UMONS
Université de Mons

Faculté Polytechnique

Cookbook : application web médicale avec Hyperledger Fabric

VANSNICK Tanguy

POLYTECH MONS

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Table des matières

Préface

Ce cookbook a pour objectif d'expliquer le fonctionnement et de détailler les étapes de création d'une ledger en utilisant Hyperledger Fabric. Hyperledger Fabric est un framework open source développé par la Linux Foundation qui offre une série de paramètres et de fonctionnalités permettant de créer une ledger distribué adapté à une utilisation en entreprise. Ce cookbook s'appuie sur l'architecture de démarrage proposée par Hyperledger Fabric, ainsi que sur d'autres contributions provenant de la communauté GitHub.

Le rapport est structuré en plusieurs chapitres, chacun comprenant une section explicative et une section dédiée à l'implémentation. La section d'implémentation met l'accent sur les éléments clés, et si nécessaire, des liens vers GitHub et d'autres documentations seront fournis pour approfondir le sujet.

Chapitre 1

Installation packages et prérequis

Dans ce premier chapitre, nous aborderons l'installation de tous les éléments nécessaires pour assurer le bon fonctionnement de l'architecture. Cela comprend les prérequis avec leurs versions appropriées, ainsi que les composants de l'architecture de base.

1.1 Prérequis

Avant de passer à la partie Hyperledger, plusieurs outils doivent être installés. Hyperledger Fabric prend en charge différents langages de programmation tels que Go, Node.js et Java. Dans notre cas, nous avons utilisé Node.js, donc les autres langages n'ont pas été installés.

Pour travailler avec Hyperledger Fabric, les outils suivant doivent être installés :

Docker et docker-compose : Ces outils nous permettront d'utiliser des images et des conteneurs Docker pour nos pairs (peers) et nos organisations. Docker Desktop peut être installé avec le lien suivant : Lien d'installation de Docker Desktop.

Node.js : Nous utiliserons Node.js pour créer notre application et nos chaincodes. Toutes les versions de Node.js ne sont pas compatibles avec Hyperledger Fabric. Nous avons utilisé la version 8.13 pour notre projet. Node.js peut être installé à partir du lien suivant : Lien d'installation de Node.js. Ensuite, dans le terminal, à l'aide des commandes suivante, il est possible d'inter-changer rapidement la version de Node.js souhaité :

Listing 1.1 – Nodejs choix de version

```
sudo npm install -g n
sudo n 8.13
```

Pour obtenir plus d'informations sur les prérequis, veuillez consulter la page suivante : Lien vers la page des prérequis. Il est également important de faire attention à la version de Hyperledger utilisée. Au cours de notre implémentation, nous avons essayé de travailler avec une machine équipée d'une puce arm64 (puce M1). Cependant, nous n'avons pas réussi à lancer l'architecture sur cette configuration.

1.2 Installation des packages

Pour commencer avec Hyperledger Fabric, nous allons installer le dossier de démarrage en utilisant la commande suivante dans le terminal, à l'emplacement où nous souhaitons travailler :

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Une fois cette commande exécutée, un dossier "fabric-samples" sera téléchargé, contenant une architecture de départ servant d'exemple. Comme ce dossier contient plusieurs sous-dossiers, il est important de comprendre leurs différentes fonctions.

En plus du dossier de démarrage, nous aurons également besoin de cloner deux autres référentiels GitHub. Le premier a été implémenté spécifiquement pour ce travail de fin d'études, tandis que le second a été utilisé comme point de départ et référence tout au long du projet. Ce dernier est accompagné d'une série de vidéos expliquant les différentes étapes :

— GitHub TFE
— GitHub de référence

Chapitre 2

Les dossiers

Dans ce chapitre, une brève description de chaque dossier du répertoire est évoqué. Ces descriptions ont pour but de connaître la fonction de chaque compartiment ainsi que les différents éléments qu'ils contient.

2.1 Dossier binaries

Le répertoire "fabric-samples" contient les binaires nécessaires pour travailler avec Hyperledger Fabric. Pour faciliter l'accès à ces binaires, Hyperledger recommande de créer une variable d'environnement "PATH" qui pointe vers ce dossier. Pour notre projet, nous avons décidé d'ajouter ce dossier directement dans le répertoire de notre projet afin d'éviter d'éventuelles erreurs. Voici une description des binaires que nous avons utilisés au cours de notre projet :

1. configtxgen : configtxgen est utilisé pour générer les éléments cryptographiques liés aux transactions dans Hyperledger Fabric. Il permet de créer les canaux, les profils de consortium, les profils d'ordre et les blocs genesis.
2. configtxlator : ce binaire est utilisé pour générer et manipuler la configuration des canaux dans Hyperledger Fabric. Il permet de convertir la configuration des canaux entre différents formats et d'examiner les détails de la configuration.
3. cryptogen : il est employé pour générer les éléments cryptographiques nécessaires à la création de pairs (peers) et d'ordonneurs (orderers) dans Hyperledger Fabric. Il permet de générer les clés, les certificats et les artefacts nécessaires pour sécuriser les communications dans le réseau.
4. orderer : il représente un ordonneur dans Hyperledger Fabric, qui est responsable de la validation des transactions et de la création des blocs dans un canal. Il peut être utilisé pour lancer et configurer un ordonneur dans un réseau Hyperledger Fabric.
5. peer : Ce binaire représente un pair dans Hyperledger Fabric, qui est responsable de l'exécution des chaincodes (contrats intelligents) et de la gestion de l'état du grand livre dans un canal. Il peut être utilisé pour lancer et configurer un pair dans un réseau Hyperledger Fabric.
6. fabric-ca-client : Ce binaire est utilisé pour interagir avec le service d'autorité de certification (CA) dans Hyperledger Fabric. Il permet d'identifier, d'enregistrer et de gérer les certificats d'utilisateur dans le réseau.

