**Setting the Stage with a Business Scenario**

The first two chapters were focused on setting the stage and defining the landscape of a blockchain project. We now understand how the technology works within a business framework and how the various Hyperledger projects aim to solve the problem of time and trust.

With an understanding of the components that make up Hyperledger Fabric, we will now delve into application design and implementation considerations. The next few chapters will take you through the steps of creating your very own smart contract and then integrating it to an application.

In order to make these exercises relevant, we will leverage a business use case with its roots in some older civilizations: trading and letters of credit.

The chapter's objective will be to introduce the business concept of letter of credit, walk you through the sample scenario we selected, and conclude by setting up our development environment.

In this chapter, we will:

* Explore letters of credit
* Review our simplified business scenario
* Set up our development environment

**Trading and letter of credit**

Step back in history to a time when merchants traveled across continents to buy cloth in one country to sell in another country. As a Florentine wool merchant, you might make a journey to Amsterdam to buy fine wool in that newly formed city-state, whose port collected resources from the whole of Northern Europe and beyond. You could then transport the wool to Florence, where it could be sold to tailors making fine garments for their wealthy clients. We're talking about 1300 AD—a time when it was not safe to carry gold or other precious metals as a form of currency to buy and sell goods. What was necessary was a form of currency that worked across country boundaries, one that could be used in Amsterdam and Florence, or anywhere!  
  
Marco Polo had been to China and had seen how commerce was conducted in that thriving economy. At the heart of the successful Khan empire were advanced financial techniques that we would recognize today. Fiat currencies, paper money, promissory notes, and letters of credit all arrived in Europe by way of China. Marco Polo brought these ideas back to Europe—they helped form and grow a merchant banking industry for a Europe emerging after the fall of the Roman Empire.

**The importance of trust in facilitating trade**

Our Florentine merchant could now contact his banker to say that he wanted to buy wool in Amsterdam, and the bank would in return give him a letter of credit, in exchange for payment on account. This letter could have various stipulations, such as the maximum amount for the trade, how it would be paid (at once or in parts), what goods it could be used for, and so forth. The merchant would now travel to Amsterdam, and after selecting wool from a wool merchant, he would offer the letter or credit as payment. The Amsterdam merchant would happily exchange the wool for the letter because Florentine bankers were famed throughout Europe as being trustworthy when it came to money. The Amsterdam merchant could bring the letter of credit to his banker, who in turn would credit their account. Of course, the Florentine and Amsterdam bankers charged their respective clients—the merchants—for this service! It was good for everyone.

Periodically, Amsterdam bankers and the Florentine bankers would meet up to settle their accounts, but this was of no importance to the wool trader and wool merchant. Effectively, what was happening was that the Florentine and Amstel merchants were using the trust between their respective bankers to establish a trust relationship with each other—a very sophisticated idea when you think about it. This is why the letter of credit process remains a fundamental way of conducting business worldwide to this day.

**The letter of credit process today**

However, over time, due to massive globalization of trade and the explosion of the financial industry, the number of financial institutions involved in the letter of credit process has exploded! Nowadays, there could be over 20 intermediary financial institutions involved in the process. This requires coordination of many people and systems, resulting in excessive time, cost, and risk throughout the process for both merchants and banks alike.

The promise of blockchain is to provide a logically singular but physically distributed system that provides a platform for a low-friction letter of credit process. The characteristics of such a system would include greater transparency, timeliness, and automation (resulting in lower cost), and new features such as incremental payment.

**Business scenario and use case**

International trade includes the kinds of situations that illustrate the inefficiencies and distrust in real-world processes that blockchains were designed to mitigate. So, we have selected an element of an import-export scenario with simplified versions of transactions carried out in the real world as our canonical use case for practical exercises in the next few chapters.

**Overview**

The scenario we will describe involves a simple transaction: the sale of goods from one party to another. This transaction is complicated by the fact that the buyer and the seller live in different countries, so there is no common trusted intermediary to ensure that the exporter gets the money he was promised and the importer gets the goods. Such trade arrangements in today's world rely on:

* Intermediaries that facilitate payments and physical transfer of goods
* Processes that have evolved over time to enable exporters and importers to hedge their bets and reduce the risks involved

**Real-world processes**

The intermediaries that facilitate payment are the respective banks of the exporter and the importer. In this case, the trade arrangement is fulfilled by the trusted relationships between a bank and its client, and between the two banks. Such banks typically have international connections and reputations to maintain. Therefore, a commitment (or promise) by the importer's bank to make a payment to the exporter's bank is sufficient to trigger the process. The goods are dispatched by the exporter through a reputed international carrier after obtaining regulatory clearances from the exporting country's government.

