploy the developed chain code and will prepare our application to start interacting with the network by proposing network transactions.

## **Getting started:**

Before we go further deep into this chapter, you may wish to cross-check whether you have cloud-based instances [three] (or) you may have your local virtual instances or docker desktop is good to go. The best part is making sure all prerequisites have been installed on the instances. As I mentioned in the previous chapter, we will be dealing with servers. So we will need many software dependencies to make this project run smoothly. Below are the mentioned compulsory dependencies.

* SSH and CURL,
* Docker and Docker-Compose,
* Nodejs with NVM,
* Python,
* Golang and GIT,
* Oracle virtual box with three virtual servers or docker desktop

Once you have the listed prerequisites have been installed, you can proceed to download Hyperledger Fabric docker images.

Clone the fabric-samples repo

git clone https://github.com/hyperledger/fabric-samples.git   
# Fetch bootstrap.sh from fabric repository using  
curl -sS https://raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh -o ./scripts/bootstrap.sh  
  
# Change file mode to executable  
chmod +x ./scripts/bootstrap.sh  
  
# Download binaries and docker images  
./scripts/bootstrap.sh [version] [ca version] [thirdparty\_version]  
./scripts/bootstrap.sh 1.4.6 1.4.6 0.4.18   
-------------OR-------------  
  
curl -sSL http://bit.ly/2ysbOFE | bash -s -- 1.4.6 1.4.6 0.4.18

## **Define Business Network**

Business Network Definition involves identifying different stakeholders for our use-case and defining their roles, for our Invoice Financing Network. We will have two different types of Organizations (Grouping different Business Entities) that form a network to carry out their business in the form of transactions. These Organizations will interact with the invoice finance platform and perform transactions to fulfill their business. By this, it ensures the traditional way of a trading invoice in a secure and trusted manner with different parties.

For our use case, we will have two organizations.

* OpenData (clients)
* FactoringCompany (TCIB)

OpenDataOrganization consists of registered borrowers that make use of the platform to fulfill their needs with the help of factoring companies. They will interact with OpenData applications typically issue their invoices or proposals to the blockchain platform.

TCIBOrganization is a factoring company that makes use of the platform in order to interact with TCIB applications typically and can propose a bid to the listed unpaid invoices. Once a bid is accepted by the client then TCIB proceeds to fund the client in two installments.

**STEP1:** Let's create a network configuration file. Create a file named congiftx.yml

|  |
| --- |
| Organizations:   - &OrdererOrg  Name: OrdererOrg  ID: OrdererMSP  MSPDir: crypto-config/ordererOrganizations/finance.com/msp   - &opendata  Name: opendata  ID: opendataMSP  MSPDir: crypto-config/peerOrganizations/opendata/msp  AnchorPeers:  - Host: opendata-peer0-opendata  Port: 7051   - &tcib  Name: tcib  ID: tcibMSP  MSPDir: crypto-config/peerOrganizations/tcib/msp  AnchorPeers:  - Host: tcib-peer0-tcib  Port: 7051  Capabilities:  Channel: &ChannelCapabilities  V1\_3: true  Orderer: &OrdererCapabilities  V1\_1: true  Application: &ApplicationCapabilities  V1\_3: true  V1\_2: false  V1\_1: false  Application: &ApplicationDefaults   # We will fill the data at the Profile section below  Organizations:  Orderer: &OrdererDefaults   # Orderer Type: The orderer implementation to start  # Available types are "solo" , "kafka" and "raft-etcd"  OrdererType: solo   Addresses:  - ordererorg-orderer-ordererorg:7050   # Batch Timeout: The amount of time to wait before creating a batch  BatchTimeout: 2s   # Batch Size: Controls the number of messages batched into a block  BatchSize:   # Max Message Count: The maximum number of messages to permit in a batch  MaxMessageCount: 40   # Absolute Max Bytes: The absolute maximum number of bytes allowed for  # the serialized messages in a batch.  AbsoluteMaxBytes: 98 MB   # Preferred Max Bytes: The preferred maximum number of bytes allowed for  # the serialized messages in a batch. A message larger than the preferred  # max bytes will result in a batch larger than preferred max bytes.  PreferredMaxBytes: 4354 KB   Kafka:  # Brokers: A list of Kafka brokers to which the orderer connects  # **NOTE:** Use IP:port notation  Brokers:  - 127.0.0.1:9092   Organizations: # We will fill this at the orderer section in the profiles  Profiles:   financeGroupOrdererGenesis:  Capabilities:  <<: \*ChannelCapabilities  Orderer:  <<: \*OrdererDefaults  Organizations:  - \*OrdererOrg  Capabilities:  <<: \*OrdererCapabilities  Consortiums:  financeConsortium:  Organizations:  - \*opendata  - \*tcib   opendataChannel:  Consortium: financeConsortium  Application:  <<: \*ApplicationDefaults  Organizations:  - \*opendata  - \*tcib  Capabilities:  <<: \*ApplicationCapabilities |

Files under this location MSPDir: crypto-config/peerOrganizations/tcib/msp

|  |
| --- |
| ├── admincerts │ └── Admin@tcib-cert.pem ├── cacerts │ └── ca.tcib-cert.pem └── tlscacerts  └── tlsca.tcib-cert.pem |

|  |
| --- |
| - &OrdererOrg Name: OrdererOrg ID: OrdererMSP MSPDir: crypto-config/ordererOrganizations/finance.com/msp |

**Note**: OpenData organization or TCIB organization can host an orderer. But, one dedicated organization can also host an orderer. We are mapping the MSP folder that will have admin certificates for channel and genesis block. To identify an organization admin at later stages. But this is no longer needed using EnableNodeOUs: true in the **crypto-config.yml** Mainly this admin identity is used to create a **channel** in the network.

|  |
| --- |
| AnchorPeers: - Host: BCY-peer0-BCY - Port: 7051 |

***Note****: Anchor peers are used for cross-organizational gossip communication.*

**Note**: Block generation in the hyperledger fabric mainly depends on BatchTimeout & Batch Size

**BatchTimeout**: Time to create a block

**Batch Size:** Transactions per block**. deals with the first come first serve.**

**Step2**: Let's create a crypto configuration file. Create a file named crypto-config.yml

Hyperledger fabric is a certificate-based permissioned network which is similar to PKI [Public-Key-Infrastructure] https. However, one can generate crypto keys and certificates by using **crytogen\_tool** or using a certificate authority. In this project, we will be using cryptogen\_tool for the sake of simplicity. However, using certificate authority to generate crypto materials, are explained in the SDK’s section.

|  |
| --- |
| **OrdererOrgs**:  - **Name**: Orderer  **Domain**: finance.com  **Specs**:  - **Hostname**: ordererorg-orderer-ordererorg  - **SANS**:  - localhost  **PeerOrgs**:  - **Name**: opendata  **Domain**: opendata  **Template**:  **Count**: 1  **SANS**:  - localhost  - opendata-peer0-opendata  **Users**:  **Count**: 1   - **Name**: tcib  **Domain**: tcib  **Template**:  **Count**: 1  **SANS**:  - localhost  - tcib-peer0-tcib  **Users**:  **Count**: 1 |

**Note:** SANS is an alias or subject alternative name when you deploy a peer as a service, other services need to identify this peer by using a name and this name will be encoded to the certificate itself and peers validate it for the security reasons. A template is related to cryptogen\_tool, how many peers you want to host under a particular organization to generate certificates.

**Step 3**: Execute the below commands in order to get the artifacts

**Cryptogen**: A tool to pre-generate the certificates for identities

**Configtxgen**: A tool to generate the initial configuration of the consortium

**FABRIC\_CFG\_PATH**: IN order to get the supplied files: path

|  |
| --- |
| > Mkdir artifacts  > export FABRIC\_CFG\_PATH=${PWD}   > cryptogen generate --config=./cryptogen.yaml  > configtxgen -profile financeGroupOrdererGenesis -outputBlock ./artifacts/genesis.block  > configtxgen -profile opendataChannel -outputCreateChannelTx ./artifacts/financechannel.tx -channelID financechannel |

By this, we have generated all artifacts related to the invoice financing project and we can proceed to write compose files instead of running each service. We can write a compose file to deploy all services such as CA, Orderer, Peers, CouchDB in one go.

**STEP4**: Write docker-compose deployment file

Fabric Provides us to use CA, Peer, Orderer, CouchDB as docker images in order to skip all dependencies installation on the current environments. All Fabric entities have all required values as defaults. However, using environment variables we can override the default variables. The fabric uses viper([spf13/viper: Go configuration with fangs](https://github.com/spf13/viper)) for managing the variables.

**CA**: We can see what variables are needed by the CA to run successfully in this project below and values to those variables are available in the github repo.

|  |
| --- |
| image: hyperledger/fabric-ca:1.4.6  Environment: #Refer GitHub Repo for the values for ENV's  - FABRIC\_CA\_HOME  - FABRIC\_CA\_SERVER\_CA\_NAME  - FABRIC\_CA\_SERVER\_CA\_CERTFILE   - FABRIC\_CA\_SERVER\_CA\_KEYFILE  - FABRIC\_CA\_SERVER\_TLS\_ENABLED  - FABRIC\_CA\_SERVER\_TLS\_CERTFILE   - FABRIC\_CA\_SERVER\_TLS\_KEYFILE   ports:  - "7054:7054"  command: sh -c 'fabric-ca-server start -b tcibadmin:tcibpw -d' |

In order to not supply the root certificate and private key of the CA just replace start with init. On the fly, identities will be created by the CA itself. In order to run the CA server with custom port then supply **FABRIC\_CA\_SERVER\_PORT** and to use Postgres or MySQL DB instead of the in-memory database then supply **FABRIC\_CA\_SERVER\_DB\_TYPE, FABRIC\_CA\_SERVER\_DB\_DATASOURCE**

**Orderer**: We can see what variables are needed by the Orderer to run successfully in this project below and values to those variables are available in the github repo.

|  |
| --- |
| image: hyperledger/fabric-orderer:1.4.6  environment:  - ORDERER\_GENERAL\_LOGLEVEL  - ORDERER\_GENERAL\_LISTENADDRESS = 0.0.0.0 #listen on any IP available   - ORDERER\_GENERAL\_GENESISMETHOD  - ORDERER\_GENERAL\_GENESISFILE  - ORDERER\_GENERAL\_LOCALMSPID = OrdererMSP #Each entity has its own MSP  - ORDERER\_GENERAL\_LOCALMSPDIR  - ORDERER\_GENERAL\_TLS\_ENABLED  - ORDERER\_GENERAL\_TLS\_PRIVATEKEY   - ORDERER\_GENERAL\_TLS\_CERTIFICATE   - ORDERER\_GENERAL\_TLS\_ROOTCAS # List of Trusted RootCA certs for auth  working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderers  command: orderer  ports:  - 7050:7050 |

ORDERER\_GENERAL\_GENESISMETHOD The method by which the genesis block for the orderer system channel is specified. Available options are:

* Provisional
* file

Provisional: Utilizes a genesis profile, specified by GenesisProfile, to dynamically generate a new genesis block.

File: Uses the file provided by GenesisFile as the genesis block.

ORDERER\_GENERAL\_GENESISPROFILE The profile to use to dynamically generate the genesis block to use when initializing the orderer system channel.

ORDERER\_GENERAL\_GENESISFILE The file containing the genesis block to use when initializing the orderer system channel.

CouchDB:

|  |
| --- |
| image: hyperledger/fabric-couchdb  environment:  - COUCHDB\_USER=dbuser  - COUCHDB\_PASSWORD=dbpass@17788  ports:  - "5984:5984" |

Peer: We can see what variables are needed by the Peer to run successfully in this project below.

|  |
| --- |
| image: hyperledger/fabric-peer  environment:  - PEER\_HOST=tcib-peer0-tcib  - PEER\_NAME=tcib-peer0-tcib  - CORE\_PEER\_ID=tcib-peer0-tcib  - CORE\_PEER\_LOCALMSPID=tcibMSP  - CORE\_PEER\_ADDRESS=tcib-peer0-tcib:7051  - CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock  - CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=tcib\_finance  - CORE\_LOGGING\_LEVEL=DEBUG  - CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=tcib-peer0-tcib:7051  - CORE\_PEER\_GOSSIP\_USELEADERELECTION=true  - CORE\_PEER\_GOSSIP\_ORGLEADER=false  - CORE\_PEER\_MSPCONFIGPATH=/etc/hyperledger/crypto/peer/msp  - CORE\_PEER\_TLS\_ENABLED=true  - CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/crypto/peer/tls/server.key  - CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/crypto/peer/tls/server.crt  - CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/crypto/peer/tls/ca.crt  - CORE\_LEDGER\_STATE\_STATEDATABASE=CouchDB  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS=couchdb1:5984  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_USERNAME=dbuser  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_PASSWORD=dbpass@17788  working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer  command: peer node start  ports:  - 7051:7051  - 7053:7053 |

CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE is needed in order to start chaincode containers on the same bridge network as the peers.

CORE\_VM\_ENDPOINT please refer to the docker sockets chapter.

CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT This is an endpoint that is published to peers outside of the organization. If this isn't set, the peer will not be known to other organizations. Must be set if we enable private data concepts.

CORE\_PEER\_GOSSIP\_USELEADERELECTION Whenever peers initialize a dynamic algorithm for leader selection, where the leader in the peer to establish a connection with the ordering service and use the delivery protocol to pull ledger blocks from the ordering service. Leader Peer is responsible for delivering the blocks from orderer to other peers in the leader peer organization.

CORE\_PEER\_GOSSIP\_ORGLEADER if (true) Statically defines peer to be an organization "leader", where this means that the current peer will maintain a connection with ordering service and disseminate block across peers in its own organization.

**Note**: orgLeader and useLeaderElection parameters are mutually exclusive. Setting both to true would result in the termination of the peer since at a time both cannot be true and this leads to an undefined state.

## **Design Network Topology**

Basically, Network Topology is a high-level design or a blueprint with different deployment entities that communicate over the network with each other to complete a proposed transaction.

|  |
| --- |
|  |

## **Deploy Consortium**:

Once we are ready with configuration files such as channel.tx, genesis.block , certificates for all entities, and docker-compose file then, we are ready to deploy the Fabric consortium.

**Clone the GitHub repo**:

|  |
| --- |
| > cd chapter-8 > hyperledger-fabric > opendata > docker-compose up -d > cd chapter-8 > hyperledger-fabric > tcib > docker-compose up -d |

|  |
| --- |
| ➜ tcib git:(master) ✗ docker-compose up -d WARNING: The Docker Engine you're using is running in swarm mode.  Compose does not use swarm mode to deploy services to multiple nodes in a swarm. All containers will be scheduled on the current node.  To deploy your application across the swarm, use `docker stack deploy`.  Creating network "tcib\_finance" with the default driver Creating couchdb1 ... done Creating tcib\_ordererorg-orderer-ordererorg\_1 ... done Creating tcib\_ca\_tcib\_1 ... done Creating tcib\_tcib-peer0-tcib\_1 ... done ➜ opendata git:(master) ✗ docker-compose up -d Creating network "opendata\_finance" with the default driver Creating couchdb2 ... done Creating opendata\_ca\_opendata\_1 ... done Creating opendata\_opendata-peer0-opendata\_1 ... done |

**Note:** We can ignore the warnings as we will deploy our next project in the swarm network.

**Monitor docker logs:** Once We deploy the docker-compose, let us see the logs using a utility tool called logspout. We can pass arguments while running the below-mentioned script. 1) Network Name 2) log server to be exposed on the port. Forgot to mention then fallback to default values.

**Note**: Monitor the logs based on a particular network. So, watch out for the network that is created by the docker-compose. BY default, docker creates a network and all containers are mapped to particular networks. We need to update this at the peer service using CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE. Below mentioned script runs only once docker services are up and running.

if [ -z "$1" ]; then DOCKER\_NETWORK=tcib\_finance else DOCKER\_NETWORK="$1" fi  
  
if [ -z "$2" ]; then PORT=8000 else PORT="$2" fi  
  
echo Starting monitoring on all containers on the network ${DOCKER\_NETWORK}  
  
docker kill logspout 2> /dev/null 1>&2 || true docker rm logspout 2> /dev/null 1>&2 || true  
  
docker run -d --name="logspout" \  
 --volume=/var/run/docker.sock:/var/run/docker.sock \  
 --publish=127.0.0.1:${PORT}:80 \  
 --network ${DOCKER\_NETWORK} \  
 gliderlabs/logspout sleep 3  
  
curl http://127.0.0.1:${PORT}/logs

**Deploy** logspout**:**

|  |
| --- |
| > cd chapter-8 > hyperledger-fabric > chmod +x monitor.sh  > c./monitor.sh tcib\_finance > c./monitor.sh opendata\_finance |

Expected output below pasted for your reference

➜ tcib git:(master) ✗ ./monitor.sh

Starting monitoring on all containers on the network tcib\_finance efc319ce4a073a1d75525e63bde88945810fd913f85afdb7ee520484f2f98b09 opendata\_ca\_opendata\_1|2020/02/15 11:36:48 [DEBUG] Cleaning up expired nonces for CA 'ca-opendata' tcib\_tcib-peer0-tcib\_1|2020-02-15 11:38:53.074 UTC [endorser] callChaincode -> INFO 0a0 [financechannel][c2078b34] Entry chaincode: name:"invoice1"

## **Building an application:**

We have defined our consortium network and deployed it successfully. Now it's time to start developing applications to interact with our network. We can develop an application [To interact with fabric] from scratch or we can just integrate with the existing running business application. It's completely loose coupling and totally user-friendly because of SDK.

**Step1:** Let's create a service connection profile file and configuration file. Create a file named network-config.yml and config file named config.json. Mentioned only the high level of the file content. Please check GitHub repo for full implementation.

{  
 "name": "finance-tcib",  
 "x-type": "hlfv1",  
 "description": "finance-tcib-admin-portal",  
 "version": "1.0",  
 "channels": {},  
 "organizations": {},  
 "orderers": {},  
 "peers": {},  
 "certificateAuthorities": {}  
}

{  
 "host":"localhost",  
 "port":"6000",  
 "channelName":"financechannel",  
 "CC\_SRC\_PATH":"../chaincode",  
 "chaincodeName":"policy",  
 "admins":[  
 {  
 "username":"tcibadmin",  
 "secret":"tcibpw" }  
 ],  
 "orgName":"tcib",  
 "clientIndy":"tcibClient",  
 "peerName":"tcib-peer0-tcib"  
}

**STEP2**: Application, further steps. Let's go further in depth and finish all the preliminary tasks. We have to get a client identity before proceeding with further operations with fabric. Getting the client object is very easy. The network-config.yml file plays a major role in the creation of a client object. Below code is sufficient to generate a client object. We will be using this object at every stage, whenever we interact with fabric.

const hfc = require('fabric-client');  
const file = network-config.yaml;

hfc.setConfigSetting('network-connection-profile-path',path.join(\_\_dirname, file));  
let client = hfc.loadFromConfig(hfc.getConfigSetting('network-connection- profile-path'));  
await client.initCredentialStores();  
return client;

const envelope = fs.readFileSync(path.join(\_\_dirname, channelConfigPath));  
const channelConfig = client.extractChannelConfig(envelope);

let signature = client.signChannelConfig(channelConfig);

channelConfigPath which is nothing but a local path where our channel file[channel.tx] exists. We created this file in the previous steps. Once we successfully mapped the respective file, we can proceed to sign the payload using the organization's admin private key. Luckily, we have our admin credentials loaded with client objects. So, just proceed for signing payload. Below mentioned detailed objects which we will send to orderer as a transaction.

let request = {  
config: channelConfig,  
signatures: [signature],  
name: channelName,  
txId: client.newTransactionID(true)  
   
};  
const result = await client.createChannel(request);

We can confirm whether the channel has been created successfully or not using the result object. The result object is the output from orderer. We will have a status object in the result object. If status equals SUCCESS, then the channel has been created successfully.

**STEP3**: Once the channel has been created successfully, we are allowed to onboard our peers to the channel and then, we will proceed to install chaincode operation and finally instantiate chaincode. Operation.

const channel = client.getChannel(channel\_name);  
let genesis\_block = await channel.getGenesisBlock(request);

let join\_request = {   
targets: peers,  
txId: client.newTransactionID(true),

block: genesis\_block

};

let result = channel.joinChannel(join\_request);

**Peers’** variable is an array of peer names. At runtime, the client object will inject peer details into the ongoing request. If we miss the peer names then the client object will inject all peers that are associated with a particular channel from the connection profile. Finally, the client sends the final payload to the channel. We would need genesis block to make sure we are joining our peers in the right channel. We can confirm whether peers have been joined successfully or not using a result object. The result object is the output from the channel. We will have a status object in the result object. If status equals SUCCESS, then peers have been joined successfully.

const request = {   
targets : peers,   
chaincodePath: chaincodePath,  
chaincodeId: chaincodeName,  
chaincodeType: chaincodeType,  
chaincodeVersion: chaincodeVersion

};

let results = await client.installChaincode(request);

const request = {

targets : peers,

chaincodeId: chaincodeName,  
 chaincodeType: chaincodeType,  
 chaincodeVersion: chaincodeVersion,  
 args: args,

txId: tx\_id

};

let results = await channel.sendInstantiateProposal(request, 60000);

Once peers have been joined successfully, we are allowed to go for install chaincode onto the peers and onboard chaincode into the channel. This process is also called a **chaincode instantiation**. By now, you must be confident enough to make a transaction with fabric.

**User** We need a user identity in order to invoke the chaincode which means to push the data fabric checks for user identity. So let us write some snippets in order to register the user. First let us get the CA admin as an object. Admins variable is defined in the config.json file.

var admins = hfc.getConfigSetting('admins');  
 let adminUserObj = await client.setUserContext({  
 username: admins[0].username,  
 password: admins[0].secret  
 });

Once we get the CA admin, we are ready to register a new user. IN the fabric-samples fab car this code snippet is present in registerUser.js

let affiliationService = caClient.newAffiliationService();  
 let secret = await caClient.register({  
 enrollmentID: username,  
 affiliation: userOrg.toLowerCase() + '.department1'  
}, adminUserObj);

Once we register a particular user we should proceed for enrollment. Here as every operation proceeding for enrollment in order to get the user credentials is a bit heck. Let us use some advantages provided by fabric-client.

user = await client.setUserContext({  
 username: username,  
 password: secret  
});

Now we will have a question. We will create a set of users and how can we tell the client to use a particular user identity while invoking the chaincode ?

All we have to do is just pass the username to the getUserContext getUserContext

let user = await client.getUserContext(username, true);  
 if (!user) {  
 throw new Error(util.format('Alert! User was not found :', username));  
 } else {  
 logger.debug('User %s was found to be registered and enrolled', username);  
 }

Client Interaction with fabric involves mainly two operations.

1. Invoke transaction 2. Query transaction

Proposing a transaction to fabric is not straightforward like installing chaincode or joining peers. There are several steps involved in the fabric ecosystem, right from the client proposing a transaction to peer validating this transaction in the block and committing into peer ledger. This whole process is called invoking and we have another operation called query transaction. As the name suggests it fetch queried data as a transaction.

const request = {  
 chaincodeId: chaincodeName,  
 fcn: fcn,  
 args: args,  
 chainId: channelName,  
 txId: tx\_id  
 };  
let results = await channel.sendTransactionProposal(request);

const proposalResponses = results[0];

const proposal = results[1];

const orderer\_request = {

txId: tx\_id,

proposalResponses:proposalResponses,  
 proposal: proposal

};  
const result = channel.sendTransaction(orderer\_request);  
var request = {

chaincodeId: chaincodeName,  
 fcn: fcn,  
 args: args

};  
let response\_payloads = await channel.queryByChaincode(request);

When the client proposes a transaction, first it reaches the endorsing peers. Endorsing peers simulate proposed transactions without disturbing the ledger and signs the simulated result with the private key, then it is sent back to the client itself. The **results** object has both the endorsement results and the actual proposal. Proposal responses contain endorsement results. If proposal responses are good then the client submits data [proposal, proposal responses] as a payload to the orderer. We can confirm whether the proposed transaction has been committed successfully to the block or not by using the result object. The result object is the output from the orderer. We will have a status object in the result object. If status equals SUCCESS, then the proposed transaction has been committed successfully. Query data always fetch the latest data from the block.

We are ready with the functions in Application and now let us proceed to write chaincode.

## **Chain Code:**

Chaincode development starts with a transaction definition, transaction definition is a high- level blueprint. One can easily understand the whole process by looking at the chaincode logic because chaincode logic is always directly proportional to business logic. So, we will start developing chaincode with transaction definitions.

|  |
| --- |
| // InvoiceBid struct **type** InvoiceBid **struct** {  BidID **string** `json:"bidID"`  InvoiceID **string** `json:"invoiceID"`  Fee **string** `json:"fee"`  Financier **string** `json:"financier"`  AdvanceAmount **string** `json:"advanceAmount"` }  //Invoice struct **type** Invoice **struct** {  InvoiceID **string** `json:"invoiceID"`  CustomersDetails **string** `json:"customersDetails"`  Issuer **string** `json:"issuer"`  IssueDate **string** `json:"issueDate"`  InvoiceValue **float32** `json:"invoiceValue"`  CurrentState **string** `json:"currentState"`  InvoiceBid InvoiceBid `json:"invoiceBid"` } |

According to the use case, there are 7 identified fields. These fields are the keys in the key-value pair of a blockchain ledger. When OpenData clients have an invoice, which will be financed later in the near future. Any OpenData authorized client can post the latest invoice into our financing platform. Any factoring company can make a bid to the invoice and the client can select a bid that is more favorable, which results in the client can be paid immediately some value. Once customers of the client pay the full amount to the factoring company then it will close the deal with the client. Hyperledger fabric supports various languages to write chain-codes in, but Golang is the most popular one. For the sake of simplicity, we create one invoice and one bid to complete the flow. However, this chaincode logic can be expanded further.

Let’s, deep dive into chaincode development. Before going further, an important point needs to be highlighted here. There is a global object present in the hyperledger fabric chaincode ecosystem which is called shim object. Package shim provides rich APIs for the chaincode to access chaincode state variables, transaction context and chaincode could be able to call other chaincodes. Below repo mentioned a detailed shim object.

|  |
| --- |
| *https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim* |

We can import shim global objects by using go imports.

|  |
| --- |
| import ( "bytes" "encoding/json" "fmt" "github.com/hyperledger/fabric/core/chaincode/shim" sc "github.com/hyperledger/fabric/protos/peer"  ) |

Golang is a statically typed language and it is a compiled language. Like java, Golang also needs a main function. As the execution starts from here. While we create a new smart contract struct, we need to make sure that all methods from the interface such as Invoke and Init are present. Since the start function in the shim package accepts the Chaincode interface. Let us see the chaincode interface

|  |
| --- |
| type Chaincode interface {  // Init is called during Instantiate transaction after the chaincode container  // has been established for the first time, allowing the chaincode to  // initialize its internal data  Init(stub ChaincodeStubInterface) pb.Response   // Invoke is called to update or query the ledger in a proposal transaction.  // Updated state variables are not committed to the ledger until the  // transaction is committed.  Invoke(stub ChaincodeStubInterface) pb.Response } |

type SmartContract struct {}  
  
func main() {  
  
 // Create a new Smart Contract  
 err := shim.Start(new(SmartContract))  
 if err != nil {  
 fmt.Printf("Error creating new Smart Contract: %s", err)  
 }  
}

Let us first write the Init and functions:

func (s \*SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response {  
 return shim.Success(nil)  
} // we are not auto populating any data through init func   
  
func (s \*SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {  
 return s.Controller(APIstub)  
}

func (s \*SmartContract) Controller(APIstub shim.ChaincodeStubInterface) sc.Response {  
  
 function, args := APIstub.GetFunctionAndParameters()  
 if len(args) < 1 {  
 str := fmt.Sprintf("Invalid request")  
 return shim.Error(str)  
 }  
  
 fmt.Println(function)  
 fmt.Println(args)  
  
 //guard  
 authorized := s.RequestAuth(APIstub, function, args)  
  
 if !authorized {  
 str := fmt.Sprintf("Unauthorized operation in request")  
 return shim.Error(str)  
 }  
  
 return s.InvokeController(APIstub, function, args)  
}

//RequestAuth to check if the request is valid or not  
func (s \*SmartContract) RequestAuth(APIstub shim.ChaincodeStubInterface, function string, args []string) bool {  
 // One can have any type of check here..  
 return true  
  
}

// InvokeController request controller  
func (s \*SmartContract) InvokeController(APIstub shim.ChaincodeStubInterface, function string, args []string) sc.Response {  
  
 // Route to the appropriate handler function to interact with the ledger appropriately  
 if function == "query" {  
 return s.query(APIstub, args)  
 } else if function == "createInvoice" {  
 return s.createInvoice(APIstub, args)  
 } else if function == "createInvoiceBid" {  
 return s.createInvoiceBid(APIstub, args)  
 } else if function == "delete" {  
 return s.delete(APIstub, args)  
 } else if function == "update" {  
 return s.update(APIstub, args)  
 }  
 return shim.Error("Invalid Smart Contract function name.")  
}

Let us proceed with the business methods such as **createInvoice** **createInvoiceBid selectBid**

// createInvoice record for the request  
func (s \*SmartContract) createInvoice(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  
 fmt.Println(args)  
 if len(args) != 1 {  
 str := fmt.Sprintf("Invalid request : invalid number of arguments!")  
 return shim.Error(str)  
 }  
 data := Invoice{}  
 if err := json.Unmarshal([]byte(args[0]), &data); err != nil {  
 str := fmt.Sprintf("JSON Parsing exception: %+v", err)  
 return shim.Error(str)  
 }  
  
 fmt.Printf("%v", data)  
 UniqueID := data.InvoiceID  
  
 dataAsBytes, err := APIstub.GetState(UniqueID)  
 if err != nil {  
 return shim.Error("Failed to get invoice: " + err.Error())  
 } else if dataAsBytes != nil {  
 fmt.Println("This invoice already exists")  
 return shim.Error("This invoice already exists")  
 }  
 dataAsBytes, err = json.Marshal(data)  
 if err != nil {  
 str := fmt.Sprintf("Can not marshal %+v", err.Error())  
 return shim.Error(str)  
 }  
 if err := APIstub.PutState(UniqueID, dataAsBytes); err != nil {  
 str := fmt.Sprintf("Problem has been occurred while save")  
 return shim.Error(str)  
 }  
 fmt.Println(fmt.Sprintf("Successfully tested %s", dataAsBytes))  
 return shim.Success(dataAsBytes)  
}

// createInvoiceBid record for the request  
func (s \*SmartContract) createInvoiceBid(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  
 fmt.Println(args)  
 if len(args) != 1 {  
 str := fmt.Sprintf("Invalid request : invalid number of arguments!")  
 return shim.Error(str)  
 }  
 data := InvoiceBid{}  
 if err := json.Unmarshal([]byte(args[0]), &data); err != nil {  
 str := fmt.Sprintf("JSON Parsing exception: %+v", err)  
 return shim.Error(str)  
 }  
 fmt.Printf("%v", data)  
 UniqueID := data.BidID  
 dataAsBytes, err := APIstub.GetState(UniqueID)  
 if err != nil {  
 return shim.Error("Failed to get invoice: " + err.Error())  
 } else if dataAsBytes != nil {  
 fmt.Println("This invoice already exists")  
 return shim.Error("This invoice already exists")  
 }  
 dataAsBytes, err = json.Marshal(data)  
 if err != nil {  
 str := fmt.Sprintf("Can not marshal %+v", err.Error())  
 return shim.Error(str)  
 }  
 if err := APIstub.PutState(UniqueID, dataAsBytes) ; err != nil {  
 str := fmt.Sprintf("Problem has been occurred while saving")  
 return shim.Error(str)  
 }  
 fmt.Println(fmt.Sprintf("Successfully tested %s", dataAsBytes))  
 return shim.Success(dataAsBytes)  
}

|  |
| --- |
| // selectBid record as per the request func (s \*SmartContract) selectBid(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  fmt.Println(args)  if len(args) < 2 {  str := fmt.Sprintf("Invalid request : invalid number of arguments!")  return shim.Error(str)  }  invoiceBytes, err := APIstub.GetState(args[0])  if err != nil {  str := fmt.Sprintf("Problem while checking the information")  return shim.Error(str)  } else if invoiceBytes == nil {  str := fmt.Sprintf("Information does not exists for Invoice")  return shim.Error(str)  }  invoiceData := &Invoice{}  if err := json.Unmarshal(invoiceBytes, &invoiceData); err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  bidBytes, err := APIstub.GetState(args[1])  if err != nil {  str := fmt.Sprintf("Problem has been occured while get")  return shim.Error(str)  } else if bidBytes == nil {  str := fmt.Sprintf("Info does not exists for Invoice")  return shim.Error(str)  }  bidData := InvoiceBid{}  if err := json.Unmarshal(bidBytes, &bidData); err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  invoiceData.InvoiceBid = bidData  finalInvoiceAsBytes, err := json.Marshal(invoiceData)  if err != nil {  str := fmt.Sprintf("Can not marshal %+v", err.Error())  return shim.Error(str)  }  err = APIstub.PutState(args[0], finalInvoiceAsBytes)  if err != nil {  str := fmt.Sprintf("Can not put state %+v", err.Error())  return shim.Error(str)  }  fmt.Println(fmt.Sprintf("Successfully tested %s", []byte(args[0])))   return shim.Success([]byte("Success")) } |

Query an invoice at any time during the lifecycle of an invoice. By using the shim object

|  |
| --- |
| var APIstub = shim.ChaincodeStubInterface; data, \_ := APIstub.GetState(args[0])  func (s \*SmartContract) query(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {   if len(args) != 1 {  return shim.Error("Incorrect number of arguments. Expecting 1")  }   objAsBytes, \_ := APIstub.GetState(args[0])  return shim.Success(objAsBytes) } |

Delete an invoice at any time during the lifecycle of an invoice. By using the shim object

|  |
| --- |
| var APIstub = shim.ChaincodeStubInterface; if err := APIstub.DelState(args[0]); err != nil {  return shim.Error("Failed to delete state")  } |

## **Interact with fabric:**

Let us run the OpenData client applications which are built with nodejs-express

|  |
| --- |
| *> CD portals > opendata  ➜ opendata git:(master) ✗ npm start [2020-02-15 15:14:47.096] [INFO] opendata-admin - ------------------- SERVER STARTED ----------------------- [2020-02-15 15:14:47.099] [INFO] opendata-admin - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* http://localhost:5000 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

Create User:

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/createUser \  -H "content-type: application/json" \  -d '{  "username":"bingo",  "orgName":"opendata" }' |

Create Channel:

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/createChannel \  -H "content-type: application/json" \  -d '{  "channelName":"financechannel",  "userName": "bingo",  "orgName": "opendata",  "channelConfigPath":"../crypto/financechannel.tx" }' |

Join Peers:

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/joinPeers \  -H "content-type: application/json" \  -d '{  "channelName":"financechannel",  "userName": "bingo",  "orgName": "opendata",  "peers": ["opendata-peer0-opendata"] }' |

Install Chaincode:

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/installChaincode \  -H "content-type: application/json" \  -d '{  "chaincodeName": "invoice",  "chaincodePath": "github.com/invoice/go",  "chaincodeType": "Golang ",  "chaincodeVersion": "v0",  "userName": "bingo",  "orgName": "opendata",  "peers": ["opendata-peer0-opendata"] }' |

InstantiateChaincode:

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/instantiateChaincode \  -H "content-type: application/json" \  -d '{  "chaincodeName": "invoice",  "chaincodeVersion": "v0",  "userName": "bingo",  "orgName": "opendata",  "channelName": "financechannel",  "chaincodeType": "Golang ",  "args": [""],  "peers": ["opendata-peer0-opendata"] }' |

Create Invoice:

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"financechannel",  "chaincodeName":"invoice",  "fcn": "createInvoice",  "userName": "bingo",  "orgName": "opendata",  "args": ["{\r\n \"invoiceID\": \"12345\",\r\n \"customersDetails\": \"khanz suppliers, marhaba suppliers\",\r\n \"issuer\":\"mint jewlers\",\r\n \"issueDate\":\"14-04-2020\",\r\n \"invoiceValue\": \"345655757\",\r\n \"currentState\":\"Invoice Submitted\"\r\n}"],  "peers": ["opendata-peer0-opendata"] }' |

Let us run the TCIB client applications which are built with nodejs-express

|  |
| --- |
| *> CD portals > tcib  ➜ tcib git:(master) ✗ npm start [2020-02-15 15:14:47.096] [INFO] tcib-admin - ------------------- SERVER STARTED ----------------------- [2020-02-15 15:14:47.099] [INFO] tcib-admin - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* http://localhost:6000 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

CreateUser:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/createUser \  -H "content-type: application/json" \  -d '{  "username":"tcibClient",  "orgName":"tcib" }' |

Join Peers:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/joinPeers \  -H "content-type: application/json" \  -d '{  "channelName":"financechannel",  "userName": "tcibClient",  "orgName": "tcib",  "peers": ["tcib-peer0-tcib"] }' |

Submit Bid:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"financechannel",  "chaincodeName":"invoice",  "fcn": "createInvoiceBid",  "userName": "tcibClient",  "orgName": "tcib",  "args": ["{\r\n \"bidID\": \"abcde\",\r\n \"invoiceID\": \"12345\",\r\n \"fee\": \"5000\", \r\n \"financier\":\"TCIB\",\r\n \"advanceAmount\":\"305566666\"\r\n}"],  "peers": ["tcib-peer0-tcib"] }' |

Map Bid With Invoice: By the client from OpenData Organization

|  |
| --- |
| curl -s -X POST \  http://localhost:5000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"financechannel",  "chaincodeName":"invoice1",  "fcn": "selectBid",  "userName": "opendataClient",  "orgName": "opendata",  "args": ["12345", "abcde"],  "peers": ["tcib-peer0-tcib"] }' |

**Invoice lifecycle**

Assume that channel is created, peers joined and chaincode is installed and the platform is ready to serve the requests. Client Submit Invoice to the platform through the OpenData organization. InvoiceID is “12345”, customer details are khanz suppliers and marhaba suppliers. Customers are important to the invoice if ignored then no factoring company would interest to finance the invoice as clear potential risk is involved. Below payload mentioned. Once the invoice is submitted. Factoring companies can see the invoices and submit a bid. If the client is ok with the bid, then he can accept the bid which is called map bid with the invoice.

|  |
| --- |
| //Invoice {  "invoiceID": "12345",  "customersDetails": "khanz suppliers, marhaba suppliers",  "issuer":"mint jewlers",  "issueDate":"14-04-2020",  "invoiceValue": "345655757",  "currentState":"Invoice Submitted" } |

|  |
| --- |
| {  "bidID": "abcde",  "invoiceID": "12345",  "fee": "5000",  "financier":"TCIB",  "advanceAmount":"305566666" } |

## **Tips on Errors:**

**Failed to create the channel** ‘**public**’

NodeJS Logs: [2018–10–04 12:19:28.548] [DEBUG] Create-Channel — response ::{“status”:”BAD\_REQUEST”,”info”:”error authorizing update: error validating DeltaSet: policy for [Group] /Channel/Application not satisfied: Failed to reach implicit threshold of 1 sub-policies, required 1 remaining”} [2018–10–04 12:19:28.548] [ERROR] Create-Channel —

!!!!!!!!! Failed to create the channel ‘public’ !!!!!!!!!

Orderer Logs: reg channel creation error

2018–10–04 08:19:28.617 UTC [orderer/common/broadcast] Handle -> WARN 132 [channel: public] Rejecting broadcast of config message from 10.255.0.2:57714 because of error: error authorizing update: error validating DeltaSet: policy for [Group] /Channel/Application not satisfied: Failed to reach implicit threshold of 1 sub-policies, required 1 remaining.

**Reason**: The transaction is not signed with the trusted admin key.

Solution and Tip: To create a channel, we must need admin credentials, such as confidential private key & public certificate (x.509), a channel has an inbuilt policy, only organization admin can create a channel. However, we can add more functionalities to this policy. If a network with three organizations, then we can add channel policy at the time of defining the network, in such a way that the majority of admin signatures needed to create a channel.