2.2 Dossier api 2.0

Ce dossier contient tous les éléments nécessaires pour le lancement de l'application en local, notamment dans notre cas où l'application est basée sur Node.js. On y retrouve tous les codes JavaScript liés à cette fonction, ainsi que les fichiers de connexion avec l'architecture créée.

2.3 Dossier artifacts

C'est dans ce dossier que tous les éléments de cryptages sont définit, générés et stockés. Les fichiers docker-compose s'y trouvent également. C'est donc à partir de dossier que l'ont retrouve la configuration du réseau ainsi que sont lancement dans divers conteneurs docker.

2.4 Dossier script

Le dossier "script" contient une série de fonctions shell. C'est fonctions sont communes à plusieurs codes, elles ont donc été groupées pour faciliter leur exécution. Les codes shell (.sh) sont utilisés dans le but d'automatiser diverses actions, telles que la configuration et la génération d'éléments cryptographiques.

Chapitre 3

Architecture

Dans ce cookbook, nous utilisons l'architecture que nous avons proposé dans le rapport de TFE. Pour en savoir plus concernant les choix et les justifications, le rapport est accessible sur le GitHub. Pour en savoir plus sur le fonctionnement global et les différentes possibilités d'implémentations, nous vous recommandons de consulter la documentation ou le rapport de TFE. La documentation est disponible via le lien suivant : Lien de la documentation

Pour cette architecture, nous avons défini deux organisations : (R1 et R2) et un orderer (R0). CC1 représente la configuration du réseau qui a été acceptée par les deux organisations. CC1 définit également les rôles de chaque organisation lors de la création des différents channels.

Lorsqu'un channel C1 est créé, les peers des organisations concernées rejoignent le channel. Chaque organisation possède 2 peers, tandis que l'orderer en possède 3. Chaque nœud possède une copie de la ledger L1. Les nœuds peuvent interagir avec les channels à travers les applications A1 et A2. Chaque organisation possède un certificat d'autorité (CA) qui génère les certificats requis entre les nœuds et l'organisation.

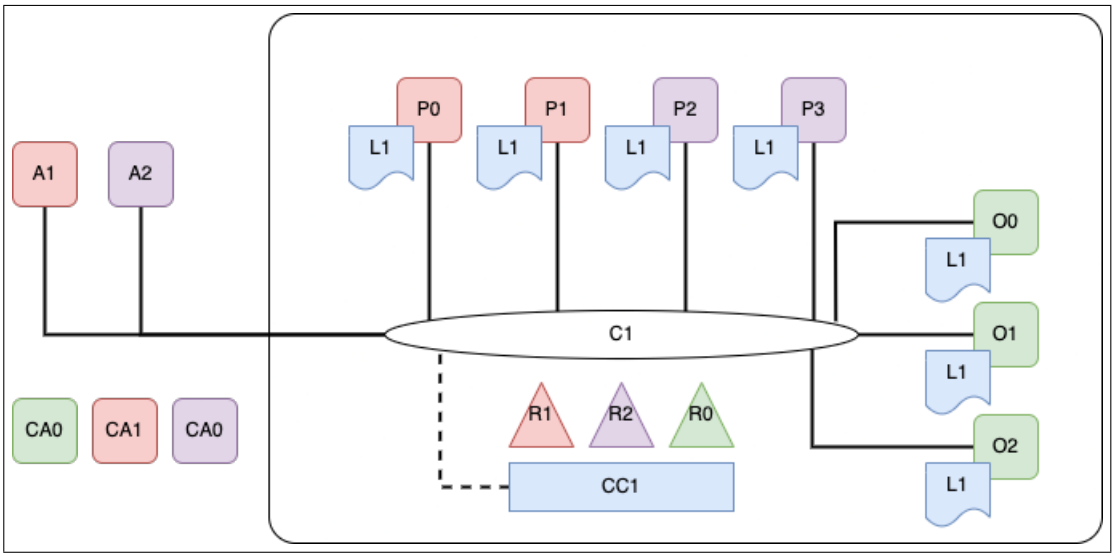

FIGURE 3.1 – Architecture réseau

3.1 Création des éléments cryptographiques

Avant de pouvoir démarrer le réseau, il est nécessaire de générer les éléments cryptographiques requis. Pour cela, nous utiliserons l'outil binaire "cryptogen" pour générer les éléments cryptographiques de chaque nœud. Ensuite, pour configurer le réseau, nous utiliserons l'outil binaire "configtxgen".

3.1.1 Cryptogen

Cryptogen est un outil utilisé dans un environnement de test pour générer les certificats et les clés nécessaires aux organisations et aux peers de ledgers. Dans notre cas, le fichier de configuration des éléments cryptographiques est nommé "crypto-config.yaml". Ce fichier YAML permet de spécifier les informations nécessaires pour l'orderer ainsi que pour les différents peers du réseau. Il doit être configuré avec les paramètres appropriés pour chaque entité du réseau, conformément aux exigences de sécurité et de configuration spécifique à l'environnement.