Proof of delivery to the carrier is sufficient to clear payment from the importer's bank to the exporter's bank, and such clearance is not contingent on the goods reaching their intended destination (it is assumed that the goods are insured against loss or damage in transit.) The promise made by the importer's bank to pay the exporter's bank specifies a list of documents that are required as proof of dispatch, and the precise method of payment to be made immediately or over a period. Various regulatory requirements must be fulfilled by the exporter before getting documentary clearances that allow them to hand off the goods to the carrier.

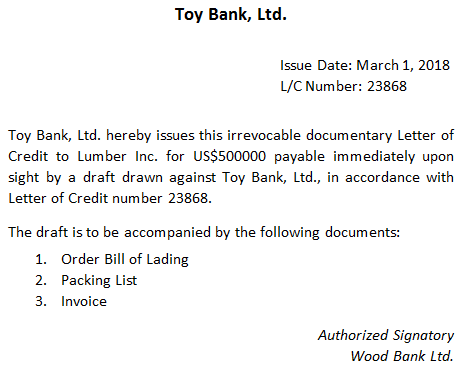
**Simplified and modified processes**

Our use case will follow a simplified version of the preceding process, with certain variations to demonstrate the value of blockchain in facilitating this trade. A payment promise is made by the importer's bank to the exporter's bank in two installments. The exporter obtains a clearance certificate from the regulatory authority, hands off the goods to the carrier, and then obtains a receipt. The production of the receipt triggers the first payment installment from the importer's bank to the exporter's bank. When the shipment has reached the destination port, the second and final payment installments are made, and the process concludes.

**Terms used in trade finance and logistics**

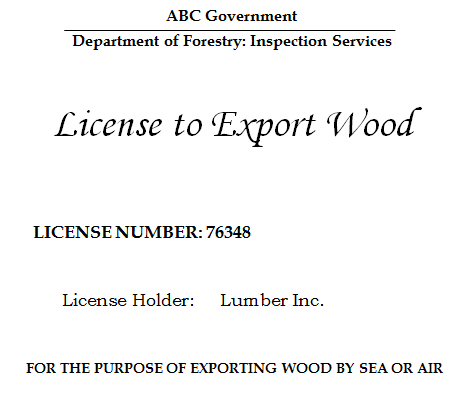
The following terms are used to refer to certain instruments and artifacts that are in play in our trade scenario. The application we will build in this chapter uses very simplified forms of these instruments:

* **Letter of credit**: As we have seen at the beginning of the chapter, this refers to a bank's promise to pay an exporter upon presentation of documentary proof of goods having been shipped. Called **L/C** for short, this document is issued by the importer's bank at the request of its client: the importer. The L/C states the list of documents that constitute proof of shipment, the amount to be paid, and the beneficiary (the exporter in our case) of that amount. A sample L/C is illustrated in the following screenshot:

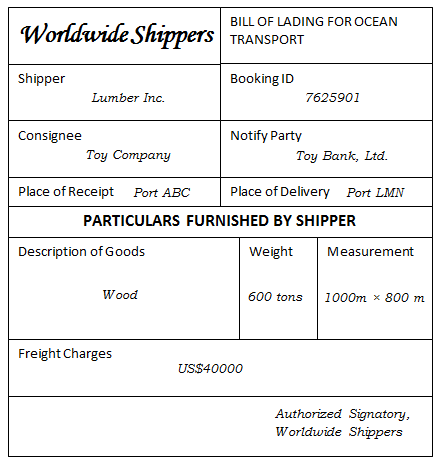


We will introduce small variations in our use case to make this instrument comprehensible to the reader.