## **Troubleshooting:**

**Problem1:** If you are not able to execute the **cryptogen** command, then probably your instance would have missed the bin path.

**Solution**: Please follow the below steps to set the bin path.

|  |
| --- |
| sudo nano ~/.bashrc export PATH=/ubuntu/home/fabric-samples/bin Source ~/.bashrc |

**Problem2**: While doing the project, you might end with creating the same instances many times, which leads to multiple certificates of the same certificate authority. So, removing previous certificates from local KVS is advisable. Search for the folder name "fabric-client-kv-opendata" and delete it.

|  |
| --- |
| sudo rm -rf fabric-client-kv-opendata sudo rm -rf fabric-client-kv-tcib |

Stop running container: docker stop -f $(docker ps -aq)

Remove dead or stopped containers: docker rm -f $(docker ps -aq)

Remove chaincode images:

|  |
| --- |
| docker rmi -f $(docker images | grep dev | awk '{print $3}') |

Good to remove previous dead containers or running containers, before proceeding with fresh containers.

## **Summary**

Overall, in this chapter, we have learned about trading an invoice using blockchain technology and we analyzed problems in the present invoice financing sector and finally, we proposed our solution with Hyperledger fabric which will mitigate all fraud happening in this sector. We learned all the way from defining a business network to interacting with the network. Now you must be confident enough to at least build a proof-of-concept for your own use case.

In the upcoming chapters, we will explore in-depth concepts of fabric and will gradually increase our learning curve step by step. We will cover all concepts such as Kafka consensus with multi-order setup, raft consensus with multi-order setup, docker swarm as a container orchestrator, Kubernetes as a container orchestrator, etc.

In the next chapter, we will get to know about an innovative use case Prevent Subscriber Fraud in the Telcom with Kafka consensus.

# Chapter9: Running fabric Consortium with Kafka consensus using docker swarm

**SubTitle**: Prevent Subscriber Fraud in the Telecom with Kafka consensus

(CSPs) Communications service providers often encounter a lot of issues related to roaming subscribers on CSP networks because they do not have clear visibility into their roaming subscribers’ activity on partnered networks. Nowadays, the majority of CSPs include third-party clearinghouses’ involvement in order to reconcile the payments of roaming customers. Money and time are the major factors in this area. Cramming and slamming are two major issues for most CSPs, and records show that costs more than $40 billion USD annually which is more than 5% of the total revenue. This is basically referred to as a subscriber fraud. Fraudulent subscribers can access a victim's home CSP network while cloning the roaming subscriber’s identity. Once Fraudulent gains access, he will raise charges made on the victim’s behalf by the Fraudulent.

Blockchain brings these CSPs onto a single blockchain Hyperledger network, which enables direct exchange of information with transactions that are immutable and executed based on a consensus model that uses smart contract rules. This improves the CSP’s visibility into the subscriber activity. This enables a quick payment reconciliation, and reduces fraudulent activities.

In this chapter we will build a consortium that consists of Two organizations,

* One channel,
* Multiple orderers,
* Kafka consensus mechanism,
* One peer per organization and
* One smart contract.

## **Use Case in Detail**:

When a subscriber from India travels to Dubai. Indian region is the home network and the Dubai region is the roaming network(international). In India, suppose airtel is the CSP and Dubai du is the CSP. Once a subscriber moves out of the airtel network and enters into du’s network then the subscriber gets full services from du’s network until the subscriber goes back to the home country. In this situation criminals from Dubai clone the subscriber identity while he is in Dubai and access the subscriber’s Indian Airtel network and make a huge bill. Airtel network assumes that the Indian subscriber comes back home and provides all services in the home country (+) the original subscriber is still in Dubai and still, Airtel has to pay for the du services. Because of this criminal, both networks need to approach the clearinghouses in order to reconcile the payments for the services.

Hyperledger Fabric consortium where airtel and du are two organizations and share the complete data of the subscribers in a real-time manner then we can mitigate the subscriber fraud and readers furthermore can expand the use case to eliminate third parties for reconciling services and much more.

## **Technical Requirements:**

Technical requirements analysis always begins with a business use case and type of environment. We need a minimum of two VM’s in order to set up a swarm cluster. Our main focus will be on the ubuntu 16.04 but, the selection of instances is completely on users' choice. A user can go with a local virtual machine or Aws Ec2 instance or Google Container Engine. In a nutshell, A virtual instance from a local environment or cloud-based environment requires at least 4gb RAM and 40 GB ROM. Not very surprisingly, most of us are using virtual machines by using an Oracle virtual box on a computer or laptop. This is more than enough! I already developed this project for your reference. Feel free to have a look at any time. I would strongly recommend you to do this project by yourself and use the below-mentioned GitHub repository as a reference.

<https://github.com/narendranathreddythota/masteringhyperledgerfabric/tree/master/chapter-9>

## **Project overview:**

We witnessed a very easy setup with docker-compose and solo consensus. Present chapter contains medium level concepts, as we go step by step further chapters, we will explore more and more concepts from fabric. This is mainly to build confidence and learning curve realization. Once the problem description is clearly articulated, the first step in the project is the design and define a business consortium. Next, we will be dealing with the deployment of our network. After deployment, our main focus will be on-chain code development. Finally, we will deploy the developed chain code and will prepare our application to start interacting with the network by proposing network transactions.

## **Getting started:**

Before we go further deep into this chapter, you may wish to cross-check whether you have cloud-based instances [three] (or) you may have your local virtual instances[three] is a must to proceed further. The best part is making sure all prerequisites have been installed on the instances. As I mentioned in the previous chapter, we will be dealing with servers. So, we will need many software dependencies to make this project run smoothly. Below are the mentioned compulsory dependencies.

* SSH and CURL,
* Docker and Docker-Compose,
* Nodejs with NVM,
* Python,
* Golang and GIT,
* Oracle virtual box with three virtual servers or docker desktop

Once you have the listed prerequisites have been installed, you can proceed to download Hyperledger Fabric docker images.

Clone the fabric-samples repo.

|  |
| --- |
| git clone https://github.com/hyperledger/fabric-samples.git  # Fetch bootstrap.sh from fabric repository using curl -sS https://raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh -o ./scripts/bootstrap.sh  # Change file mode to executable chmod +x ./scripts/bootstrap.sh  # Download binaries and docker images ./scripts/bootstrap.sh [version] [ca version] [thirdparty\_version] ./scripts/bootstrap.sh 1.4.6 1.4.6 0.4.18  -------------OR-------------  curl -sSL http://bit.ly/2ysbOFE | bash -s -- 1.4.6 1.4.6 0.4.18 |

## 

## **Define Business Network**

Business Network Definition involves identifying different stakeholders for our use-case and defining their roles, for our CSP Anti Subscriber Fraud Network. We will have two different types of Organizations (Grouping different Business Entities) that form a network to carry out their business in the form of transactions. These Organizations will interact with the CSP platform and perform transactions to fulfill their business requirements. By this, it ensures the authenticity of the subscribers in a secure and trusted manner with different parties.

For our use case, we will have two organizations.

* Airtel (Indian Telecom Company)
* du (UAE Telecom Company)

Airtel is a telecom company which is based out of India which makes use of the platform whenever a user appears in the Indian zone belongs to airtel services category by means of airtel subscribers or airtel partner subscriber. Airtel authenticates the user using the Blockchain Platform to know the real-time status of the subscriber.

Du is a telecom company which is based out of UAE which makes use of the platform whenever a user appears in the UAE zone belonging to the du services category by means of du subscribers or du partner subscriber. Du authenticates the user using the the Blockchain Platform to know the real-time status of the subscriber.

**STEP1:** Let's create a network configuration file. Create a file named congiftx.yml

|  |
| --- |
| Organizations:  - &du  Name: du  ID: duMSP  MSPDir: crypto-config/peerOrganizations/du/msp  AnchorPeers:  - Host: du-peer  Port: 7051  - &airtel  Name: airtel  ID: airtelMSP  MSPDir: crypto-config/peerOrganizations/airtel/msp  AnchorPeers:  - Host: airtel-peer  Port: 7051  Capabilities:  Channel: &ChannelCapabilities  V1\_3: true  Orderer: &OrdererCapabilities  V1\_1: true  Application: &ApplicationCapabilities  V1\_3: true  V1\_2: false  V1\_1: false  Application: &ApplicationDefaults   # Organizations is the list of orgs which are defined as participants on  # the application side of the network  Organizations:  Orderer: &OrdererDefaults   # Orderer Type: The orderer implementation to start  # Available types are "solo", "kafka" and "raft-etcd"  OrdererType: kafka   Addresses:  - du-orderer:7050  - airtel-orderer:7050   # Batch Timeout: The amount of time to wait before creating a batch  BatchTimeout: 2s   # Batch Size: Controls the number of messages batched into a block  BatchSize:   # Max Message Count: The maximum number of messages to permit in a batch  MaxMessageCount: 40   # Absolute Max Bytes: The absolute maximum number of bytes allowed for  # the serialized messages in a batch.  AbsoluteMaxBytes: 98 MB   # Preferred Max Bytes: The preferred maximum number of bytes allowed for  # the serialized messages in a batch. A message larger than the preferred  # max bytes will result in a batch larger than preferred max bytes.  PreferredMaxBytes: 4354 KB   Kafka:  # Brokers: A list of Kafka brokers to which the orderer connects  # NOTE: Use IP:port notation  Brokers:  - kafka0:9092  - kafka1:9092  - kafka2:9092   # Organizations is the list of orgs which are defined as participants on  # the orderer side of the network  Organizations: # We will fill this at the orderer section in the profiles   Channel: &ChannelDefaults  # Policies defines the set of policies at this level of the config tree  # For Channel policies, their canonical path is  # /Channel/<PolicyName>  Policies:  # Who may invoke the 'Deliver' API  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  # Who may invoke the 'Broadcast' API  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  # By default, who may modify elements at this config level  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"    # Capabilities describes the channel level capabilities, see the  # dedicated Capabilities section elsewhere in this file for a full  # description  Capabilities:  <<: \*ChannelCapabilities  Profiles:   netGroupOrdererGenesis:  <<: \*ChannelDefaults  Orderer:  <<: \*OrdererDefaults  Organizations:  - \*du  - \*airtel  Capabilities:  <<: \*OrdererCapabilities  Consortiums:  netConsortium:  Organizations:  - \*du  - \*airtel netGroupChannel:  Consortium: netConsortium  <<: \*ChannelDefaults  Application:  <<: \*ApplicationDefaults  Organizations:  - \*du  - \*airtel  Capabilities:  <<: \*ApplicationCapabilities |

Files under this location MSPDir: crypto-config/peerOrganizations/airtel/msp

|  |
| --- |
| ├── admincerts │ └── Admin@airtel-cert.pem ├── cacerts │ └── ca.airtel-cert.pem └── tlscacerts  └── tlsca.airtel-cert.pem  3 directories, 3 files |

**Note**: Unlike the previous chapter we no longer have a dedicated organization for ordering service. We are mapping the MSP folder that will have admin certificates for channel and genesis block. To identify an organization admin at later stages. But this is no longer needed using EnableNodeOUs: true in the **crypto-config.yml** Mainly this admin identity is used to create a **channel** in the network.

|  |
| --- |
| AnchorPeers:  - Host: airtel-peer  Port: 7051 |

***Note****: Anchor peers are used for cross-organizational gossip communication.*

**Note**: Block generation in the Hyperledger fabric mainly depends on BatchTimeout & Batch Size

**BatchTimeout**: Time to create a block

**Batch Size:** Transactions per block**. deals with the first come first serve.**

**STEP2**: Let's create a crypto configuration file. Create a file named crypto-config.yml

Hyperledger fabric is a certificate-based permissioned network which is similar to PKI [Public-Key-Infrastructure] https. However, one can generate crypto keys and certificates by using **crytogen\_tool** or using a certificate authority. In this project, we will be using cryptogen\_tool for the sake of simplicity. However, using certificate authority to generate crypto materials is explained in the SDK’s section.

|  |
| --- |
| PeerOrgs:  - Name: du  Domain: du  Template:  Count: 2  SANS:  - localhost  - du-peer  - du-orderer  Users:  Count: 1   - Name: airtel  Domain: airtel  Template:  Count: 2  SANS:  - localhost  - airtel-peer  - airtel-orderer  Users:  Count: 1 |

**Notice**: There is no OrdererOrg present in the cryptogen yaml file

**Note:** SANS is an alias or subject alternative name when you deploy a peer as a service, other services need to identify this peer by using a name and this name will be encoded to the certificate itself and peers validate it for the security reasons. A template is related to cryptogen\_tool, how many peers you want to host under a particular organization to generate certificates.

**STEP3**: Execute the below commands in order to get the artifacts

**Cryptogen**: A tool to pre-generate the certificates for identities

**Configtxgen**: A tool to generate the initial configuration of the consortium

FABRIC\_CFG\_PATH: IN order to get the supplied files: path

|  |
| --- |
| > mkdir artifacts  > export FABRIC\_CFG\_PATH=${PWD}   > cryptogen generate --config=./cryptogen.yaml  > configtxgen -profile financeGroupOrdererGenesis -outputBlock ./artifacts/genesis.block  > configtxgen -profile netGroupChannel -outputCreateChannelTx ./artifacts/cspnetchannel.tx -channelID cspnetchannel |

By this we have generated all artifacts related to the CSP Network project and we can proceed to write compose files instead of running each service in a swarm network. We can write a compose file to deploy all services such as CA, Orderer, Peers, CouchDB, Kafka, and zookeeper in one go.

**STEP4**: Write docker-compose deployment file

Fabric Provides us to use Zookeeper, Kafka, CA, Peer, Orderer, CouchDB as docker images in order to skip required dependencies installation on the current deployment environments. All Fabric images have come up with all the required values as defaults. However, using docker environment variables we can override the default variables and some variables we must override. The fabric uses viper([spf13/viper: Go configuration with fangs](https://github.com/spf13/viper)) for managing the variables.

Zookeeper: Zookeeper will need to be in a cluster in order to maintain the Kafka cluster. Let us see the zookeeper manifest. Zookeeper is a KVS store which means key-value-store. It stores a distributed configuration and helps as a synchronization service.

|  |
| --- |
| zookeeper0:  image: hyperledger/fabric-zookeeper:0.4.18  environment:  - ZOO\_SERVERS=server.1=zookeeper0:2888:3888 server.2=zookeeper1:2888:3888 server.3=zookeeper2:2888:3888  - ZOO\_MY\_ID=1  restart: always  hostname: zookeeper0  ports:  - 2181:2181  - 2888:2888  - 3888:3888 |

|  |
| --- |
| zookeeper1:  image: hyperledger/fabric-zookeeper:0.4.13  environment:  - ZOO\_SERVERS=server.1=zookeeper0:2888:3888 server.2=zookeeper1:2888:3888 server.3=zookeeper2:2888:3888  - ZOO\_MY\_ID=2  restart: always  hostname: zookeeper1  ports:  - 12181:2181  - 12888:2888  - 13888:3888 |

|  |
| --- |
| *zookeeper2:  image: hyperledger/fabric-zookeeper:0.4.13  environment:  - ZOO\_SERVERS=server.1=zookeeper0:2888:3888 server.2=zookeeper1:2888:3888 server.3=zookeeper2:2888:3888  - ZOO\_MY\_ID=3  restart: always  hostname: zookeeper2  ports:  - 22181:2181  - 22888:2888  - 23888:3888* |

Kafka: Kafka is a messaging system like other messaging services such as RabbitMQ, NATS. Kafka needs to be in a cluster for high availability. The Kafka cluster will be maintained using the zookeeper cluster. Kafka needs a zookeeper to make a leadership election. Kafka uses zookeeper for service discovery of other kafka brokers

|  |
| --- |
| kafka0:   image: hyperledger/fabric-kafka:0.4.18  restart: always  environment:  - KAFKA\_MESSAGE\_MAX\_BYTES=103809024 # 99 \* 1024 \* 1024 B  - KAFKA\_REPLICA\_FETCH\_MAX\_BYTES=103809024 # 99 \* 1024 \* 1024 B  - KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE=false  - KAFKA\_MIN\_INSYNC\_REPLICAS=2  - KAFKA\_DEFAULT\_REPLICATION\_FACTOR=3  - KAFKA\_ZOOKEEPER\_CONNECT=zookeeper0:2181,zookeeper1:2181,zookeeper2:2181  - KAFKA\_BROKER\_ID=0  - KAFKA\_ADVERTISED\_HOST\_NAME=kafka0  - KAFKA\_ADVERTISED\_PORT=9092  - KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://kafka0:9092  ports:  - 9092:9092  - 9093:9093 |

|  |
| --- |
| kafka1:  image: hyperledger/fabric-kafka:0.4.13  restart: always  environment:  - KAFKA\_MESSAGE\_MAX\_BYTES=103809024 # 99 \* 1024 \* 1024 B  - KAFKA\_REPLICA\_FETCH\_MAX\_BYTES=103809024 # 99 \* 1024 \* 1024 B  - KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE=false  - KAFKA\_MIN\_INSYNC\_REPLICAS=2  - KAFKA\_DEFAULT\_REPLICATION\_FACTOR=3  - KAFKA\_ZOOKEEPER\_CONNECT=zookeeper0:2181,zookeeper1:2181,zookeeper2:2181  - KAFKA\_BROKER\_ID=1  - KAFKA\_ADVERTISED\_HOST\_NAME=kafka1  - KAFKA\_ADVERTISED\_PORT=9092  - KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://kafka1:9092  ports:  - 10092:9092  - 10093:9093 |

|  |
| --- |
| kafka2:  image: hyperledger/fabric-kafka:0.4.13  restart: always  environment:  - KAFKA\_MESSAGE\_MAX\_BYTES=103809024 # 99 \* 1024 \* 1024 B  - KAFKA\_REPLICA\_FETCH\_MAX\_BYTES=103809024 # 99 \* 1024 \* 1024 B  - KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE=false  - KAFKA\_MIN\_INSYNC\_REPLICAS=2  - KAFKA\_DEFAULT\_REPLICATION\_FACTOR=3  - KAFKA\_ZOOKEEPER\_CONNECT=zookeeper0:2181,zookeeper1:2181,zookeeper2:2181  - KAFKA\_BROKER\_ID=2  - KAFKA\_ADVERTISED\_HOST\_NAME=kafka2  - KAFKA\_ADVERTISED\_PORT=9092  - KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://kafka2:9092  ports:  - 11092:9092  - 11093:9093 |

KAFKA\_ZOOKEEPER\_CONNECT, KAFKA\_BROKER\_ID, KAFKA\_ADVERTISED\_HOST\_NAME, KAFKA\_ADVERTISED\_LISTENERS These variables make each broker connect each other initially along with zookeeper.

**Orderer**: Unlike the previous chapter. Ordering service is in a cluster and there is no separate ordering service organization. We can see what variables are needed by the Orderer to run successfully in this project below and values to those variables are available in the github repo.

Zookeeper --> Kafka --> Orderer --> Peer

|  |
| --- |
| airtel-orderer:   image: hyperledger/fabric-orderer:1.4.6  environment:  - ORDERER\_GENERAL\_LOGLEVEL  - ORDERER\_GENERAL\_LISTENADDRESS = 0.0.0.0 #listen on any IP available   - ORDERER\_GENERAL\_GENESISMETHOD  - ORDERER\_GENERAL\_GENESISFILE  - ORDERER\_GENERAL\_LOCALMSPID = airtelMSP  - ORDERER\_GENERAL\_LOCALMSPDIR  - ORDERER\_GENERAL\_TLS\_ENABLED  - ORDERER\_GENERAL\_TLS\_PRIVATEKEY   - ORDERER\_GENERAL\_TLS\_CERTIFICATE   - ORDERER\_GENERAL\_TLS\_ROOTCAS # List of Trusted RootCA certs for auth  working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderers  command: orderer  ports:  - 7050:7050 |

|  |
| --- |
| du-orderer:   image: hyperledger/fabric-orderer:1.4.6  environment:  - ORDERER\_GENERAL\_LOGLEVEL  - ORDERER\_GENERAL\_LISTENADDRESS = 0.0.0.0 #listen on any IP available   - ORDERER\_GENERAL\_GENESISMETHOD  - ORDERER\_GENERAL\_GENESISFILE  - ORDERER\_GENERAL\_LOCALMSPID = duMSP  - ORDERER\_GENERAL\_LOCALMSPDIR  - ORDERER\_GENERAL\_TLS\_ENABLED  - ORDERER\_GENERAL\_TLS\_PRIVATEKEY   - ORDERER\_GENERAL\_TLS\_CERTIFICATE   - ORDERER\_GENERAL\_TLS\_ROOTCAS # List of Trusted RootCA certs for auth  working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderers  command: orderer  ports:  - 8050:7050 |

ORDERER\_GENERAL\_GENESISMETHOD The method by which the genesis block for the orderer system channel is specified. Available options are :

Provisional

file

Provisional: Utilizes a genesis profile, specified by GenesisProfile, to dynamically generate a new genesis block.

File: Uses the file provided by GenesisFile as the genesis block.

ORDERER\_GENERAL\_GENESISPROFILE The profile to use to dynamically generate the genesis block to use when initializing the orderer system channel.

ORDERER\_GENERAL\_GENESISFILE The file containing the genesis block to use when initializing the orderer system channel.

CouchDB:

|  |
| --- |
| image: hyperledger/fabric-couchdb  environment:  - COUCHDB\_USER=dbuser  - COUCHDB\_PASSWORD=dbpass@17788  ports:  - "5984:5984" |

Peer: We can see what variables are needed by both organizations airtel and du .

|  |
| --- |
| airtel-peer:   image: hyperledger/fabric-peer:1.4.6  environment:  - PEER\_HOST=airtel-peer  - PEER\_NAME=airtel-peer  - CORE\_PEER\_ID=airtel-peer  - CORE\_PEER\_LOCALMSPID=airtelMSP  - CORE\_PEER\_ADDRESS=airtel-peer:7051  - CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock  - CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=cspnet  - CORE\_LOGGING\_LEVEL=DEBUG  - CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=airtel-peer:7051  - CORE\_PEER\_GOSSIP\_USELEADERELECTION=true  - CORE\_PEER\_GOSSIP\_ORGLEADER=false  - CORE\_PEER\_MSPCONFIGPATH=/etc/hyperledger/crypto/peer/msp  - CORE\_PEER\_TLS\_ENABLED=true  - CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/crypto/peer/tls/server.key  - CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/crypto/peer/tls/server.crt  - CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/crypto/peer/tls/ca.crt  - CORE\_LEDGER\_STATE\_STATEDATABASE=CouchDB  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS=couchdb0:5984  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_USERNAME=dbuser  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_PASSWORD=dbpass@17788  working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer  command: peer node start  ports:  - 7051:7051  - 7053:7053 |

|  |
| --- |
| du-peer:   image: hyperledger/fabric-peer:1.4.6  environment:  - PEER\_HOST=du-peer  - PEER\_NAME=du-peer  - CORE\_PEER\_ID=du-peer  - CORE\_PEER\_LOCALMSPID=duMSP  - CORE\_PEER\_ADDRESS=du-peer:7051  - CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock  - CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=cspnet  - CORE\_LOGGING\_LEVEL=DEBUG  - CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=du-peer:7051  - CORE\_PEER\_GOSSIP\_USELEADERELECTION=true  - CORE\_PEER\_GOSSIP\_ORGLEADER=false  - CORE\_PEER\_MSPCONFIGPATH=/etc/hyperledger/crypto/peer/msp  - CORE\_PEER\_TLS\_ENABLED=true  - CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/crypto/peer/tls/server.key  - CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/crypto/peer/tls/server.crt  - CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/crypto/peer/tls/ca.crt  - CORE\_LEDGER\_STATE\_STATEDATABASE=CouchDB  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS=couchdb1:5984  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_USERNAME=dbuser  - CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_PASSWORD=dbpass@17788  working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer  command: peer node start  ports:  - 7051:7051  - 7053:7053 |

CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE is needed in order to start chaincode containers on the same bridge network as the peers. CORE\_VM\_ENDPOINT please refer to the docker sockets chapter. CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT This is an endpoint that is published to peers outside of the organization. If this isn't set, the peer will not be known to other organizations. Must be set if we enable private data concepts.

CORE\_PEER\_GOSSIP\_USELEADERELECTION Whenever peers initialize a dynamic algorithm for leader selection, where the leader in the peer to establish a connection with the ordering service and use the delivery protocol to pull ledger blocks from the ordering service. Leader Peer is responsible for delivering the blocks from orderer to other peers in the leader peer organization.

CORE\_PEER\_GOSSIP\_ORGLEADER if (true) Statically defines peer to be an organization "leader", where this means that the current peer will maintain a connection with ordering service and disseminate block across peers in its own organization.

**Note**: orgLeader and useLeaderElection parameters are mutually exclusive. Setting both to true would result in the termination of the peer since at a time both cannot be true and this leads to an undefined state.

## **Design Network Topology**

Basically, Network Topology is a high-level design or a blueprint with different deployment entities that communicate over the network with each other to complete a proposed transaction.

|  |
| --- |
|  |

## Authentication Flow:

|  |
| --- |
|  |

## **Deploy Consortium**:

Once we are ready with configuration files such as channel.tx, genesis.block , certificates for all entities, and docker-compose file then, we are ready to deploy the Fabric consortium.

**Clone the Github repo**: <https://github.com/narendranathreddythota/masteringhyperledgerfabric>

We are using docker swarm so let's complete the prerequisites before we proceed further. We can complete the deployment using a standalone single VM or two VM’s or a standalone laptop.

**STEP1**: Initialize the docker swarm

|  |
| --- |
| > IN the VM1:  docker swarm init --advertise-addr 10.1.1.68 (Only if VM has more than one IP)  --advertise-addr If a VM has two IP'S  1 PUBLIC IP (INTERNET FACING 2 PRIVATE IP (INTERNAL IP Reachable to internal VM's) |

**STEP2**: Initialization will happen in any of one VM and other VM’s need to be part of the swarm network

|  |
| --- |
| IN the VM1:  ➜ narendranathreddy ✗ docker swarm join-token manager To add a manager to this swarm, run the following command:   docker swarm join --token SWMTKN-1-4y4okz1r5n89sjut8hhhq2rbfof8li5e26bl80llb5rxzk1cfu-ah9ubqa1m31am56ju154na71u 10.1.1.68:2377 |

We have the token generated and all we have to do is execute the command along with token in another VM

|  |
| --- |
| *IN the VM2:  ➜ narendranathreddy ✗ docker swarm join --token SWMTKN-1-4y4okz1r5n89sjut8hhhq2rbfof8li5e26bl80llb5rxzk1cfu-ah9ubqa1m31am56ju154na71u 10.1.1.68:2377  This node joins the swarm network* |

Let us confirm this by using below command

|  |
| --- |
| ➜ narendranathreddy ✗ docker node ls    ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION lutnhtvd8u0ahcn4j5t0uxse2 \* node-1 Ready Active Leader 19.03.5 kutnhtdd8y0ahcn4j5t0uxse2 \* node-2 Ready Active 19.03.5 |

Let us proceed with the deployment of the consortium

|  |
| --- |
| > cd chapter-9 > hyperledger-fabric > kafka > docker stack deploy --compose-file=docker-compose.yaml csp  > cd chapter-9 > hyperledger-fabric > airtel > docker stack deploy --compose-file=docker-compose.yaml csp  > cd chapter-9 > hyperledger-fabric > du > docker stack deploy --compose-file=docker-compose.yaml csp |

Once we have deployed successfully. When we see the service list, it should show us the available replicas for each service as 1/1

|  |
| --- |
| *➜ narendranathreddy ✗ docker service ls  ID NAME MODE REPLICAS IMAGE PORTS dsak7b6hgyjw csp\_airtel-orderer replicated 1/1 hyperledger/fabric-orderer:latest \*:7050->7050/tcp lm44mqhju36d csp\_airtel-peer replicated 1/1 hyperledger/fabric-peer:latest \*:7051->7051/tcp, \*:7053->7053/tcp apdww3hn2590 csp\_ca\_airtel replicated 1/1 hyperledger/fabric-ca:latest \*:7054->7054/tcp qkff1ila6fye csp\_ca\_du replicated 1/1 hyperledger/fabric-ca:latest \*:8054->7054/tcp nvgr5afypiik csp\_couchdb1 replicated 1/1 hyperledger/fabric-couchdb:latest \*:5984->5984/tcp u675a114fkzr csp\_couchdb2 replicated 1/1 hyperledger/fabric-couchdb:latest \*:6984->5984/tcp 3d5sy8do5tf7 csp\_du-orderer replicated 1/1 hyperledger/fabric-orderer:latest \*:8050->7050/tcp k7n7xoyrcmw4 csp\_du-peer replicated 1/1 hyperledger/fabric-peer:latest \*:8051->7051/tcp, \*:8053->7053/tcp x30pzlka5aqk csp\_kafka0 replicated 1/1 hyperledger/fabric-kafka:0.4.13 \*:9092-9093->9092-9093/tcp b9qmq6s0lspo csp\_kafka1 replicated 1/1 hyperledger/fabric-kafka:0.4.13 \*:10092-10093->9092-9093/tcp pqd6ev8oypog csp\_kafka2 replicated 1/1 hyperledger/fabric-kafka:0.4.13 \*:11092-11093->9092-9093/tcp q5itkhglqn1e csp\_zookeeper0 replicated 1/1 hyperledger/fabric-zookeeper:0.4.13 \*:2181->2181/tcp, \*:2888->2888/tcp, \*:3888->3888/tcp a3lda9032m41 csp\_zookeeper1 replicated 1/1 hyperledger/fabric-zookeeper:0.4.13 \*:12181->2181/tcp, \*:12888->2888/tcp, \*:13888->3888/tcp m3x3orn1azxh csp\_zookeeper2 replicated 1/1 hyperledger/fabric-zookeeper:0.4.13 \*:22181->2181/tcp, \*:22888->2888/tcp, \*:23888->3888/tcp* |

## **Monitoring Swarm Network**:

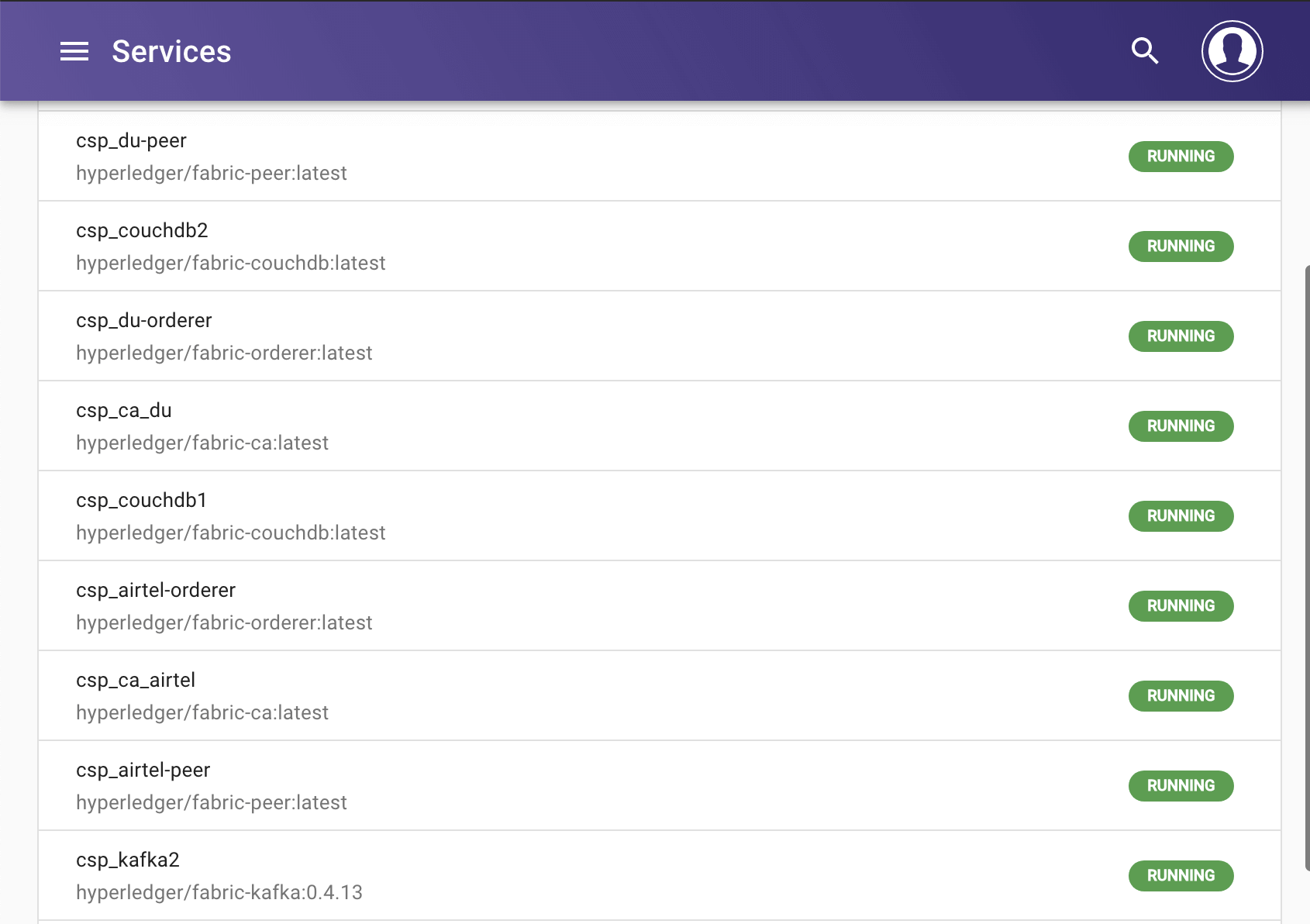
Monitoring using logspout such a big network is a nightmare. Let us use some of the best open-source tools. The best tool is swarmpit https://swarmpit.io. Using a swarm pit we can control the complete stack or even we can create a completely new stack from scratch. Resource monitoring will be easy as swarmpit Displays information about the use of hardware (CPU, memory, disk) in real-time. We can manage services that are present inside the stack or even we can create a new service from scratch. We can use swarmpit ACL, which means different uses with different privileges to manage the swarmpit. Installation Instructions: Installation happens with just one click. That is the power of docker.

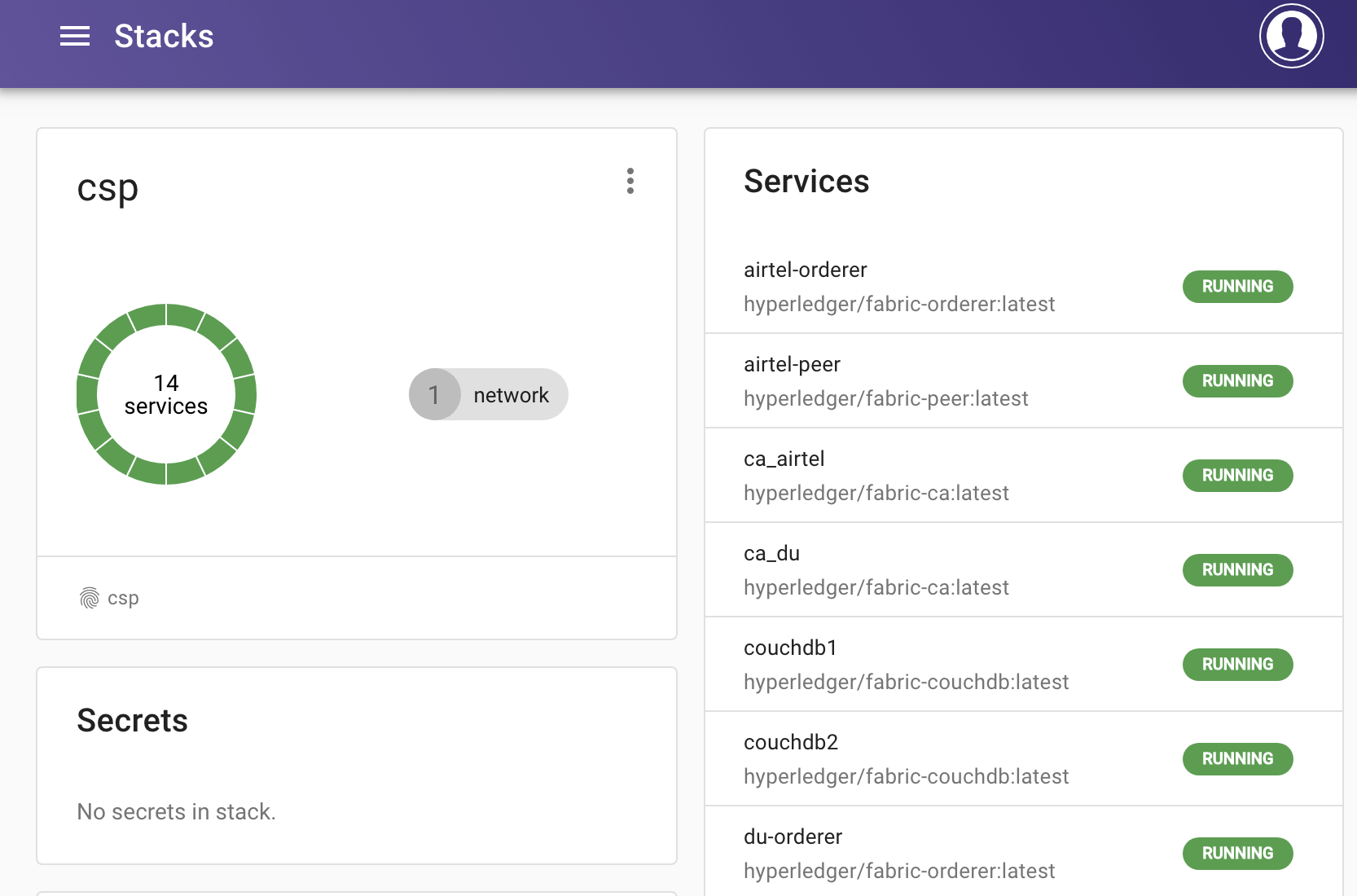
|  |
| --- |
| docker run -it --rm \  --name swarmpit-installer \  --volume /var/run/docker.sock:/var/run/docker.sock \ swarmpit/install:1.8 |

The successful installation depicts below screenshot.

|  |
| --- |
| \_\_\_\_\_ \_\_\_\_ \_ \_ \_\_ \_ \_\_ \_\_\_ \_ \_\_ (\_) |\_  / \_\_\ \ /\ / / \_` | '\_\_| '\_ ` \_ \| '\_ \| | \_\_| \\_\_ \\ V V / (\_| | | | | | | | | |\_) | | |\_  |\_\_\_/ \\_/\\_/ \\_\_,\_|\_| |\_| |\_| |\_| .\_\_/|\_|\\_\_|  |\_|  Welcome to Swarmpit Version: 1.8 Branch: 1.8 Application setup Enter stack name [swarmpit]: swarmpit Enter application port [888]: 888 Enter database volume driver [local]: local Enter admin username [admin]: admin Enter admin password (min 8 characters long): admin123 DONE. Application deployment Creating network swarmpit\_net Creating service swarmpit\_agent Creating service swarmpit\_app Creating service swarmpit\_db Creating service swarmpit\_influxdb DONE. Starting swarmpit..............DONE. Initializing swarmpit...DONE.  Summary Username: admin Password: admin123 Swarmpit is running on port :888 Enjoy :) |

|  |
| --- |
|  |





**Note**: For some reasons swarmpit may fail initially

|  |
| --- |
| Starting swarmpit.......................FAILED! Swarmpit is not responding for a long time. Aborting installation...:( Please check logs and cluster status for details. |

**Tip**: Try to remove the stack and redeploy docker stack rm swarmpit

## **Building an application**

We have defined our consortium network and deployed it successfully. Now it's time to start developing applications to interact with our network. We can develop an application [To interact with fabric] from scratch or we can just integrate with the existing running business application. It's completely loose coupling and totally user-friendly because of SDK.