Dans cet exemple, nous définissons la création de 3 orderers (OrdererOrgs) et 2 organisations (PeerOrgs) dans le fichier de configuration "crypto-config.yaml". Dans la section "Specs" de ce fichier, nous pouvons spécifier tous les nœuds que nous souhaitons générer pour notre ledger. Pour les orderers, nous avons donc défini la création de 3 instances.

```
OrdererOrgs:
  - Name: Orderer
    Domain: example.com
    EnableNodeOUs: true
    Specs:
      - Hostname: orderer
        SANS:
            - "localhost"
            - "127.0.0.1"
      - Hostname: orderer2
        SANS:
          - "localhost"
          - "127.0.0.1"
      - Hostname: orderer3
        SANS:
            - "localhost"
            - "127.0.0.1"
```

Pour les organisations, nous pouvons définir le nombre de pairs souhaités en utilisant la commande "Template" dans le fichier de configuration. De plus, le nombre d'utilisateurs peut également être défini dans la section "Users". Dans notre cas, nous avons défini un seul utilisateur pour chaque organisation.

```
PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
  - Name: Org2
    Domain: org2.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
```

Une fois que tous les éléments sont définis dans le fichier YAML, nous pouvons générer l'ensemble des certificats et clés cryptographiques. Pour ce faire, il suffit d'utiliser la commande suivante dans le terminal pour appeler le fichier "cryptogen" :

```
cryptogen generate --config=./crypto-config.yaml
```

Cette commande peut varier en fonction de l'emplacement où le dossier "./bin" est stocké. Une fois que la commande est exécutée, un dossier "crypto-config" est créé. À l'intérieur de ce dossier, on retrouve les certificats et clés cryptographiques générés pour chaque organisation, pair et utilisateur. Dans la figure 3.2, on peut voir tous les fichiers générés.

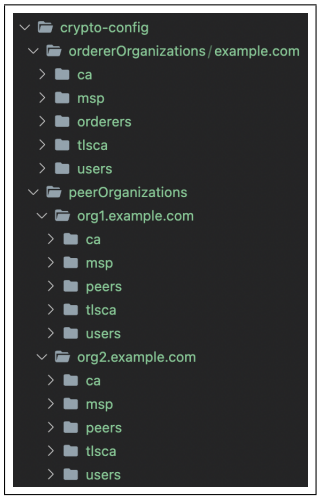

FIGURE 3.2 – Dossier crypto-config

3.2 Configtxgen

Nous allons utiliser l'outil binaire "configtxgen" pour configurer et définir les différentes conventions du réseau. Dans le fichier "configtx.yaml", nous allons donc définir les cinq artefacts suivants :

1. Organizations
2. Applications
3. Capabilities
4. Orderer
5. Channel
6. Profiles

3.2.1 Organisations

Cette rubrique regroupe la liste des organisations utilisées. Chaque organisation possède un nom, un ID et un MSP directory reprenant l'ensemble des clés cryptographiques générées précédemment. Le champ "policies" définit les accès. Nous avons énuméré 3 policies différentes :

1. Readers : ont uniquement un accès en lecture.
2. Writers : ont uniquement un accès en écriture.
3. Admins : ont accès à l'ensemble du réseau.

Dans l'exemple ci-dessous, nous avons repris les 2 organisations que nous avions générées lors de l'étape précédente. Le code ci-dessous montre la configuration de chaque organisation.

```
Organizations:
    - &OrdererOrg
        Name: OrdererOrg
        ID: OrdererMSP
        MSPDir: crypto-config/ordererOrganizations/example.com/msp
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('OrdererMSP.member')"
            Writers:
                Type: Signature
                Rule: "OR('OrdererMSP.member')"
            Admins:
                Type: Signature
                Rule: "OR('OrdererMSP.admin')"

    - &Org1
        Name: Org1MSP
        ID: Org1MSP
```

```
        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org1MSP.admin')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org1MSP.peer')"

        AnchorPeers:
            - Host: peer0.org1.example.com
              Port: 7051

    - &Org2
        Name: Org2MSP
        ID: Org2MSP
        MSPDir: crypto-config/peerOrganizations/org2.example.com/msp
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('Org2MSP.admin', 'Org2MSP.peer', 'Org2MSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org2MSP.admin', 'Org2MSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org2MSP.admin')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org2MSP.peer')"

        AnchorPeers:
            - Host: peer0.org2.example.com
              Port: 9051
```

3.2.2 Capabilities

La section "Capabilities" définit les différentes versions compatibles avec notre architecture. Dans le code ci-dessous, on retrouve les version que nous avons décidé d'employer.

```
Capabilities:
  Channel: &ChannelCapabilities
    V2_0: true
  Orderer: &OrdererCapabilities
    V2_0: true
  Application: &ApplicationCapabilities
    V2_0: true
```

3.2.3 Application

Comme nous n'avons pas encore défini la couche d'application, il n'est pas possible de configurer entièrement cette couche pour le moment. Cependant, il est possible de définir les différentes "policies" comme nous l'avons fait dans la partie "organizations".

```
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
```

3.2.4 Orderer

Dans ce bloc, les paramètres de l'orderer sont définis. Étant donné que l'orderer est responsable de la construction et de la proposition des blocs aux différents peers du réseau, c'est ici que la configuration du bloc a lieu. Le protocole de consensus est également défini dans cette section.