* Firstly, the L/C will be issued to the exporter's bank rather than directly to the exporter.
* Secondly, the L/C states that payment will be made in two identical installments, the first upon production of two documents and the second upon the goods reaching the destination.
* **Export license**: This refers to the approval given by the regulatory authority in the exporter's country for the shipment of the specified goods. In this book, we will refer to it as E/L for short. A sample E/L is illustrated in the following screenshot:



* **Bill of lading**: This is a document issued by the carrier to the exporter once it takes possession of the shipment. Called B/L for short, it simultaneously serves as a receipt, a contract obliging the carrier to transport the goods to a specified destination in return for a fee, and a title of ownership of the goods. This document is also listed in the L/C and serves as proof of shipment that will automatically trigger a payment clearance. A sample B/L is illustrated in the following screenshot:

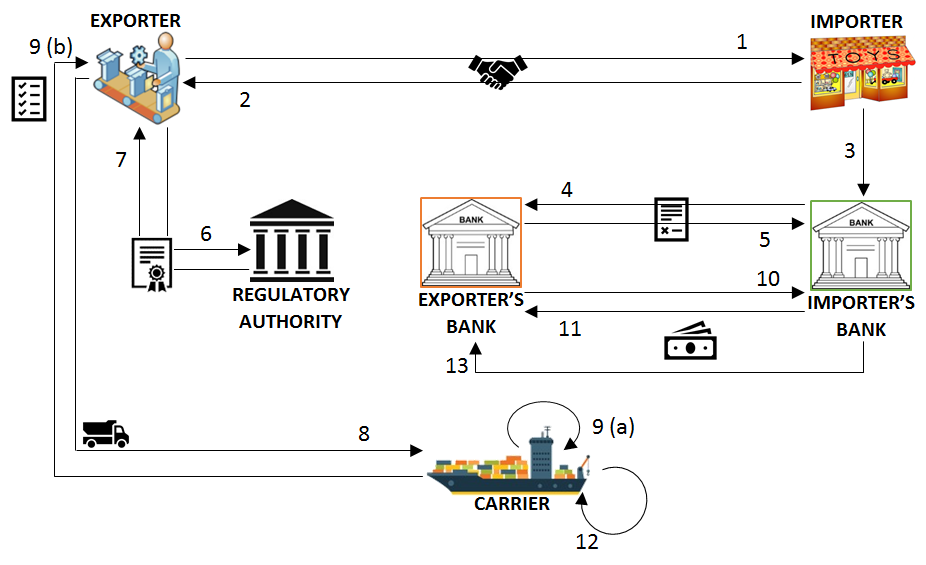


**Shared process workflow**

Every instance of a test case scenario presented in this chapter takes a long period of time to complete, involves interactions among different sets of entities at different times, and has many different moving parts that are difficult to keep track of. We hope to simplify this process using our workflow. Implemented on a blockchain, the sequences of transactions described in the following steps (and illustrated in the following diagram) can be carried out in an irrevocable and non-repudiable manner. In this sequence of events, we assume a straight, linear narrative where parties are in agreement with each other and nothing untoward happens; guards are built in the process only to catch errors.

The transactions in our workflow are as follows:

1. Importer requests goods from the exporter in exchange of money
2. Exporter accepts the trade deal
3. Importer asks its bank for an L/C in favor of the exporter
4. The importer's bank supplies an L/C in favor of the exporter, and payable to the latter's bank
5. The exporter's bank accepts the L/C on behalf of the exporter
6. Exporter applies for an E/L from the regulatory authority
7. Regulatory authority supplies an E/L to the exporter
8. Exporter prepares a shipment and hands it off to the carrier
9. The carrier accepts the goods after validating the E/L, and then supplies a B/L to the exporter
10. The exporter's bank claims half the payment from the importer's bank
11. The importer's bank transfers half the amount to the exporter's bank
12. The carrier ships the goods to the destination
13. The importer's bank pays the remaining amount to the exporter's bank



**Shared assets and data**

The participants in the previous workflow must have some information in common that gives them a view into the trade arrangement and its progress at any given moment.