**Step1:** Let's create a service connection profile file and configuration file. Create a file named network-config.yml and config file named config.json. Mentioned only the high level of the file content. Please check GitHub repo for full implementation.

|  |
| --- |
| {  "name": "csp-airtel",  "x-type": "hlfv1",  "description": "csp-airtel-admin-portal",  "version": "1.0",  "channels": {},  "organizations": {},  "orderers": {},  "peers": {},  "certificateAuthorities": {} } |

|  |
| --- |
| {  "host":"localhost",  "port":"6000",  "channelName":"cspnetchannel",  "CC\_SRC\_PATH":"../chaincode",  "chaincodeName":"policy",  "admins":[  {  "username":"airteladmin",  "secret":"airtelpw"  }  ],  "orgName":"airtel",  "clientIndy":"airtelClient",  "peerName":"airtel-peer" } |

**STEP2**: Application, further steps. Let's go further in depth and finish all the preliminary tasks. We have to get a client identity before proceeding with further operations with fabric. Getting the client object is very easy. The network-config.yml file plays a major role in the creation of a client object. Below code is sufficient to generate a client object. We will be using this object at every stage, whenever we interact with fabric.

|  |
| --- |
| const hfc = require('fabric-client'); const file = network-config.yaml; hfc.setConfigSetting('network-connection-profile-path',path.join(\_\_dirname, file)); let client = hfc.loadFromConfig(hfc.getConfigSetting('network-connection- profile-path')); await client.initCredentialStores(); return client;  const envelope = fs.readFileSync(path.join(\_\_dirname, channelConfigPath)); const channelConfig = client.extractChannelConfig(envelope);  let signature = client.signChannelConfig(channelConfig); |

channelConfigPath which is nothing but a local path where our channel file[channel.tx] exists. We created this file in the previous steps. Once we successfully mapped the respective file, we can proceed to sign the payload using the organization's admin private key. Luckily we have our admin credentials loaded with client objects. So, just proceed for signing payload. Below mentioned detailed objects which we will send to orderer as a transaction.

|  |
| --- |
| let request = { config: channelConfig, signatures: [signature], name: channelName, txId: client.newTransactionID(true)   }; const result = await client.createChannel(request); |

We can confirm whether the channel has been created successfully or not using the result object. The result object is the output from orderer. We will have a status object in the result object. If status equals SUCCESS, then the channel has been created successfully.

**STEP3**: Once the channel has been created successfully, we are allowed to onboard our peers to the channel and then, we will proceed to install chaincode operation and finally instantiate chaincode. operation.

|  |
| --- |
| const channel = client.getChannel(channel\_name); let genesis\_block = await channel.getGenesisBlock(request); let join\_request = {  targets: peers, txId: client.newTransactionID(true), block: genesis\_block  };  let result = channel.joinChannel(join\_request); |

**Peers** variable is an array of peer names. At runtime, the client object will inject peer details into the ongoing request. If we miss the peer names then the client object inject all peers that are associated to a particular channel from the connection profile will be injected. Finally, the client sends the final payload to the channel. We would need genesis block to make sure we are joining our peers in the right channel. We can confirm whether peers have been joined successfully or not using a result object. The result object is the output from the channel. We will have a status object in the result object. If status equals SUCCESS then peers have been joined successfully.

|  |
| --- |
| const request =  {  targets : peers,  chaincodePath: chaincodePath, chaincodeId: chaincodeName, chaincodeType: chaincodeType, chaincodeVersion: chaincodeVersion  }; let results = await client.installChaincode(request);  const request = {   targets : peers,   chaincodeId: chaincodeName,  chaincodeType: chaincodeType,  chaincodeVersion: chaincodeVersion,  args: args,   txId: tx\_id  };  let results = await channel.sendInstantiateProposal(request, 60000); |

Once peers have been joined successfully, we are allowed to go for install chaincode onto the peers and onboard chaincode into the channel. This process is also called a **chaincode instantiation**. By now, you must be confident enough to make a transaction with fabric.

**User** We need a user identity in order to invoke the chaincode which means to push the data fabric checks for user identity. So let us write some snippets in order to register the user. First let us get the CA admin as an object. Admins variable is defined in the config.json file.

|  |
| --- |
| var admins = hfc.getConfigSetting('admins');  let adminUserObj = await client.setUserContext({  username: admins[0].username,  password: admins[0].secret  }); |

Once we get the CA admin we are ready to register a new user. IN the fabric-samples fab car this code snippet is present in registerUser.js

|  |
| --- |
| let affiliationService = caClient.newAffiliationService();  let secret = await caClient.register({  enrollmentID: username,  affiliation: userOrg.toLowerCase() + '.department1' }, adminUserObj); |

Once we register a particular user we should proceed for enrollment. Here as every operation proceeding for enrollment in order to get the user credentials is a bit heck. Let us use some advantages provided by fabric-client.

|  |
| --- |
| user = await client.setUserContext({  username: username,  password: secret  }); |

Now we will have a question. We will create a set of users and how we can tell the client to use a particular user identity while invoking the chaincode ?

All we have to do is just pass the username to the getUserContext getUserContext

|  |
| --- |
| let user = await client.getUserContext(username, true);  if (!user) {  throw new Error(util.format('Alert! User was not found :', username));  } else {  logger.debug('User %s was found to be registered and enrolled', username);  } |

Client Interaction with fabric involves mainly two operations.

1. Invoke transaction 2. Query transaction

Proposing a transaction to fabric is not straightforward like installing chaincode or joining peers. There are several steps involved in the fabric ecosystem, right from the client proposing a transaction to peer validating this transaction in the block and committing into peer ledger. This whole process is called invoking and we have another operation called query transaction. As the name suggests it fetch queried data as a transaction.

|  |
| --- |
| Const request =   {  chaincodeId: chaincodeName,  fcn: fcn,  args: args,  chainId: channelName,  txId: tx\_id  }; let results = await channel.sendTransactionProposal(request);  const proposalResponses = results[0]; const proposal = results[1];   const orderer\_request =   {   txId: tx\_id,   proposalResponses:proposalResponses,  proposal: proposal   }; const result = channel.sendTransaction(orderer\_request);   var request =   {   chaincodeId: chaincodeName,  fcn: fcn,  args: args   }; let response\_payloads = await channel.queryByChaincode(request); |

When the client proposes a transaction, first it reaches to the endorsing peers. Endorsing peers simulates proposed transactions without disturbing the ledger and Sign the simulated result with the private key, then it is sent back to the client itself. The **results** object has both the endorsement results and the actual proposal. Proposal responses contain endorsement results. If proposal responses are good then the client submits data[proposal, proposal responses] as a payload to the orderer. We can confirm whether the proposed transaction has been committed successfully to the block or not by using the result object. The result object is the output from the orderer. We will have a status object in the result object. If status equals SUCCESS, then the proposed transaction has been committed successfully. Query data always fetches the latest data from the block.

We are ready with the functions in Application and now let us proceed to write chaincode.

## **Chain Code**:

Chaincode development starts with a transaction definition, transaction definition is a high- level blueprint. One can easily understand the whole process by looking at the chaincode logic because chaincode logic is always directly proportional to business logic. So we will start developing chaincode with transaction definitions.

|  |
| --- |
| //Subscriber structure type Subscriber struct {  Msisdn string `json:"msisdn"`  Address string `json:"address"`  AtHome bool `json:"atHome"`  Status string `json:"status"`  location string `json:"Location"`  Latitude string `json:"latitude"`  Longitude string `json:"longitude"`  IsRoaming bool `json:"isRoaming"`  HomeOperatorName string `json:"homeOperatorName"`  HomeOperatorRegion string `json:"homeOperatorRegion"`  RoamingOperatorName string `json:"roamingOperatorName"`  RoamingOperatorRegion string `json:"roamingOperatorRegion"` } |

According to the use case, there are 12 identified fields. These fields are the keys in the key- value pair of a blockchain ledger. Whenever a new subscriber(8848284834) is added to the home region with a new identity. Concerned CSP creates a new record in the CSP network and marks the field AtHome as true and IsRoaming false. Suppose if the same subscriber (8848284834) moves to a roaming region then the home CSP invokes the toRoaming function in order to share the subscriber status instantly with the roaming partners. If the subscriber identity is cloned at the roaming location, then the home CSP and roaming CSP can detect the subscriber fraud instantly. Hyperledger fabric supports various languages to write chain-codes in, but Golang is the most popular one.

Let’s, deep dive into chaincode development. Before going further, an important point needs to be highlighted here. There is a global object present in the Hyperledger fabric chaincode ecosystem which is called shim object. Package shim provides rich APIs for the chaincode to access chaincode state variables, transaction context and chaincode could be able to call other chaincodes. Below repo mentioned a detailed shim object.

|  |
| --- |
| *https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim* |

We can import shim global objects by using go imports.

|  |
| --- |
| import ( "bytes" "encoding/json" "fmt" "github.com/hyperledger/fabric/core/chaincode/shim" sc "github.com/hyperledger/fabric/protos/peer"  ) |

Golang is a statically typed language and it is a compiled language. Like java, Golang also needs a main function. As the execution starts from here. While we create a new smart contract struct, we need to make sure that all methods from the interface such as Invoke and Init are present. Since the start function in the shim package accepts the Chaincode interface. Let us see the chaincode interface.

|  |
| --- |
| type Chaincode interface {  // Init is called during Instantiate transaction after the chaincode container  // has been established for the first time, allowing the chaincode to  // initialize its internal data  Init(stub ChaincodeStubInterface) pb.Response   // Invoke is called to update or query the ledger in a proposal transaction.  // Updated state variables are not committed to the ledger until the  // transaction is committed.  Invoke(stub ChaincodeStubInterface) pb.Response } |

|  |
| --- |
| type SmartContract struct {}  func main() {   // Create a new Smart Contract  err := shim.Start(new(SmartContract))  if err != nil {  fmt.Printf("Error creating new Smart Contract: %s", err)  } } |

Let us first write the Init and functions

|  |
| --- |
| func (s \*SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response {  return shim.Success(nil) } // we are not auto populating any data through init func   func (s \*SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {  return s.Controller(APIstub) } |

|  |
| --- |
| func (s \*SmartContract) Controller(APIstub shim.ChaincodeStubInterface) sc.Response {   function, args := APIstub.GetFunctionAndParameters()  if len(args) < 1 {  str := fmt.Sprintf("Invalid request")  return shim.Error(str)  }   fmt.Println(function)  fmt.Println(args)   //guard  authorized := s.RequestAuth(APIstub, function, args)   if !authorized {  str := fmt.Sprintf("Unauthorized operation in request")  return shim.Error(str)  }   return s.InvokeController(APIstub, function, args) } |

|  |
| --- |
| //RequestAuth to check if the request is valid or not func (s \*SmartContract) RequestAuth(APIstub shim.ChaincodeStubInterface, function string, args []string) bool {  // One can have any type of check here..  return true } |

|  |
| --- |
| // InvokeController request controller func (s \*SmartContract) InvokeController(APIstub shim.ChaincodeStubInterface, function string, args []string) sc.Response {   // Route to the appropriate handler function to interact with the ledger appropriately  if function == "query" {  return s.query(APIstub, args)  } else if function == "createSubscriber" {  return s.createSubscriber(APIstub, args)  } else if function == "authenticate" {  return s.authenticate(APIstub, args)  } else if function == "toRoaming" {  return s.toRoaming(APIstub, args)  } else if function == "delete" {  return s.delete(APIstub, args)  } else if function == "update" {  return s.update(APIstub, args)  }   return shim.Error("Invalid Smart Contract function name.") } |

Let us proceed with the business methods such as **createSubscriber authenticate toRoaming**

|  |
| --- |
| // createSubscriber record for the request func (s \*SmartContract) createSubscriber(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  fmt.Println(args)  if len(args) != 1 {  str := fmt.Sprintf("Invalid request : invalid number of arguments!")  return shim.Error(str)  }  data := Subscriber{}  err := json.Unmarshal([]byte(args[0]), &data)  if err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  fmt.Printf("%v", data)  UniqueID := data.Msisdn  dataAsBytes, err := APIstub.GetState(UniqueID)  if err != nil {  return shim.Error("Failed to get subscriber: " + err.Error())  } else if dataAsBytes != nil {  fmt.Println("This subscriber already exists")  return shim.Error("This subscriber already exists")  }  dataAsBytes, err = json.Marshal(data)  if err != nil {  str := fmt.Sprintf("Can not marshal %+v", err.Error())  return shim.Error(str)  }  err = APIstub.PutState(UniqueID, dataAsBytes)  if err != nil {  str := fmt.Sprintf("Problem occured while saving the information")  return shim.Error(str)  }  fmt.Println(fmt.Sprintf("Successfully created %s", dataAsBytes))  return shim.Success(dataAsBytes) } |

|  |
| --- |
| // toRoaming record as per the request func (s \*SmartContract) toRoaming(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {   fmt.Println(args)  fmt.Println(len(args))   if len(args) < 3 {  str := fmt.Sprintf("Invalid request : invalid number of arguments!")  return shim.Error(str)  }   simBytes, err := APIstub.GetState(args[0])   if err != nil {  str := fmt.Sprintf("Problem while checking the information")  return shim.Error(str)  } else if simBytes == nil {  str := fmt.Sprintf("Information does not exists for sim")  return shim.Error(str)  }  simData := &Subscriber{}  if err := json.Unmarshal(simBytes, &simData); err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  simData.IsRoaming = true  simData.AtHome = false  simData.RoamingOperatorName = args[1]  simData.RoamingOperatorRegion = args[2]  finalSimAsBytes, err := json.Marshal(simData)  if err != nil {  str := fmt.Sprintf("Can not marshal %+v", err.Error())  return shim.Error(str)  }   err = APIstub.PutState(args[0], finalSimAsBytes)  if err != nil {  str := fmt.Sprintf("Can not put state %+v", err.Error())  return shim.Error(str)  }   fmt.Println(fmt.Sprintf("Successfully updated %s", []byte(args[0])))   return shim.Success([]byte("Success")) } |

|  |
| --- |
| // authenticate record for the request func (s \*SmartContract) authenticate(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  fmt.Println(args)  if len(args) != 2 {  str := fmt.Sprintf("Invalid request : invalid number of arguments!")  return shim.Error(str)  }   dataAsBytes, err := APIstub.GetState(args[0])  if err != nil {  return shim.Error("Failed to get subscriber: " + err.Error())  } else if dataAsBytes == nil {  fmt.Println("This subscriber does not exists")  return shim.Error("This subscriber does not exists")  }  data := Subscriber{}  if err := json.Unmarshal(dataAsBytes, &data); err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  if data.HomeOperatorRegion == args[1] {  if data.AtHome == false && data.IsRoaming == true {  return shim.Success([]byte("UnAuthorized!!"))  }  return shim.Success([]byte("Authorized"))  } else {  if data.AtHome == false && data.IsRoaming == true {  return shim.Success([]byte("Authorized!!"))  }  return shim.Success([]byte("UnAuthorized"))  }  fmt.Println(fmt.Sprintf("Successfully authenticated %s", dataAsBytes))  return shim.Success(dataAsBytes) } |

Query subscriber details at any time after creating a record in the CSP network. By using the shim object.

|  |
| --- |
| var APIstub = shim.ChaincodeStubInterface; data, \_ := APIstub.GetState(args[0])  func (s \*SmartContract) query(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {   if len(args) != 1 {  return shim.Error("Incorrect number of arguments. Expecting 1")  }   objAsBytes, \_ := APIstub.GetState(args[0])  return shim.Success(objAsBytes) } |

Removing the subscriber from the CSP network. By using the shim object

|  |
| --- |
| var APIstub = shim.ChaincodeStubInterface; if err := APIstub.DelState(args[0]); err != nil {   return shim.Error("Failed to delete state")   } |

## 

## **Interact with fabric:**

Let us run the airtel client applications which are built with nodejs-express

|  |
| --- |
| *> CD portals > airtel  ➜ airtel git:(master) ✗ npm start [2020-02-15 15:14:47.096] [INFO] airtel-admin - ------------------- SERVER STARTED ----------------------- [2020-02-15 15:14:47.099] [INFO] airtel-admin - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* http://localhost:6000 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

Create User:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/createUser \  -H "content-type: application/json" \  -d '{  "username":"airtelClient",  "orgName":"airtel" }' |

Create Channel:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/createChannel \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "userName": "airtelClient",  "orgName": "airtel",  "channelConfigPath":"../crypto/netchannel.tx" }' |

Join Peers:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/joinPeers \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "userName": "airtelClient",  "orgName": "airtel",  "peers": ["airtel-peer"] }' |

Install Chaincode:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/installChaincode \  -H "content-type: application/json" \  -d '{  "chaincodeName": "csp",  "chaincodePath": "github.com/csp/go",  "chaincodeType": "Golang ",  "chaincodeVersion": "v0",  "userName": "airtelClient",  "orgName": "airtel",  "peers": ["airtel-peer"] }' |

InstantiateChaincode:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/instantiateChaincode \  -H "content-type: application/json" \  -d '{  "chaincodeName": "csp",  "chaincodeVersion": "v0",  "userName": "airtelClient",  "orgName": "airtel",  "channelName": "netchannel",  "chaincodeType": "Golang ",  "args": [""],  "peers": ["airtel-peer"] }' |

Create Subscriber:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "chaincodeName":"csp",  "fcn": "createSubscriber",  "userName": "airtelClient",  "orgName": "airtel",  "args": ["{\r\n \"msisdn\": \"8848284834\",\r\n \"address\": \"Somewhere in the Andhra Pradesh, some PIN code\",\r\n \"atHome\": true,\r\n \"status\":\"in-force\",\r\n \"Location\": \"Kadapa, Andhrapradesh\",\r\n \"latitude\":\"86245826\",\r\n \"longitude\": \"24756247\",\r\n \"isRoaming\":false,\r\n \"HomeOperatorName\":\"airtel\",\r\n \"HomeOperatorRegion\":\"IN\",\r\n \"RoamingOperatorName\":\"\",\r\n \"RoamingOperatorRegion\":\"\"\r\n}"],  "peers": ["airtel-peer"] }' |

Query:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/query \   -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "chaincodeName":"csp",  "fcn": "query",   "userName": "airtelClient",  "orgName": "airtel",  "args": ["8848284834"],   "peers": ["airtel-peer"] }' {"address":"Somewhere in the Andhra Pradesh, some PIN code","atHome":true,"homeOperatorName":"airtel","homeOperatorRegion":"IN","isRoaming":false,"latitude":"86245826","longitude":"24756247","msisdn":"8848284834","roamingOperatorName":"","roamingOperatorRegion":"","status":"in-force"} |

Let us run the du client applications which are built with nodejs-express

|  |
| --- |
| *> CD portals > du  ➜ du git:(master) ✗ npm start [2020-02-15 15:14:47.096] [INFO] du-admin - ------------------- SERVER STARTED ----------------------- [2020-02-15 15:14:47.099] [INFO] du-admin - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* http://localhost:7000 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

CreateUser:

|  |
| --- |
| curl -s -X POST \  http://localhost:7000/createUser \  -H "content-type: application/json" \  -d '{  "username":"duClient",  "orgName":"du" }' |

Join Peers:

|  |
| --- |
| curl -s -X POST \  http://localhost:7000/joinPeers \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "userName": "duClient",  "orgName": "du",  "peers": ["du-peer"] }' |

Now the subscriber moves to roaming network location and home CSP automatically change the information:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "chaincodeName":"csp",  "fcn": "toRoaming",  "userName": "airtelClient",  "orgName": "airtel",  "args": ["8848284834", "du", "UAE"],  "peers": ["airtel-peer"] }' {"address":"Somewhere in the Andhra Pradesh, some PIN code","atHome":false,"homeOperatorName":"airtel","homeOperatorRegion":"IN","isRoaming":true,"latitude":"86245826","longitude":"24756247","msisdn":"8848284834","roamingOperatorName":"du","roamingOperatorRegion":"UAE","status":"in-force"} |

Initiate Authenticate by roaming CSP:

|  |
| --- |
| curl -s -X POST \  http://localhost:7000/query \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "chaincodeName":"csp",  "fcn": "authenticate",  "userName": "duClient",  "orgName": "du",  "args": ["8848284834", "UAE"], # 'IN' if Home CSP initiate authenticate  "peers": ["du-peer"] }' |

Let us see the UnAuthorized scenario:

|  |
| --- |
| curl -s -X POST \  http://localhost:7000/query \  -H "content-type: application/json" \  -d '{  "channelName":"netchannel",  "chaincodeName":"csp",  "fcn": "authenticate",  "userName": "duClient",  "orgName": "du",  "args": ["8848284834", "IN"],  "peers": ["du-peer"] }' # Result: UnAuthorized. Actual user is still in UAE, fraud user try to utilise services of the original user and our Blockchain platform mitigate the fraud instantly |

If the subscriber’s Identity has been cloned by the hackers from the roaming location and is trying to use it on the home network. Home network CSP will validate with the Hyperledger fabric network and instantly mitigate the fraud. When a subscriber leaves a roaming location. Roaming partner du will update the status to is roaming false and home country will accept the incoming home connection.

If a cloned subscriber enters into home location while the original subscriber on the roaming network, then home CSP authenticate against is Roaming is still true then home CSP mark the status as a cloned subscriber.

## **Tips on Errors:**

## Error: connection issue

|  |
| --- |
| error: [Remote.js]: Error: Failed to connect before the deadline [2018-10-24 13:09:48.602] [ERROR] Join-Channel -- Error: Failed to connect before the deadline at checkState (/home/server/Downloads/fabric-ca-network/TwoOrgs-ThirdLater/Org3/client-node/node\_modules/grpc/src/client.js:838:16) [2018-10-24 13:09:48.602] [ERROR] Join-Channel -- Failed to join all peers to channel. cause:Error: Failed to connect before the deadline Reason: grpcs IP and port is correct but ssl target name has been changed |

**Tip**: Check orderer ports. If orderer name is my-org1-orderer then ssl-target-name should also be my-org1-orderer.

## **Summary**

Overall, in this chapter, we have learned about knowing the status of the subscribers instantly using blockchain technology and we analyzed problems in the present CSP networks and finally, we proposed our solution with Hyperledger fabric which will mitigate all fraud happening in this sector. We learned all the way from defining a business network to interacting with the network. Now you must be confident enough to at least build a proof-of-concept for your own use case.

In the upcoming chapter, we will explore in-depth concepts of fabric and will gradually increase our learning curve step by step. We will cover all concepts such as raft consensus with multi-order setup, Kubernetes as a container orchestrator, etc.

In the next chapter, we will get to know about an innovative use case International Courier services with track and trace facility with raft consensus.

# 

# Chapter10: Running fabric Consortium with Raft consensus using Kubernetes

**SubTitle**: International Courier services with track and trace facility

Blockchain gives transparency which drives trust, accountability, and collaboration among everyone connected. Blockchain enables shippers, freight forwarders, and consumers to connect each party in their value chain from suppliers and production sites, to distribution centers and retail partners with a holistic and permanent record of every, a single transaction that takes place. These records are stored and accessible to everyone within the network. This level of transparency and permanency can be helpful for manufacturers that have to manage product origins, traceability, potential recalls, and even perishable goods that have a limited shelf life.

In this chapter, we will build a consortium that consists of

* Two organizations,
* One channel,
* Multiple orderers with raft consensus mechanism,
* One peer per organization and
* One smart contract.

## **Use Case in Detail**:

Assumes goods are now at the shipper place in India and consignments need to be delivered at the consumer’s locations in Spain Madrid. Once goods are crossed Indian waters then there is no clear tracking for the consumer as it crosses the borders and now two different tracking systems need to update through APIs. A consortium that enables Indian logistics and Spanish logistics share the locations of the goods instantly which enables consumers and distributors to track the goods in a real-time manner. Let us consider generic names for two companies such as company1, company2

Hyperledger Fabric consortium where company1 from India and company2 from Spain can share the complete journey of the consignment details in a transparent way with instantaneous results.

## **Technical Requirements**

As this chapter is completely focused on Kubernetes, let us first understand the ways to create Kubernetes clusters. Kubernetes clusters can be created in (n) No of ways, few are cloud-based solutions which mean ready to deploy. Using some tools, we can create clusters on our infrastructure. Kubernetes official page has enough info regarding the installation. <https://kubernetes.io/docs/setup>

**Some Tools used to create Kubernetes**:

* Kops
* KRIB
* Kubeadm
* Kubespray

**Using Cloud Solutions**:

* Kind (Kubernetes in Docker)
* AKS (Azure Kubernetes Service)
* GKE (Google Kubernetes Engine)
* DK (Digitalocean Kubernetes)
* Kubernetes on Alibaba Cloud
* Kubernetes on Centurylink Cloud
* Tencent Kubernetes Engine, Etc.

As a learner of Kubernetes, we have a very easy way of Kubernetes using two awesome tools

* Minikube (Using Virtualization)
* Kind (Using docker > Kubernetes in docker)

We need a computer/laptop that enables virtualization and an oracle box or kvm or hyper kit installed on the computer in order to create a minikube single node kubernetes cluster. For the sake of simplicity, we are going to use minikube but the selection of the infrastructure and type of Kubernetes cluster completely depends on the user. The environment requires at least 8gb RAM and 200 GB ROM. I already developed this project for your reference. Feel free to have a look at any time. I would strongly recommend you to do this project by yourself and use the below-mentioned GitHub repository as a reference. Detailed Installation instructions for minikube are below mentioned.

[**Install Minikube**](https://kubernetes.io/docs/tasks/tools/install-minikube/)

GitHub: <https://github.com/narendranathreddythota/masteringhyperledgerfabric/tree/master/chapter-10>

**Note**: Once we deploy Hyperledger fabric consortium. We can access our services such as orderers, peers, and ca, using minikube IP. But if a user deploys the same consortium in the cloud-based Kubernetes cluster such as AKS, GKE, etc., then there are different ways to access the cluster, we will see at the deployment section.

## **Project overview:**

We witnessed a medium level setup with docker swarm and Kafka consensus. The present chapter contains high-level concepts, as we go step by step further chapters, we will explore more and more concepts from fabric. This is mainly to build confidence and learning curve realization. Once the problem description is clearly articulated, the first step in the project is the design and define a business consortium. Next, we will be dealing with the deployment of our network. After deployment, our main focus will be on-chain code development. Finally, we will deploy the developed chain code and will prepare our application to start interacting with the network by proposing network transactions.

## **Getting started**

Before we go further deep into this chapter, you may wish to cross-check whether you have an oracle virtual box or hyper kit. The best part is making sure all prerequisites have been installed on the instances. As I mentioned in the previous chapter, we will be dealing with servers. So, we will need many software dependencies to make this project run smoothly. Below are the mentioned compulsory dependencies.

* SSH and CURL,
* Docker and Docker Desktop,
* Nodejs with NVM,
* Python,
* Golang and GIT,
* Oracle virtual box or hyper kit or kvm

## **Define Business Network**

Business Network Definition involves identifying different stakeholders for our use-case and defining their roles for our supply chain consortium. We will have two different types of Organizations (Grouping different Business Entities) that form a network to carry out their business in the form of transactions. These Organizations will interact with the supply chain consortium and perform transactions to fulfill their business requirements. By this, it ensures the authenticity and traceability of the consignments throughout the lifecycle of the chain.

For our use case, we will have two organizations.

* Company1(Indian supply chain management Company)
* Company2(Spanish supply chain management Company)

**STEP1:** Let's create a network configuration file. Create a file named congiftx.yml

|  |
| --- |
| Organizations:  - &company1  Name: company1  ID: company1MSP  MSPDir: crypto-config/peerOrganizations/company1/msp  AnchorPeers:  - Host: company1-peer  Port: 30051  Policies:  Readers:  Type: Signature  Rule: "OR('company1MSP.member')"  Writers:  Type: Signature  Rule: "OR('company1MSP.member')"  Admins:  Type: Signature  Rule: "OR('company1MSP.admin')"  Endorsement:  Type: Signature  Rule: "OR('company1MSP.admin')"  - &company2  Name: company2  ID: company2MSP  MSPDir: crypto-config/peerOrganizations/company2/msp  AnchorPeers:  - Host: company2-peer  Port: 30061  Policies:  Readers:  Type: Signature  Rule: "OR('company2MSP.member')"  Writers:  Type: Signature  Rule: "OR('company2MSP.member')"  Admins:  Type: Signature  Rule: "OR('company2MSP.admin')  Endorsement:  Type: Signature  Rule: "OR('company2MSP.admin')"  Capabilities:  Channel: &ChannelCapabilities  V1\_4\_3: true  V1\_3: false  V1\_1: false  Orderer: &OrdererCapabilities  V1\_4\_2: true  V1\_1: false  Application: &ApplicationCapabilities  V1\_4\_2: true  V1\_3: false  V1\_2: false  V1\_1: false  Application: &ApplicationDefaults  Organizations:  Policies:  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  LifecycleEndorsement:  Type: ImplicitMeta  Rule: "MAJORITY Endorsement"  Endorsement:  Type: ImplicitMeta  Rule: "MAJORITY Endorsement"  Capabilities:  <<: \*ApplicationCapabilities  Orderer: &OrdererDefaults   # Orderer Type: The orderer implementation to start  # Available types are "solo", "kafka" and "raft"  OrdererType: etcdraft  EtcdRaft:  Consenters:  - Host: orderer1  Port: 30050  ClientTLSCert: crypto-config/peerOrganizations/company1/peers/peer0.company1/tls/server.crt  ServerTLSCert: crypto-config/peerOrganizations/company1/peers/peer0.company1/tls/server.crt  - Host: orderer2  Port: 30060  ClientTLSCert: crypto-config/peerOrganizations/company1/peers/peer2.company1/tls/server.crt  ServerTLSCert: crypto-config/peerOrganizations/company1/peers/peer2.company1/tls/server.crt  - Host: orderer3  Port: 30070  ClientTLSCert: crypto-config/peerOrganizations/company1/peers/peer3.company1/tls/server.crt  ServerTLSCert: crypto-config/peerOrganizations/company1/peers/peer3.company1/tls/server.crt  Options:  TickInterval: 500ms  ElectionTick: 10  HeartbeatTick: 1  Addresses:  - orderer1:30050  - orderer2:30060  - orderer3:30070  BatchTimeout: 2s  BatchSize:  MaxMessageCount: 40  AbsoluteMaxBytes: 98 MB  PreferredMaxBytes: 4354 KB  Organizations:  Policies:  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  BlockValidation:  Type: ImplicitMeta  Rule: "ANY Writers"  Channel: &ChannelDefaults  Policies:  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  Capabilities:  <<: \*ChannelCapabilities  Profiles:   transportGroupOrdererGenesis:  <<: \*ChannelDefaults  Orderer:  <<: \*OrdererDefaults  Organizations:  - \*company1  Capabilities:  <<: \*OrdererCapabilities  Application:  <<: \*ApplicationDefaults  Organizations:  - <<: \*company1  Consortiums:  transportConsortium:  Organizations:  - \*company1  - \*company2   transportGroupChannel:  Consortium: transportConsortium  <<: \*ChannelDefaults  Application:  <<: \*ApplicationDefaults  Organizations:  - \*company1  - \*company2 |

Files under this location MSPDir: crypto-config/peerOrganizations/company1/msp

|  |
| --- |
| ├── admincerts │ └── Admin@company1-cert.pem ├── cacerts │ └── ca.company1-cert.pem └── tlscacerts  └── tlsca.company1-cert.pem 3 directories, 3 files |

**Note**: We have only company1 orderers in the consortium and there is no orderer from the company2 (Just another use case). We are mapping the MSP folder that will have admin certificates for channel and genesis block. To identify an organization admin at later stages. But this is no longer needed using EnableNodeOUs: true in the **crypto-config.yml** Mainly this admin identity is used to complete admin related tasks in the consortium.

**STEP2**: Let's create a crypto configuration file. Create a file named crypto-config.yml

Hyperledger fabric is a certificate-based permissioned network which is similar to PKI [Public-Key-Infrastructure] https. However, one can generate crypto keys and certificates by using **crytogen\_tool** or using a certificate authority. In this project, we will be using cryptogen\_tool for the sake of simplicity. However, using certificate authority to generate crypto materials, are explained in the SDK’s section

|  |
| --- |
| PeerOrgs:  - Name: company1  Domain: company1  Template:  Count: 4  SANS:  - company1-peer  - orderer1  - orderer2  - orderer3  Users:  Count: 1   - Name: company2  Domain: company2  Template:  Count: 1  SANS:  - company2-peer  Users:  Count: 1 |

**Note:** SANS is an alias or subject alternative name when you deploy a peer as a service, other services need to identify this peer by using a name and this name will be encoded to the certificate itself and peers validate it for the security reasons. A template is related to cryptogen\_tool, how many peers you want to host under a particular organization to generate certificates.

**STEP3**: Execute the below commands in order to get the artifacts

**Cryptogen**: A tool to pre-generate the certificates for identities

**Configtxgen**: A tool to generate the initial configuration of the consortium

FABRIC\_CFG\_PATH: IN order to get the supplied files: path

|  |
| --- |
| > mkdir artifacts  > export FABRIC\_CFG\_PATH=${PWD}   > cryptogen generate --config=./cryptogen.yaml  > configtxgen -profile financeGroupOrdererGenesis -outputBlock ./artifacts/genesis.block  > configtxgen -profile netGroupChannel -outputCreateChannelTx ./artifacts/cspnetchannel.tx -channelID cspnetchannel |

By this, we have generated all artifacts related to the supply chain Network project and we can proceed to write Kubernetes manifest files.

**STEP4**: Write Kubernetes manifests

Fabric Provides us to use Zookeeper, Kafka, CA, Peer, Orderer, CouchDB as docker images in order to skip required dependencies installation on the current deployment environments. All Fabric images have come up with all the required values as defaults. However, using environment variables we can override the default variables and some variables we must override. The fabric uses viper([spf13/viper: Go configuration with fangs](https://github.com/spf13/viper)) for managing the variables.

As we already know what configmap, secret, service, deployment, replica set etc. we can start developing manifests. Feel free to refer to these terms at any time in the container orchestrator section.

**Let us write CA Deployment**:

The syntax of the deployment is as below.

apiVersion Like docker, Kubernetes also expose its functionalities through API. As we know Kubernetes master has an API-server and kubectl, SDKs can interact with the API-server. Kubernetes is a very giant open-source project with many releases. To make it easier we have API versioning created and apps/v1 is an API path. Once we select the API version and now, we need to select the resource type by using Kind, metadata is the meta-information. Spec is the Specification of the desired behavior of the Deployment. Template describes the pods that will be created. PodSpec is a description of a pod. Volumes List of volumes that can be mounted by containers belonging to the pod. containers List of containers belonging to the pod. volumeMounts Pod volumes to mount into the container's filesystem. Selector Route service traffic to pods with label keys and values matching this selector.

|  |
| --- |
| apiVersion: kind: metadata:  name: spec:  replicas:   selector:  matchLabels:  template:  metadata:  labels:  spec:  Containers:  volumes: |

Now let us fill the template with detailed information. For a better view please refer to the GitHub manifest files.

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: ca-comp1-dep spec:  replicas: 1  selector:  matchLabels:  app: ca-company1  template:  metadata:  labels:  app: ca-company1  tier: identity  spec:  containers:  - name: ca  image: hyperledger/fabric-ca:1.4.6  env:  - name: FABRIC\_CA\_HOME  value: /etc/hyperledger/fabric-ca-server  - name: FABRIC\_CA\_SERVER\_CA\_NAME  value: ca-company1  - name: FABRIC\_CA\_SERVER\_TLS\_ENABLED  value: "true"  - name: FABRIC\_CA\_SERVER\_CA\_CERTFILE  value: /etc/hyperledger/fabric-ca-server-config/ca.company1-cert.pem  - name: FABRIC\_CA\_SERVER\_CA\_KEYFILE  value: /etc/hyperledger/fabric-ca-server-config/cc459d7cf9730213e2046d2058bd07989d8c2bdfdb9ed8b0bcf8a0b9bd6ca00c\_sk  - name: FABRIC\_CA\_SERVER\_CA\_CERTFILE  value: /etc/hyperledger/fabric-ca-server-config/ca.company1-cert.pem  - name: FABRIC\_CA\_SERVER\_CA\_KEYFILE  value: /etc/hyperledger/fabric-ca-server-config/cc459d7cf9730213e2046d2058bd07989d8c2bdfdb9ed8b0bcf8a0b9bd6ca00c\_sk  ports:  - containerPort: 7054  command: ["sh"]  args: ["-c", "fabric-ca-server start -b company1admin:company1pw -d"]  volumeMounts:  - mountPath: "/etc/hyperledger/fabric-ca-server-config/"  name: ca-volume  readOnly: true  volumes:  - name: ca-volume   secret:  secretName: company1-ca |

Secrets, we can create secrets from a directory or using data. In our case let us create private keys as secret by pointing to the ca key store directory. The same way configmap can be created by referencing a directory or data. When we point a directory secret or configmap kubernetes collects all files present in that directory and create files in the volume else, we can refer to a particular file instead of pointing to a directory.

|  |
| --- |
| > kubectl create secret generic company1-ca --from-file=../../artifacts/crypto-config/peerOrganizations/company1/ca/ |

|  |
| --- |
| > kubectl create configmap company1-cacrt --from-file=../../artifacts/crypto-config/peerOrganizations/company1/ca/ca.company1-cert.pem |

|  |
| --- |
| > kubectl create configmap company1-tlsca --from-file=../../artifacts/crypto-config/peerOrganizations/company1/tlsca/tlsca.company1-cert.pem |

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: ca-comp2-dep spec:  replicas: 1  selector:  matchLabels:  app: ca-company2  template:  metadata:  labels:  app: ca-company2  tier: identity  spec:  containers:  - name: ca  image: hyperledger/fabric-ca:1.4.6  env:  - name: FABRIC\_CA\_HOME  value: /etc/hyperledger/fabric-ca-server  - name: FABRIC\_CA\_SERVER\_CA\_NAME  value: ca-company2  - name: FABRIC\_CA\_SERVER\_TLS\_ENABLED  value: "true"  - name: FABRIC\_CA\_SERVER\_CA\_CERTFILE  value: /etc/hyperledger/fabric-ca-server-config/ca.company2-cert.pem  - name: FABRIC\_CA\_SERVER\_CA\_KEYFILE  value: /etc/hyperledger/fabric-ca-server-config/364092ad6e1a4a7e483c72b9e57287f05ae39504f37eefe4a5640d85cc9db90e\_sk  - name: FABRIC\_CA\_SERVER\_CA\_CERTFILE  value: /etc/hyperledger/fabric-ca-server-config/ca.company2-cert.pem  - name: FABRIC\_CA\_SERVER\_CA\_KEYFILE  value: /etc/hyperledger/fabric-ca-server-config/364092ad6e1a4a7e483c72b9e57287f05ae39504f37eefe4a5640d85cc9db90e\_sk  ports:  - containerPort: 7054  command: ["sh"]  args: ["-c", "fabric-ca-server start -b company2admin:company2pw -d"]  volumeMounts:  - mountPath: "/etc/hyperledger/fabric-ca-server-config/"  name: ca-volume  readOnly: true  volumes:  - name: ca-volume  secret:  secretName: comp2-ca |

|  |
| --- |
| > kubectl create secret generic company2-ca --from-file=../../artifacts/crypto-config/peerOrganizations/company2/ca/ |

|  |
| --- |
| > kubectl create configmap company2-cacrt --from-file=../../artifacts/crypto-config/peerOrganizations/company2/ca/ca.company2-cert.pem |

|  |
| --- |
| > kubectl create configmap company2-tlsca --from-file=../../artifacts/crypto-config/peerOrganizations/company2/tlsca/tlsca.company2-cert.pem |

Once we complete the deployment. Let us create a service in order to expose the container created in the pod under deployment. Before we proceed let us see the syntax of the service.

|  |
| --- |
| apiVersion: v1 kind: Service metadata: spec: selector: type: ports: |

We already know about apiVersion , kind , metadata , spec. Let us see the remaining pieces in the cake. The type determines how the Service is exposed. Defaults to ClusterIP. Valid options are ExternalName, ClusterIP, NodePort, and LoadBalancer. We already saw these options in the container orchestration section. "ExternalName '' maps to the specified externalName. "ClusterIP" allocates a cluster-internal IP (PrivateIP) address for load-balancing to endpoints. Endpoints are determined by the selector or if that is not specified, by manual construction of an Endpoints object. "NodePort" builds on ClusterIP and allocates a port on every node which routes to the clusterIP. "LoadBalancer" builds on NodePort and creates an external load-balancer (if supported in the current cloud) which routes to the clusterIP. Ports The list of ports that are exposed by this service. The IP protocol for this port. Supports "TCP", "UDP", and "SCTP". The default is TCP. Number or name of the port to access on the pods targeted by the service. NodePort has arranged 30000-32767, which means that a service can be exposed as a NodePort only in the specified range of ports.