1. OrdererType : c'est le type de orderer que l'on souhaite utilisé. Fabric en propose 3 pour l'instant solo, kafka et etcdraft. En fonction de celui sélectionnés, des champs supplémentaires devront être complétés.
2. BatchTimeout : représente le temps à attendre après la première transaction avant qu'un nouveau bloc ne soit créé.
3. BatchSize : configure le nombre maximum de messages, la taille souhaitée et la taille maximale d'un bloc.

```
Orderer: &OrdererDefaults
    OrdererType: etcdraft
    EtcdRaft:
        Consenters:
        - Host: orderer.example.com
          Port: 7050
          ClientTLSCert: crypto-config/ordererOrganizations/example.com/
↪   orderers/orderer.example.com/tls/server.crt
          ServerTLSCert: crypto-config/ordererOrganizations/example.com/
↪   orderers/orderer.example.com/tls/server.crt
    Addresses:
        - orderer.example.com:7050
    BatchTimeout: 2s
    BatchSize:
        MaxMessageCount: 10
        AbsoluteMaxBytes: 99 MB
        PreferredMaxBytes: 512 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"
```

3.2.5 Channel

La section channel définit les paramètres de configuration par défaut des différents canaux de communications. La configuration peut varier entre chaque channel. Dans l'exemple, nous avons uniquement définit un seul channel.

```
Channel: &ChannelDefaults
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"

    Capabilities:
        <<: *ChannelCapabilities
```

3.2.6 Profiles

Il est possible de créer plusieurs profiles. Chaque profile est représenté par une sous-section. Chacun de ces profiles est utilisé pour la configuration de composants spécifiques. Les attributs de chaque profiles varient par rapport à leur fonction :

— Genesis profile : génère le bloc genesis de la chaîne.
— Channel profile : construit la couche au dessus du réseau.

Les différents profiles vont uniquement référencés les éléments de configuration que nous avons définit plus précédemment. Dans l'exemple ci-dessous, nous avons crée un profile pour le genesis block et un second pour le channel.

```
Profiles:
    BasicChannel:
        Consortium: SampleConsortium
        <<: *ChannelDefaults
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
            Capabilities:
                <<: *ApplicationCapabilities

    OrdererGenesis:
        <<: *ChannelDefaults
```

```
        Capabilities:
            <<: *ChannelCapabilities
        Orderer:
            <<: *OrdererDefaults
            OrdererType: etcdraft
            EtcdRaft:
                Consenters:
                - Host: orderer.example.com
                  Port: 7050
                  ClientTLSCert: crypto-config/ordererOrganizations/
↪   example.com/orderers/orderer.example.com/tls/server.crt
                  ServerTLSCert: crypto-config/ordererOrganizations/
↪   example.com/orderers/orderer.example.com/tls/server.crt
                - Host: orderer2.example.com
                  Port: 8050
                  ClientTLSCert: crypto-config/ordererOrganizations/
↪   example.com/orderers/orderer2.example.com/tls/server.crt
                  ServerTLSCert: crypto-config/ordererOrganizations/
↪   example.com/orderers/orderer2.example.com/tls/server.crt
                - Host: orderer3.example.com
                  Port: 9050
                  ClientTLSCert: crypto-config/ordererOrganizations/
↪   example.com/orderers/orderer3.example.com/tls/server.crt
                  ServerTLSCert: crypto-config/ordererOrganizations/
↪   example.com/orderers/orderer3.example.com/tls/server.crt
            Addresses:
                - orderer.example.com:7050
                - orderer2.example.com:8050
                - orderer3.example.com:9050

            Organizations:
            - *OrdererOrg
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                - *Org1
                - *Org2
```

3.2.7 Génération des éléments

Maintenant que tous les paramètres de configurations ont été définit, le genesis block et le channel peuvent être générés. Les éléments générés seront stocké dans un nouveau dossier appelé "channel-artifacts".

```
mkdir channel-artifacts
```

Une fois le dossier créé, la commande de génération du bloc genesis peut être exécuté. Pour cette commande, nous devons faire appel au binary "configtxgen". Il faut préciser le profile à exécuter ("OrdererGenesis"). Un nom de channel doit également être définit.

```
configtxgen -profile OrdererGenesis -configPath . -channelID
sys-channel -outputBlock ./genesis.block

configtxgen -profile BasicChannel -configPath . -channelID
mychannel -outputCreateChannelTx ./mychannel.tx
```

Ensuite nous devons également générer les anchors peer pour chaque organisation. Les anchors peers permettent la communication entre différentes organisations.

```
configtxgen -profile BasicChannel -configPath . -channelID mychannel
-outputAnchorPeerUpdate ./Org1MSPanchors.tx -asOrg Org1MSP

configtxgen -profile BasicChannel -configPath . -channelID mychannel
-outputAnchorPeerUpdate ./Org2MSPanchors.tx -asOrg Org2MSP
```