The following is a table of the assets owned by the participants, which are shared with each other to drive the process from one stage to the next. This includes documentary and monetary assets:

|  |  |
| --- | --- |
| **Asset type** | **Asset attributes** |
| Letter of credit | ID, issue date, expiration date, issuer, beneficiary, amount, and a list of documents |
| Bill of lading | ID, shipper (exporter), consignee (importer), party to notify (importer's bank), places of receipt and delivery, description of goods, and freight amount |
| Export license | ID, issue date, expiration date, beneficiary, license holder, and description of goods |
| Payment | Amount in standard currency units |

The following are the data elements that circumscribe the options available to participants in each stage:

|  |  |
| --- | --- |
| **Data type** | **Data attributes** |
| Trade agreement | Requested by importer and accepted by exporter |
| Letter of credit | Requested by importer, issued by importer's bank, and accepted by exporter's bank |
| Export license | Requested by exporter and issued by regulatory authority |
| Shipment | Prepared by exporter, accepted by carrier, and current position or location |

**Participants' roles and capabilities**

There are six categories of participants in our scenario: which are exporter, importer, exporter's bank, importer's bank, carrier, and regulatory authority. The terms in this set refer to the roles an entity can assume in a trade deal; for example, a company exporting goods in one instance may be an importer in another. The capabilities and restrictions of each role are also detailed in the following list:

* Only an importer may apply for an L/C
* Only an importer's bank may supply an L/C
* Only an exporter's bank may accept an L/C
* Only an exporter may request an E/L
* Only a regulatory authority may supply an E/L
* Only an exporter may prepare a shipment
* Only a carrier may supply a B/L
* Only a carrier may update a shipment location
* Only an importer's bank may send money, and only an exporter's bank may receive money

**Benefits of blockchain applications over current real-world processes**

The risks inherent in transferring goods or making payments in the absence of safeguards (such as a trusted mediator) inspired the involvement of banks and led to the creation of the letter of credit and bill of lading. A consequence of these processes was not just additional cost (banks charge commission to issue letters of credit) or additional overhead. Applying and waiting for export licenses to be awarded also increases the turnaround time. In an ideal trade scenario, only the process of preparing and shipping the goods would take time. Recently, the adoption of SWIFT messaging over manual communication has made the document application and collection processes more efficient, but it has not fundamentally changed the game. A blockchain, on the other hand, with its (almost) instantaneous transaction commitments and assurance guarantees, opens possibilities that did not previously exist.

As an example, the one variation we introduced in our use case was payment by installments, which cannot be implemented in the legacy framework because there is no guaranteed way of knowing and sharing information about a shipment's progress. Such a variation would be deemed too risky in this case, which is why payments are linked purely to documentary evidence. By getting all participants in a trade agreement on a single blockchain implementing a common smart contract, we can provide a single shared source of truth that will minimize risk and simultaneously increase accountability.

In subsequent chapters, we will demonstrate in detail how our use case is implemented on the Hyperledger Fabric and Composer platforms. The reader will be able to appreciate both the simplicity and elegance of the implementation, which can then be used as a guide for other applications to revamp their archaic processes using this exciting new technology. However, before jumping into the code, we will look at the design of a Hyperledger network and we will set up our development environment.

**Setting up the development environment**

As you already know by now, an instance of a Hyperledger Fabric blockchain is referred to as a channel, which is a log of transactions linked to each other in a cryptographically secure manner. To design and run a blockchain application, the first step is to determine how many channels are required. For our trade application, we will use one channel, which will maintain the history of trades carried out among the different participants.

*A Fabric peer may belong to multiple channels, which from the application's perspective will be oblivious to each other, but which help a single peer run transactions in different applications on behalf of its owners (or clients). A channel may run multiple smart contracts, each of which may be an independent application or linked together in a multi-contract application. In this chapter, and in this book, we will walk the reader through the design of a single-channel, single-contract application for simplicity's sake. It is up to the reader to design more complex applications, relying on the information provided in this book as well as in the Fabric documentation.*

Before we delve into the mechanics of setting up our system to install an application and run transactions on our smart contract, we will describe how to create and launch a network on which the application will be installed. A sample network structure will be used to illustrate trade operations throughout this chapter (in [Chapter 9](https://www.safaribooksonline.com/library/view/hands-on-blockchain-with/9781788994521/dd98c2ee-1318-45c4-a118-9955bacf2995.xhtml), Life in a Blockchain Network, you will see how this sample network can be modified as the requirements change and evolve).