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: ca-company1 spec:  selector:  app: ca-company1  tier: identity type: NodePort ports:  - name: endpoint  protocol: TCP  port: 7054  targetPort: 7054  nodePort: 30054 |

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: ca-company2 spec:  selector:  app: ca-company2  tier: identity type: NodePort ports:  - name: endpoint  protocol: TCP  port: 7054  targetPort: 7054  nodePort: 30064 |

CouchDB: As couchDB is the alternative to levelDB and only for rich queries as a world state store. We will expose couchDB service as ClusterIP as it does not need to be available outside of the cluster and this service is being utilized only by the peer. couchDB 1 for company1 and couchDB 2 for company 2.

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: couchdb1 spec:  replicas: 1  selector:  matchLabels:  app: couchdb1  template:  metadata:  labels:  app: couchdb1  tier: database  track: stable  spec:  containers:  - name: couchdb1  image: "hyperledger/fabric-couchdb"  env:  - name: COUCHDB\_USER  value: "user1"  - name: COUCHDB\_PASSWORD  value: "user1pw"  ports:  - name: couchdb1-port  containerPort: 5984 |

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: couchdb1 spec:  selector:  app: couchdb1  tier: database  ports:  - protocol: TCP  targetPort: couchdb1-port  port: 5984  name: couchdb1-port  type: ClusterIP |

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: couchdb2 spec:  replicas: 1  selector:  matchLabels:  app: couchdb2  template:  metadata:  labels:  app: couchdb2  tier: database  track: stable  spec:  containers:  - name: couchdb2  image: "hyperledger/fabric-couchdb"  env:  - name: COUCHDB\_USER  value: "user1"  - name: COUCHDB\_PASSWORD  value: "user1pw"  ports:  - name: couchdb2-port  containerPort: 5984 |

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: couchdb2 spec:  selector:  app: couchdb2  tier: database  ports:  - protocol: TCP  targetPort: couchdb2-port  port: 6984  name: couchdb2-port  type: ClusterIP |

Orderer: Ordering service is powered by a raft cluster and there is no separate ordering service organization. We can see what variables are needed by the Orderer to run successfully in this project below. As raft suggests having an odd number of nodes in a cluster and hyperledger suggests having minimum 3 orderers. Hence, we will be creating a total of three orderers, for the sake of simplicity here only one orderer is mentioned and for the other two orderers please refer to the GitHub repo.

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: orderer1 spec:  replicas: 1  selector:  matchLabels:  app: orderer1  template:  metadata:  labels:  app: orderer1  tier: backend  track: stable  spec:  containers:  - name: orderer1  image: "hyperledger/fabric-orderer:1.4.6"  ports:  - name: orderer-port  containerPort: 7050  workingDir: /opt/gopath/src/github.com/hyperledger/fabric/orderers  command: ["orderer"]  env:  - name: ORDERER\_HOST  value: "orderer1"  - name: ORDERER\_GENERAL\_LOGLEVEL  value: "debug"  - name: ORDERER\_GENERAL\_LISTENADDRESS  value: "0.0.0.0"  - name: ORDERER\_GENERAL\_LISTENPORT  value: "7050"  - name: ORDERER\_GENERAL\_GENESISMETHOD  value: "file"  - name: ORDERER\_GENERAL\_GENESISFILE  value: "/etc/hyperledger/crypto/orderer/genesis.block"  - name: ORDERER\_GENERAL\_LOCALMSPID  value: "company1MSP"  - name: GODEBUG  value: netdns=go  - name: ORDERER\_GENERAL\_LOCALMSPDIR  value: "/etc/hyperledger/crypto/orderer/msp"  - name: ORDERER\_GENERAL\_TLS\_ENABLED  value: "true"  - name: ORDERER\_GENERAL\_TLS\_PRIVATEKEY  value: "/etc/hyperledger/crypto/orderer/tls/server.key"  - name: ORDERER\_GENERAL\_TLS\_CERTIFICATE  value: "/etc/hyperledger/crypto/orderer/tls/server.crt"  - name: ORDERER\_GENERAL\_TLS\_ROOTCAS  value: "[/etc/hyperledger/crypto/orderer/tls/ca.crt]"  - name: ORDERER\_GENERAL\_CLUSTER\_CLIENTPRIVATEKEY  value: "/etc/hyperledger/crypto/orderer/tls/server.key"  - name: ORDERER\_GENERAL\_CLUSTER\_CLIENTCERTIFICATE  value: "/etc/hyperledger/crypto/orderer/tls/server.crt"  - name: ORDERER\_GENERAL\_CLUSTER\_ROOTCAS  value: "[/etc/hyperledger/crypto/orderer/tls/ca.crt]"  volumeMounts:  - name: config-genesis  mountPath: /etc/hyperledger/crypto/orderer/  - name: config-admin  mountPath: /etc/hyperledger/crypto/orderer/msp/admincerts/  - name: config-cacerts  mountPath: /etc/hyperledger/crypto/orderer/msp/cacerts/  - name: config-keystore  mountPath: /etc/hyperledger/crypto/orderer/msp/keystore/  - name: config-signcerts  mountPath: /etc/hyperledger/crypto/orderer/msp/signcerts/  - name: config-tlscacerts  mountPath: /etc/hyperledger/crypto/orderer/msp/tlscacerts/  - name: config-tls  mountPath: /etc/hyperledger/crypto/orderer/tls/   volumes:  - name: config-genesis  configMap:  name: genesis  - name: config-admin  configMap:  name: company1-admin  - name: config-cacerts  configMap:  name: company1-cacrt  - name: config-keystore  secret:  secretName: comp1-ordr1-key  - name: config-signcerts  configMap:  name: comp1-ordr1-crt  - name: config-tlscacerts  configMap:  name: company1-tlsca  - name: config-tls  secret:  secretName: comp1-ordr1-tls |

**Production Tip**: Please do not forget to persist the ledger data, raft WAL data and raft snapshot otherwise consortium will end up into serious problems. We can use PV and PVC from Kubernetes in order to complete the persistence. PV is a storage resource provisioned by an administrator. It is analogous to a node. PVC PersistentVolumeClaim is a user's request for and claims to a persistent volume.

|  |
| --- |
| apiVersion: v1 kind: PersistentVolume metadata:  name: orderer1-pv spec:  capacity:  storage: 100Gi  accessModes:  - ReadWriteOnce  persistentVolumeReclaimPolicy: Delete  storageClassName: manual  hostPath:  path: /mnt/orderer |

|  |
| --- |
| apiVersion: v1 kind: PersistentVolumeClaim metadata:  name: orderer1-pvc spec:  storageClassName: manual  accessModes:  - ReadWriteOnce  resources:  requests:  storage: 100Gi |

Let us see how volume mounts are being referenced in the manifest:

|  |
| --- |
| volumeMounts:  - mountPath: /var/hyperledger/production/orderer  name: orderer1-volume  volumes:  - name: orderer1-volume  persistentVolumeClaim:  claimName: orderer1-pvc |

We are supplying all the required artifacts, such as genesis.block, orderer private key, certificate, TLS certificate through secrets and configmaps. We can do this in (N) no of ways. Let us see one example of supplying artifacts instead of pointing to the directory. Let us create a configmap of the organization admin certificate by using data.

|  |
| --- |
| apiVersion: v1 kind: ConfigMap metadata:  name: company1-admin data:  company1\_admin-cert.pem: |-  -----BEGIN CERTIFICATE-----  MIICAjCCAaigAwIBAgIRAPNr0zp6evJ3ELydfR38ROIwCgYIKoZIzj0EAwIwYzEL  MAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDVNhbiBG  cmFuY2lzY28xETAPBgNVBAoTCGNvbXBhbnkxMRQwEgYDVQQDEwtjYS5jb21wYW55  MTAeFw0yMDAzMDQxNTMzMDBaFw0zMDAzMDIxNTMzMDBaMFMxCzAJBgNVBAYTAlVT  MRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1TYW4gRnJhbmNpc2NvMRcw  FQYDVQQDDA5BZG1pbkBjb21wYW55MTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IA  BCnlujqydjLm+e5UAhucNUeaGuXRfYIS19k/kNNv32H3IpqHuJOCvzYYxDRyI4R+  Ohga6BjiXchJySWGvJgpZcajTTBLMA4GA1UdDwEB/wQEAwIHgDAMBgNVHRMBAf8E  AjAAMCsGA1UdIwQkMCKAIMxFnXz5cwIT4gRtIFi9B5idjCvf257YsLz4oLm9bKAM  MAoGCCqGSM49BAMCA0gAMEUCIQDBtYkWXkE4P6ANH7OcMKjJcT9twOP2JZfJOEjJ  obGcLgIgffUQBxdX4I4JPZ0F3SyggEDk2v0lRWnX/dj3x3LUFHk=  -----END CERTIFICATE----- |

|  |
| --- |
| After this |- whatever the data presents, configmap just creates a file with that data |

|  |
| --- |
| #ORG Admin Certificate > kubectl create -f msp-admin-configMap.yaml  # Orderer genesis block > kubectl create configmap genesis --from-file=../../artifacts/genesis.block  #Orderer Private Key > kubectl create secret generic comp1-ordr1-key --from-file=../../artifacts/crypto-config/peerOrganizations/company1/peers/peer0.company1/msp/keystore  #Orderer sign Certificate > kubectl create configmap comp1-ordr1-crt --from-file=../../artifacts/crypto-config/peerOrganizations/company1/peers/peer0.company1/msp/signcerts  #Orderer TLS Private key and Certificate > kubectl create secret generic comp1-ordr1-tls --from-file=../../artifacts/crypto-config/peerOrganizations/company1/peers/peer0.company1/tls |

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: orderer1 spec:  selector:  app: orderer1  tier: backend  ports:  - protocol: TCP  targetPort: orderer-port #We can refer a string or a port NO   port: 30050  nodePort: 30050  name: orderer-port  type: NodePort |

Peer: The final piece is the peer. Peer exposes chaincode port, peer port, event port. It is good to expose all these through service. It is completely optional. 0.0.0.0 which is a nonroutable meta address. When we are not sure about the selection of IP and 0.0.0.0 means all IPv4 addresses on the local machine

If a host has two ip addresses, 192.168.12.14 and 10.1.2.1, and a server running on the host listens on 0.0.0.0, it will be reachable at both of those IPs.

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: company1-peer-dep spec:  replicas: 1  selector:  matchLabels:  app: comp1-peer  template:  metadata:  labels:  app: comp1-peer  tier: backend  track: stable  spec:  containers:  - name: peer1  image: "hyperledger/fabric-peer:1.4.6"  ports:  - name: peer1-port  containerPort: 7051  - name: peer1-chaincode  containerPort: 7052  - name: peer1-event  containerPort: 7053  workingDir: /opt/gopath/src/github.com/hyperledger/fabric/peer  command: ["peer"]  args: ["node","start"]  env:  - name: CORE\_VM\_ENDPOINT  value: "unix:///host/var/run/docker.sock"  - name: CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE  value: "bridge"  - name: GODEBUG  value: "netdns=go"  - name: CORE\_PEER\_ADDRESSAUTODETECT  value: "true"  - name: CORE\_PEER\_ID  value: "company1-peer"  - name: CORE\_PEER\_ADDRESS  value: "company1-peer:30051"  - name: CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT  value: "company1-peer:30051"  - name: CORE\_PEER\_CHAINCODELISTENADDRESS  value: "0.0.0.0:7052"  - name: CORE\_PEER\_GOSSIP\_BOOTSTRAP  value: "company1-peer:30051"  - name: CORE\_PEER\_LISTENADDRESS  value: "0.0.0.0:7051"  - name: CORE\_PEER\_EVENTS\_ADDRESS  value: "0.0.0.0:7053"  - name: CORE\_PEER\_LOCALMSPID  value: "company1MSP"  - name: CORE\_LOGGING\_GOSSIP  value: "INFO"  - name: CORE\_LOGGING\_PEER\_GOSSIP  value: "INFO"  - name: CORE\_LOGGING\_MSP  value: "INFO"  - name: CORE\_LOGGING\_POLICIES  value: "DEBUG"  - name: CORE\_LOGGING\_CAUTHDSL  value: "DEBUG"  - name: CORE\_PEER\_GOSSIP\_USELEADERELECTION  value: "true"  - name: CORE\_PEER\_GOSSIP\_ORGLEADER  value: "false"  - name: CORE\_PEER\_GOSSIP\_ORGLEADER  value: "false"  - name: CORE\_PEER\_MSPCONFIGPATH  value: "/etc/hyperledger/crypto/peer/msp"  - name: CORE\_PEER\_TLS\_ENABLED  value: "true"  - name: CORE\_PEER\_TLS\_KEY\_FILE  value: "/etc/hyperledger/crypto/peer/tls/server.key"  - name: CORE\_PEER\_TLS\_CERT\_FILE  value: "/etc/hyperledger/crypto/peer/tls/server.crt"  - name: CORE\_PEER\_TLS\_ROOTCERT\_FILE  value: "/etc/hyperledger/crypto/peer/tls/ca.crt"  # -- About Couch --  - name: CORE\_LEDGER\_STATE\_STATEDATABASE  value: "CouchDB"  - name: CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS  value: "couchdb1:5984"  - name: CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_USERNAME  value: ""  - name: CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_PASSWORD  value: ""  - name: CORE\_VM\_DOCKER\_ATTACHSTDOUT  value: "true"  volumeMounts:  - name: host  mountPath: /host/var/run/  - name: config-admin  mountPath: /etc/hyperledger/crypto/peer/msp/admincerts/  - name: config-cacerts  mountPath: /etc/hyperledger/crypto/peer/msp/cacerts/  - name: config-keystore  mountPath: /etc/hyperledger/crypto/peer/msp/keystore/  - name: config-signcerts  mountPath: /etc/hyperledger/crypto/peer/msp/signcerts/  - name: config-tlscacerts  mountPath: /etc/hyperledger/crypto/peer/msp/tlscacerts/  - name: config-tls  mountPath: /etc/hyperledger/crypto/peer/tls/   volumes:  - name: host  hostPath:  path: /var/run  - name: config-admin  configMap:  name: company1-admin  - name: config-cacerts  configMap:  name: company1-cacrt  - name: config-keystore  secret:  secretName: comp1-peer-key  - name: config-signcerts  configMap:  name: comp1-peer-crt  - name: config-tlscacerts  configMap:  name: company1-tlsca  - name: config-tls  secret:  secretName: comp1-peer-tls |

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: company1-peer spec:  selector:  app: comp1-peer  tier: backend  ports:  - protocol: TCP  targetPort: peer1-port  port: 30051  nodePort: 30051  name: peer1-port  - protocol: TCP  targetPort: peer1-event  port: 30053  nodePort: 30053  name: peer1-event  type: NodePort |

## **Design Network Topology**

Basically, Network Topology is a high-level design or a blueprint with different deployment entities that communicate over the network with each other to complete a proposed transaction.

|  |
| --- |
|  |

## **Deploy Consortium**:

Once we are ready with configuration files such as channel.tx, genesis.block, certificates for all entities, and Kubernetes manifests then, we are ready to deploy the Fabric consortium.

Clone the GitHub repo if you have not done already:

<https://github.com/narendranathreddythota/masteringhyperledgerfabric>

**STEP1**: Start the minikube **> minikube start**

|  |
| --- |
| ➜ minikube start --vm-driver=hyperkit --cpus=4 --memory=4000mb 😄 minikube v1.5.2 on Darwin 10.15.3 🔥 Creating hyperkit VM (CPUs=2, Memory=2000MB, Disk=20000MB) ... 🐳 Preparing Kubernetes v1.16.2 on Docker '18.09.9' ... 🚜 Pulling images ... 🚀 Launching Kubernetes ... ⌛ Waiting for: apiserver 🏄 Done! kubectl is now configured to use "minikube" |

**STEP2**: Status of the minikube and IP

|  |
| --- |
| **>** minikube status  host: Running  kubelet: Running  apiserver: Running  kubeconfig: Configured **>** minikube ip -> 192.168.64.16 |

**STEP3**: Verify using kubectl get all

|  |
| --- |
| NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 51d |

**FYI**: We have a cheat sheet from Linux academy

<https://linuxacademy.com/site-content/uploads/2019/04/Kubernetes-Cheat-Sheet_07182019.pdf>

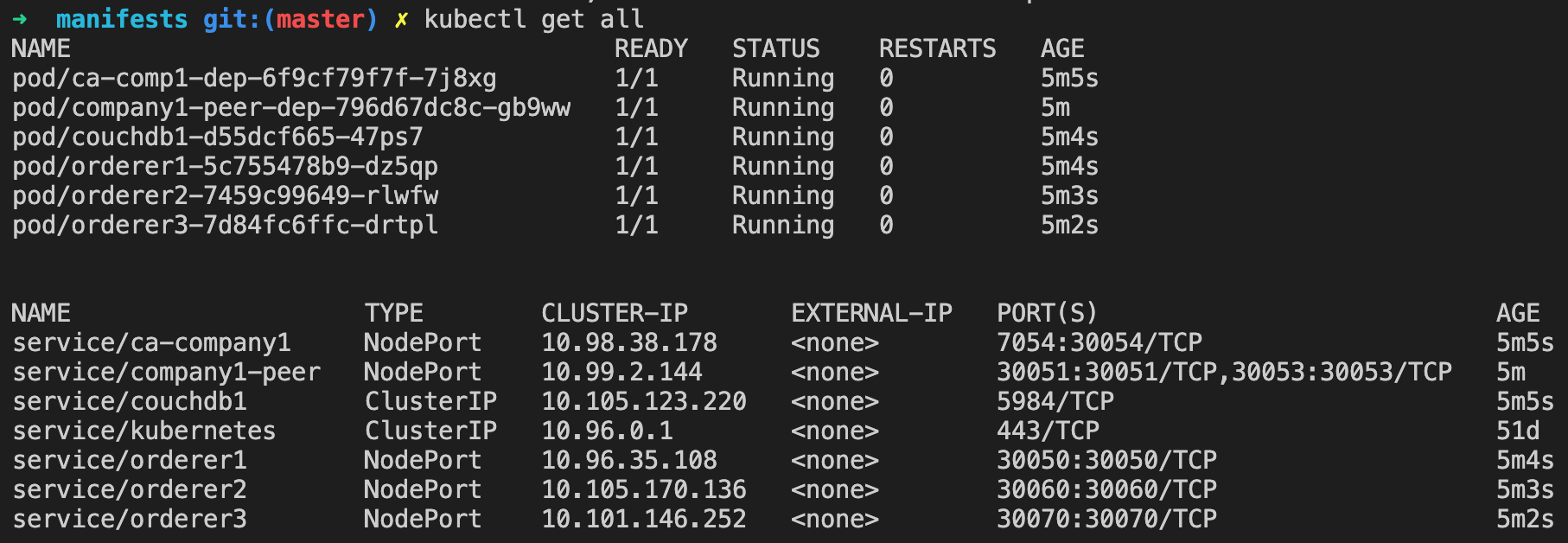
**STEP4**: Now deploy the consortium. Wait, you will have a question in your mind! We have created many resources such as orderer deployment, orderer service, peer deployment, peer service, their secrets, and configmaps. It will be difficult to deploy each and every resource. I have created a script. It is a very easy script.

Script location for company1: command line arguments 1) create 2) delete

|  |
| --- |
| **>** cd Mastering-HLF/chapter-10/hyperledger-fabric/company1/manifests  **>** chmod +x run.sh **>** ./run.sh create #delete to destroy |

|  |
| --- |
|  |

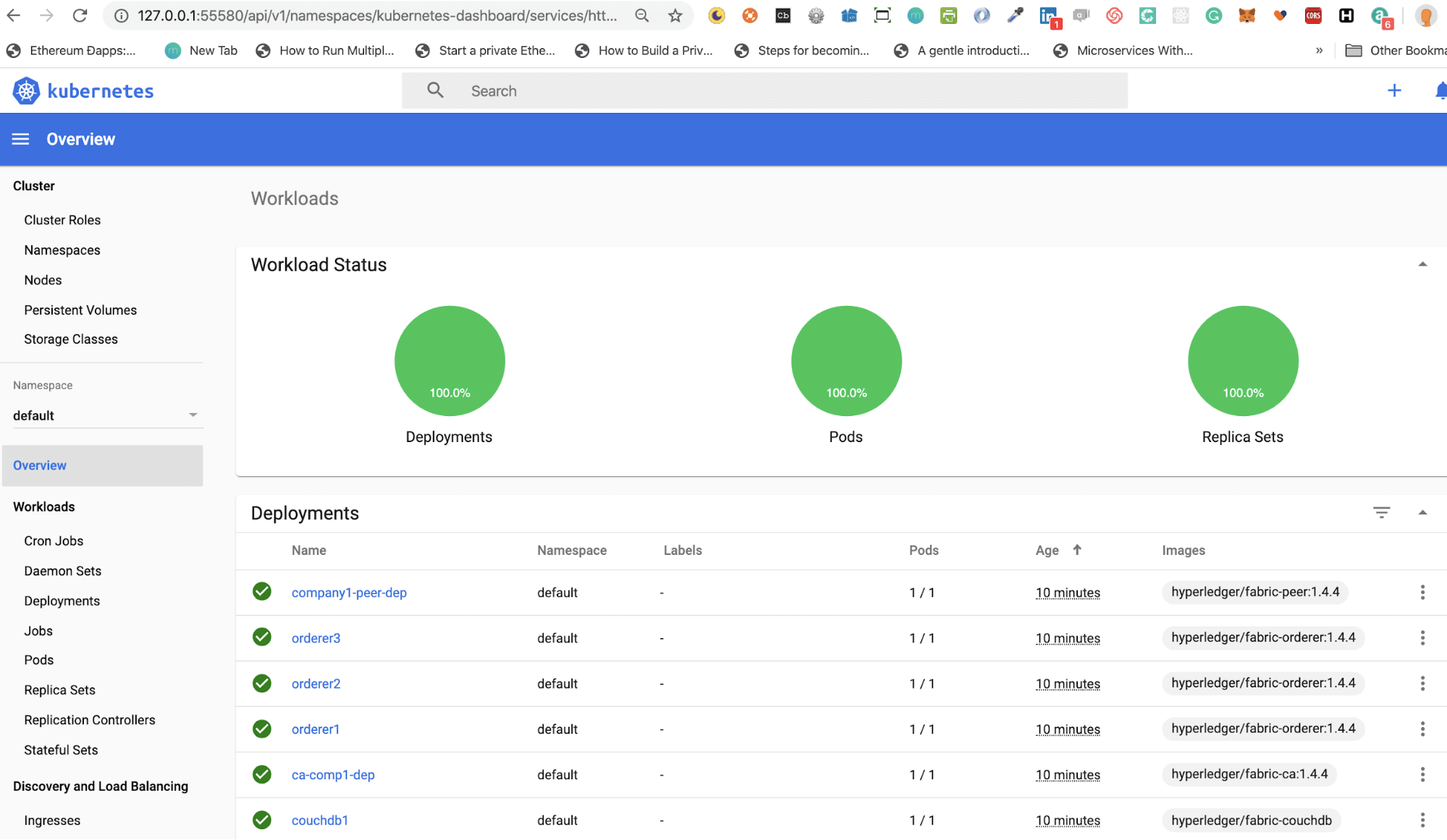
On successful deployment, kubectl get all give us the status

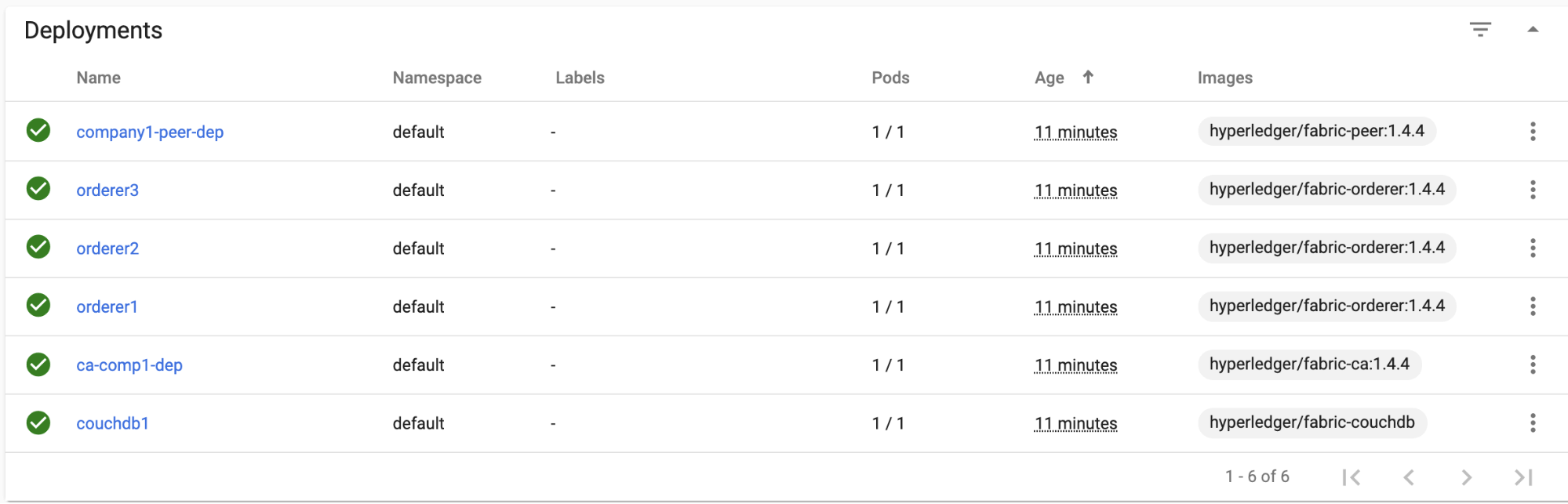


## Monitoring Kubernetes:

We can monitor our Kubernetes cluster in (n) no of ways. There is a default Kubernetes dashboard where we can see complete details of our cluster. We can see all Kubernetes related information. Pods, deployments, services, secrets, configmaps, etc. Not just seeing the information of resources but also, we can interact with the resources. Ex: for the pod, pod logs, exec into the container, edit, delete. Filter resources using namespaces. List no of nodes and resource utilization.

|  |
| --- |
| ➜ manifests git:(master) ✗ minikube dashboard 🤔 Verifying dashboard health ... 🚀 Launching proxy ... 🤔 Verifying proxy health ... 🎉 Opening http://127.0.0.1:55580/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in your default browser... |





## Monitoring Kubernetes using Weave Net:

Weave Scope is a visualization and monitoring tool for Docker and Kubernetes. It provides a top-down view into your app as well as your entire infrastructure and allows you to diagnose any problems with your distributed containerized app, in real-time, as it is being deployed to a cloud provider. Weave Scope generates a map of your processes, containers, and hosts, so that you can understand, monitor, and control your applications. Drill down on nodes in any topology, and view contextual details for your containers, hosts, and processes. With Weave Scope, you can control the entire container lifecycle across your cluster hosts from a single UI. Start, stop, pause, and restart containers from the details panel and toggle filters for stopped containers in the container’s view.

To install Scope on your Kubernetes cluster, run the following commands:

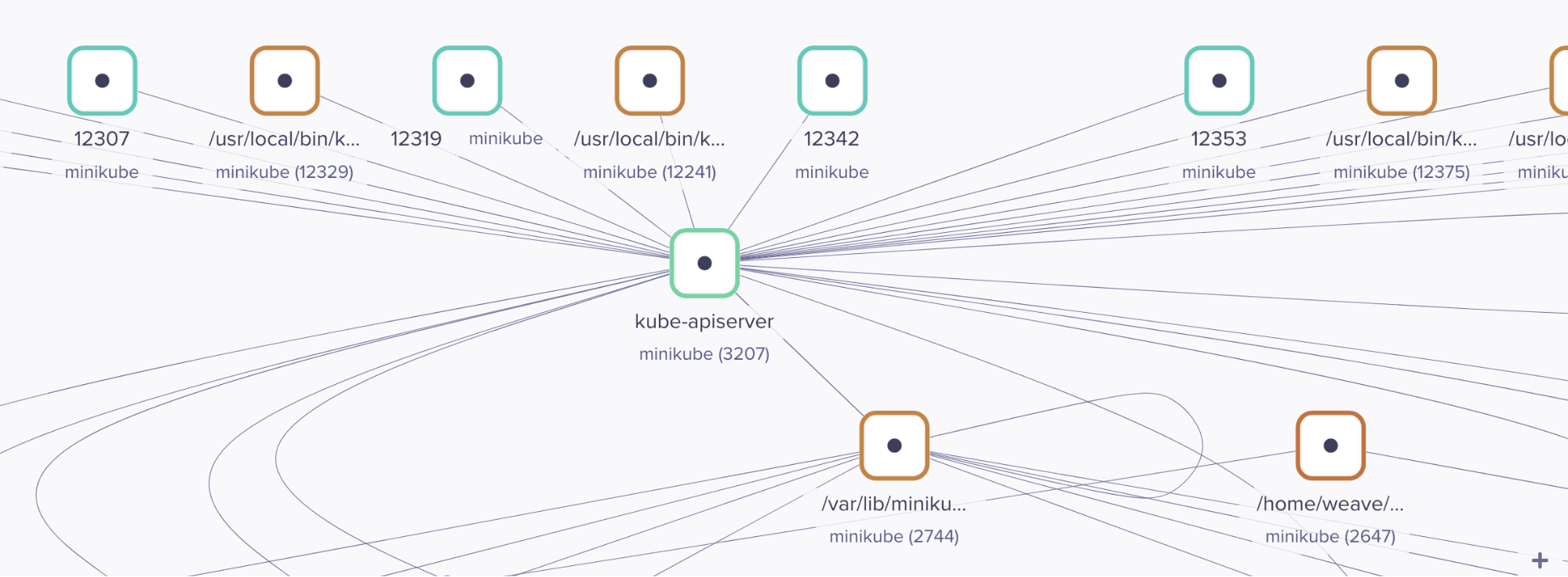
|  |
| --- |
| ➜ manifests git:(master) ✗ kubectl apply -f "https://cloud.weave.works/k8s/scope.yaml?k8s-version=$(kubectl version | base64 | tr -d '\n')" serviceaccount/weave-net created clusterrole.rbac.authorization.k8s.io/weave-net created clusterrolebinding.rbac.authorization.k8s.io/weave-net created role.rbac.authorization.k8s.io/weave-net created rolebinding.rbac.authorization.k8s.io/weave-net created daemonset.apps/weave-net created |

|  |
| --- |
| > kubectl get pods --all-namespaces kube-system weave-net-wfpk2 2/2 Running 0 2m44s |

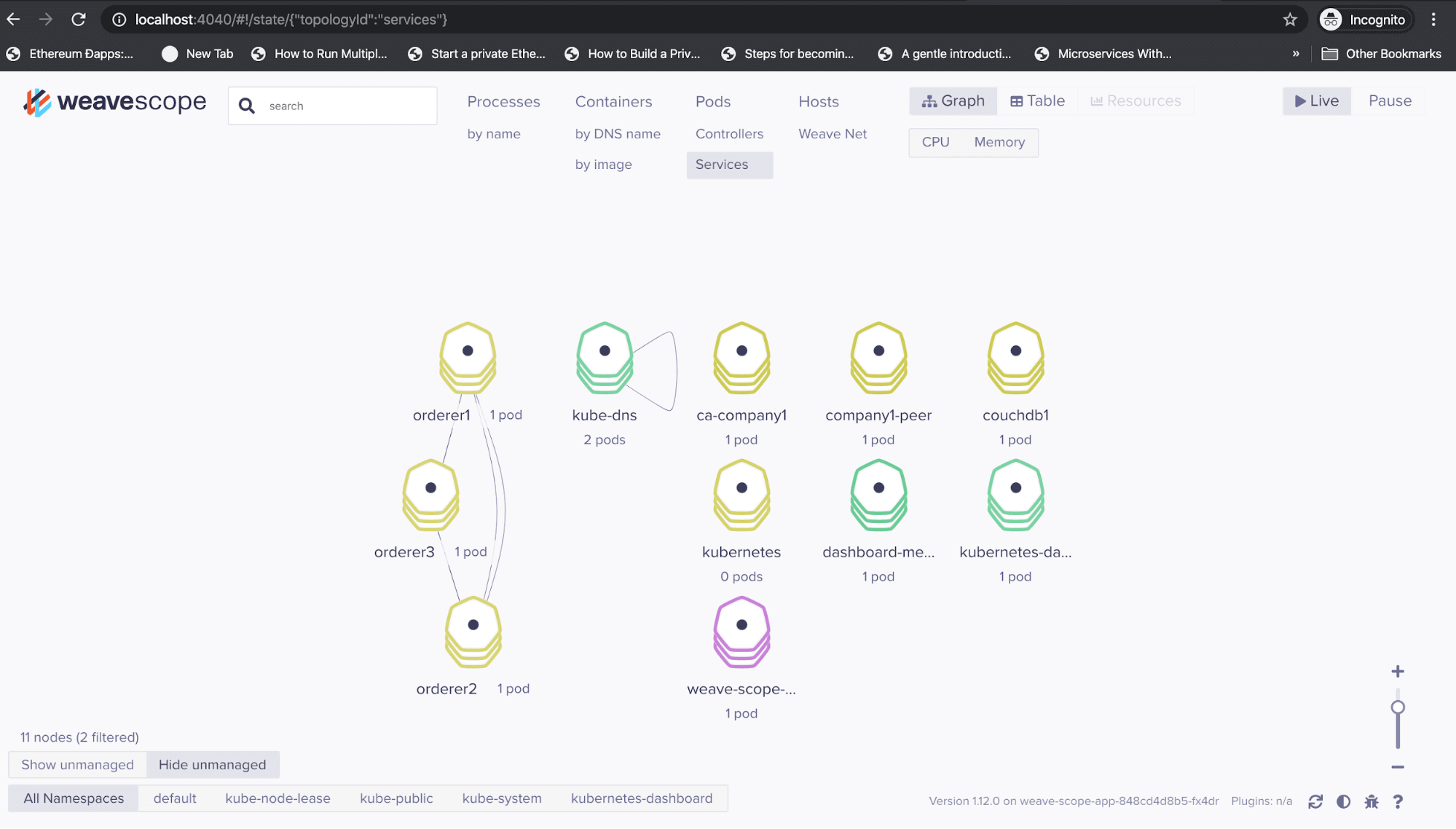
Access weave scope by using below command:

|  |
| --- |
| kubectl port-forward -n weave "$(kubectl get -n weave pod --selector=weave-scope-component=app -o jsonpath='{.items..metadata.name}')" 4040 |

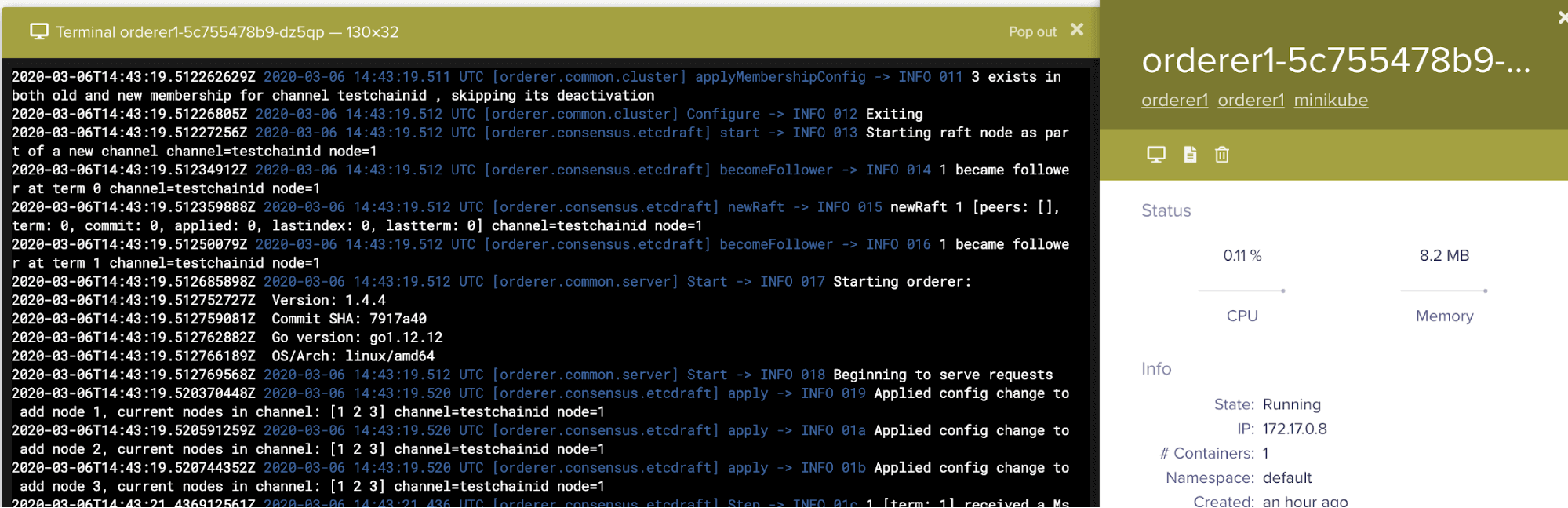
Distributed tracing of the process in the Kubernetes cluster. We can see the communication tracing with api-server.



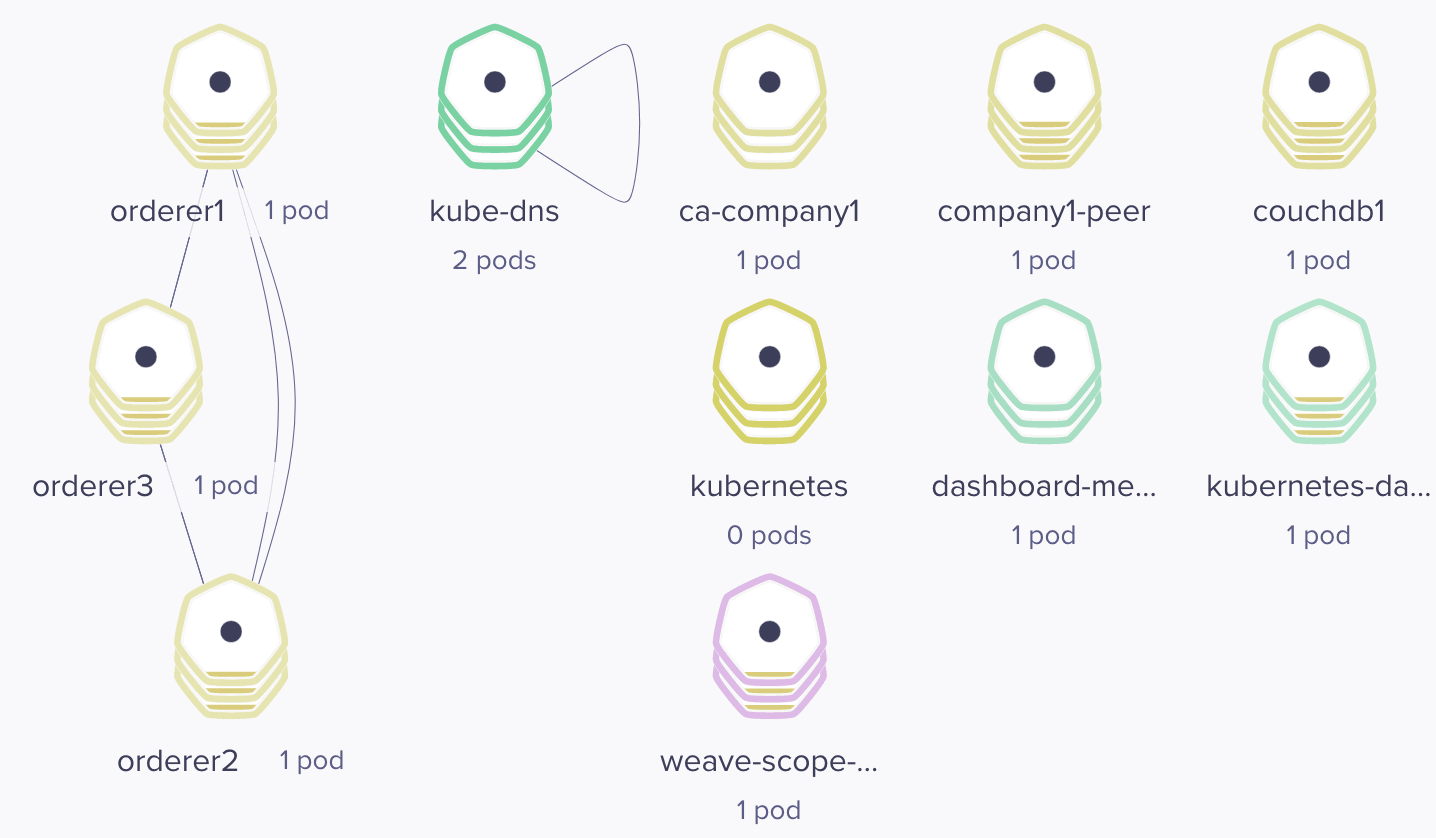
The Overview of the weave scope dashboard:



We can get the logs of any pod by just selecting a particular pod and a big dialog window will be opened in which we can get a chance to select logs, describe, delete. Let us see this in a screenshot below.