3.3 Automatisation

Maintenant que tous les éléments ont été définit, nous pouvons créer un script .sh afin de regrouper l'ensemble des commandes en un seul ficher. Dans ce fichier on retrouve l'ensemble des commandes de terminal utilisé précédemment.

```
1  chmod -R 0755 ./crypto-config
2  # Delete existing artifacts
3  rm -rf ./crypto-config
4  rm genesis.block mychannel.tx
5  rm -rf ../../channel-artifacts/*
6
7  #Generate Crypto artifactes for organizations
8  ../../bin/cryptogen generate --config=./crypto-config.yaml
   ↪  --output=./crypto-config/
9
10 SYS_CHANNEL="sys-channel"
11 CHANNEL_NAME="mychannel"
12
13 # Generate System Genesis block
14 ../../bin/configtxgen -profile OrdererGenesis -configPath . -channelID
   ↪  $SYS_CHANNEL  -outputBlock ./genesis.block
15
16 # Generate channel configuration block
17 ../../bin/configtxgen -profile BasicChannel -configPath .
   ↪  -outputCreateChannelTx ./mychannel.tx -channelID $CHANNEL_NAME
18
```

```
19 echo "#######    Generating anchor peer update for Org1MSP  ##########"
20 ../../bin/configtxgen -profile BasicChannel -configPath .
    ↪  -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID $CHANNEL_NAME
    ↪  -asOrg Org1MSP
21
22 echo "#######    Generating anchor peer update for Org2MSP  ##########"
23 ../../bin/configtxgen -profile BasicChannel -configPath .
    ↪  -outputAnchorPeersUpdate ./Org2MSPanchors.tx -channelID $CHANNEL_NAME
    ↪  -asOrg Org2MSP
```

Chapitre 4

2 : Configuration Docker

Comme le réseau est configuré et que tous les éléments cryptographique sont généré, il est maintenant possible de construire les conteneurs des différents éléments. Pour ce faire, nous avons créé un fichier "docker-compose.yaml". Ce fichier contient les éléments de configuration pour :

- Peer
- Orderer
- base de données

Durant ce chapitre, nous allons uniquement montrer des parties de codes. Le code complet est disponible sur GitHub.

4.1 Peer et Orderer

Les peers et orderers ont une structure similaire. Pour les peers, nous utilisons l'image docker "hyperledger/fabric-peer", tandis que pour les orderers, nous faisons appel à l'image "hyperledger/fabric-orderer". Dans la section "environnement", on trouve plusieurs paramètres de configuration tels que les répertoires des éléments cryptographiques, les ports utilisés, etc.

```
peer0.org1.example.com:
container_name: peer0.org1.example.com
extends:
  file: base.yaml
  service: peer-base
environment:
  ### voir GitHub ###
depends_on:
  - couchdb0
ports:
  - 7051:7051
volumes:
  - ./channel/crypto-config/peerOrganizations/org1.example.com/peers/
  ↪ peer0.org1.example.com/msp:/etc/hyperledger/crypto/peer/msp
  - ./channel/crypto-config/peerOrganizations/org1.example.com/peers/
  ↪ peer0.org1.example.com/tls:/etc/hyperledger/crypto/peer/tls
```

```
  - /var/run/docker.sock:/host/var/run/docker.sock
  - ./channel/:/etc/hyperledger/channel/
networks:
  - test
```

```
orderer.example.com:
container_name: orderer.example.com
image: hyperledger/fabric-orderer:2.1
dns_search: .
environment:
  ### Voir GitHub ###
working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderers
command: orderer
ports:
  - 7050:7050
  - 8443:8443
networks:
  - test
volumes:
  - ./channel/genesis.block:/var/hyperledger/orderer/genesis.block
  - ./channel/crypto-config/ordererOrganizations/example.com/orderers/
  orderer.example.com/msp:/var/hyperledger/orderer/msp
  - ./channel/crypto-config/ordererOrganizations/example.com/orderers/
  orderer.example.com/tls:/var/hyperledger/orderer/tls
```

4.2 Base de données

Hyperledger propose deux types de bases de données : LevelDB et CouchDB. LevelDB est la base de données par défaut, fonctionnant selon le principe clé-valeur, ce qui signifie que seules les requêtes avec les clés sont possibles. CouchDB, quant à elle, est une solution alternative où les données sont stockées sous format JSON, permettant ainsi des requêtes plus efficaces.

Il est crucial de choisir le type de base de données avant de configurer le réseau, car il n'est pas possible de changer de type de base de données à cause de problèmes de compatibilité. De plus, tous les peers du réseau doivent avoir le même type de base de données.

Dans notre cas, nous avons choisi d'utiliser CouchDB. Chaque peer doit être connecté à une base de données, donc nous en avons créé quatre. Pour cela, nous utilisons l'image Docker "hyperledger/ fabric-couchdb". Dans la section "environnement", il est possible de définir un identifiant et un mot de passe pour la base de données.

```
uchdb0:
container_name: couchdb0
image: hyperledger/fabric-couchdb
environment:
  - COUCHDB_USER=
```

```
6     - COUCHDB_PASSWORD=
7   ports:
8     - 5984:5984
9   networks:
10    - test
```

4.3 Démarrage des conteneurs

Une fois que tous les nœuds et bases de données sont correctement configurés, nous pouvons maintenant exécuter le fichier "docker-compose.yaml" et démarrer les différents conteneurs en utilisant la commande suivante :

```
docker-compose up -d
```

Après l'exécution de cette commande, tous les conteneurs devraient être opérationnels. Si vous ne souhaitez plus travailler avec les conteneurs, vous pouvez les arrêter en utilisant la commande suivante depuis le terminal :

```
docker-compose down
```