**Designing a network**

The first step in determining a Hyperledger Fabric network structure for one's application is listing the participating organizations. Logically, an organization is a security domain and a unit of identity and credentials. It governs one or more network peers, and depends on a **membership service provider** (**MSP**) to issue identities and certificates for the peers as well as clients for smart contract access privileges. The ordering service, which is the cornerstone of a Fabric network, is typically assigned its own organization. The following diagram illustrates a typical peer network structure with clients, MSPs, and logical organization groupings.

The criterion for the approval of a transaction (or invocation) is an endorsement policy (which we will revisit later in this chapter). It is framed in terms of the organizations that are participating in the application network, and not the peers themselves:

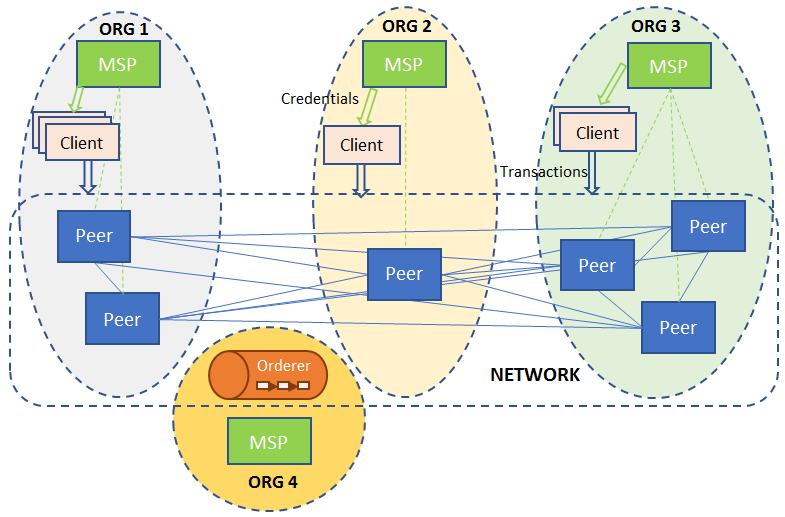


Figure ? Blockchain network with peers distributed among organizations, and clients obtaining credentials from organizations to submit queries and invocations to the chaincode

The set of peers, the organizations they belong to, and the membership service providers serving each organization must be decided beforehand so that the appropriate services can be installed and run on those machines.

Our sample trade network will consist of four organizations, representing the exporter, importer, carrier, and regulator, respectively. The latter two represent the carrier and regulator entities, respectively. The exporter organization, however, represents both the exporting entity and its bank. Similarly, the importer organization represents the importing entity and its bank. Grouping entities with parties they trust into a single organization makes sense from both the perspective of security and cost. Running a Fabric peer is a heavy and costly business, so it is sufficient for a bank, which likely has more resources and a large clientele, to run such a peer on behalf of itself and its clients. A trading entity obtains the right to submit transactions or read the ledger state from its organizations in the role of a client. Our blockchain network therefore needs four peers, each belonging to a different organization. Apart from the peers, our network consists of one MSP for each of the four organizations, and an ordering service running in solo mode.

*In a production application, the ordering service should be set up as a Kafka cluster on Zookeeper, but for the purpose of demonstrating how to build a blockchain application, the ordering service can be treated as a black box.*

The ordering service belongs to its own separate organization with an MSP. The Organizations with their MSPs, peers, and clients of our trading network are illustrated in the following diagram:

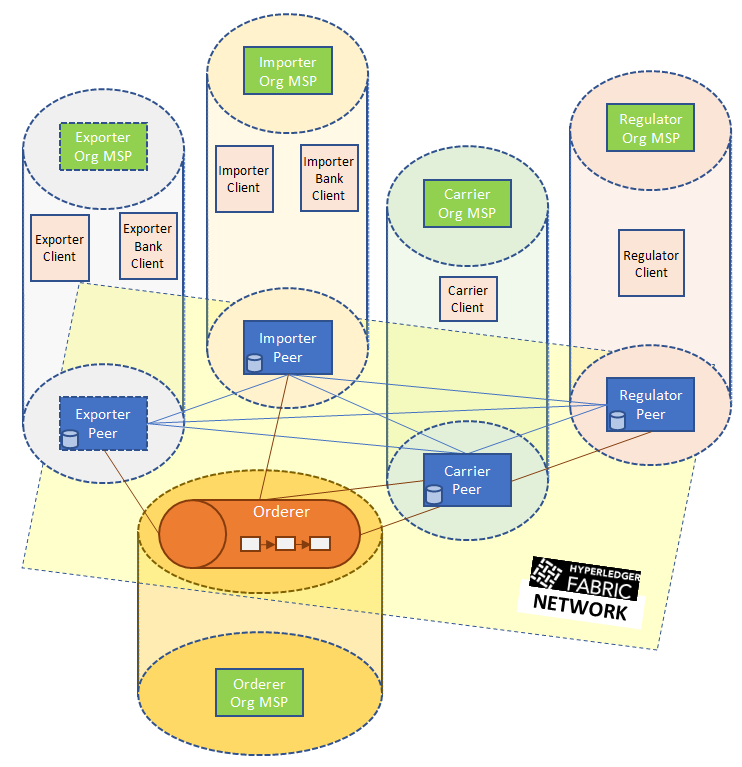


Figure 3.2: A trade network with peers, an orderer, and clients in their respective organizations

The reader may wonder how, if a trading party and its banker belong to the same organization, the application can distinguish the two (the exporter from the exporter’s bank, and the importer from importer’s bank) for the purpose of controlling access to the smart contract and ledger. Two ways of doing this are as follows:

Embedding access control logic in the middleware and application layers (which we will describe later in this chapter), whereby users can be distinguished by their IDs (or login names) and an access control list mapping IDs to permitted chaincode functions is maintained.

Having an organization’s MSP, acting as a CA server, embed distinguishing attributes within the certificates it issues to members of an organization. The access control logic can be implemented in the middleware or even in the chaincode to parse the attributes and permit or disallow an operation as per application policy.

These mechanisms are not implemented in our application, in which bankers and clients are indistinguishable to the smart contract and the middleware layers. But the reader may treat this as an exercise, which should be straightforward for someone skilled at developing secure client-server applications.

**Installing prerequisites**

With the design of the network in hand, lets install the pre-requisite tools:

1. Ensure that you have the latest version of:
   * Docker using <https://docs.docker.com/install/>
   * Docker-Compose using: <https://docs.docker.com/compose/install/>
2. We will be using GitHub to share the source code of our tutorial. To access GitHub, the Git client needs to be installed and configured with authentication to GitHub. For more information, visit GitHub's official website at <https://help.github.com/articles/set-up-git/>.
3. Install the software required for the business network example: <https://hyperledger.github.io/composer/latest/installing/installing-prereqs>.  
   The instructions above are for the Mac and Linux. Note that when using Windows, we recommend the use of a solution like Vagrant to run the development environment in a virtual machine.
4. Fabric is implemented in the Go language. Note that:
   * Go is syntactically similar to C++
   * We will also use Go to write chaincodes
   * Go can be installed from <https://golang.org/>
5. Next, we need to set up our environmental variables.

GOPATH points to a workspace for the go source code, for example:

$ export GOPATH=$HOME/go

PATH needs to include the Go bin directory used to store libraries and executables, as we can see in the following snippet:

$ export PATH=$PATH:$GOPATH/bin

**Forking and cloning the trade-finance-logistics repository**

Now we need to get our own copy of the original source code by forking the repository on GitHub. Then, we can clone the source code into a local machine directory with the following steps:

1. **In GitHub, navigate to the following repository**: <https://github.com/HyperledgerHandsOn/trade-finance-logistics>
2. **Fork the repository**: Use the Fork button at the top-right corner of the page to create a copy of the source code to your account
3. **Get the clone URL**: Navigate to your fork of the trade-finance-logistics repository. Click on the Clone or download button, and copy the URL
4. **Clone the repository**: In the Go workspace, clone the repository as follows:

$ cd $GOPATH/src

$ git clone https://github.com/YOUR-USERNAME/trade-finance-logistics

We now have a local copy of all the trade-finance-logistics tutorial materials.

**Creating and running a network configuration**

The code to configure and launch our network can be found in the network folder in our repository (this is an adaptation of [fabric-samples/first-network](https://github.com/hyperledger/fabric-samples/tree/master/first-network)). For this exercise, we will run the entire network on a single physical or virtual machine, with the various network elements running in suitably configured Docker containers. It is assumed that the reader has a basic familiarity with containerization using Docker and configurations using Docker-compose. Once the prerequisites listed in the previous section are met, it is sufficient to run the commands in that section without any extra knowledge or configuration required of the reader.