We can clearly see the connection and communication tracing in the below-mentioned screenshot. All three raft orderers are in sync. Weave scope showed us beautifully in the below image.



## **Building an application:**

We have defined our consortium network and deployed it successfully. Now it's time to start developing applications to interact with our network. We can develop an application [To interact with fabric] from scratch or we can just integrate with the existing running business application. It's completely loose coupling and totally user-friendly because of SDK.

**Step1:** Let's create a service connection profile file and configuration file. Create a file named network-config.yml and config file named config.json. Mentioned only the high level of the file content. Please check the GitHub repo for full implementation.

|  |
| --- |
| {  "name": "supplychain-company1",  "x-type": "hlfv1",  "description": "company1-admin-portal",  "version": "1.0",  "channels": {},  "organizations": {},  "orderers": {},  "peers": {},  "certificateAuthorities": {} } |

|  |
| --- |
| {  "channelName":"transportchannel",  "CC\_SRC\_PATH":"../chaincode",  "chaincodeName":"shipmement",  "admins":[  {  "username":"company1admin",  "secret":"company1pw"  }  ],  "orgName":"company1",  "clientIndy":"company1Client",  "peerName":"company1-peer" } |

**STEP2**: By Chapter, chapter9 we have already learned Application further steps so we can skip for now. Feel free to revisit at any time.

## **Chain Code**:

Chaincode development starts with a transaction definition, transaction definition is a high- level blueprint. One can easily understand the whole process by looking at the chaincode logic because chaincode logic is always directly proportional to business logic. So, we will start developing chaincode with transaction definitions. Let us see the complete shipping process in one image below. So that when we see chaincode logic it will be easy to understand. In the supply chain, there are many use cases. For the sake of simplicity, we will take a very small use case called track and trace. Our main focus is learning HLF with Kubernetes.



Source: <https://transporteca.co.uk/>

|  |
| --- |
| //Shipment structure type Shipment struct {  ShipmentID string `json:"shipmentID"`  DeliveryCountry string `json:"deliveryCountry"`  SourceCountry string `json:"sourceCountry"`  ShipmentWeight string `json:"shipmentWeight"`  DeliveryLocation string `json:"deliveryLocation"`  SourceLocation string `json:"sourceLocation"`  ClearenceStatus string `json:"clearenceStatus"`  ShipmentRange string `json:"shipmentRange"`  Status string `json:"status"`  Location string `json:"location"`  Latitude string `json:"latitude"`  Longitude string `json:"longitude"` } |

According to the use case, there are 11 identified fields. These fields are the keys in the key-value pair of a blockchain ledger. Whenever a new shipment is added at the source location. Concerned companies will create a record in the supply chain consortium. All the required fields should be filled by the source shipping company. As the shipment travels the status will be updated instantly. This allows consumers to easily track their shipments. Hyperledger fabric supports various languages to write chain-codes in, but Golang is the most popular one.

Let’s, deep dive into chaincode development. Before going further, an important point needs to be highlighted here. There is a global object present in the Hyperledger fabric chaincode ecosystem which is called shim object. Package shim provides rich APIs for the chaincode to access chaincode state variables, transaction context and chaincode could be able to call other chaincodes. Below repo mentioned a detailed shim object.

|  |
| --- |
| *https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim* |

We can import shim global objects by using go imports.

|  |
| --- |
| import ( "bytes" "encoding/json" "fmt" "github.com/hyperledger/fabric/core/chaincode/shim" sc "github.com/hyperledger/fabric/protos/peer"  ) |

Golang is a statically typed language and it is a compiled language. Like java, Golang also needs a main function. As the execution starts from here. While we create a new smart contract struct, we need to make sure that all methods from the interface such as Invoke and Init are present. Since the start function in the shim package accepts the Chaincode interface. Let us see the chaincode interface.

|  |
| --- |
| type Chaincode interface {  // Init is called during Instantiate transaction after the chaincode container  // has been established for the first time, allowing the chaincode to  // initialize its internal data  Init(stub ChaincodeStubInterface) pb.Response   // Invoke is called to update or query the ledger in a proposal transaction.  // Updated state variables are not committed to the ledger until the  // transaction is committed.  Invoke(stub ChaincodeStubInterface) pb.Response } |

|  |
| --- |
| type SmartContract struct {}  func main() {   // Create a new Smart Contract  err := shim.Start(new(SmartContract))  if err != nil {  fmt.Printf("Error creating new Smart Contract: %s", err)  } } |

Let us first write the Init and Invoke functions

|  |
| --- |
| func (s \*SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response {  return shim.Success(nil) } // we are not auto populating any data through init func   func (s \*SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {  return s.Controller(APIstub) } |

|  |
| --- |
| function, args := APIstub.GetFunctionAndParameters()  if len(args) < 1 {  str := fmt.Sprintf("Invalid request")  return shim.Error(str)  }   fmt.Println(function)  fmt.Println(args)   //guard  authorized := s.RequestAuth(APIstub, function, args)   if !authorized {  str := fmt.Sprintf("Unauthorized operation in request")  return shim.Error(str)  }   return s.InvokeController(APIstub, function, args) } |

|  |
| --- |
| //RequestAuth to check if the request is valid or not func (s \*SmartContract) RequestAuth(APIstub shim.ChaincodeStubInterface, function string, args []string) bool {  // One can have any type of check here..  return true } |

|  |
| --- |
| // InvokeController request controller func (s \*SmartContract) InvokeController(APIstub shim.ChaincodeStubInterface, function string, args []string) sc.Response {   // Route to the appropriate handler function to interact with the ledger appropriately  if function == "query" {  return s.query(APIstub, args)  } else if function == "createSubscriber" {  return s.createSubscriber(APIstub, args)  } else if function == "authenticate" {  return s.authenticate(APIstub, args)  } else if function == "toRoaming" {  return s.toRoaming(APIstub, args)  } else if function == "delete" {  return s.delete(APIstub, args)  } else if function == "update" {  return s.update(APIstub, args)  }   return shim.Error("Invalid Smart Contract function name.") } |

Let us proceed with the business methods such as **createShipment update query**

|  |
| --- |
| // createShipment record for the request func (s \*SmartContract) createShipment(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {   if len(args) != 1 {  str := fmt.Sprintf("Invalid request : invalid number of arguments!")  return shim.Error(str)  }  data := Shipment{}  err := json.Unmarshal([]byte(args[0]), &data)  if err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  fmt.Printf("%v", data)  UniqueID := data.ShipmentID  dataAsBytes, err := APIstub.GetState(UniqueID)  if err != nil {  return shim.Error("Failed to get Shipment: " + err.Error())  } else if dataAsBytes != nil {  fmt.Println("This Shipment already exists")  return shim.Error("This Shipment already exists")  }  dataAsBytes, err = json.Marshal(data)  if err != nil {  str := fmt.Sprintf("Can not marshal %+v", err.Error())  return shim.Error(str)  }  err = APIstub.PutState(UniqueID, dataAsBytes)  if err != nil {  str := fmt.Sprintf("Problem while saving the information!!")  return shim.Error(str)  }  fmt.Println(fmt.Sprintf("Successfully created %s", dataAsBytes))  return shim.Success(dataAsBytes) } |

Query shipment details at any time after creating a record in the supply chain network. By using the shim object.

|  |
| --- |
| var APIstub = shim.ChaincodeStubInterface; data, \_ := APIstub.GetState(args[0])  func (s \*SmartContract) query(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {   if len(args) != 1 {  return shim.Error("Incorrect number of arguments. Expecting 1")  }   objAsBytes, \_ := APIstub.GetState(args[0])  return shim.Success(objAsBytes) } |

|  |
| --- |
| // Update record as per the request func (s \*SmartContract) update(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  if len(args) < 1 {  str := fmt.Sprintf("Invalid request : invalid number of arguments!")  return shim.Error(str)  }  data := &Shipment{}  if err := json.Unmarshal([]byte(args[0]), &data); err != nil {  str := fmt.Sprintf("JSON Parsing exception: %+v", err)  return shim.Error(str)  }  UniqueID := data.ShipmentID   dataAsBytes, err := APIstub.GetState(UniqueID)  if err != nil {  return shim.Error(err)  } else if dataAsBytes == nil {  str := fmt.Sprintf("Information does not exists for Shipment ID:%v", data.ShipmentID)  return shim.Error(str)  }  err = APIstub.PutState(UniqueID, []byte(args[0]))  if err != nil {  str := fmt.Sprintf("Can not put state %+v", err.Error())  return shim.Error(str)  }  return shim.Success([]byte("Successfully updated")) } |

Removing the shipment from the supply chain network. By using the shim object

|  |
| --- |
| var APIstub = shim.ChaincodeStubInterface; if err := APIstub.DelState(args[0]); err != nil {  return shim.Error("Failed to delete state")  } |

## **Interact with fabric:**

Let us run the company1 client applications which are built with nodejs-express

|  |
| --- |
| *> CD portals > company1  ➜ company1 git:(master) ✗ npm start [2020-02-15 15:14:47.096] [INFO] company1-admin - ------------------- SERVER STARTED ----------------------- [2020-02-15 15:14:47.099] [INFO] company1-admin - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* http://localhost:6000 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

Create User:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/createUser \  -H "content-type: application/json" \  -d '{  "username":"company1Client",  "orgName":"company1" }' |

Create Channel:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/createChannel \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "userName": "company1Client",  "orgName": "company1",  "channelConfigPath":"../crypto/transportchannel.tx" }' |

Join Peers:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/joinPeers \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "userName": "company1Client",  "orgName": "company1",  "peers": ["company1-peer"] }' |

Install Chaincode:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/installChaincode \  -H "content-type: application/json" \  -d '{  "chaincodeName": "shipment",  "chaincodePath": "github.com/shipment/go",  "chaincodeType": "Golang ",  "chaincodeVersion": "v0",  "userName": "company1Client",  "orgName": "company1",  "peers": ["company1-peer"] }' |

InstantiateChaincode:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/instantiateChaincode \  -H "content-type: application/json" \  -d '{  "chaincodeName": "shipment",  "chaincodeVersion": "v0",  "userName": "company1Client",  "orgName": "company1",  "channelName": "transportchannel",  "chaincodeType": "Golang ",  "args": [""],  "peers": ["company1-peer"] }' |

Create Shipment:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "chaincodeName":"shipment",  "fcn": "createShipment",  "userName": "company1Client",  "orgName": "company1",  "args": ["{\n \"shipmentID\": \"1245343636\",\n \"deliveryCountry\": \"Spain\",\n \"sourceCountry\": \"India\",\n \"shipmentWeight\":\"10,000kg\",\n \"deliveryLocation\": \"Spain, Madrid, sol\",\n \"sourceLocation\":\"India, Hyderabad, HitechCity\",\n \"clearenceStatus\":\"PENDING\",\n \"shipmentRange\": \"Origin\",\n \"status\":\"CREATED\",\n \"location\":\"India\",\n \"latitude\":\"24524624426\",\n \"longitude\":\"245245425\"\n}"],  "peers": ["company1-peer"] }' |

Query:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/query \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "chaincodeName":"shipment",  "fcn": "query",  "userName": "company1Client",  "orgName": "company1",  "args": ["1245343636"],  "peers": ["company1-peer"] }' {"clearenceStatus":"PENDING","deliveryCountry":"Spain","deliveryLocation":"Spain, Madrid, sol","latitude":"24524624426","location":"India","longitude":"245245425","shipmentID":"1245343636","shipmentRange":"Origin","shipmentWeight":"10,000kg","sourceCountry":"India","sourceLocation":"India, Hyderabad, HitechCity","status":"CREATED"} |

Update Shipment:

|  |
| --- |
| curl -s -X POST \  http://localhost:6000/invoke \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "chaincodeName":"shipment",  "fcn": "update",  "userName": "company2Client",  "orgName": "company2",  "args": ["{\n \"shipmentID\": \"1245343636\",\n \"deliveryCountry\": \"Spain\",\n \"sourceCountry\": \"India\",\n \"shipmentWeight\":\"10,000kg\",\n \"deliveryLocation\": \"Spain, Madrid, sol\",\n \"sourceLocation\":\"India, Hyderabad, HitechCity\",\n \"clearenceStatus\":\"PENDING\",\n \"shipmentRange\": \"Crossed Indian Waters\",\n \"status\":\"SHIPPED\",\n \"location\":\"Dubai\",\n \"latitude\":\"24524624426\",\n \"longitude\":\"245245425\"\n}"],  "peers": ["company2-peer"] }' |

Let us run the company2 client applications which are built with nodejs-express

|  |
| --- |
| *> CD portals > company2  ➜ company2 git:(master) ✗ npm start [2020-02-15 15:14:47.096] [INFO] company2-admin - ------------------- SERVER STARTED ----------------------- [2020-02-15 15:14:47.099] [INFO] company2-admin - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* http://localhost:7000 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

Create User:

|  |
| --- |
| curl -s -X POST \  http://localhost:7000/createUser \  -H "content-type: application/json" \  -d '{  "username":"company2Client",  "orgName":"company2" }' |

Join Peers:

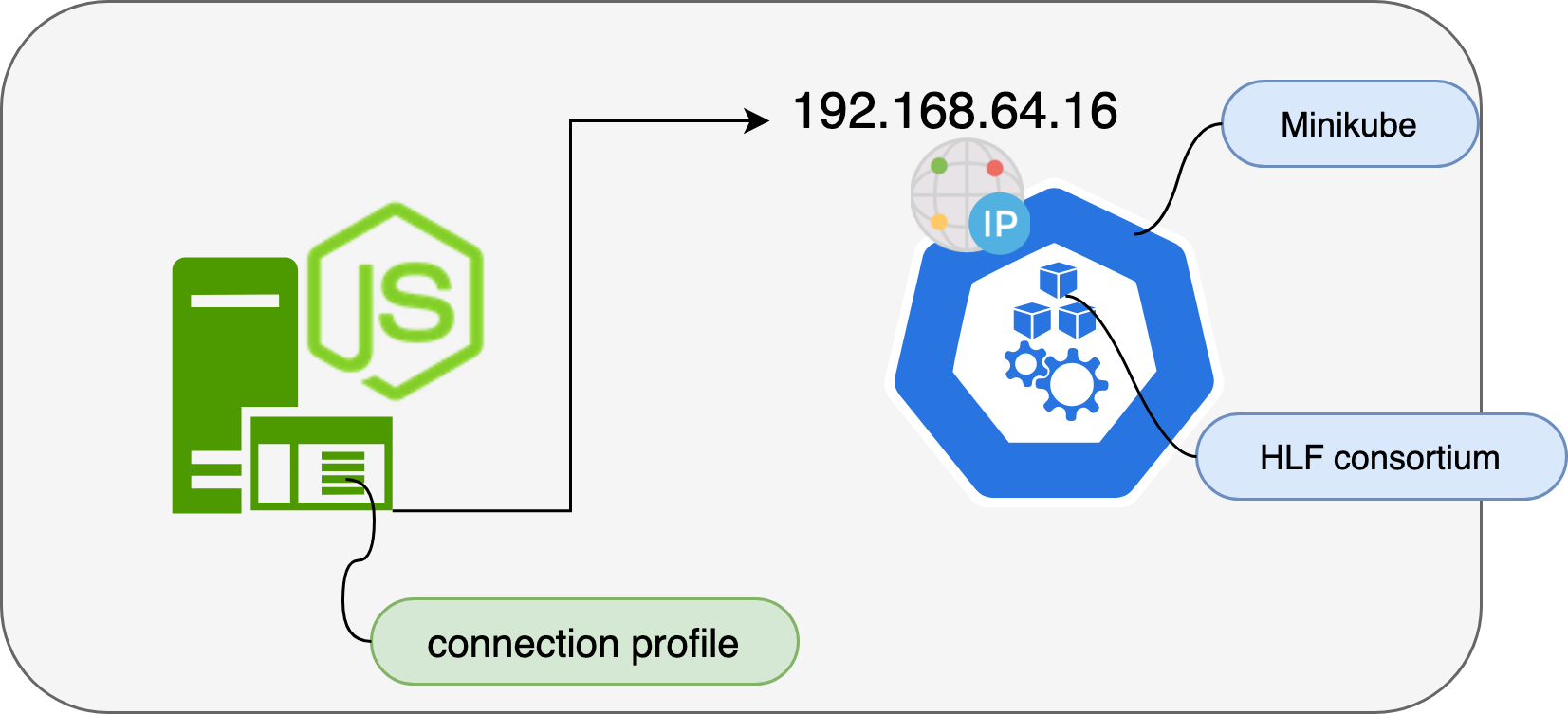
|  |
| --- |
| curl -s -X POST \  http://localhost:7000/joinPeers \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "userName": "company2Client",  "orgName": "company2",  "peers": ["company2-peer"] }' |

Query:

|  |
| --- |
| curl -s -X POST \  http://localhost:7000/query \  -H "content-type: application/json" \  -d '{  "channelName":"transportchannel",  "chaincodeName":"shipment",  "fcn": "query",  "userName": "company1Client",  "orgName": "company2",  "args": ["1245343636"],  "peers": ["company2-peer"] }' {"clearenceStatus":"PENDING","deliveryCountry":"Spain","deliveryLocation":"Spain, Madrid, sol","latitude":"24524624426","location":"Dubai","longitude":"245245425","shipmentID":"1245343636","shipmentRange":"Crossed Indian Waters","shipmentWeight":"10,000kg","sourceCountry":"India","sourceLocation":"India, Hyderabad, HitechCity","status":"SHIPPED"} |

**Let us see the architecture of the current stack with minikube**:

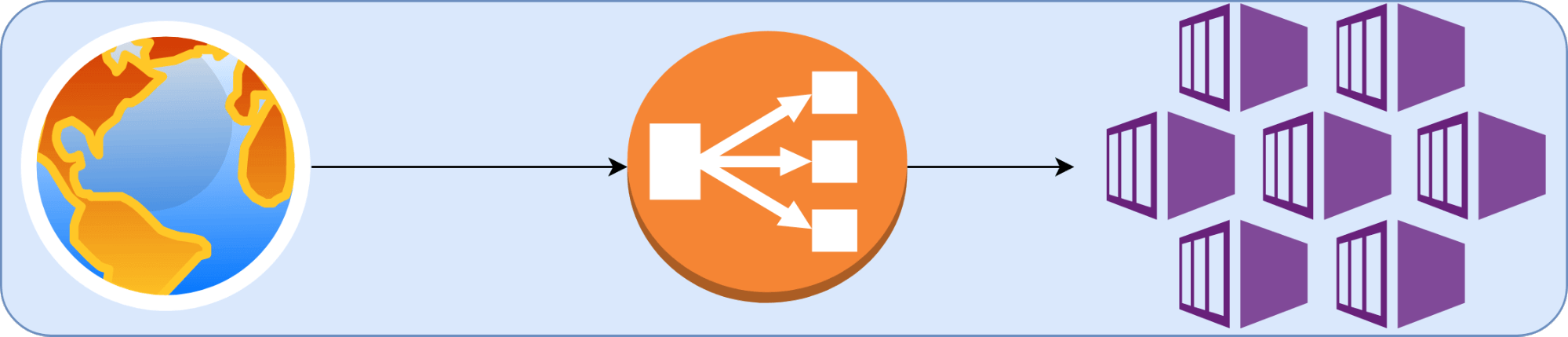
As we clearly understand with the below image, that minikube has one node so all services deployed as NodePort will be available with the node IP. So, Peer which is deployed to the minikube will be able to access as at 30061 and the complete endpoint is as the following grpcs://192.168.64.16:30061



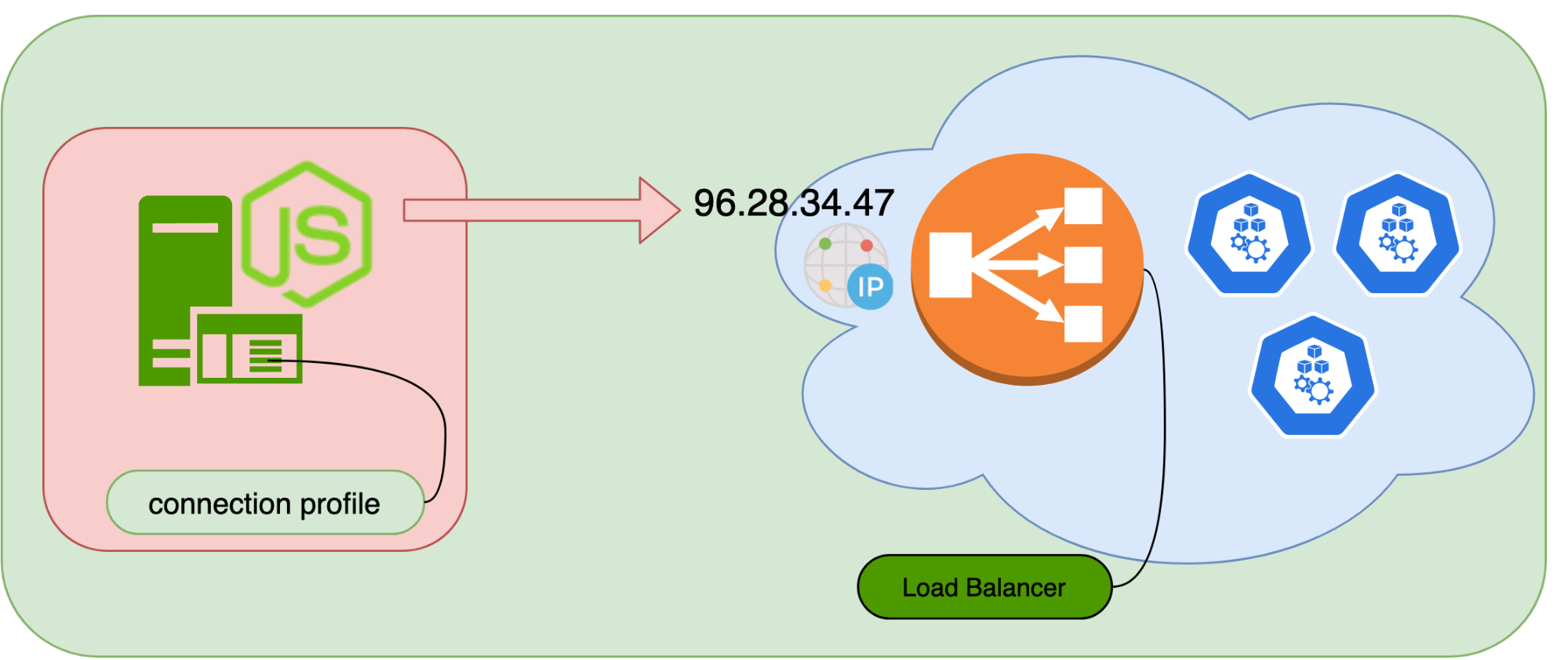
**Let us see the architecture of the cloud stack**:

**Type1**: Access HLF consortium from outside the luster

Now we clearly understand with the below image, that cloud kubernetes service many nodes and all services deployed as NodePort or cluster IP will be available through the ingress controller or load balancer. Popular load balancers are Kubernetes Community Edition, Traffic, Nginx, etc. A peer that is deployed to the cluster will be able to access as at 96.28.34.47:30061 and the complete endpoint is as the following grpcs://96.28.34.47:30061 .

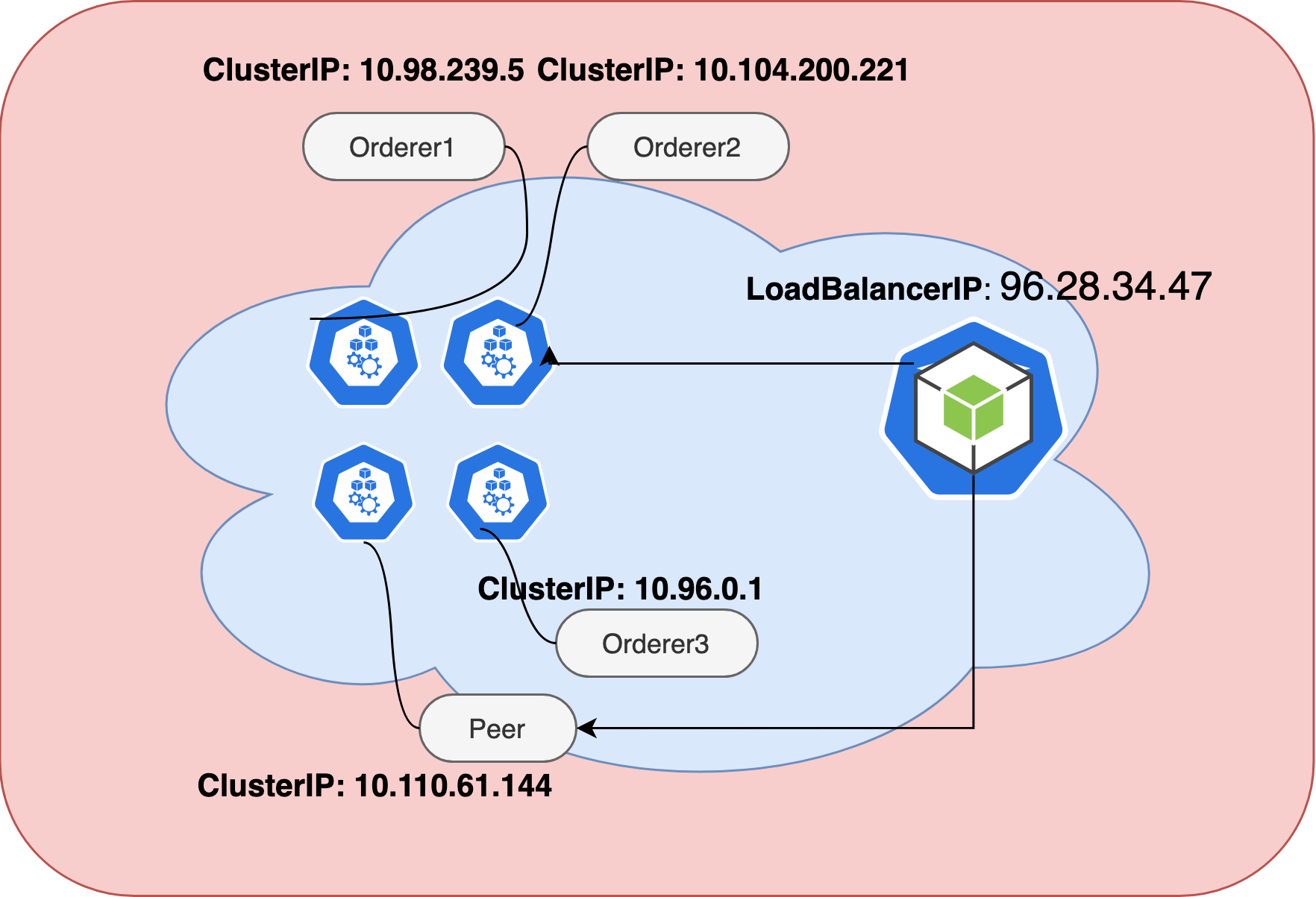


|  |
| --- |
| internet  |  [ Load Balancer ]  --|-----|--  [ Services ] |



**Type2**: Access HLF from Inside the cluster

Accessing our consortium from the inside cluster, which means our NodeJS Application specially designed for a gateway to the HLF or our business application which has already HLF SDK is also present inside the cluster. This approach is more suitable for those who do not want to expose their HLF to the outside world. In this approach Peer which is deployed to the cluster will be able to access as at 10.110.61.144:30061 which is clusterIP and the complete endpoint is as the following grpcs://10.110.61.144:30061 . ClusterIP is accessible only from the inside of the cluster. So there is no need to create an ingress or load balancer.



## **Summary:**

Overall, in this chapter, we learned about knowing the status of the shipments instantly using blockchain technology. We have set up our consortium using raft consensus with multi-order setup, Kubernetes as a container orchestrator. We have learned many ways to create Kubernetes clusters and identified that minikube is the best in terms of demonstration or POC purpose and we figured out other ways to architect Hyperledger fabric consortium and application part with cloud-based Kubernetes clusters. Creating manifests for orderer, peer, ca and couch are now easy. Explored many concepts in the Kubernetes such as deployment, service, configmap, secrets, etc. We have learned all the way from defining a business network to interacting with the network. Now you must be confident enough to at least build a proof-of-concept or production-grade Hyperledger fabric consortium and SDK application in order to interact with the consortium which is running in any cloud, for your own use case.

In the upcoming chapter, we will explore furthermore interesting topics with and in Hyperledger fabric. We will gradually increase our learning curve step by step.

In the next chapter, we will explore one of the most interesting topic private data concepts and Access control lists (ACL)

# 

# Chapter11: Private Data Concept, Consortium level ACL (Access Control Lists) and Raft Consensus

This chapter is covered with three miscellaneous concepts. Completely theory and additional practical snippets covered. Private data concept, access controls, and raft consensus mechanism these three are very crazy topics in the Hyperledger fabric whichever version. Hyperledger fabric is a consortium-based permissioned blockchain network where groups of organizations form together to exchange some information with having some rules and regulations. This is governed by the access controls at the consortium level, channel level, and application level. Even in the consortia privacy across members is important at some particular time. Some information necessarily remains confidential between some sets of organizations. This is where enabling private data inside the consortium is a must. When we take organization as a whole entity or workspace, we will end up dealing with multiple identities, multiple users from different departments in an organization, so there is also a need to enable access control at the organization level. Once we create a channel and organizations are part of the channel, there will be a need to enable access control at the channel level as different organizations are exchanging information.

When a single database is being used by the server then all transactions are created, updated are done perfectly. Due to load balancing and fault tolerance, we put our database in a cluster mode. It means that more than one database will be in a cluster and then create and update operations are initiated, data across the databases in a cluster needs to be in a consistent mode and this is called a distributed consistency problem.

In this chapter, we will see above all problems and answers in a detailed manner with a clear explanation.

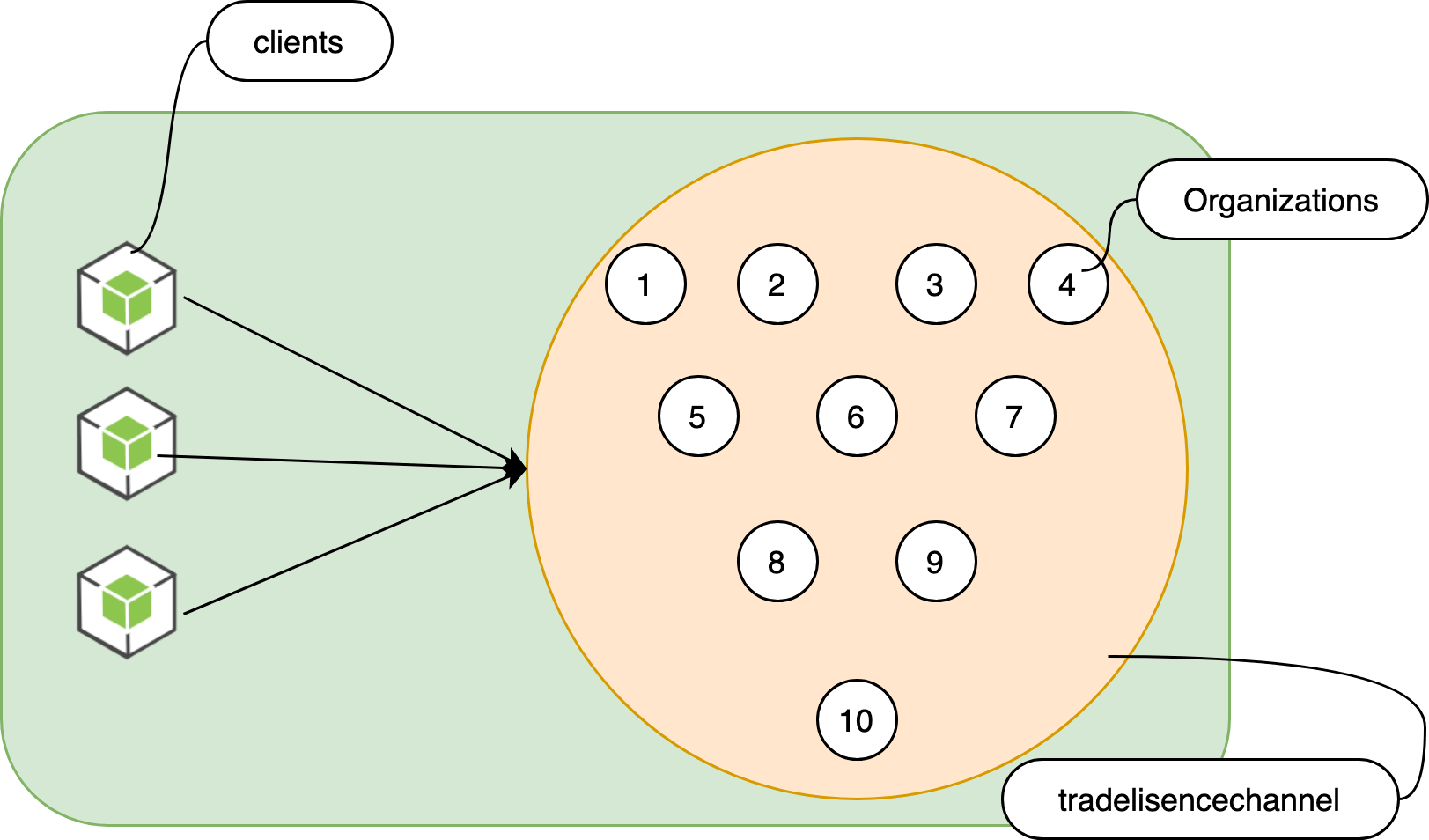
## Private Data Concept:

### **Theory**:

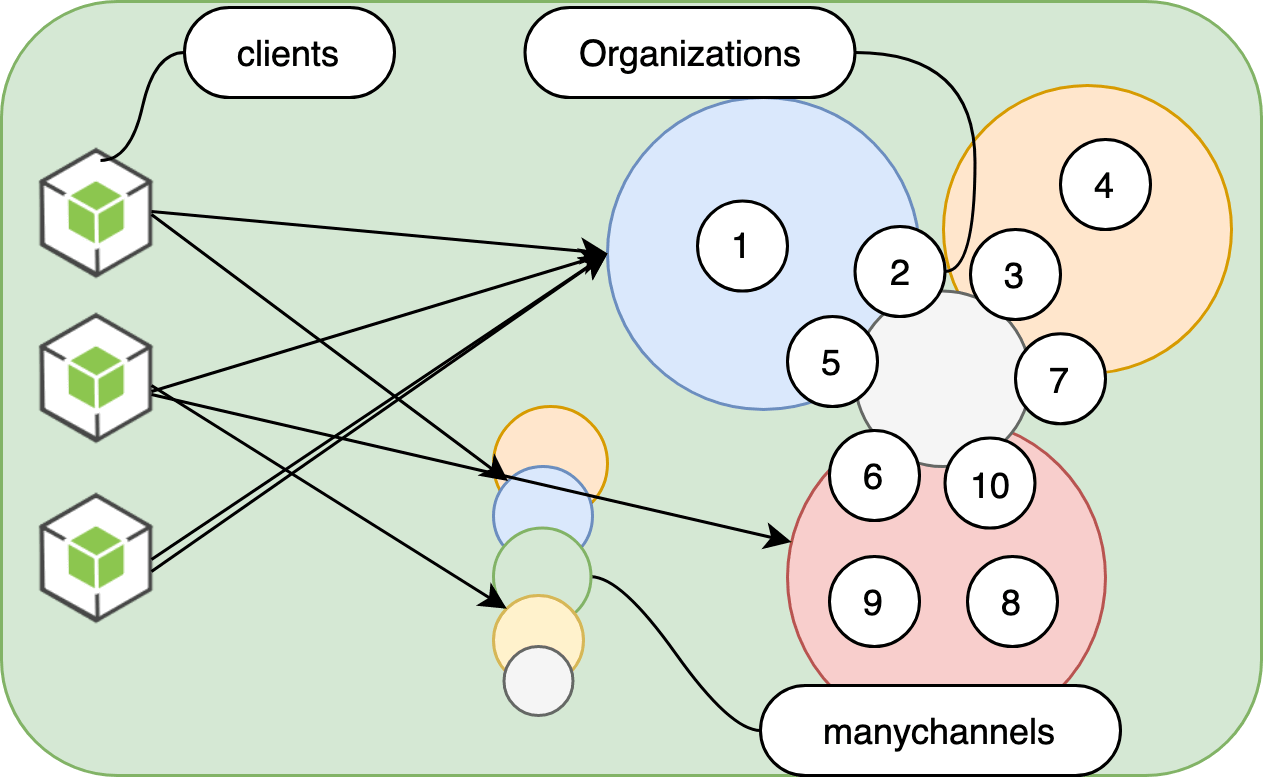
Once an organization joins a channel through peer’s aka nodes. Peers are able to sync all transactions in the channel ledger. In the fabric, channel means a separate ledger. Multiple channels lead to multiple ledgers and are maintained by the organization peers which are part of a particular channel. One of the popular statements regarding the data is data is a new oil and we have to keep our data safe and secure. By default, once organization peers join the channel it syncs all transactions within that channel ledger, which results in a major privacy breach. In order to understand this in a detailed manner then take this example.

Suppose there is an island in this world and this island is famous for trading. Businesspeople around the world started setting up their company by registering trade licenses with the government on this island. Soon it becomes crazy and the single government cannot afford to serve new company requests. Therefore, all new company registrations are in pending mode. So, the government started a decentralized way of registering which means instead of one entity serving all requests thereby forming a total of 10 semi-government bodies started serving new requests. The real problem comes into the picture. Decentralization in the concepts only but technically no single network where all semi governments exchange licenses with each other. All licenses issued by all 10 semi-government entities are exchanged with very difficult through web applications, APIs, etc. So the government started implementing a Hyperledger fabric consortium with 10 organizations. All organizations joined the tradelines channel and started transacting.