Chapitre 5

Création du channel

Nous disposons maintenant d'un réseau fonctionnel avec plusieurs nœuds et organisations. Dans ce chapitre, nous allons créer un canal commun pour nos deux organisations. Un script shell intitulé "create-channel.sh" est disponible, regroupant l'ensemble des commandes à exécuter. Ce fichier comprend trois fonctions distinctes :

1. La fonction "createChannel" a pour objectif de créer le canal en fournissant le bloc génésis ainsi que les paramètres de configuration nécessaires.
2. La fonction "joinChannel" permet d'ajouter tous les peers souhaitant rejoindre le canal lors de son exécution.
3. La fonction "updateAnchorPeer" est utilisée pour mettre à jour les "anchor peers" afin d'assurer la communication entre les différents nœuds du réseau.

```
createChannel(){
    rm -rf ./channel-artifacts/*
    setGlobalsForPeer0Org1

    bin/peer channel create -o localhost:7050 -c $CHANNEL_NAME \
    --ordererTLSHostnameOverride orderer.example.com \
    -f ./artifacts/channel/${CHANNEL_NAME}.tx --outputBlock
     ↪  ./channel-artifacts/${CHANNEL_NAME}.block \
    --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA
}
```

```
joinChannel(){
    setGlobalsForPeer0Org1
    bin/peer channel join -b ./channel-artifacts/$CHANNEL_NAME.block

    setGlobalsForPeer1Org1
    bin/peer channel join -b ./channel-artifacts/$CHANNEL_NAME.block

    setGlobalsForPeer0Org2
    bin/peer channel join -b ./channel-artifacts/$CHANNEL_NAME.block

    setGlobalsForPeer1Org2
```

```
    bin/peer channel join -b ./channel-artifacts/$CHANNEL_NAME.block

}
```

```
updateAnchorPeers(){
    setGlobalsForPeer0Org1
    bin/peer channel update -o localhost:7050 --ordererTLSHostnameOverride
    ↪ orderer.example.com -c $CHANNEL_NAME -f
    ↪ ./artifacts/channel/${CORE_PEER_LOCALMSPID}anchors.tx --tls
    ↪ $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA

    setGlobalsForPeer0Org2
    bin/peer channel update -o localhost:7050 --ordererTLSHostnameOverride
    ↪ orderer.example.com -c $CHANNEL_NAME -f
    ↪ ./artifacts/channel/${CORE_PEER_LOCALMSPID}anchors.tx --tls
    ↪ $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA

}
```

Une fois que toutes les fonctions sont définies, il suffit de les appeler dans le fichier et d'exécuter ce dernier. Pour l'exécuter, il faut entrer le nom du fichier .sh dans le terminal :

```
./creatChannel.sh
```

Si aucune erreur ne s'est produite, les peers font maintenant partie du channel "mychannel". Pour vérifier cela, nous pouvons entrer dans l'un des conteneurs et vérifier s'il est bien lié au channel "mychannel" à l'aide des commandes suivantes :

```
docker exec -it peer0.org1.example.com sh
peer channel list
```

Si le nom "mychannel" figure dans la liste, cela signifie que le peer0 de l'organisation 1 fait bien partie du channel. Dans le cas contraire, cela implique qu'il n'a pas été enrôlé ou que le channel n'existe pas.

Chapitre 6

Création du chaincode

Un chaincode est un programme qui peut être écrit en GO, Node.js ou Java. Le chaincode s'exécute dans un conteneur Docker isolé des autres nœuds du réseau. Il est responsable de l'initialisation et de la gestion de l'état de la ledger au travers des transactions soumises par les pairs (peers) du channel.

En général, un chaincode possède une business logique qui est liée aux besoins spécifiques du réseau et de ses membres. Les états définis par un chaincode ne sont pas accessibles par un autre chaincode. Si le chaincode appelé appartient à un autre channel, alors il ne peut être que lu.

Le chapitre sur les chaincodes peut être divisé en deux parties. Dans la première partie, nous allons examiner l'implémentation du chaincode et comment il peut être déployé. Dans la seconde partie, nous verrons comment interagir avec les différentes fonctions du chaincode.

6.1 Création du chaincode

Dans ce cookbook, nous avons choisi d'implémenter le chaincode en Node.js, car c'est le langage le plus adapté pour la création d'applications web et le traitement en temps réel. Pour l'implémentation du chaincode, nous avons utilisé le template de fabric-samples disponible via le lien suivant : Lien de fabric-samples. Une fois le chaincode placé dans un dossier, nous avons installé npm en utilisant la commande suivante :

```
npm install
```

Après avoir récupéré le template, le chaincode a été modifié pour correspondre à la business logique souhaitée. La classe du chaincode se trouve dans le dossier "lib", et dans notre cas, le fichier est nommé "capsule.js". Ce fichier contient plusieurs fonctions avec lesquelles un utilisateur peut interagir. Les fonctions mentionnées sont disponibles dans la documentation suivante : docs chaincode.

6.1.1 Initialisation

En tant que première fonction, nous avons créé une fonction qui remplit la ledger avec des données initiales au format JSON. Nous avons défini six éléments : ID, SensorID, SensorType, TimeStamp, Patient et Value. Une fois que la liste des éléments à ajouter est définie, ils sont ajoutés un par un à la ledger.

```
async InitLedger(ctx) {
    const assets = [
        {
            ID: "capsule1",
            SensorID: 101,
            SensorType: 'heart',
            TimeStamp: '2021-04',
            Patient: 'Said',
            value: 300,
        },
    ];
    for (const asset of assets) {
        asset.docType = 'asset';
        await ctx.stub.putState(asset.ID,
         ↪ Buffer.from(stringify(sortKeysRecursive(asset))));
    }
}
```

6.1.2 Ajout de données

La deuxième fonction que nous avons mise en place est la fonction "CreateAsset", qui a pour objectif d'ajouter de nouvelles données qui n'existent pas encore dans la ledger. Pour ce faire, nous devons entrer les paramètres d'un nouvel asset, tels que l'ID, SensorID, SensorType, TimeStamp, Patient et Value. Une fois que l'asset est défini, il est ajouté à la chaîne de la même manière que dans la fonction d'initialisation. Une amélioration possible serait de vérifier si l'asset n'existe pas déjà dans la chaîne avant de l'ajouter.

```
async CreateAsset(ctx, id, sensorId, sensorType, timeStamp, patient, value) {
const exists = await this.AssetExists(ctx, id);
if (exists) {
    throw new Error(`The asset ${id} already exists`);
}

const asset = {
    ID: id,
    SensorID: sensorId,
    SensorType: sensorType,
    TimeStamp: timeStamp,
    Patient: patient,
    value: value,
};
await ctx.stub.putState(id,
 ↪ Buffer.from(stringify(sortKeysRecursive(asset))));
return JSON.stringify(asset);
}
```

6.1.3 Requête

Maintenant que nous avons une fonction pour initialiser la ledger et une autre pour ajouter de nouveaux éléments, nous allons créer une troisième fonction permettant de lire un asset appartenant à la ledger. Cette fonction sera appelée "ReadAsset". L'utilisateur devra fournir l'ID de l'asset qu'il souhaite analyser en entrée. En retour, il recevra sous la forme d'un JSON, toutes les informations appartenant à cet asset. Cette fonction permettra ainsi de consulter les données d'un asset spécifique dans la ledger.

```
async ReadAsset(ctx, id) {
    const assetJSON = await ctx.stub.getState(id); // get the asset from
     ↪ chaincode state
    if (!assetJSON || assetJSON.length === 0) {
        throw new Error(`The asset ${id} does not exist`);
    }
    return assetJSON.toString();
}
```

6.2 Déploiement du chaincode

Une fois que le chaincode est rédigé et fonctionnel, il peut être déployé. Le déploiement se fait en une série d'étapes, dont l'ensemble est détaillé dans le fichier "deployChaincode.sh" afin d'automatiser le processus. Les étapes comprennent la compilation du chaincode, la création du package du chaincode, l'installation du package sur les peers du réseau, l'instantiation du chaincode sur le channel spécifié, et enfin, la vérification du déploiement réussi du chaincode. Ce fichier de déploiement permet de simplifier et d'automatiser le processus de déploiement du chaincode sur le réseau.

6.2.1 Création du package

Pour pouvoir envoyer le chaincode aux différents nœuds, celui-ci doit être transformé sous la forme d'un package (tar.gz). Pour ce faire, nous utilisons la fonction "packageChaincode" qui prend en charge la conversion du chaincode dans le format souhaité. Cette fonction permet de créer un package du chaincode, compressé au format tar.gz, qui peut être distribué et installé sur les nœuds du channel.

```
packageChaincode() {
    set -x
    bin/peer lifecycle chaincode package ${CC_NAME}.tar.gz \
            --path ${CC_SRC_PATH} --lang ${CC_RUNTIME_LANGUAGE} \
            --label ${CC_NAME}_${CC_VERSION} >&log.txt
    res=$?
    PACKAGE_ID=$(bin/peer lifecycle chaincode calculatepackageid
     ↪ ${CC_NAME}.tar.gz)
    { set +x; } 2>/dev/null
    cat log.txt
    verifyResult $res "Chaincode packaging has failed"
```

```
    successln "Chaincode is packaged"
}
```

6.2.2 Installation chaincode

Une fois le package créé, nous pouvons l'installer sur plusieurs nœuds. Dans l'exemple ci-dessous, le chaincode est installé uniquement sur le peer 0 de l'organisation 1 et le peer 0 de l'organisation 2.

```
installChaincode() {
    setGlobalsForPeer0Org1
    set -x
    bin/peer lifecycle chaincode queryinstalled --output json | jq -r 'try
    ↪  (.installed_chaincodes[].package_id)' | grep ^${PACKAGE_ID}$
    ↪  >&log.txt
    if test $? -ne 0; then
        bin/peer lifecycle chaincode install ${CC_NAME}.tar.gz >&log.txt
        res=$?
    fi
    { set +x; } 2>/dev/null
    cat log.txt
    verifyResult $res "Chaincode installation on peer0.org1 has failed"
    successln "Chaincode is installed on peer0.org1"

    setGlobalsForPeer0Org2
    set -x
    bin/peer lifecycle chaincode queryinstalled --output json | jq -r 'try
    ↪  (.installed_chaincodes[].package_id)' | grep ^${PACKAGE_ID}$
    ↪  >&log.txt
    if test $? -ne 0; then
        bin/peer lifecycle chaincode install ${CC_NAME}.tar.gz >&log.txt
        res=$?
    fi
    { set +x; } 2>/dev/null
    cat log.txt
    verifyResult $res "Chaincode installation on peer0.org2 has failed"
    successln "Chaincode is installed on peer0.org2"
}
```

6.2.3 Installation des requêtes

Après avoir installé le chaincode sur les différents nœuds, les nœuds doivent approuver la définition du chaincode, qui comprend le nom, la version et la politique d'approbation. Par défaut, la politique d'approbation requiert qu'une majorité de nœuds approuvent le chaincode avant qu'il puisse être utilisé. Étant donné que nous avons installé le chaincode seulement sur deux nœuds, pour qu'il y ait une majorité, le chaincode doit être validé par ces deux nœuds.

Si une organisation a le chaincode installé, elle doit inclure le packageID dans la définition du chaincode approuvé par l'organisation. Le packageID est récupéré à l'aide de la commande suivante :

```
queryInstalled() {
    setGlobalsForPeer0Org1
    set -x
    bin/peer lifecycle chaincode queryinstalled --output json | jq -r 'try
     ↪  (.installed_chaincodes[].package_id)' | grep ^${PACKAGE_ID}$
     ↪  >&log.txt
    res=$?
    { set +x; } 2>/dev/null
    cat log.txt
    verifyResult $res "Query installed on peer0.org1 has failed"
    successln "Query installed successful on peer0.org1 on channel"
}
```

À l'aide du pacakgeID, le chaincode peut être approuvé au niveau des organisations. L'approbation est distribué entre les différents noeuds au sien de l'organisation. à l'aide de gossip. La définition du chaincode est approuvé avec la commande "approveformyorg".

```
approveForMyOrg1() {
    setGlobalsForPeer0Org1
    set -x
    bin/peer lifecycle chaincode approveformyorg -o localhost:7050 \
            --ordererTLSHostnameOverride orderer.example.com --tls \
            --cafile "$ORDERER_CA" --channelID $CHANNEL_NAME --name
             ↪  ${CC_NAME} --version ${CC_VERSION} \
            --package-id ${PACKAGE_ID} \
            --sequence ${CC_SEQUENCE} ${INIT_REQUIRED} ${CC_END_POLICY}
             ↪  ${CC_COLL_CONFIG} >&log.txt
    res=$?
    { set +x; } 2>/dev/null
    cat log.txt
    verifyResult $res "Chaincode definition approved on peer0.org1 on channel
     ↪  '$CHANNEL_NAME' failed"
    successln "Chaincode definition approved on peer0.org1 on channel
     ↪  '$CHANNEL_NAME'"
}
```

6.2.4 Commit de la définition du chaincode

Une fois qu'un nombre suffisant d'organisations ont approuvé la définition du chaincode, l'organisation peut valider la définition du chaincode sur le channel. Dans notre cas, puisque les deux organisations ont envoyé une approbation, la définition du chaincode est prête à être validée (commit) sur le channel.

```
commitChaincodeDefinition() {
    set -x
    bin/peer lifecycle chaincode commit -o localhost:7050 \
            --ordererTLSHostnameOverride orderer.example.com --tls \
            --cafile "$ORDERER_CA" --channelID $CHANNEL_NAME --name
             ↪  ${CC_NAME} \
            --peerAddresses localhost:7051 --tlsRootCertFiles $PEER0_ORG1_CA
             ↪  \
            --peerAddresses localhost:9051 --tlsRootCertFiles $PEER0_ORG2_CA
             ↪  \
            --version ${CC_VERSION} --sequence ${CC_SEQUENCE}
             ↪  ${INIT_REQUIRED} ${CC_END_POLICY} ${CC_COLL_CONFIG} >&log.txt
    res=$?
    { set +x; } 2>/dev/null
    cat log.txt
    verifyResult $res "Chaincode definition commit failed on peer0.hospital
     ↪  on channel '$CHANNEL_NAME' failed"
    successln "Chaincode definition committed on channel '$CHANNEL_NAME'"
}
```

6.2.5 Chaincode invoke

Une fois que la définition du chaincode a été commit sur le channel, le chaincode est prêt à être invoqué par l'application client. La commande suivante peut être utilisée pour générer une série d'assets dans la ledger

```
chaincodeInvoke() {
    setGlobalsForPeer0Org1

    bin/peer chaincode invoke -o localhost:7050 \
        --ordererTLSHostnameOverride orderer.example.com \
        --tls $CORE_PEER_TLS_ENABLED \
        --cafile $ORDERER_CA \
        -C $CHANNEL_NAME -n ${CC_NAME} \
        --peerAddresses localhost:7051 --tlsRootCertFiles $PEER0_ORG1_CA \
        --peerAddresses localhost:9051 --tlsRootCertFiles $PEER0_ORG2_CA \
        -c '{"function": "InitLedger","Args":[]}'
}
```

6.2.6 Conclusion

Nous avons parcouru les étapes majeures du déploiement d'un chaincode au sein d'un channel. Dans la documentation , on retrouve l'exemple présenté ci-dessous avec une série d'explications et de commandes supplémentaires. Une section est également dédié à la mise à jour du chaincode.

Nous avons passé en revue les principales étapes du déploiement d'un chaincode au sein d'un channel. La documentation suivante contient un exemple détaillé avec des explications et des commandes supplémentaires : Lien de documentation. Une section est également dédiée à la mise à jour du chaincode, vous fournissant ainsi un guide complet pour gérer toutes les étapes du cycle de vie d'un chaincode dans un réseau Hyperledger Fabric.