<https://github.com/hyperledger/fabric-samples/tree/master/first-network>

# Preparing the network

We need to perform the following steps before generating network cryptographic material:

1. Clone the Fabric (<https://github.com/hyperledger/fabric/>) source code repository
2. Run make docker to build Docker images for the peers and orderers
3. Run make configtxgen cryptogen to generate the necessary tools to run the network creation commands described in this section
4. Clone the Fabric-CA (<https://github.com/hyperledger/fabric-ca>) source code repository
5. Run make docker to build the Docker images for the MSPs

# Generating network cryptographic material

The first step in the configuration of a network involves the creation of certificates and signing keys for the MSP of each peer and orderer organization, and for TLS-based communication. We also need to create certificates and keys for each peer and orderer node to be able to communicate with each other and with their respective MSPs. The configuration for this must be specified in a crypto-config.yaml file in the network folder in our code repository. This file contains the organization structure (see more details in the channel artifacts configuration section later), the number of peers in each organization, and the default number of users in an organization for whom certificates and keys must be created (note that an admin user is created by default). As an example, see the definition of the Importer’s organization in the file as follows:

PeerOrgs:  
- Name: ImporterOrg  
 Domain: importerorg.trade.com  
 EnableNodeOUs: true  
 Template:  
 Count: 1  
 Users:  
 Count: 2

This configuration indicates that the organization labeled ImporterOrg will contain one peer. Two non-admin users will also be created. The organization domain name to be used by the peer is also defined.

To generate cryptographic material for all the organizations, run the cryptogen command as follows:

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

The output is saved to the crypto-config folder.

**Generating channel artifacts**

To create a network according to an organization's structure, and to bootstrap a channel, we will need to generate the following artifacts:

* A genesis block, containing organization-specific certificates that serve to initialize the Fabric blockchain.
* Channel configuration information.
* Anchor peer configurations for each organization. An anchor peer serves as a fulcrum within an organization, for cross-organization ledger syncing using the Fabric gossip protocol.

Like the crypto-config.yaml file, channel properties are specified in a file labeled configtx.yaml, which in our source code can be found in the network folder. The high-level organization of our trade network can be found in the Profiles section as follows:

Profiles:  
 FourOrgsTradeOrdererGenesis:  
 Capabilities:  
 <<: \*ChannelCapabilities  
 Orderer:  
 <<: \*OrdererDefaults  
 Organizations:  
 - \*TradeOrdererOrg  
 Capabilities:  
 <<: \*OrdererCapabilities  
 Consortiums:  
 TradeConsortium:  
 Organizations:  
 - \*ExporterOrg  
 - \*ImporterOrg  
 - \*CarrierOrg  
 - \*RegulatorOrg  
 FourOrgsTradeChannel:  
 Consortium: TradeConsortium  
 Application:  
 <<: \*ApplicationDefaults  
 Organizations:  
 - \*ExporterOrg  
 - \*ImporterOrg  
 - \*CarrierOrg  
 - \*RegulatorOrg  
 Capabilities:  
 <<: \*ApplicationCapabilities

As we can see, the channel we are going to create is named FourOrgsTradeChannel, which is defined in the profile. The four organizations participating in this channel are labeled ExporterOrg, ImporterOrg, CarrierOrg, and RegulatorOrg, each of which refers to a subsection defined in the Organizations section. The orderer belongs to its own organization called TradeOrdererOrg. Each organization section contains information about its MSP (ID as well as the location of the cryptographic material, such as keys and certificates), and the hostname and port information for its anchor peers. As an example, the ExporterOrg section contains the following:

- &ExporterOrg  
 Name: ExporterOrgMSP  
 ID: ExporterOrgMSP  
 MSPDir: crypto-config/peerOrganizations/exporterorg.trade.com/msp  
 AnchorPeers:  
 - Host: peer0.exporterorg.trade.com  
 Port: 7051

As you can see, the MSPDir variable (representing a folder) in this specification references the cryptographic material we generated earlier using the cryptogen tool.