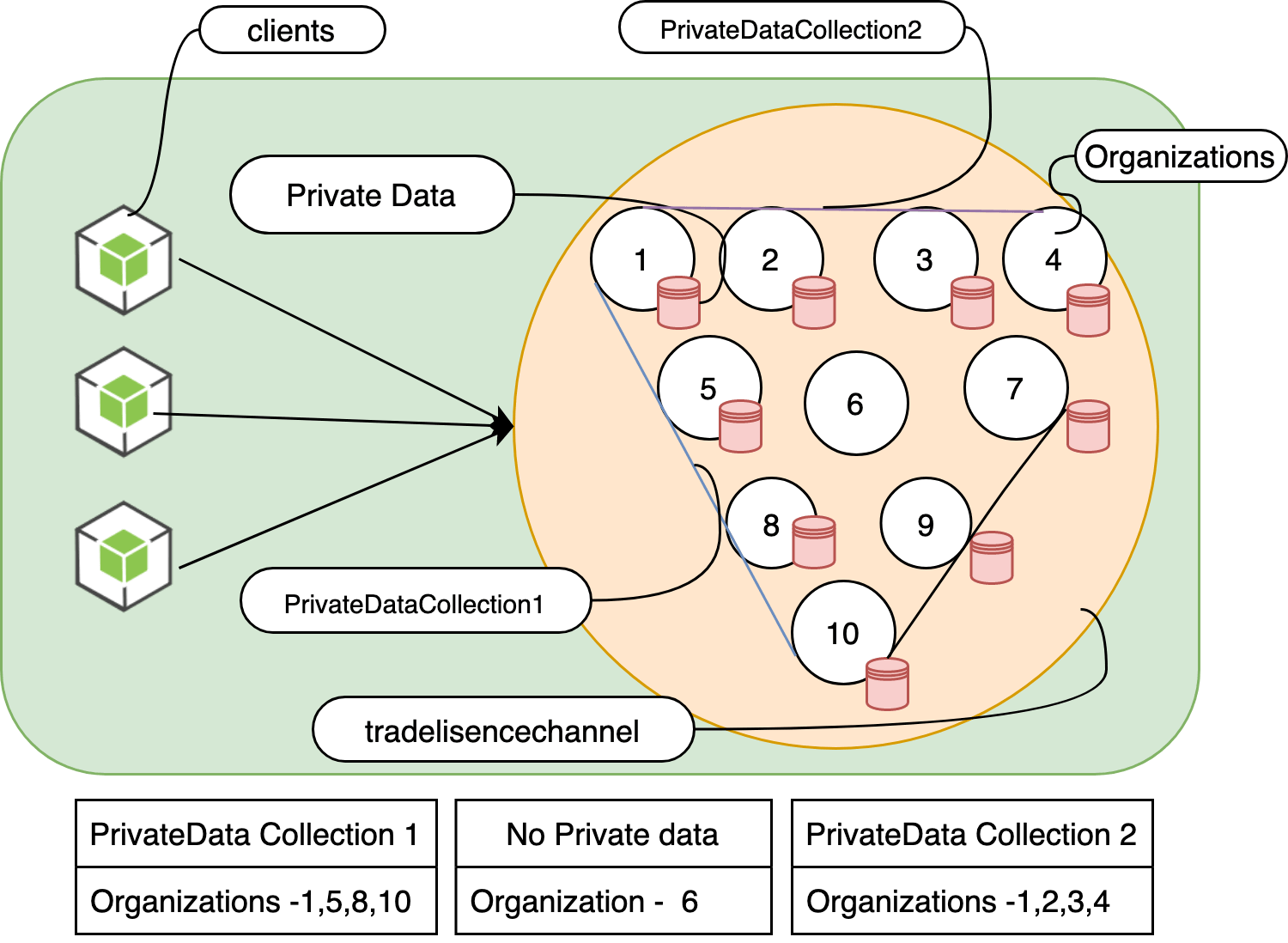
These 10 entities are offering individual companies with exciting offers in order to attract more business individuals. Since all 10 entities are on the same channel. By default, all transactions performed in the channel are exposed to channel organizations and each organization is knowing other organization offers, secrets, etc.



Soon after hyperledger fabric administrators decided to start creating multiple channels as below. We can clearly see the mess with multiple channels.



Soon Hyperledger fabric administrators understood the mess with having multiple channels and started implementing private data in a channel.



In the above screenshot. Private data collection-1 has organizations 1,5,8,10 decided to keep some data private, which means other organizations are not able to see the data but they keep hash of the data as proof. The private data sent to organizations present in the collection-1 through gossip protocol and this data is stored in a private database of authorized organizations. Ordering service is also unable to see private data. Hash of the data which is endorsed, ordered, and committed to the ledgers of every peer on the channel. The hash serves as evidence for auditing purposes.

### **Practical**:

STEP1: When we create an application channel. By default, application channel is unaware of Anchor peers, unlike system channel where anchor peers are learned through genesis block. So, we need to update anchor peers for all application channels. As anchor peers are for cross-organization communication purposes.

STEP2:

### **Create a collection definition JSON file**

A collection definition describes:

* who can persist data,
* how many peers the data is distributed to,
* how many peers are required to disseminate the private data, and
* how long the private data is persisted in the private database.

Chaincode APIs will map the collection to the private data by the collection name.

A collection definition is composed of the following five properties.

name: Name of the collection.

policy: Defines the organization peers allowed to persist the collection data.

requiredPeerCount: Number of peers required to disseminate the private data as a condition of the endorsement of the chaincode

maxPeerCount: For data redundancy purposes, the number of other peers that the current endorsing peer will attempt to distribute the data to. If an endorsing peer goes down, these other peers are available at commit time if there are requests to pull the private data.

blockToLive: For very sensitive information such as pricing or personal information, this value represents how long the data should live on the private database in terms of blocks. The data will be purged after this specified number of blocks on the private database. To keep private data indefinitely, that is, to never purge private data, set the blockToLive property to 0.

|  |
| --- |
| [  {  "name": "collection-public",  "policy": {  "identities": [  {  "role": {  "name": "member",  "mspId": "Org1MSP"  }  },  {  "role": {  "name": "member",  "mspId": "Org2MSP"  }  }  ],  "policy": {  "1-of": [  {  "signed-by": 0  },  {  "signed-by": 1  }  ]  }  },  "requiredPeerCount": 1,  "maxPeerCount": 2,  "blockToLive": 100  },  {  "name": "collection-private",  "policy": {  "identities": [  {  "role": {  "name": "member",  "mspId": "Org1MSP"  }  }  ],  "policy": {  "1-of": [  {  "signed-by": 0  }  ]  }  },  "requiredPeerCount": 0, #If 1 means Org1 should have two peers  "maxPeerCount": 1,  "blockToLive": 100 # 0 -> never purge private data  } ] |

Deploy the collection config and chaincode: Install and instantiate chaincode with a collection definition

Client applications or CLI interact with the blockchain ledger which means push data to the ledger or query some data from the ledger, through chaincode. As such we need to install and instantiate the chaincode on every peer that will execute and endorse transactions. When instantiated a chaincode on a channel the collection will be associated with that chaincode.

* Install chaincode. No specific parameter needed to support private data.
* Instantiate chaincode. To support private data, the request must include the collections-config attribute.

|  |
| --- |
| const collectionsConfigLoc = path.resolve(\_\_dirname, collection\_definition\_json\_filelocation); const request = {  chaincodeId: chaincodeId,  chaincodeType: chaincodeType,  chaincodeVersion: chaincodeVersion,  fcn: functionName,  args: args,  txId: tx\_id,  'collections-config': collectionsConfigLoc }; |

Once it is deployed. Ready to use by using the chaincode stub methods such as GetPrivate and PutPrivateData, bypassing the collection name. Below mentioned code snippets for Golang chaincode.

|  |
| --- |
| func (s \*ChaincodeStub) PutPrivateData(collection string, key string, value []byte) error  APIstub := shim.ChaincodeStubInterface  coll := "collection-public"  err := APIstub.PutPrivateData(coll, tradeKey, dataAsBytes)  if err != nil {  str := fmt.Sprintf("Problem while saving the information")  return shim.Error(str)  } |

|  |
| --- |
| func (s \*ChaincodeStub) GetPrivateData(collection string, key string) ([]byte, error)  APIstub := shim.ChaincodeStubInterface  coll := "collection-public"  objAsBytes, err := APIstub.GetPrivateData(coll, args[0])  if err != nil {  jsonRes := "{\"Error\":\"Failed to get for " + args[0] + "\"}"  return shim.Error(jsonRes)  } |

A beautiful article available [here](https://hyperledger.github.io/fabric-sdk-node/release-1.4/tutorial-private-data.html) from fabric-sdk-node

## Access Control Lists:

An ACL specifies which users or system processes are granted **access** to objects, as well as what operations are allowed on given objects. - **From wiki**

Similarly, Hyperledger fabric also has ACL. ACL present at the channel level, chaincode level, Organization level. It is a fact that Fabric is a very complex system and it is very difficult to define policies for each resource and each operation. So, fabric comes up with defaults which are very high level, and policies are associated with those defaults as default policies and they come with a chance to be overridden by the custom policies which we will learn in the near future.

The defaults are divided into three categories and each default associated policies are applied and we can override them.

* Application Defaults -> Default Policies
* Channel Defaults -> Default Policies
* Orderer Defaults -> Default Policies

**Policy**:

Policies are a beautiful creation. Policies allow the set of identities associated with a client request to be checked against the policy associated with the resource needed to fulfill the request. In a way to block unauthorized access. Policies can be structured in one of two ways: As Signature policies, As an ImplicitMeta policy.

Signature Policy:

As the name itself clearly tells us, this policy totally depends on the signature. When a particular resource is applied with a signature policy then that particular resource identifies specific clients with signing signatures and based on the policy definition it takes a decision whether grants access or block.

For example, Endorsement policies are used to determine whether a transaction has been appropriatly endorsed or not just by verifying the signatures against the policy.

|  |
| --- |
| Policies:  MyPolicy:  Type: Signature  Rule: "Org1.Peer OR Org2.Peer" |

Signature policies support arbitrary combinations of N Out Of, OR, AND. It also supports general statements such as the following “An admin of org ORGA and two other admins, or 2 of 4 org admins”.

ImplicitMeta Policies:

Implicit Meta policies are the aggregators which means these types of policies aggregate the result of policies. These policies are very clear with clear words. ANY, ALL, MAJORITY

So the following representation

|  |  |  |  |
| --- | --- | --- | --- |
| POLICY | TYPE-1 | TYPE-2 | TYPE-3 |
| ANY | ANY Admins | ANY Readers | ANY Writers |
| ALL | ALL Admins | ALL Readers | ALL Writers |
| MAJORITY | MAJORITY Admins | MAJORITY  Readers | MAJORITY  Writers |

**Writers**: Writers are able to propose the ledger updates

**Readers:** Readers are able to read the ledger

**Admins**: Admins are able to access or modify sensible information

|  |
| --- |
| Policies:  MyPolicy:  Type: ImplicitMeta  Rule: "ANY Admins" |

**Types of Policies**:

Policies defined at the application level are for example:

* Who can invoke chaincode on peers
* Who can listen to Block Events
* Policies for chaincode to chaincode invocation
* who can access lifecycle system chaincode functions such as ChaincodeExists GetDeploymentSpec GetChaincodeData
* who can access configuration system chaincode functions such as GetConfigBlock GetConfigTree SimulateConfigTreeUpdate
* who can access query system chaincode functions such as GetBlockByNumber GetChainInfo GetBlockByHash GetTransactionByID GetBlockByTxID

Policies defined at the channel level and Orderer level are for example:

1. Who can modify the configuration of the channel
2. who can claim as writers, readers, admins
3. What signatures must be included in the block

**ACL**:

ACLs are formatted as a key-value pair consisting of a resource function name followed by a value string. To see what this looks like. The conical paths of different entities are the following

|  |
| --- |
| For Channel := /Channel/<PolicyName>  For Orderer := /Channel/Orderer/<PolicyName>  For Application := /Channel/Application/<PolicyName>   /Channel/<Application|Orderer>/<OrgName>/<PolicyName> |

Let us see by combining these two topics: ACL and Policy. A sample reference brought from [configtx.yaml file](https://github.com/hyperledger/fabric/blob/release-2.0/sampleconfig/configtx.yaml). The below-mentioned ACL Policies are referenced in the configtx.yaml at &ACLsDefault

|  |
| --- |
| Application: &ApplicationDefaults   ACLs: &ACLsDefault  # ACL policy for qscc's "GetBlockByNumber" function  qscc/GetBlockByNumber: /Channel/Application/Readers   # ACL policy for invoking chaincodes on peer   peer/Propose: /Channel/Application/Writers   # ACL policy for qscc's "GetBlockByTxID" function  qscc/GetBlockByTxID: /Channel/Application/MyPolicy1   Policies: &ApplicationDefaultPolicies  LifecycleEndorsement:  Type: ImplicitMeta  Rule: "MAJORITY Endorsement"  Endorsement:  Type: ImplicitMeta  Rule: "MAJORITY Endorsement"  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  MyPolicy1:  Type: Signature  Rule: "OR('Org1.admin')" |

**Create New Policy & ACL**:

Creating a new policy is very easy just to create a new ACL and apply policy. Take an example. We want to use the function GetBlockByTxID only by a particular organization admin then the following Policy and ACL will make it for us.

|  |
| --- |
| Application: &ApplicationDefaults   Policies: &ApplicationDefaultPolicies  MyPolicy1:  Type: Signature  Rule: "OR('Org1.admin')"  ACLs: &ACLsDefault  <<: \*ACLsDefault  # ACL policy for qscc's "GetBlockByTxID" function  qscc/GetBlockByTxID: /Channel/Application/MyPolicy1 |

## **Raft Consensus**:

The raft is a protocol for implementing the distributed consensus. Raft consensus is one of the powerful distributed consensus algorithms. Let us start with a distributed consensus. Take a single instance of the database and this database will take a value and store it. A client which is node js application or java application or etc., which connects to the database server and sends value to it in order to update the value. Coming to the agreement and consensus in a single instance is not a difficult task. Suppose the database server is now in a cluster along with two other database servers. The difficult task is in front of us. How do we come to a consensus when database servers are in a cluster? This is called the problem of distributed consensus.

In Raft. A node can be in one of the following three states

* Leader state
* Follower state
* Candidate state

By default, all nodes start with the follower state. There is no leader so all followers will not be able to hear heartbeats from the leader. So, all followers raise a random timer which is between 150ms to 300ms. Whichever node timer timeouts, then it enters into a candidate state and conducts a leader election in order to become a leader. This is called the election term. The other nodes stop this process once it receives a vote request from other nodes. This process will be continuing if followers do not receive heartbeats from the leader. Candidate requests votes from other nodes. Nodes then reply with votes and the candidate becomes leader only if it gets sufficient votes from other nodes.

If a Nodejs application pushes a value change it will go to the leader node first and the leader node will write the change to something called log and this will not be committed. In order for the leader node to commit the log entry, it first replicates the changelog to his follower nodes and the leader waits until it receives a majority(quorum) of nodes written the entry to their changelogs and replies back as a vote. Once it receives a majority of votes then it commits to the data store and notifies the follower nodes that entry is committed. This is called the solution to the distributed consensus problem and for raft, it is log replication.

Please check out this awesome site <http://thesecretlivesofdata.com/raft/>

### **Hyperledger fabric Raft consensus**:

The raft is said to be “crash fault-tolerant” (CFT). In other words, if there are three nodes in a channel, it can withstand the loss of one node (leaving two remaining). If you have five nodes in a channel, you can lose two nodes (leaving three remaining nodes). Every channel runs on a **separate** instance of the Raft protocol, which means each channel has one separate raft protocol present and this allows each instance from each channel to elect a different leader. While all Raft nodes must be part of the system channel. System channel is the first channel in a consortium and this information is encoded in the genesis block and administrators are allowed to give a name else it will be testchainid.

Raft nodes do not necessarily have to be part of all application channels. Channel creators and channel admins have the ability to pick a subset of the available orderers and to add or remove ordering nodes as needed (as long as only a single node is added or removed at a time). This completely depends on the quorum.

**Quorum**. Describes the minimum number of consenters or nodes or orderers that need to affirm a proposal so that transactions can be ordered. For every consenter set, this is a **majority** of nodes. In a cluster with five nodes, three must be available for there to be a quorum. If a quorum of nodes is unavailable for any reason, the ordering service cluster becomes unavailable for both read and write operations on the channel, and no new logs can be committed.

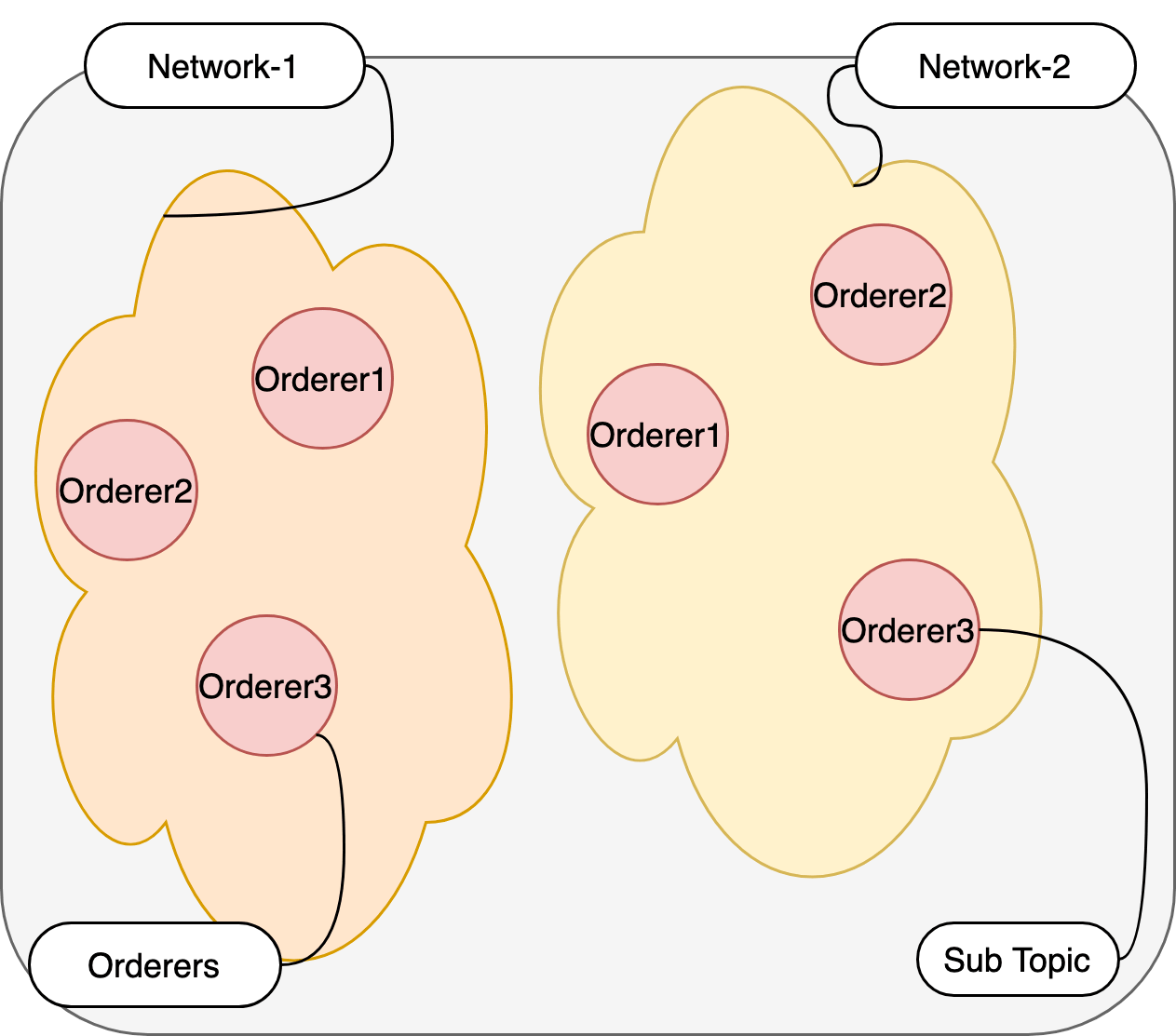
Calculate Quorum with below formula:



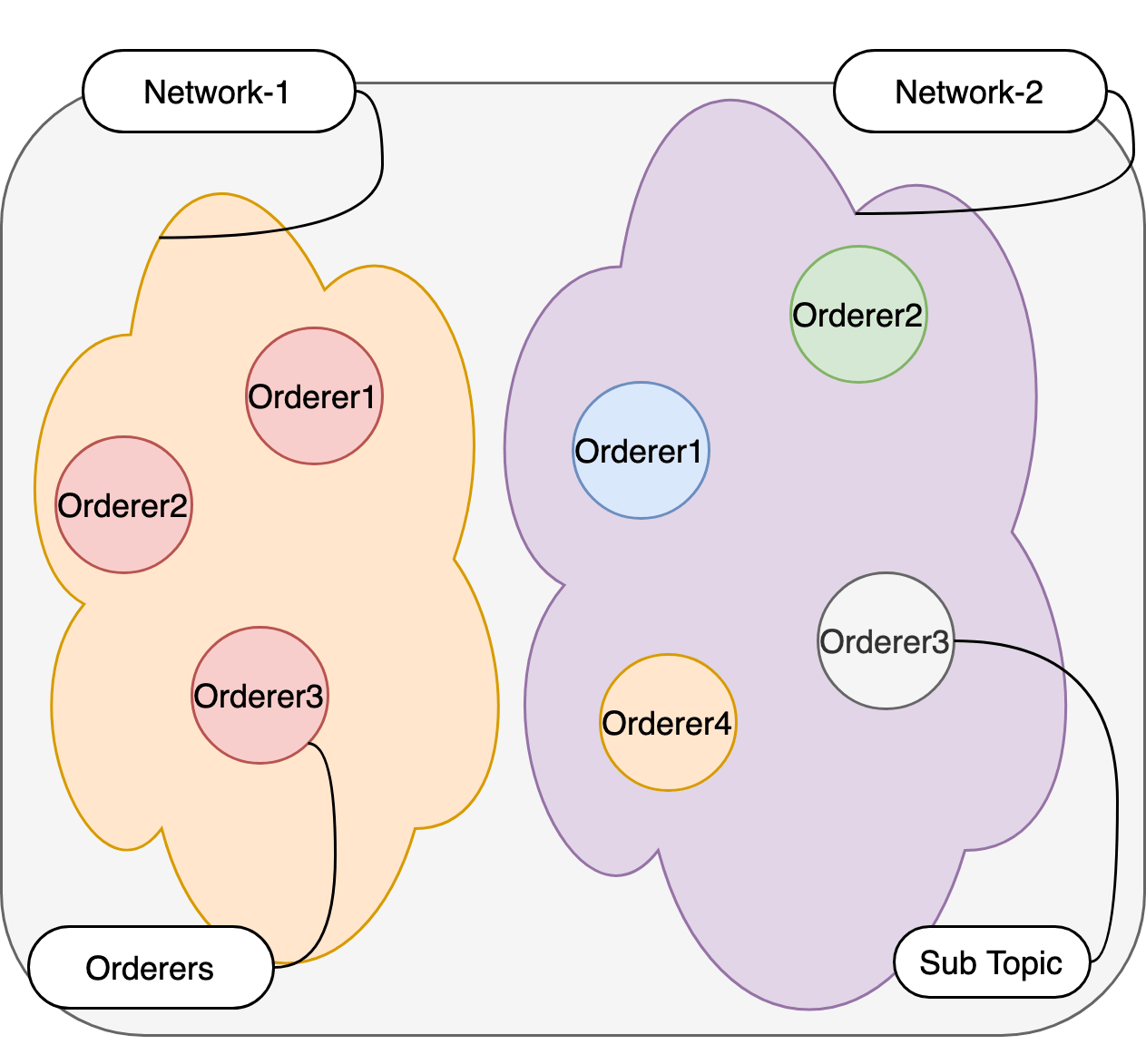
If a channel has 7 orderers then the quorum formula as below



The Quorum to stay alive is 4 and in the below image we can clearly understand that the network segment happens in such a way that each network has only 3 orderers and neither of the networks has passed the quorum and any transactions form clients which leads to failure state since a quorum is not met. There are fewer chances to meet quorum in case of network segmentation with even no of orderers.



The Quorum to stay alive is 4 and in the below image we can clearly understand that the network segment happens in such a way that one network has only 3 orderers and another network has four orderers and it meets quorum and any transactions form clients which leads to success state since it meets quorum. There are better chances to meet quorum in case of network segmentation with odd no of orderers .



If a channel has 5 orderers then Formula to calculate quorum for 5 orderers is as below



Using the main formula, we can generate a table

|  |  |  |
| --- | --- | --- |
| Orderers | Must Present | Fault Tolerance |
| 7 | 4 | 3 |
| 5 | 3 | 2 |
| 3 | 2 | 1 |
| 1 | 1 | 0 |

Do applications or peers know about the current leader of that channel?

In Raft, transactions (in the form of proposals or configuration updates) are automatically routed by the ordering node that receives the transaction to the current leader of that channel. This means that peers and applications do not need to know who the leader node is at any particular time. Only the ordering nodes need to know.

**Adding a new orderer to a Raft cluster is done by**:

1. Adding the TLS credentials of the new node to the channel through a channel configuration update transaction.
2. Fetching the latest config block of the system channel from an orderer node that’s part of the system channel.
3. Ensuring that the node that will be added is part of the system channel by checking that the config block that was fetched includes the certificate of the new orderer node that we are going to add soon.
4. Starting the new Raft node with the path to the config block in the General.BootstrapFile configuration parameter.
5. Waiting for the Raft node to replicate the blocks from existing nodes for all channels its certificates have been added to. After this step has been completed, the node begins servicing the channel.
6. Adding the endpoint of the newly added Raft node to the channel configuration of all channels.

**Note**: The new node must be added to the system channel before being added to one or more application channels.

Removing an orderer from a Raft cluster is done by:

1. Removing its endpoint from the channel config for all channels, including the system channel.
2. Removing its entry from the channel configuration for all channels including the system channel.
3. Shut down the node.

More information available [here](https://hyperledger-fabric.readthedocs.io/en/release-2.0/orderer/ordering_service.html#raft-concepts)

## **Tips on Errors:**

## Error: Failed disseminating 2 out of 2 private RWSets

|  |
| --- |
| Org2\_myorg-peer-myorg.1.05eunxk7gplx@devserver1 | 2018-10-29 10:15:08.814 UTC [gossip/gossip] learnAnchorPeers -> WARN 034 Got empty hostname, skipping connecting to anchor peer { 7051}  Org2\_myorg-peer-myorg.1.05eunxk7gplx@devserver1 | 2018-10-29 10:15:08.814 UTC [gossip/gossip] learnAnchorPeers -> WARN 035 Got invalid port (0), skipping connecting to anchor peer {myorgsea-peer-myorgsea 0}  Org2\_myorg-peer-myorg.1.05eunxk7gplx@devserver1 | 2018-10-29 10:15:08.814 UTC [gossip/gossip] learnAnchorPeers -> WARN 036 Got empty hostname, skipping connecting to anchor peer { 7051} |

**Reason**: we need to define Anchor Peers for cross Org Communication

**Note**: Anchor peers differ from leader peers. Make sure you update Anchor peers after joining peers to the channel.

### **Summary:**

Overall, in this chapter, we have learned a total of three miscellaneous interesting topics that are private data concepts, access control lists, and Raft consensus. By now we have understood the importance of private data and privacy in the channel. Policies and ACL are the keys to permissioned blockchains and by now we clearly understand the ACL and policies at the different levels. We clearly understood how the Raft is functioning. How Raft consensus is implemented in the fabric is exposed and by now we are confident about achieving quorum and formula to calculate quorum and fault tolerance.

In the upcoming chapter, we will explore the explanation and practical setup of the caliper benchmarking tool.

# Chapter12: Setup and Benchmark Blockchain Consortium Using Caliper

Caliper is a project from Hyperledger Umbrella and it is a blockchain performance benchmark framework or tool, which enables users to test different blockchain frameworks with a variety of use cases in order to get a set of performance benchmark results in different formats. Hyperledger Caliper will produce reports containing a number of performance indicators, such as TPS (Transactions Per Second), transaction latency, resource utilization, etc.

### Caliper Supported blockchain solutions are the following:

* [Hyperledger Fabric](https://github.com/hyperledger/fabric)
* [Hyperledger Besu](https://github.com/hyperledger/besu)
* [Hyperledger Burrow](https://github.com/hyperledger/burrow)
* [Hyperledger Sawtooth](https://github.com/hyperledger/sawtooth-core)
* [Ethereum](https://github.com/ethereum/)
* [FISCO BCOS](https://github.com/FISCO-BCOS/FISCO-BCOS)
* [Hyperledger Iroha](https://github.com/hyperledger/iroha)

### Caliper Supported performance metrics are the following:

* Transaction/read throughput
* Transaction/read latency (minimum, maximum, average, etc.)
* Resource consumption (CPU, Memory, Network IO, etc.)

Project Code Files URL: <https://github.com/narendranathreddythota/masteringhyperledgerfabric/tree/master/chapter-10>

## **Install and Setup Caliper**

Caliper is coming in multiple variants. The common binary which is present in all variants is caliper-CLI (command-line interface). The best way is installing using npm, as many developers are familiar with Nodejs and this installer is installing caliper-CLI globally. We can run caliper-cli from any project.

There are multiple ways to install caliper-cli as the follows:

* [Installing using NPM](https://hyperledger.github.io/caliper/vLatest/installing-caliper/#installing-from-npm)
  + local
  + global
* [Installing](https://hyperledger.github.io/caliper/vLatest/installing-caliper/#installing-from-npm) u[sing the Docker image](https://hyperledger.github.io/caliper/vLatest/installing-caliper/#using-the-docker-image)
* Installing locally from source

Let us install caliper-CLI globally using the node package manager

|  |
| --- |
| npm install -g --only=prod @hyperledger/caliper-cli |

Once we install the caliper-cli and we have to spoon-feed to the caliper about which platform we are targeting and which version of SDK to use by the caliper to interact with the targeted blockchain platform. This is called binding. The binding step is internally installing some dependencies, packages with appropriate settings, and this completely managed by the CLI. The bind command mentioned below.

|  |
| --- |
| caliper bind --caliper-bind-sut fabric --caliper-bind-sdk 1.0.0 --caliper-bind-args=-g |

Once we are ready with the benchmark configuration file, connection profile then we can start running caliper with the below command.

|  |
| --- |
| caliper benchmark run --caliper-benchconfig simple.yaml --caliper-networkconfig connection-profile.yaml |

## **Benchmark configuration file**

The benchmark configuration file is the director of the caliper and what describes in that configuration file caliper will just behave. Basically, it tells Caliper how many iterations it should execute, at what rate the TXs should be submitted, and which module will generate the TX content. It also includes settings about monitoring the platform. We can consider this file as the orchestrator of the benchmark engine. The beauty with configuration file is nearly generic and the settings are independent of the platform, so we can easily reuse them when performing testing with multiple benchmarks against different platforms or framework types or versions in the same platform.

Create a benchmark configuration file called simple.yaml

|  |
| --- |
| test:  clients:  type: local  number: 2  rounds:  - label: create a rate.  txDuration:  - 5  rateControl:  - type: fixed-backlog  opts:  unfinished\_per\_client: 5  callback: benchmark/createRate.js  - label: query a rate.  txDuration:  - 5  rateControl:  - type: fixed-backlog  opts:  unfinished\_per\_client: 5  arguments:  assets: 2  callback: benchmark/queryRate.js monitor:  type:  - docker  - process  docker:  name:  - all  process:  - command: node  arguments: fabricClientWorker.js  multiOutput: avg  interval: 1 |

The complete details of the key/values which we used in the above configuration is available at this location. We will have to mention invoke and query details under the round section. <https://hyperledger.github.io/caliper/vLatest/bench-config/>

## **Network configuration file**

Network connection profile is basically called a gateway to the fabric. We need to expose all details to the caliper in order to make benchmark tests smoothly. The content of the network configuration file is platform or framework specific. If we consider Hyperledger fabric then the connection file usually describes the topology of the consortium organization, where its peers and orderers and their endpoint addresses also, what users are present in the organization, and what chaincodes Caliper should deploy or interact with. We have seen three different use cases with different ways to create a consortium. In this chapter let us create a default consortium with Raft consensus and using docker swarm and we will utilize the default consortium in further chapters.

**STEP1:** Let's create a network configuration file. Create a file named congiftx.yml

|  |
| --- |
| Organizations:  - &default  Name: default  ID: defaultMSP  MSPDir: crypto-config/peerOrganizations/default/msp  AnchorPeers:  - Host: default-peer1-default  Port: 7051  Policies:  Readers:  Type: Signature  Rule: "OR('defaultMSP.member')"  Writers:  Type: Signature  Rule: "OR('defaultMSP.member')"  Admins:  Type: Signature  Rule: "OR('defaultMSP.admin')"  Capabilities:  Channel: &ChannelCapabilities  V1\_4\_3: true  V1\_3: false  V1\_1: false  Orderer: &OrdererCapabilities  V1\_4\_2: true  V1\_1: false  Application: &ApplicationCapabilities  V1\_4\_2: true  V1\_3: false  V1\_2: false  V1\_1: false Application: &ApplicationDefaults  Organizations:    Policies:  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  LifecycleEndorsement:  Type: ImplicitMeta  Rule: "MAJORITY Endorsement"  Endorsement:  Type: ImplicitMeta  Rule: "MAJORITY Endorsement"  Capabilities:  <<: \*ApplicationCapabilities Orderer: &OrdererDefaults  OrdererType: etcdraft  EtcdRaft:  Consenters:  - Host: default-orderer1  Port: 7050  ClientTLSCert: crypto-config/peerOrganizations/default/peers/peer0.default/tls/server.crt  ServerTLSCert: crypto-config/peerOrganizations/default/peers/peer0.default/tls/server.crt  - Host: default-orderer2  Port: 7050  ClientTLSCert: crypto-config/peerOrganizations/default/peers/peer3.default/tls/server.crt  ServerTLSCert: crypto-config/peerOrganizations/default/peers/peer3.default/tls/server.crt  - Host: default-orderer3  Port: 7050  ClientTLSCert: crypto-config/peerOrganizations/default/peers/peer4.default/tls/server.crt  ServerTLSCert: crypto-config/peerOrganizations/default/peers/peer4.default/tls/server.crt  Options:  TickInterval: 500ms  ElectionTick: 10  HeartbeatTick: 1  Addresses:  - default-orderer1:7050  - default-orderer2:7050  - default-orderer3:7050  BatchTimeout: 2s  BatchSize:  MaxMessageCount: 40  AbsoluteMaxBytes: 98 MB  PreferredMaxBytes: 4354 KB  Organizations:   Policies:  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  BlockValidation:  Type: ImplicitMeta  Rule: "ANY Writers" Channel: &ChannelDefaults  Policies:  Readers:  Type: ImplicitMeta  Rule: "ANY Readers"  Writers:  Type: ImplicitMeta  Rule: "ANY Writers"  Admins:  Type: ImplicitMeta  Rule: "MAJORITY Admins"  Capabilities:  <<: \*ChannelCapabilities Profiles:  defaultGroupOrdererGenesis:  <<: \*ChannelDefaults  Orderer:  <<: \*OrdererDefaults  Organizations:  - \*default  Capabilities:  <<: \*OrdererCapabilities  Application:  <<: \*ApplicationDefaults  Organizations:  - <<: \*default  Consortiums:  bankConsortium:  Organizations:  - \*default  defaultGroupChannel:  Consortium: bankConsortium  <<: \*ChannelDefaults  Application:  <<: \*ApplicationDefaults  Organizations:  - \*default |

**STEP2**: Let us create crypto-config file

|  |
| --- |
| PeerOrgs:  - Name: default  Domain: default  Template:  Count: 7  SANS:  - localhost  - default-peer1-default  - default-peer2-default  - default-orderer1  - default-orderer2  - default-orderer3  Users:  Count: 1 |

**STEP3**: Execute the below commands in order to get the artifacts

**Cryptogen**: A tool to pre generate the certificates for identities

**Configtxgen**: A tool to generate the initial configuration of the consortium

**FABRIC\_CFG\_PATH**: IN order to get the supplied files: path

|  |
| --- |
| > mkdir artifacts  > export FABRIC\_CFG\_PATH=${PWD}   > cryptogen generate --config=./cryptogen.yaml  > configtxgen -profile defaultGroupOrdererGenesis -outputBlock ./artifacts/genesis.block  > configtxgen -profile defaultGroupChannel -outputCreateChannelTx ./artifacts/defaultchannel.tx -channelID defaultchannel |

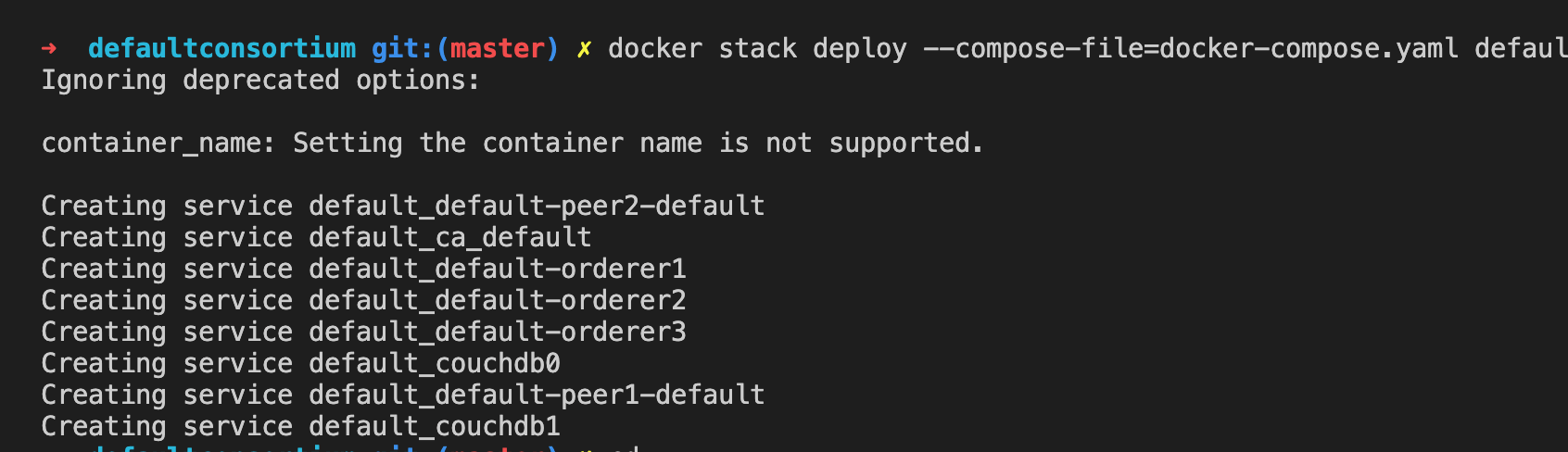
By this we have all our artifacts are ready except docker-compose file and we are skipping this step as we have seen enormous times in previous chapters and we can find the compose file in the repo

**Deploy Default Consortium**:

Once we are ready with configuration files such as channel.tx, genesis.block , certificates for all entities and docker compose file then, we are ready to deploy the Fabric consortium.

Clone the GitHub repo : <https://github.com/narendranathreddythota/masteringhyperledgerfabric>

|  |
| --- |
| > cd chapter-12 > hyperledger-fabric > defaultconsortium  > docker network create --attachable --driver overlay defaultnet  > docker stack deploy --compose-file=docker-compose.yaml default |



It's time to create a connection profile and this connection profile is almost similar to a general connection profile.

|  |
| --- |
| name: Fabric version: "1.0" mutual-tls: false  caliper:  blockchain: fabric  command:  start: sleep 3s  end: sleep 3s  info:  Version: 1.0  Size: 1 Org with 2 Peers  Orderer: Raft  Distribution: Single Host  StateDB: CouchDB  clients:  client0.default:  client:  organization: default  credentialStore:  path: /tmp/hfc-kvs/org1  cryptoStore:  path: /tmp/hfc-cvs/org1  clientPrivateKey:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/users/User1@default/msp/keystore/ad9b7f044971dd454ad3e799a5a0669083229fd72bf9dc3913a808710dba9458\_sk  clientSignedCert:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/users/User1@default/msp/signcerts/User1@default-cert.pem  channels:  defaultchannel:  created: false #Once it is created please change to true  configBinary: defaultconsortium/artifacts/defaultchannel.tx  orderers:  - default-orderer1  - default-orderer2  - default-orderer3  peers:  default-peer1-default:  eventSource: true  default-peer2-default:  eventSource: true  chaincodes:  - id: rate  version: v0  language: Golang   path: chaincode/rate  endorsement-policy:  identities:  - role:  name: member  mspId: defaultMSP  policy:  1-of:  - signed-by: 0  organizations:  default:  mspid: defaultMSP  peers:  - default-peer1-default  - default-peer2-default  certificateAuthorities:  - ca-default  adminPrivateKey:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/users/Admin@default/msp/keystore/31e43baedca2f39b7917b8f2db52f2e33f49f3630ffee671256f13f45a800d78\_sk  signedCert:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/users/Admin@default/msp/signcerts/Admin@default-cert.pem  orderers:  default-orderer1:  url: grpcs://localhost:7050  grpcOptions:  ssl-target-name-override: default-orderer1  grpc-max-send-message-length: 15  tlsCACerts:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/peers/peer0.default/tls/ca.crt  default-orderer2:  url: grpcs://localhost:8050  grpcOptions:  ssl-target-name-override: default-orderer2  grpc-max-send-message-length: 15  tlsCACerts:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/peers/peer3.default/tls/ca.crt  default-orderer3:  url: grpcs://localhost:9050  grpcOptions:  ssl-target-name-override: default-orderer3  grpc-max-send-message-length: 15  tlsCACerts:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/peers/peer4.default/tls/ca.crt  peers:  default-peer1-default:  url: grpcs://localhost:7051  grpcOptions:  ssl-target-name-override: default-peer1-default  grpc-max-send-message-length: -1  grpc.keepalive\_time\_ms: 600000  grpc.http2.min\_time\_between\_pings\_ms: 120000  grpc.http2.max\_pings\_without\_data: 0  grpc.keepalive\_permit\_without\_calls: 1  tlsCACerts:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/peers/peer1.default/tls/ca.crt  default-peer2-default:  url: grpcs://localhost:8051  grpcOptions:  ssl-target-name-override: default-peer2-default  grpc-max-send-message-length: -1  grpc.keepalive\_time\_ms: 600000  grpc.http2.min\_time\_between\_pings\_ms: 120000  grpc.http2.max\_pings\_without\_data: 0  grpc.keepalive\_permit\_without\_calls: 1  tlsCACerts:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/peers/peer2.default/tls/ca.crt  certificateAuthorities:  ca-default:  url: https://localhost:7054  httpOptions:  verify: false  tlsCACerts:  path: defaultconsortium/artifacts/crypto-config/peerOrganizations/default/ca/ca.default-cert.pem  registrar:  - enrollId: defaultadmin  enrollSecret: defaultpw |

## **Write A Sample Chaincode**

Let's make the chaincode with a simple logic. The concept is creating rates with an ability to create deals and prices associated with that deal.

|  |
| --- |
| type Rate struct {  Deal string `json:"deal"`  Price string `json:"price"` } |

Let’s, deep dive into chaincode development. Before going further, an important point needs to be highlighted here. There is a global object present in the hyperledger fabric chaincode ecosystem which is called shim object. Package shim provides rich APIs for the chaincode to access chaincode state variables, transaction context and chaincode could be able to call other chaincodes. Below repo mentioned a detailed shim object.

|  |
| --- |
| *https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim* |

We can import shim global objects by using go imports.

|  |
| --- |
| import ( "bytes" "encoding/json" "fmt" "github.com/hyperledger/fabric/core/chaincode/shim" sc "github.com/hyperledger/fabric/protos/peer"  ) |

Golang is a statically typed language and it is a compiled language. Like java, Golang also needs a main function. As the execution starts from here. While we create a new smart contract struct, we need to make sure that all methods from the interface such as Invoke and Init are present. Since the start function in the shim package accepts the Chaincode interface. Let us see the chaincode interface.

|  |
| --- |
| type Chaincode interface {  // Init is called during Instantiate transaction after the chaincode container  // has been established for the first time, allowing the chaincode to  // initialize its internal data  Init(stub ChaincodeStubInterface) pb.Response   // Invoke is called to update or query the ledger in a proposal transaction.  // Updated state variables are not committed to the ledger until the  // transaction is committed.  Invoke(stub ChaincodeStubInterface) pb.Response } |

|  |
| --- |
| type SmartContract struct {}  func main() {  // Create a new Smart Contract  err := shim.Start(new(SmartContract))  if err != nil {  fmt.Printf("Error creating new Smart Contract: %s", err)  } } |

Let us first write the Init and Invoke functions

|  |
| --- |
| func (s \*SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response {  return shim.Success(nil) } // we are not auto populating any data through init func   func (s \*SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {  function, args := APIstub.GetFunctionAndParameters()  if function == "queryRate" {  return s.queryRate(APIstub, args)  } else if function == "createRate" {  return s.createRate(APIstub, args)  }  return shim.Error("Invalid Smart Contract function name.") } |

Let us proceed with the business methods such as **createRate queryRate**

|  |
| --- |
| func (s \*SmartContract) createRate(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {  if len(args) != 3 {  return shim.Error("Incorrect number of arguments. Expecting 3")  }  var rate = Rate{Deal: args[1], Price: args[2]}  carAsBytes, \_ := json.Marshal(rate)  APIstub.PutState(args[0], carAsBytes)  fmt.Println("new Rate has been created")  return shim.Success(nil) } |

|  |
| --- |
| func (s \*SmartContract) queryRate(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {   if len(args) != 1 {  return shim.Error("Incorrect number of arguments. Expecting 1")  }   carAsBytes, \_ := APIstub.GetState(args[0])  return shim.Success(carAsBytes) } |

## **Benchmark scripts**

We need to write a total of two scripts to create and query. Caliper will take these two scripts as a base to create multiple transactions to test the load and to give a complete report. We will write our scripts in JavaScript.

Let us create createrate.js file

|  |
| --- |
| 'use strict';  module.exports.info = 'Creating rates.';  let bc, contx;  module.exports.init = function(blockchain, context, args) {  bc = blockchain;  contx = context;   return Promise.resolve(); };  module.exports.run = function() {  let args = {  chaincodeFunction: 'createRate',  chaincodeArguments: ["1","dealType", "dealPrice"]  };  return bc.invokeSmartContract(contx, 'rate', 'v0', args, 3); };  module.exports.end = function() {  return Promise.resolve(); }; |

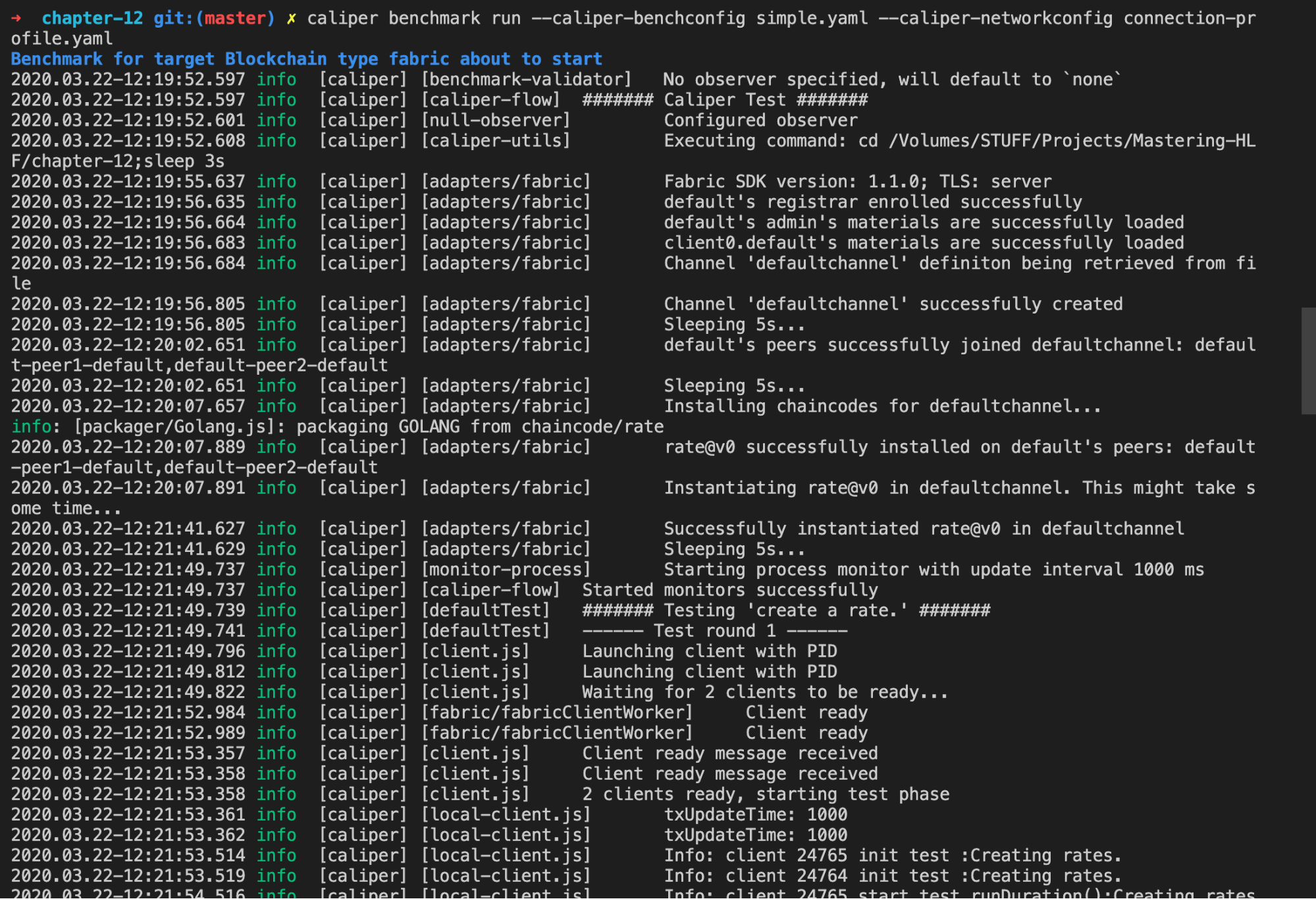
Let us create queryRate.js file

|  |
| --- |
| 'use strict'; module.exports.info = 'Querying a rates.'; let txIndex = 0; let limitIndex, bc, contx;  module.exports.init = async function(blockchain, context, args) {  bc = blockchain;  contx = context;  limitIndex = args.assets;  return Promise.resolve(); };  module.exports.run = function() {  txIndex++;  let carNumber = 'Client' + contx.clientIdx + '\_CAR' + txIndex.toString();  let args = {  chaincodeFunction: 'queryRate',  chaincodeArguments: ["1"]  };  if (txIndex === limitIndex) {  txIndex = 0;  }  return bc.bcObj.querySmartContract(contx, 'rate', 'v0', args, 3); }; module.exports.end = function() {  return Promise.resolve(); }; |

## **Run the total setup:**

Running benchmarks with the blow command. caliper benchmark run. We are running benchmarks for Hyperledger fabric. The prerequisites such as creating channel, joining peers, installing chaincode, instantiating chaincode to the channel and invoke, query will be taken care by the caliper and at the end we will have a report. This is a detailed report that contains a summary of total benchmark operation, resource consumption, performance metrics. A detailed report will be generated called report.html and caliper.log

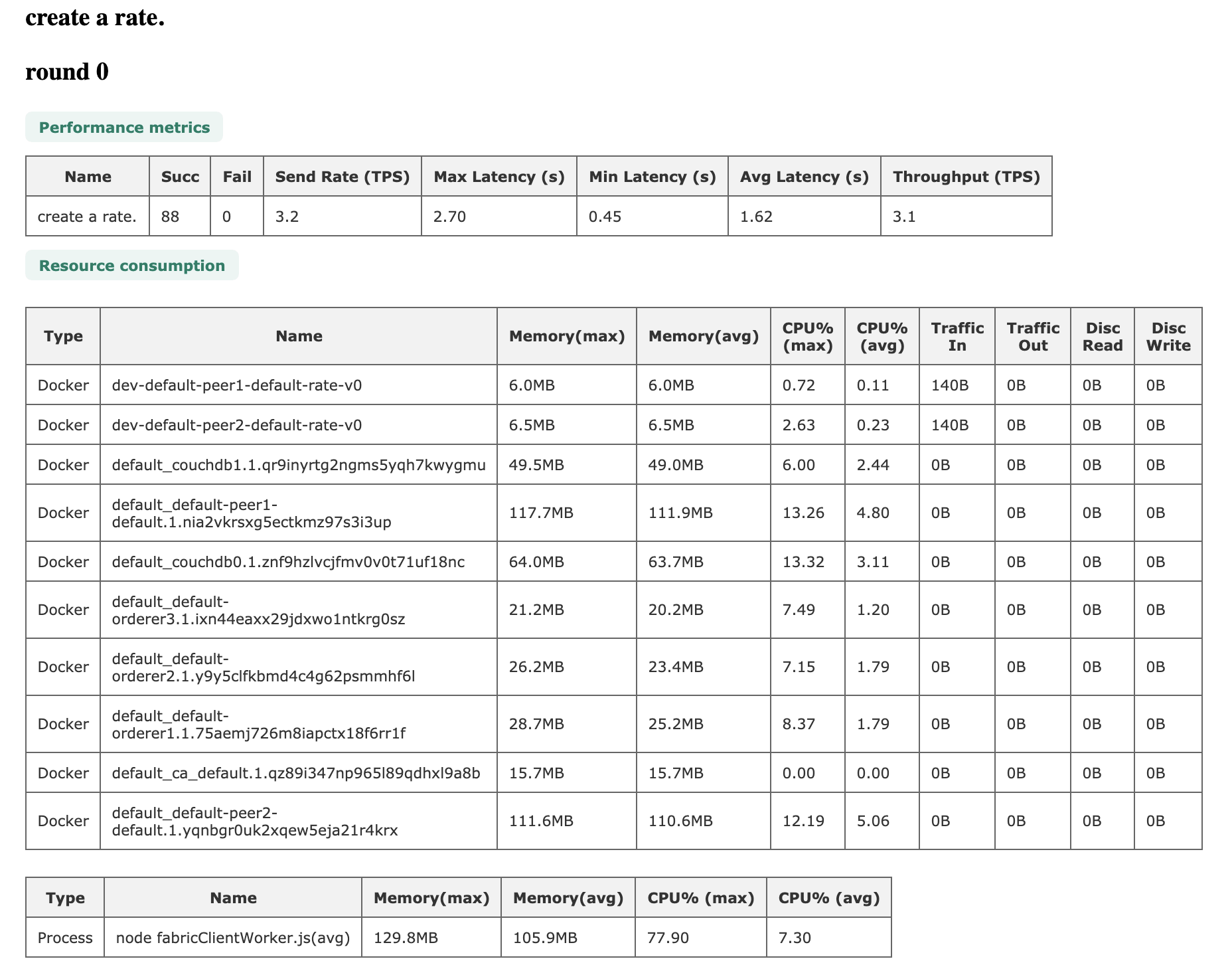
|  |
| --- |
| ➜ chapter-12 git:(master) ✗ caliper benchmark run --caliper-benchconfig simple.yaml --caliper-networkconfig connection-profile.yaml Benchmark for target Blockchain type fabric about to start 2020.03.22-12:19:52.597 info [caliper] [benchmark-validator] No observer specified, will default to `none` 2020.03.22-12:19:52.597 info [caliper] [caliper-flow] ####### Caliper Test ####### |



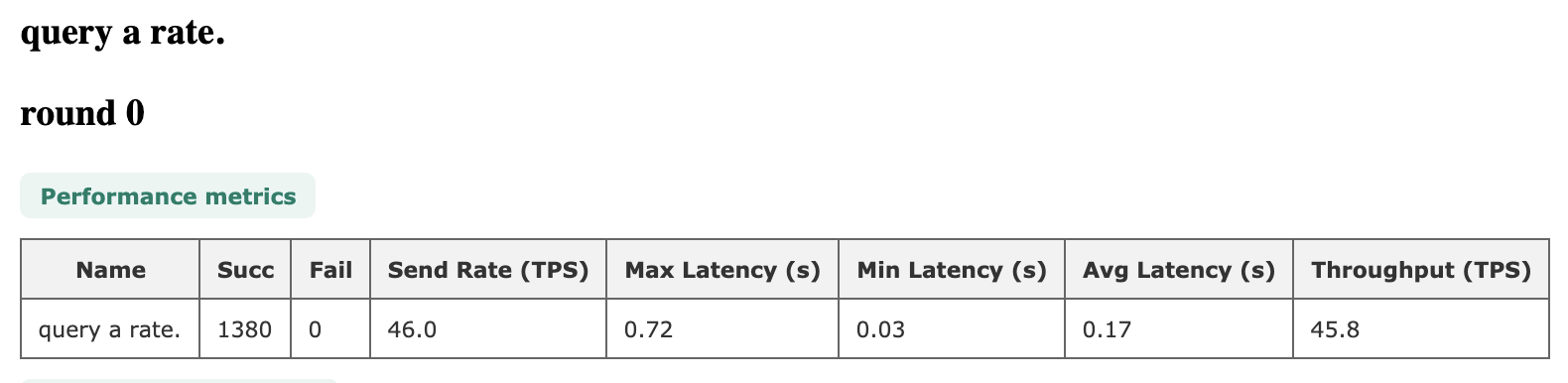
## **Report Summary:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Succ** | **Fail** | **Send Rate (TPS)** | **Max Latency (s)** | **Min Latency (s)** | **Avg Latency (s)** | **Throughput (TPS)** |
| create a rate. | 88 | 0 | 3.2 | 2.70 | 0.45 | 1.62 | 3.1 |
| query a rate. | 1380 | 0 | 46.0 | 0.72 | 0.03 | 0.17 | 45.8 |

A detailed report of resource consumption and performance metrics during the invocation by the caliper will be provided in the HTML version of the report.



With the above screenshot, we can clearly understand that the createRate has 88 success transactions and 0 failed transactions which resulted in 3.2TPS. For query, this is completely different as mentioned in the below screenshot. Query rate has 1,380 success transactions and 46 TPS. The result is very promising, which is good with one VM or node



### **Summary:**

In this chapter, we have learned the complete technical details of Hyperledger caliper. Technical Details include benchmark scripts, connection profile to the consortium itself, and a sample chaincode. We have also learned that Hyperledger caliper exposes us to complete benchmarks and server metrics that are enough to estimate the performance of a Hyperledger fabric consortium.

In the next chapter, we will learn more details about monitoring with Prometheus and Grafana.

# 

# Chapter13: Monitoring Consortium with Prometheus and Grafana

Most of the IT gigantic projects are enabled monitoring. Monitoring is very important. We need to understand what is happening at the internal level. Sometimes the server is running out of memory, resources are not sufficient, etc. Due to many reasons, monitoring is important for big products. Hyperledger fabric also opened doors for monitoring through operational HTTP servers. These servers expose us to different metrics such as consensus, chaincode, CouchDB, etcd, ledger, process, etc.

Hyperledger Fabric is giving us an opportunity to monitor all deployments such as orderer, peer just by enabling the operations server. Operations service is an HTTP server that offers Restful operations

The API exposes the following mentioned capabilities:

* Health checks
* Log level management
* StatsD or Prometheus target for operational metrics (when configured)

Syntax to configure Orderer and Peer Operation service is mentioned below:

|  |
| --- |
| Operations:  ListenAddress: 127.0.0.1:8443  TLS:  Enabled: true  PrivateKey: tls/server.key  Certificate: tls/server.crt  ClientRootCAs: []  ClientAuthRequired: false |

We need to set two environment variables in the orderer compose file **ORDERER\_OPERATIONS\_LISTENADDRESS** and **ORDERER\_METRICS\_PROVIDER**

|  |
| --- |
| #For Orderer environment:  - ORDERER\_OPERATIONS\_LISTENADDRESS= 127.0.0.1:9446 #Try 0.0.0.0 in case any issues  - ORDERER\_METRICS\_PROVIDER=prometheus ports:  - 9446:9446 |

|  |
| --- |
| #For Peer environment:  - CORE\_OPERATIONS\_LISTENADDRESS= 127.0.0.1:8444 #Try 0.0.0.0  - CORE\_METRICS\_PROVIDER=prometheus  ports:  - 9446:9446 |

Once we configure the operations service and we deploy we can access the exposed endpoints as below

|  |
| --- |
| > curl localhost:9444/healthz > {"status":"OK","time":"2020-03-22T14:59:37.839397Z"}  > curl localhost:9444/logspec > {"spec":"info"} |

It's time to write compose files for Prometheus and grafana. Let us extend the previous chapter default consortium.

|  |
| --- |
| prometheus:  image: prom/prometheus  volumes:  - ./artifacts/prometheus.yml:/etc/prometheus/prometheus.yml  ports:  - 9090:9090  deploy:  replicas: 1  restart\_policy:  condition: on-failure  delay: 10s  max\_attempts: 15  networks:  - defaultnet   grafana:  image: grafana/grafana  ports:  - 3000:3000  deploy:  replicas: 1  restart\_policy:  condition: on-failure  delay: 10s  max\_attempts: 15  networks:  - defaultnet |

Prometheus Configuration mentioned below. We have mentioned service endpoints so that Prometheus can collect metrics from the services and Grafana can query Prometheus.

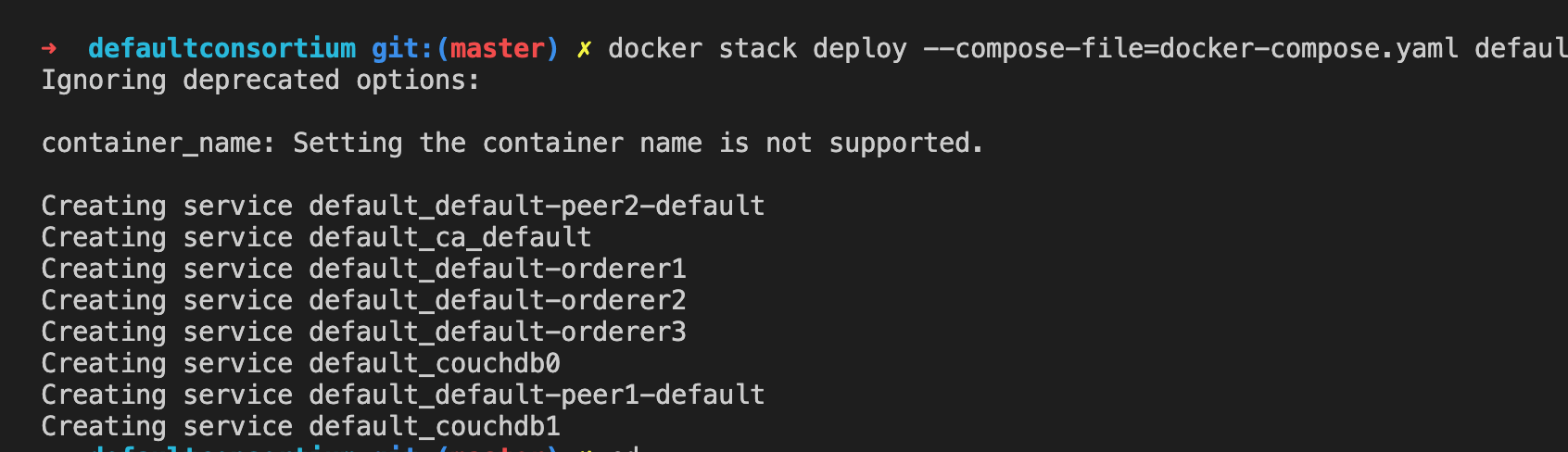
|  |
| --- |
| global:  scrape\_interval: 15s  evaluation\_interval: 15s scrape\_configs:  - job\_name: 'prometheus'  scrape\_interval: 10s  static\_configs:  - targets: ['localhost:9090']  - job\_name: 'hyperledger\_metrics'  scrape\_interval: 10s  static\_configs:  - targets: ['default-orderer1:9444', 'default-orderer2:9445', 'default-orderer3:9446', 'default-peer1-default:8444', 'default-peer2-default:8445'] |

## **Deploy Default Consortium**:

We can use the existing default consortium with the default organization. Deploy the consortium using docker swarm and Grafana and Prometheus is already configured.

GitHub: <https://github.com/narendranathreddythota/masteringhyperledgerfabric>

|  |
| --- |
| > cd chapter-12 > hyperledger-fabric > defaultconsortium  > docker stack deploy --compose-file=docker-compose.yaml default |



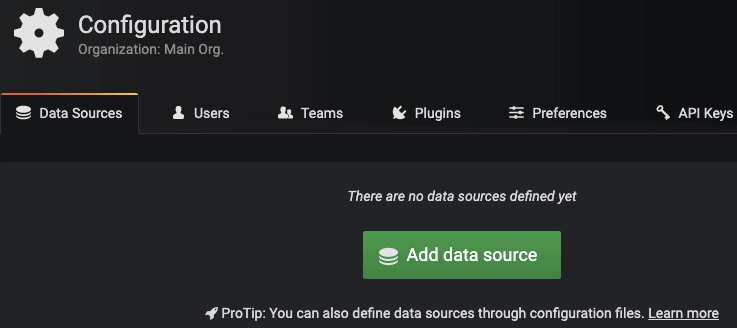
Below a screenshot of Prometheus: Prometheus is live at 9090 ports

|  |
| --- |
|  |

No, let us open Grafana and Grafana will live on 3000 port. Access with username as admin and password is admin



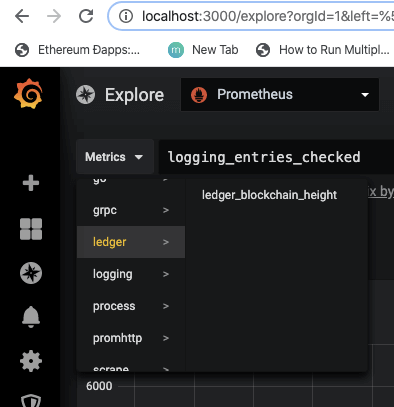
Once we login to the grafana. We have to configure it a little bit. We have to add a data source, in our case Prometheus is the data source. Below screenshots are helpful.



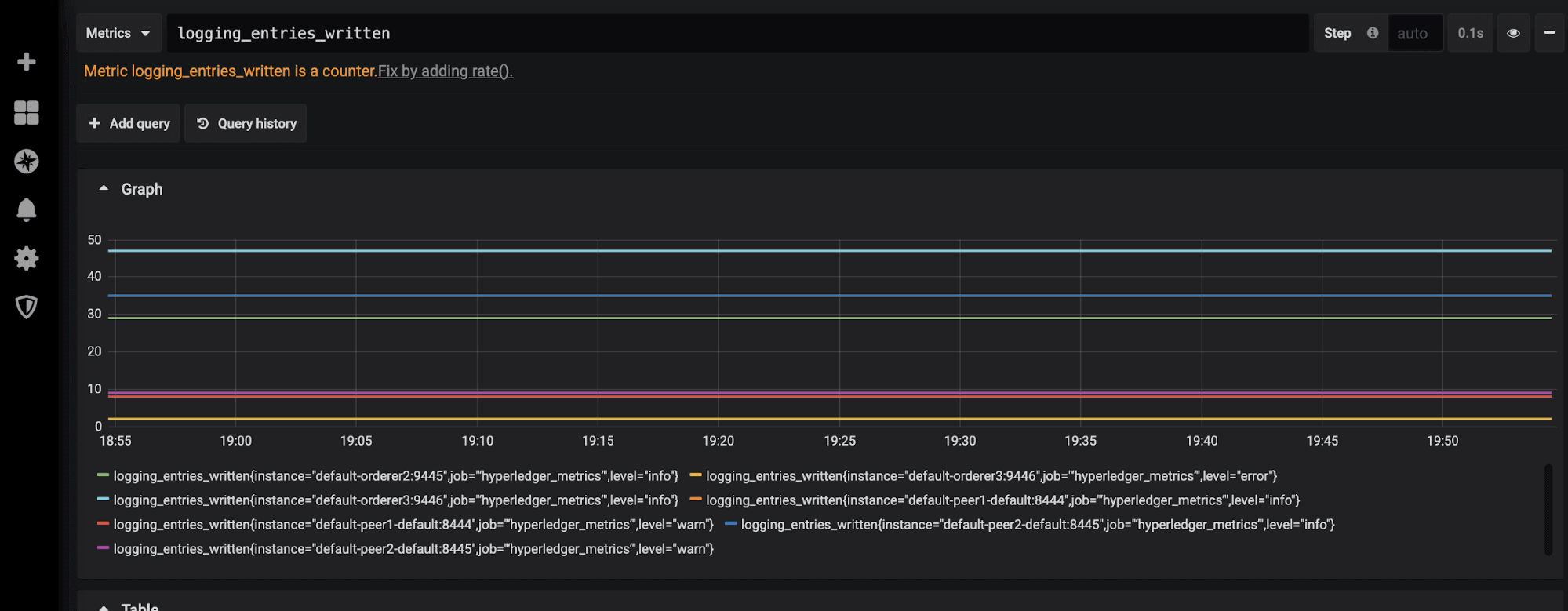
Once we have configured successfully then we can see it as an added source list in the Data Sources. By default it becomes default and we can change this at any time.



We can see the available metrics from Hyperledger fabric services -> Prometheus -> Grafana



We can monitor the complete consortium with the Prometheus and grafana with different dashboards and just by switching different metrics.



## **Summary**:

In this chapter, we have discussed briefly about server metrics and its importance in order to protect any system from entering into a disaster phase. This chapter depicts the importance of server metrics management in any applications in order to early warn the incoming disaster. We have also discussed the practical explanation of metrics with Prometheus and Grafana.

In the next chapter, we will discuss log management and the ELK stack setup.

# Chapter14: Logging Consortium with ELK Stack

It started with a log file. Logfile is used to collect events of a system and messages between systems. One can understand the detailed behavior of the system just by accessing the logs of that system. A log file can be used for monitoring of system behavior, emergency recovery aka immediate system failure recovery, to track the system flow in different components. Logs are event logs, transaction logs, message logs, etc. The log is the core fundamental part of any system. It is a best friend to us. It helps us in different situations. “Your logs have answers”

Logs in any application is a crucial component for debugging. “This code block was developed long back ago and it is working fine till today morning and it is not working now” what to do now? A common sentence from most of the developers when logging is not part of their application or not logging properly. Proper implementation of the logs is very important. Hyperledger Fabric implemented it well. There is a popular standard [RFC 5424](https://tools.ietf.org/html/rfc5424) on how to log from applications.

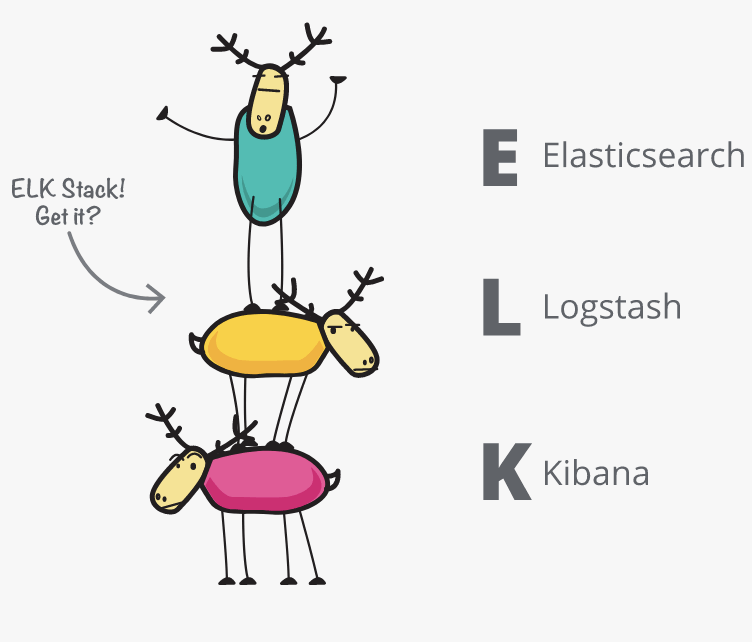
The main part here is the severity of logs as listed below:

* Emergency: the system is unusable
* Alert: action must be taken immediately
* Critical: critical conditions
* Error: error
* Warning: warning
* Notice: normal but significant
* Informational: informational
* Debug: debug-level messages

from the above standard. If we created a channel successfully then it is info. If anything happens and the channel is not created successfully then it should be Error. The beautify of fabric is we can specify which type of logs we can see from the components. For Orderer we can control this by adding the following environment variable **ORDERER\_GENERAL\_LOGLEVEL** and the values are

**FATAL | PANIC | ERROR | WARNING | INFO | DEBUG**

When we set up a consortium with at least three organizations with 7 orderers, 6 peers, 3 CA’s and business applications. The list will be increasing dramatically and finally, we will end up messing the things with different applications that produce different logs. There are some popular tools that help us to monitor logs with nice GUI and some analytical engines to search logs. One of the popular stacks for logging is ELK. ELK is the acronym for three open-source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server‑side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch.



From elastic.co

## **Setup ELK Stack**:

|  |
| --- |
|  |

ELK stack gives us the ability to analyze any dataset. Logstash has normalization capabilities. Elastic search has aggregation capabilities and Kibana has visualization by combining all powers we called our superheroes as ELK. We will use this <https://github.com/deviantony/docker-elk> for reference

We can deploy ELK using docker-compose and docker swarm. Let us use docker-swarm to deploy ELK.

STEP1: Create a docker-compose file called logging.yml

|  |
| --- |
| version: '3.3' services:  elasticsearch:  image: docker.elastic.co/elasticsearch/elasticsearch:7.6.0  ports:  - "9200:9200"  - "9300:9300"  configs:  - source: elastic\_config  target: /usr/share/elasticsearch/config/elasticsearch.yml  environment:  ES\_JAVA\_OPTS: "-Xmx256m -Xms256m"  ELASTIC\_PASSWORD: changeme  discovery.type: single-node  networks:  - elk  deploy:  mode: replicated  replicas: 1 |

|  |
| --- |
| version: '3.3'  logstash:  image: docker.elastic.co/logstash/logstash:7.6.0  ports:  - "9500:9500"  - "9600:9600"  configs:  - source: logstash\_config  target: /usr/share/logstash/config/logstash.yml  - source: logstash\_pipeline  target: /usr/share/logstash/pipeline/logstash.conf  environment:  LS\_JAVA\_OPTS: "-Xmx256m -Xms256m"  networks:  - elk  deploy:  mode: replicated  replicas: 1 |

|  |
| --- |
| kibana:  image: docker.elastic.co/kibana/kibana:7.6.0  ports:  - "5601:5601"  configs:  - source: kibana\_config  target: /usr/share/kibana/config/kibana.yml  networks:  - elk  deploy:  mode: replicated  replicas: 1 |

Now we have to create configs now instead of creating volumes just for configuration files:

|  |
| --- |
| configs:  elastic\_config:  file: ./config/elasticsearch.yaml  logstash\_config:  file: ./config/logstash.yaml  logstash\_pipeline:  file: ./config/logstash.conf  kibana\_config:  file: ./config/kibana.yaml |

Create overlay network

|  |
| --- |
| networks:  elk:  driver: overlay |

Create a folder called config

|  |
| --- |
| > mkdir config  Cd config |

**STEP2**: create configuration files

Create elasticsearch.yaml

|  |
| --- |
| cluster.name: "docker-cluster" network.host: 0.0.0.0 xpack.license.self\_generated.type: trial xpack.security.enabled: true xpack.monitoring.collection.enabled: true |

Create logstash.yaml

|  |
| --- |
| http.host: "0.0.0.0" xpack.monitoring.elasticsearch.hosts: [ "http://elasticsearch:9200" ] xpack.monitoring.enabled: true xpack.monitoring.elasticsearch.username: elastic xpack.monitoring.elasticsearch.password: changeme |

Create kibana.yaml

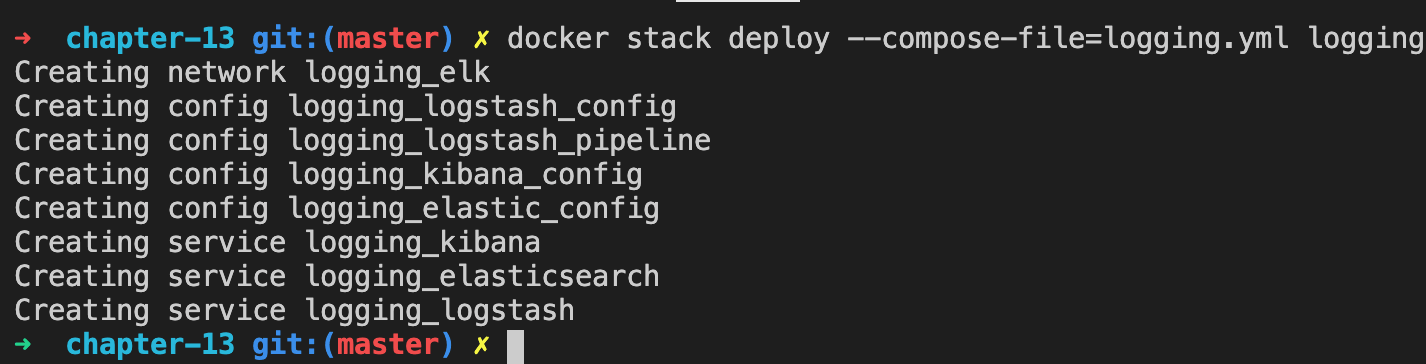
|  |
| --- |
| server.name: kibana server.host: "0" elasticsearch.hosts: [ "http://elasticsearch:9200" ] xpack.monitoring.ui.container.elasticsearch.enabled: true elasticsearch.username: elastic elasticsearch.password: changeme |

Create logstash.conf

|  |
| --- |
| input {  tcp {  port => 9500  type => syslog  }  udp {  port => 9500  codec => json  type => dockerlog  } } output {  elasticsearch {  hosts => "elasticsearch:9200"  user => "elastic"  password => "changeme"  } } |

## **Deploy and Test**:

|  |
| --- |
| > cd chapter-13  > docker stack deploy --compose-file=docker-compose.yaml logging |

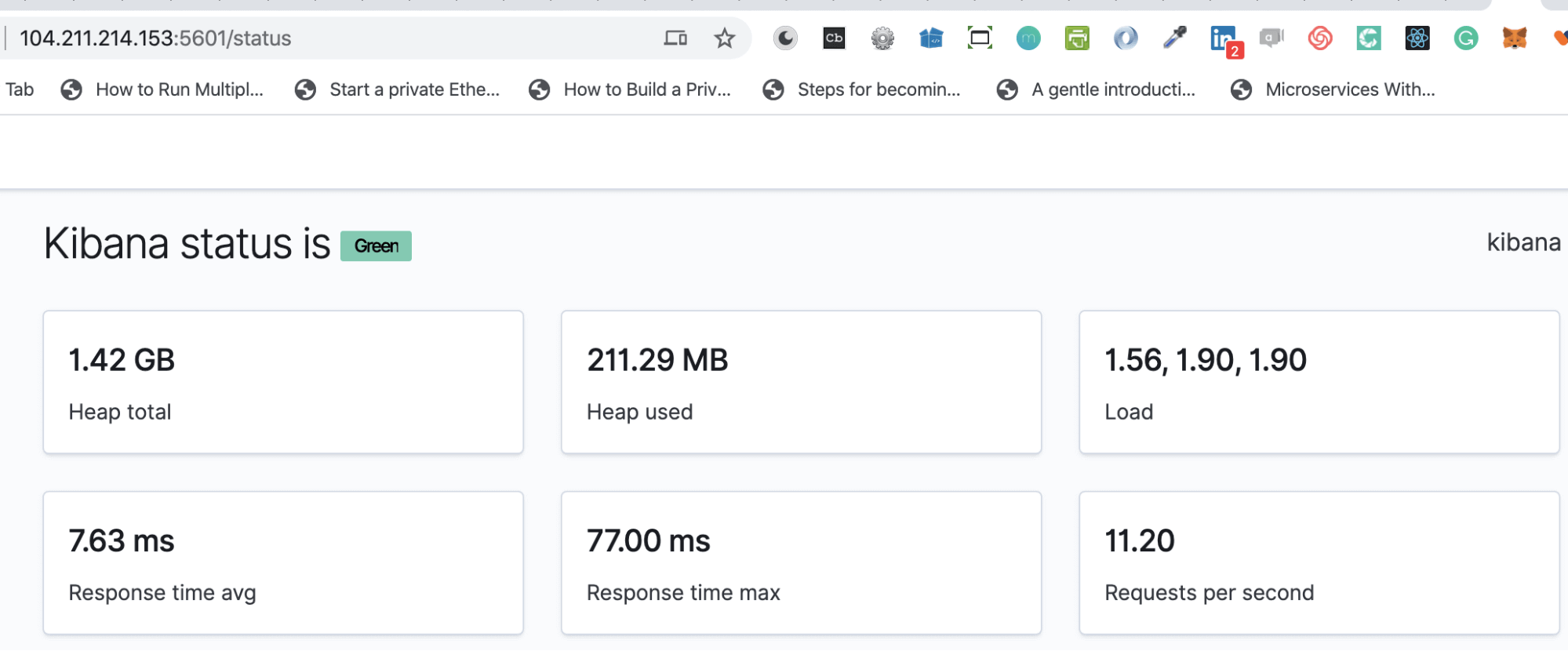


|  |
| --- |
| > Kibana will be live at http:localhost:5601 > ElasticSearch will be live at http:localhost:9200 > Logstash will be live at 9600 |

Once we deployed successfully Elastic search live at 9200 port.

|  |
| --- |
|  |

We can check status of the Kibana by localhost:5601/status

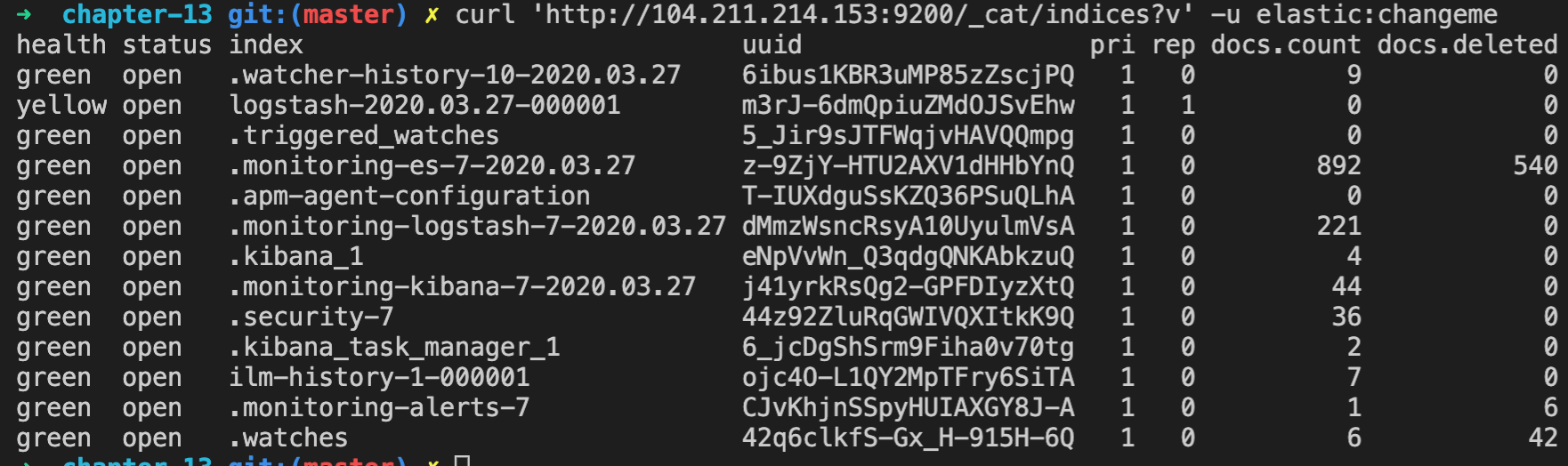


Create an index pattern via the Kibana API:

|  |
| --- |
| $ curl -XPOST -D- 'http://localhost:5601/api/saved\_objects/index-pattern' \  -H 'Content-Type: application/json' \  -H 'kbn-version: 7.6.0' \  -u elastic:changeme \  -d '{"attributes":{"title":"logstash-\*","timeFieldName":"@timestamp"}}' |

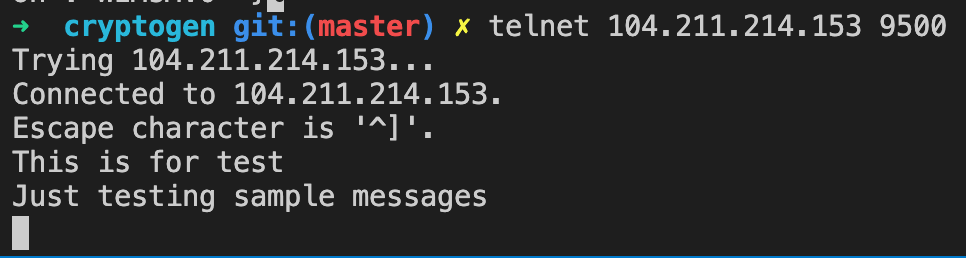
**To See Available indices:**

|  |
| --- |
| curl 'localhost:9200/\_cat/indices?v' -u elastic:changeme |



Testing the connection of ELK:

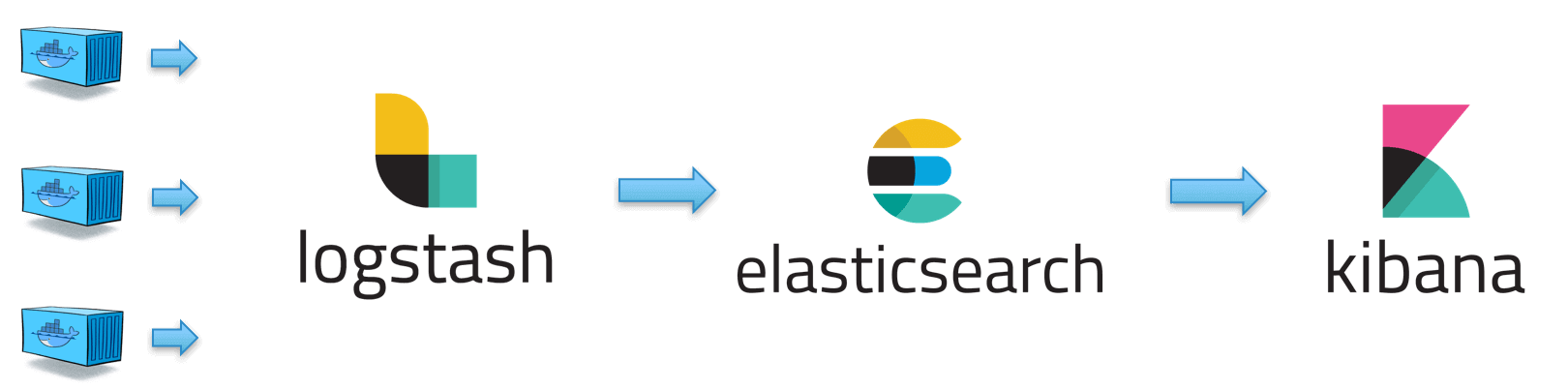
Send some TCP messages using telnet



We can see that our messages are being populated to the kibana. This means that our setup of ELK is successfully completed.

|  |
| --- |
|  |

## Shipping docker data into dockerized ELK stack:



We have created ELK stack and tested with sample images and it is time to collect docker logs and ship to Logstash. There are many ways to ship the logs to Logstash. We can use logspout. We have already seen logspout in chapter 8 “End to End with Solo consensus using docker with one use case”, that was a basic logspout. We can now see the customized logspout. This logspout collects logs from all docker containers and ships to logspout.

Create a docker-compose file called logspout.yml

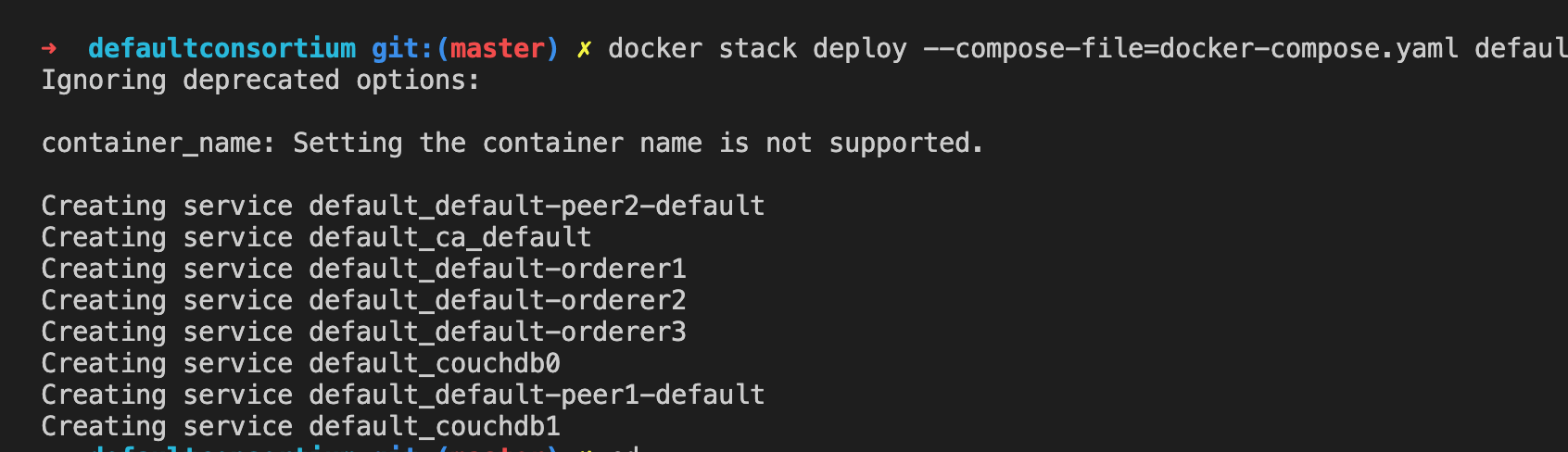
|  |
| --- |
| version: '3.3' services:  logspout:  image: narendranathreddy/logspout  volumes:  - /var/run/docker.sock:/var/run/docker.sock:ro  environment:  ROUTE\_URIS: logstash://logstash:9500  LOGSTASH\_TAGS: docker-elk  networks:  - elk  restart: on-failure networks:  elk:  driver: overlay |

**Deploy Logspout:**

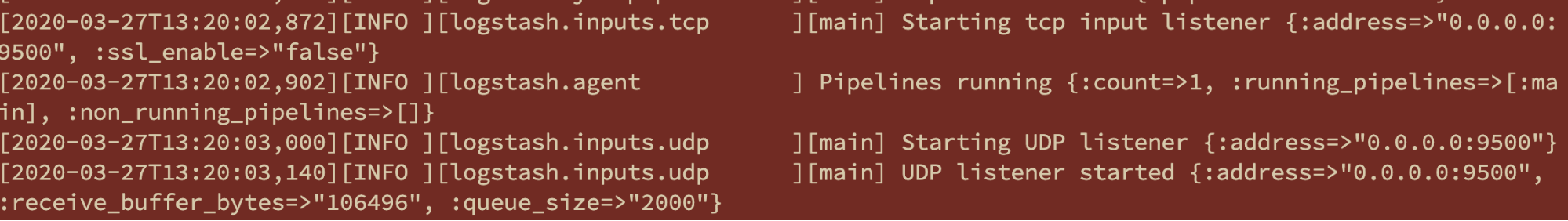
|  |
| --- |
| > cd chapter-13   > docker stack deploy --compose-file=logspout.yaml logging |

It is time to deploy our sample consortium:

|  |
| --- |
| > cd chapter-12 > hyperledger-fabric > defaultconsortium  > docker stack deploy --compose-file=docker-compose.yaml default |



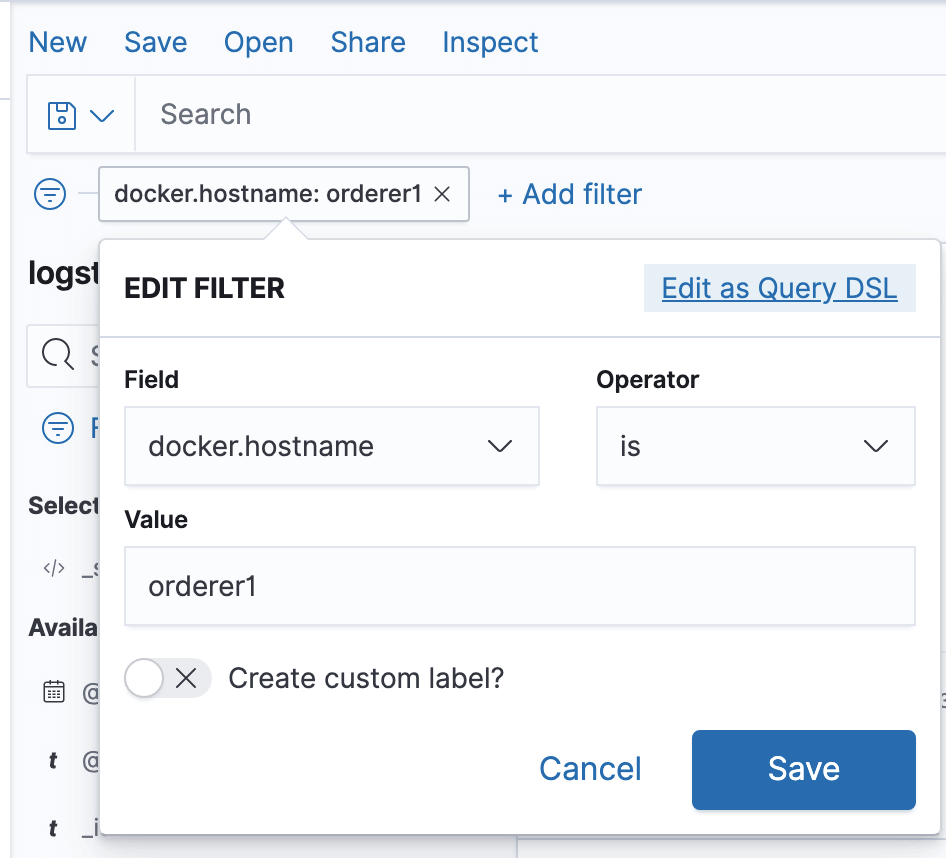
We can see logstash is getting some data from the logspout in the below screenshot



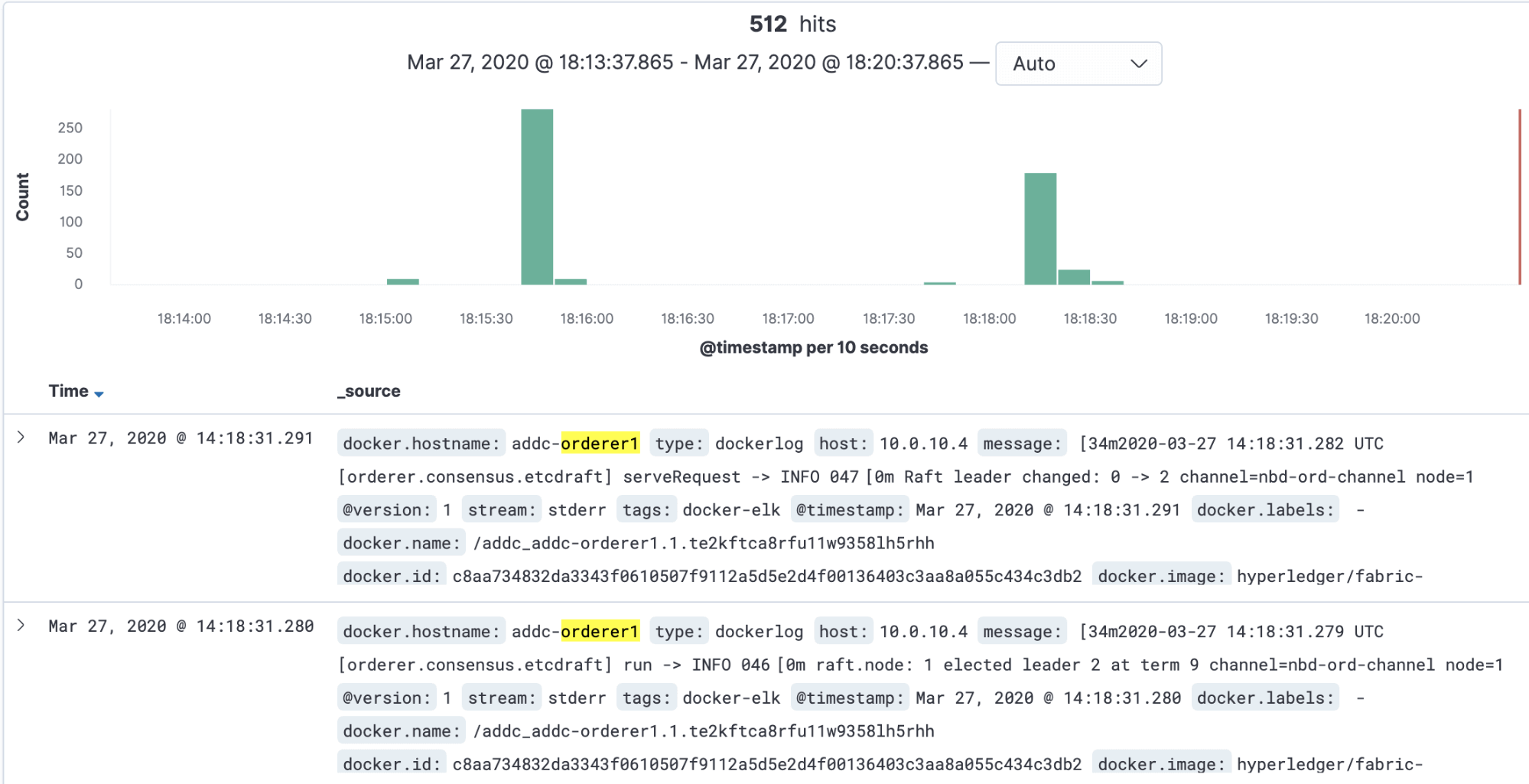
Login To Kibana with username is elastic and password is changeme and we can observe the filer options are updated with docker fields.



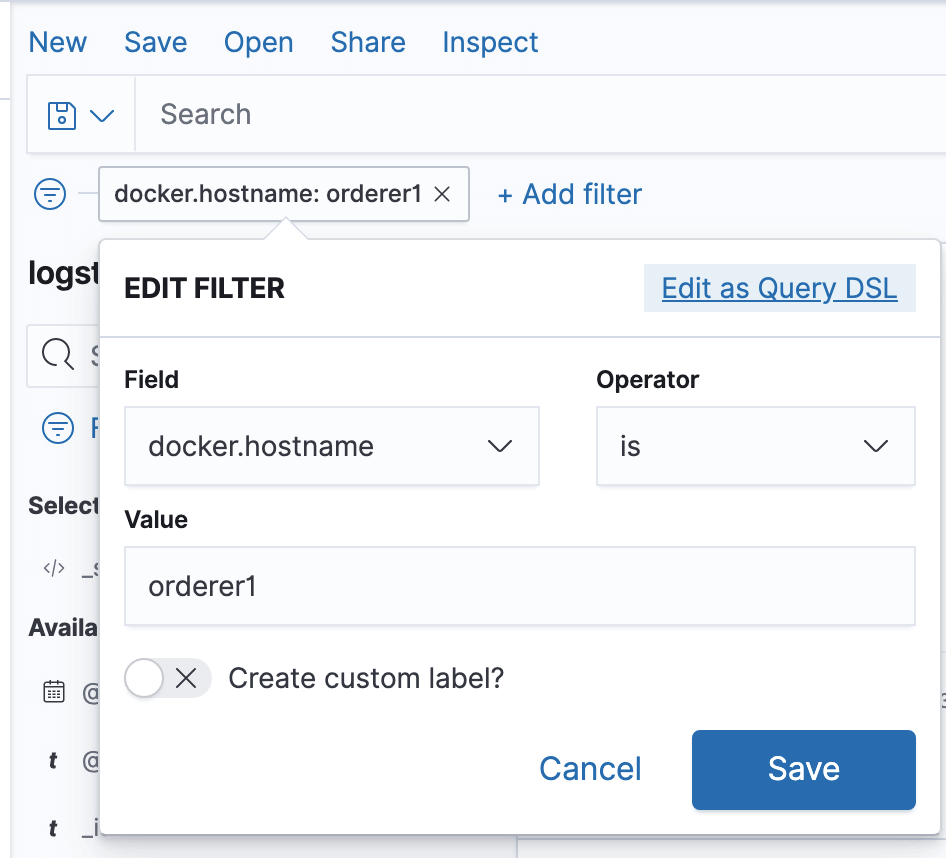
We can search particular docker logs as mentioned in the below screenshot

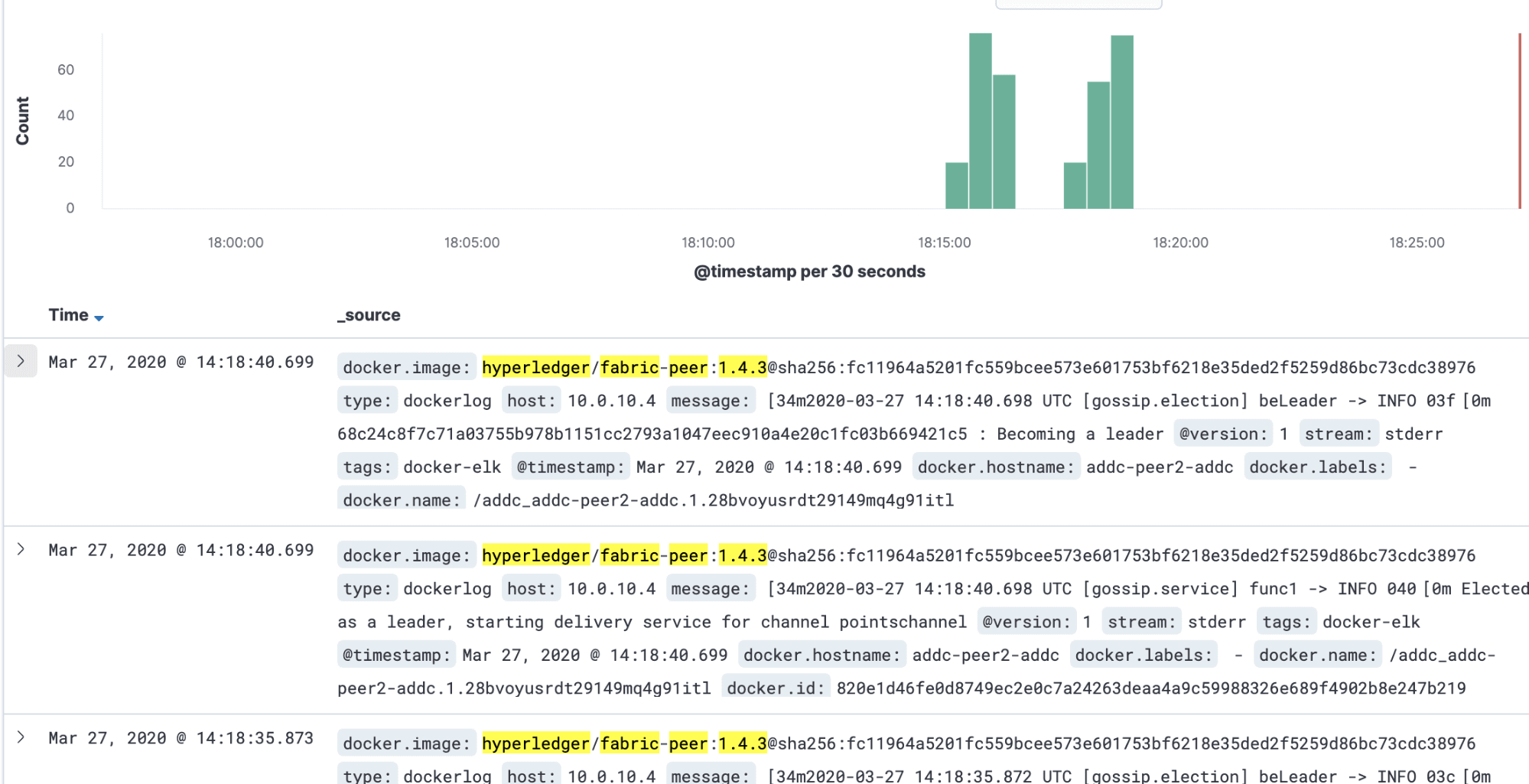


Once we apply the filter then we can see the logs of a particular orderer. Elastic search will do all needful things to us in order to ship logs to kibana all the way from a particular orderer dockerized container.



We can search partially. When we do not know full details. The complete image is hyperledger/fabric-peer:1.4.3@sha256:fc11964a5201fc559bcee573e601753bf6218e35ded2f5259d86bc73cdc38976 but we know only hyperledger/fabric-peer:1.4.3





## **Summary**:

In this chapter, we have discussed briefly about log files and its importance during disaster scenarios, application outage scenarios, etc. This chapter depicts the importance of log management in any application. We have also explored ELK stack and different ways to feed docker logs into Logstash.

In the next chapter, we will explore some details about Hyperledger Fabric v2.0

# 

# Chapter15: Glimpse of Hyperledger fabric 2.0

Generally, open-source projects like Kubernetes, docker, and of course Hyperledger fabric have new versions in their pipeline and new releases happen very frequently. These open-source projects have contributors across the world. If we observe carefully the changelog new releases have covered some advancements and some existing functionality bug fixes. If we observe Hyperledger fabric first release and present the latest release a lot of functionality changes and huge features, many bug fixes. It's been almost 4 years ever since it first released on 16 Sept 2016. The latest release is v2.0.1.

|  |
| --- |
|  |

# Do we really need to discuss 2.0?

Of course, yes fabric 2.0 has a lot of functionality and flow of execution changes. It will be a token if we have knowledge of v2.0. Let us just list the major changes

* External chaincode launcher
* Ledger data format upgrade
* Docker images with Alpine Linux
* State database cache for CouchDB
* Logger removed from chaincode shim
* Go version has been updated to 1.13.4
* The 'Solo' consensus type is deprecated
* Client Identity (CID) library has moved
* Chaincode built upon installation on peer
* Decentralized smart contract governance
* The 'Kafka' consensus type is deprecated
* Policies must be specified in configtx.yaml
* System Chaincode Plugins have been removed
* The ccenv build image no longer includes the shim
* Support for peer's Admin service has been removed
* Warn when certificates are about to expire (Upgraded)
* GetHistoryForKey returns result from newest to oldest
* New application patterns and private data enhancements
* Bash not available in Docker images with Alpine Linux
* New configuration for orderer genesismethod and genesisfile
* Orderer FileLedger location moved if specified with relative path
* The go chaincode entities extension has been removed(encryption)
* Support for configtxgen flag --outputAnchorPeersUpdate is deprecated
* Support for invoking system chaincodes from user chaincodes has been removed.
* Support for specifying orderer endpoints at the global level in channel configuration is deprecated
* The 'provisional' genesis method of generating the system channel for orderers has been removed.

That was really a big list of changes. A big learning curve is present if we see all changes carefully. Existing projects will break if we are not doing upgrades properly. It is better to keep all projects running in 1.4.X version as it is. Do no upgrade as major functionality changes are ahead in V2.0. New projects have the ability to go with v2.0. But as of April 1st 2020. SDKs are not supporting v2.0 changes, so keep in mind. Let us discuss some of the functionality changes in detail:

## **Smart Contracts Decentralized Governance**

When we have a consortium with 5 organizations with one channel and if anyone organization installs a chaincode and instantiates a channel then other organizations must install the chaincode and adhere to chaincode parameters. This leads to several problems between organizations within the channel. Fabric V2.0 comes with an idea of decentralized governance within the chaincode which means the majority of the organizations come to an agreement about chaincode parameters such as endorsement policy before it instantiates to the channel. This lifecycle flow also applies to the upgrade functionality. The majority of the organizations need to approve the upgrade logic. Moreover, we can upgrade the private data collection or policy configuration without reinstalling the chaincode.

## **Private Data Enhancements**

By default, implicit per-organization collection is present. We can deploy chaincode without defining collection for a particular organization. This gives us the ability to share private data across collections where each collection has more than one organization/member. This gives us the ability to avoid sharing private data with a particular collection that contains multiple members. Collection-level endorsement policies that override chaincode endorsement policies.

## **External Chaincode Launcher**

This is the major functionality change with chaincodes. Now, no need to peer to access the docker daemon to start/build chaincode containers. Plus, you need not necessarily run chaincode in a docker container. Fabric2.0 comes with the ability to run chain codes on our own choice of environment. We can run chaincode as service. So, the takeaways are listed below.

* Eliminate Docker daemon dependency
* Alternatives to containers
* External builder executables
* Chaincode as an external service

## **Alpine-Based Docker Images**

The Hyperledger fabric images are going to be lightweight with alpine based docker images feature. We know that alpine based docker images are secure and lightweight Linux distribution and it increases the performance dramatically.

## **Upgrading to Fabric v2.0**

Every new release fabric gives us upgrading instructions. So, for v2.0 fabric comes with detailed instructions set. Please be noted that it is a high-risk upgrading production project as you might end with corrupting the consortium and please understand the complete changes of v2.0 before upgrading otherwise we might see some sudden surprises in the production.

**Please read the following doc thoroughly**: <https://hyperledger-fabric.readthedocs.io/en/release-2.0/upgrade_to_newest_version.html>

**Fabric suggesting a four-step process**:

1. Backup the orderer ledger, peer’s ledger and MSPs that contain certificates and keys.
2. Upgrade the orderer binaries one by one to the latest Fabric version.
3. Upgrade the peer binaries one by one to the latest Fabric version.
4. Update the orderer system channel and any application channels to the latest capability levels, where available.

## **Upgrade ordering nodes**:

Orderer containers should be upgraded in a rolling fashion which means one at a time.

At a high level, the ordering node or orderer upgrade process goes as follows:

1. Stop the running ordering node.
2. As a best practice back up the ordering node’s ledger and MSP.
3. Stop and delete the ordering node container or pod.
4. Launch a new ordering node container or pod with a v2.0.0 tag.

## **Upgrade the peers:**

Peer containers should be upgraded in a rolling fashion which means one at a time. At a high level, we will perform the following steps:

1. Stop the running peer container or pod.
2. Back up the peer’s ledger and MSP.
3. Remove chaincode containers and images.
4. Stop and delete the peer container or pod.
5. Launch a new peer container or pod with a v2.0.0 tag.

Note: We do not need to relaunch the missing chaincode containers. When the peer gets a request for a chaincode, (invoke or query), it first checks if it has a copy of that chaincode running. If so, it uses it. Otherwise, as in this case, the peer launches the chaincode (rebuilding the image if required).

For detailed and complete information, please refer the following document <https://hyperledger-fabric.readthedocs.io/en/release-2.0/upgrading_your_components.html>

## **Summary**:

We have discussed how open-source projects release new versions often and we have seen the total functionality changes and bug fixes in v2.0. Some instructions on upgrading components to v2.0. The takeaway from the chapter is upgrade carefully and have a backup ledger and MSP’s. If it is not necessary then avoid upgrading to v2.0 as major functionality changes are present.

In the next chapter, we will discuss two interesting tools around Hyperledger Fabric.

# Chapter16: Some Interesting tools

Hyperledger Fabric was released on 16 Sept 2016 and the popularity of fabric is increasing day by day. Many companies and individual developers contributed some tools to support fabric developers. ChainHero is one of the major contributors to provide tools to fabric developers.

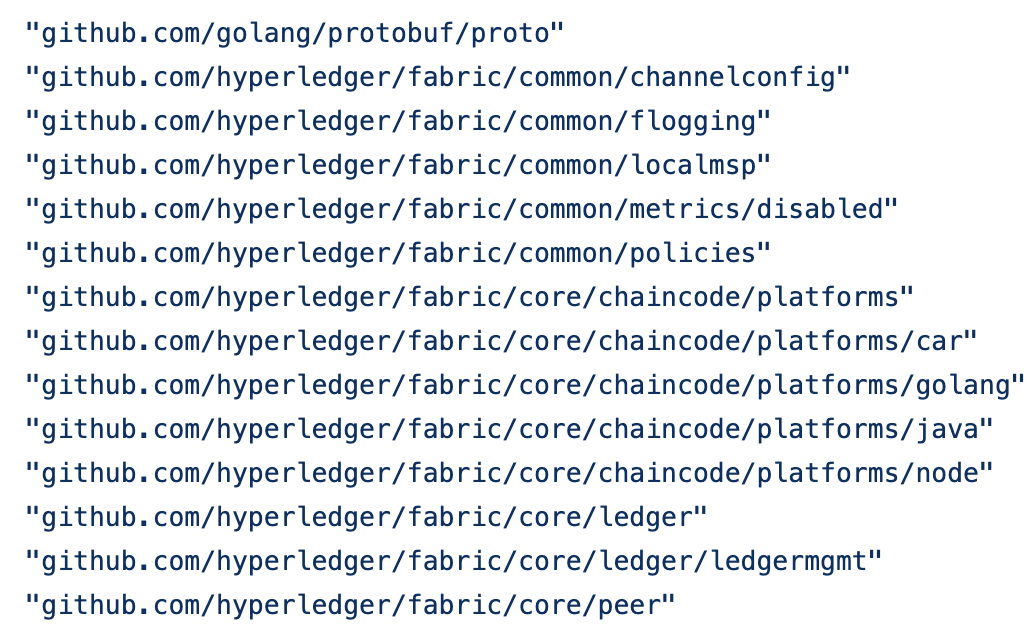
In this chapter, we will discuss two powerful tools

## Ledgerfsck - Tool to verify ledger integrity

## Security Analyzer for Hyperledger Fabric Smart Contracts

# **Ledgerfsck - Tool to verify ledger integrity**

During the validation phase, checking the integrity of blocks is also important and for this, a tool from C0rWin called ledgerfsck. The goal of this tool is to verify the Hyperledger Fabric channel ledger. It traverses ledger starting from the very first block till the last one, ledgerfsck utilizes MessageCryptoService to verify block integrity. It reads channel and resource configuration from the ledger to initialize MSP structures. When we see the source code of the project it utilizes almost all fabric packages. This project is written in Golang and thereby utilizes all packages from fabric. It utilizes all core packages from fabric and viper. Github repo: <https://github.com/C0rWin/ledgerfsck>



When we check the logic of the project. It starts with the genesis block hash and marks this as the previous hash. Next in a for loop start getting the first block and fetch the previous has and then match with genesis block hash. If both hashes are equal then it continues by updating previous hash variables to first block hash and proceed with the second block and fetch the previous hash and then compare. This operation continues until the last block and generates a report based on the operation. Quite simple and useful.

Following is the example of ledgerfsck execution:

|  |
| --- |
| > go run ledgerfsck.go #This leaves us a binary file called ledgerfsck > ./ledgerfsck --channelName mychannel \  --mspID Org1MSP \  --mspPath crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp |

**ChannelName**: the name of the channel which ledger we would like to verify

**MSPID**: the MSP ID of the organization which owns and executes the ledgerfsck

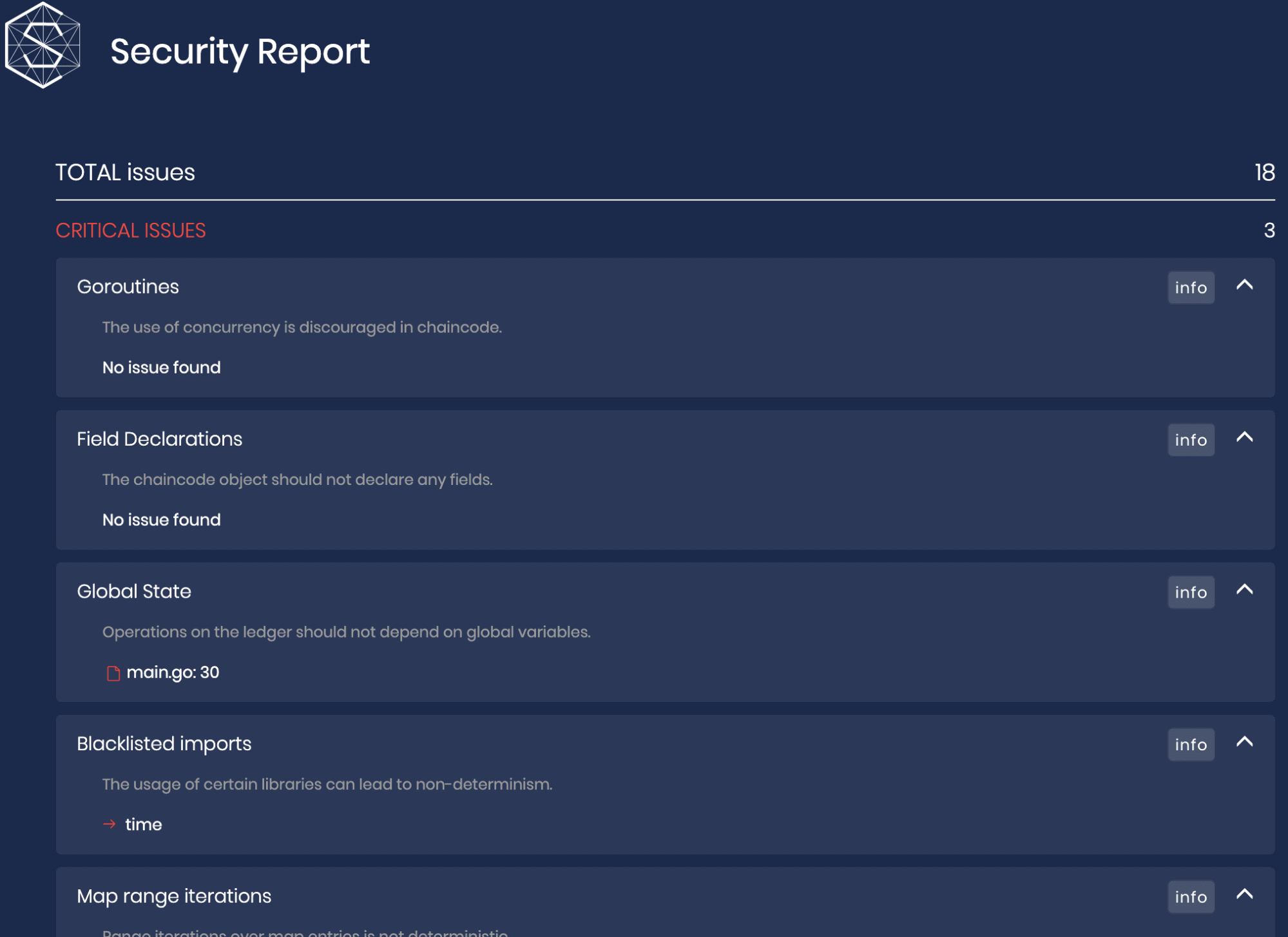
**MspPath**: the path to the MSP folder with all relevant crypto material

# **Security Analyzer**

A tool from ChainHero company. This tool is a paid version for commercial purposes and free for non-commercial purposes. This tool basically fetches the chaincode source files from the GitHub repo and starts to analyze critical issues that are present in the chain code. It basically checks whether any goroutines are present in the chaincode or not. Any potential errors are ignored. Any read operations after write operations which yield to old values. It checks whether any range iterations over map entries as it leads to non-deterministic etc. Many more

Link: <https://chaincode.chainsecurity.com/>

|  |
| --- |
|  |



## **Summary**:

In this chapter, we have discussed few interesting tools around Hyperledger fabric and we have also learned how to use those tools. The craze for HLF is immense and increasing gradually. Apart from these two tools there are many other tools present in the open-source echo system.

This is the last chapter of Mastering Hyperledger Fabric.