To generate the channel artifacts, we use the configtxgen tool. To generate the genesis block (which will be sent to the orderer during network bootstrap), run the following command from the network folder:

configtxgen -profile FourOrgsTradeOrdererGenesis -outputBlock ./channel-artifacts/genesis.block

The FourOrgsTradeOrdererGenesis keyword corresponds to the profile name in the Profiles section. The genesis block will be saved in the genesis.block file in the channel-artifacts folder. To generate the channel configuration, run the following code:

configtxgen -profile FourOrgsTradeChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -channelID tradechannel

The channel we will create is named tradechannel, and its configuration is stored in channel-artifacts/channel.tx. To generate the anchor peer configuration for the exporter organization, run:

configtxgen -profile FourOrgsTradeChannel -outputAnchorPeersUpdate ./channel-artifacts/ExporterOrgMSPanchors.tx -channelID tradechannel -asOrg ExporterOrgMSP

The same process should be repeated for the other three organizations, while changing the organization names in the preceding command.

*The environment variable FABRIC\_CFG\_PATH must be set to point to the folder that contains the configtx.yaml file in order for the configtxgen tool to work. The script file trade.sh (which we will use later) contains the following line to ensure that the YAML file is loaded from the folder in which the command is run:  
  
export FABRIC\_CFG\_PATH=${PWD}*

# Generating the configuration in one operation

For convenience, the trade.sh script is configured to generate the channel artifacts as well as the cryptographic material using the commands and configuration files described previously. Just run the following command from within the network folder:

./trade.sh generate -c tradechannel

Although you can specify any channel name here, note that the configurations used to develop the middleware later in this chapter will depend on that name.

**Composing a sample trade network**

The last command also has the effect of generating a network configuration file, docker-compose-e2e.yaml, which is used to start the network as a set of Docker containers using the docker-compose tool. The file itself depends on the statically configured files base/peer-base.yaml and base/docker-compose-base.yaml. These files collectively specify services and their attributes, and enable us to run them all in one go within Docker containers, rather than having to manually run instances of these services on one or more machines. The services we need to run are as follows:

* Four instances of a Fabric peer, one in each organization
* One instance of a Fabric orderer
* Five instances of a Fabric CA, corresponding to the MSPs of each organization

Docker images for each can be obtained from the Hyperledger project on Docker Hub (<https://hub.docker.com/u/hyperledger/>), with the images being hyperledger/fabric-peer, hyperledger/fabric-orderer, hyperledger/fabric-ca for peers, orderers, and MSPs, respectively.

The base configuration of a peer can be as follows (see base/peer-base.yaml):

peer-base:  
image: hyperledger/fabric-peer:$IMAGE\_TAG  
environment:  
 - CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock  
 - CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=${COMPOSE\_PROJECT\_NAME}\_trade  
 - CORE\_LOGGING\_LEVEL=INFO  
 - CORE\_PEER\_TLS\_ENABLED=true  
 - CORE\_PEER\_GOSSIP\_USELEADERELECTION=true  
 - CORE\_PEER\_GOSSIP\_ORGLEADER=false  
 - CORE\_PEER\_PROFILE\_ENABLED=true  
 - CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/fabric/tls/server.crt  
 - CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/fabric/tls/server.key  
 - CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/fabric/tls/ca.crt  
working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer  
command: peer node start

Fabric configuration parameters can be set here, but if you use the pre-built Docker image for fabric-peer, the defaults are sufficient to get a peer service up and running. The command to run the peer service is specified in the last line of the configuration as peer node start; if you wish to run a peer by downloading the Fabric source and building it on your local machine, this is the command you will have to run (see [Chapter 4](https://www.safaribooksonline.com/library/view/hands-on-blockchain-with/9781788994521/e71c03af-8211-4d1f-a6ac-4b8b920e429a.xhtml), Designing a Data and Transaction Model with Golang, for examples). Also make sure you configure the logging level appropriately using the CORE\_LOGGING\_LEVEL variable. In our configuration, the variable is set to INFO, which means that only informational, warning, and error messages will be logged. If you wish to debug a peer and need more extensive logging, you can set this variable to DEBUG.

*The IMAGE\_TAG variable is set to latest in the .env file in the network folder, though you can set a specific tag if you wish to pull older images.*

Furthermore, we need to configure the hostnames and ports for each peer, and sync the cryptographic material generated (using cryptogen) to the container filesystem. The peer in the exporter organization is configured in base/docker-compose-base.yaml as follows: