**Deploying a production network**

This deployment guide is a high level overview of the proper sequence for setting up production Fabric network components, in addition to best practices and a few of the many considerations to keep in mind when deploying. Note that this topic will discuss “setting up the network” as a holistic process from the perspective of a single individual. More likely than not, real world production networks will not be set up by a single individual but as a collaborative effort directed by several individuals (a collection of banks each setting up their own components, for example) instead.

The process for deploying a Fabric network is complex and presumes an understanding of Public Key Infrastructure and managing distributed systems. If you are a smart contract or application developer, you should not need this level of expertise in deploying a production level Fabric network. However, you might need to be aware of how networks are deployed in order to develop effective smart contracts and applications.

If all you need is a development environment to test chaincode, smart contracts, and applications against, check out [Using the Fabric test network](https://hyperledger-fabric.readthedocs.io/en/latest/test_network.html). It includes two organizations, each owning one peer, and a single ordering service organization that owns a single ordering node. **This test network is not meant to provide a blueprint for deploying production components, and should not be used as such, as it makes assumptions and decisions that production deployments will not make.**

The guide will give you an overview of the steps of setting up production components and a production network:

* [Step one: Decide on your network configuration](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-step-one-decide-on-your-network-configuration)
* [Step two: Set up a cluster for your resources](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-step-two-set-up-a-cluster-for-your-resources)
* [Step three: Set up your CAs](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-step-three-set-up-your-cas)
* [Step four: Use the CA to create identities and MSPs](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-step-four-use-the-ca-to-create-identities-and-msps)
* [**Step five: Deploy nodes**](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-step-five-deploy-nodes)
  + [Create a peer](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-create-a-peer)
  + [Create an ordering node](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-create-an-ordering-node)

**Step one: Decide on your network configuration**

The structure of a blockchain network must be dictated by the use case. These fundamental business decisions will vary according to your use case, but let’s consider a few scenarios.

In contrast to development environments or proofs of concept, security, resource management, and high availability become a priority when operating in production. How many nodes do you need to satisfy high availability, and in what data centers do you wish to deploy them in to satisfy both the needs of disaster recovery and data residency? How will you ensure that your private keys and roots of trust remain secure?

In addition to the above, here is a sampling of the decisions you will need to make before deploying components:

* **Certificate Authority configuration**. As part of the overall decisions you have to make about your peers (how many, how many on each channel, and so on) and about your ordering service (how many nodes, who will own them), you also have to decide on how the CAs for your organization will be deployed. Production networks should be using Transport Layer Security (TLS), which will require setting up a TLS CA and using it to generate TLS certificates. This TLS CA will need to be deployed before your enrollment CA. We’ll discuss this more in [Step three: Set up your CAs](https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html#dg-step-three-set-up-your-cas).
* **Use Organizational Units or not?** Some organizations might find it necessary to establish Organizational Units to create a separation between certain identities and MSPs created by a single CA.
* **Database type.** Some channels in a network might require all data to be modeled in a way [CouchDB as the State Database](https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_as_state_database.html) can understand, while other networks, prioritizing speed, might decide that all peers will use LevelDB. Note that channels should not have peers that use both CouchDB and LevelDB on them, as the two database types model data slightly differently.
* **Channels and private data.** Some networks might decide that [Channels](https://hyperledger-fabric.readthedocs.io/en/latest/channels.html) are the best way to ensure privacy and isolation for certain transactions. Others might decide that a single channel, along with [Private data](https://hyperledger-fabric.readthedocs.io/en/latest/private-data/private-data.html), better serves their need for privacy.
* **Container orchestration.** Different users might also make different decisions about their container orchestration, creating separate containers for their peer process, logging for the peer, CouchDB, gRPC communications, and chaincode, while other users might decide to combine some of these processes.
* **Chaincode deployment method.** Users now have the option to deploy their chaincode using either the built in build and run support, a customized build and run using the [External Builders and Launchers](https://hyperledger-fabric.readthedocs.io/en/latest/cc_launcher.html), or using an [Chaincode as an external service](https://hyperledger-fabric.readthedocs.io/en/latest/cc_service.html).
* **Using firewalls.** In a production deployment, components belonging to one organization might need access to components from other organizations, necessitating the use of firewalls and advanced networking configuration. For example, applications using the Fabric SDK require access to all endorsing peers from all organizations and the ordering services for all channels. Similarly, peers need access to the ordering service on the channels that they are receiving new blocks from.

However and wherever your components are deployed, you will need a high degree of expertise in your management system of choice (such as Kubernetes) in order to efficiently operate your network. Similarly, the structure of the network must be designed to fit the business use case and any relevant laws and regulations government of the industry in which the network will be designed to function.

This deployment guide will not go through every iteration and potential network configuration, but does give common guidelines and rules to consider.

**Step two: Set up a cluster for your resources**

Generally speaking, Fabric is agnostic to the method used to deploy and manage it. It is possible, for example, to deploy and manage a peer from a laptop. For a number of reasons, this is likely to be unadvisable, but there is nothing in Fabric that prohibits it.

As long as you have the ability to deploy containers, whether locally (or behind a firewall), or in a cloud, it should be possible to stand up components and connect them to each other. However, Kubernetes features a number of helpful tools that have made it a popular container management platform for deploying and managing Fabric networks. For more information about Kubernetes, check out [the Kubernetes documentation](https://kubernetes.io/docs). This topic will mostly limit its scope to the binaries and provide instructions that can be applied when using a Docker deployment or Kubernetes.

However and wherever you choose to deploy your components, you will need to make sure you have enough resources for the components to run effectively. The sizes you need will largely depend on your use case. If you plan to join a single peer to several high volume channels, it will need much more CPU and memory than a peer a user plans to join to a single channel. As a rough estimate, plan to dedicate approximately three times the resources to a peer as you plan to allocate to a single ordering node (as you will see below, it is recommended to deploy at least three and optimally five nodes in an ordering service). Similarly, you should need approximately a tenth of the resources for a CA as you will for a peer. You will also need to add storage to your cluster (some cloud providers may provide storage) as you cannot configure Persistent Volumes and Persistent Volume Claims without storage being set up with your cloud provider first.

By deploying a proof of concept network and testing it under load, you will have a better sense of the resources you will require.

**Managing your infrastructure**

The exact methods and tools you use to manage your backend will depend on the backend you choose. However, here are some considerations worth noting.

* Using secret objects to securely store important configuration files in your cluster. For information about Kubernetes secrets, check out [Kubernetes secrets](https://kubernetes.io/docs/concepts/configuration/secret/). You also have the option to use Hardened Security Modules (HSMs) or encrypted Persistent Volumes (PVs). Along similar lines, after deploying Fabric components, you will likely want to connect to a container on your own backend, for example using a private repo in a service like Docker Hub. In that case, you will need to code the login information in the form of a Kubernetes secret and include it in the YAML file when deploying components.
* Cluster considerations and node sizing. In step 2 above, we discussed a general outline for how to think about the sizings of nodes. Your use case, as well as a robust period of development, is the only way you will truly know how large your peers, ordering nodes, and CAs will need to be.
* How you choose to mount your volumes. It is a best practice to mount the volumes relevant to your nodes external to the place where your nodes are deployed. This will allow you to reference these volumes later on (for example, restarting a node or a container that has crashed) without having to redeploy or regenerate your crypto material.
* How you will monitor your resources. It is critical that you establish a strategy and method for monitoring the resources used by your individual nodes and the resources deployed to your cluster generally. As you join your peers to more channels, you will need likely need to increase its CPU and memory allocation. Similarly, you will need to make sure you have enough storage space for your state database and blockchain.

**Step three: Set up your CAs**

The first component that must be deployed in a Fabric network is a CA. This is because the certificates associated with a node (not just for the node itself but also the certificates identifying who can administer the node) must be created before the node itself can be deployed. While it is not necessary to use the Fabric CA to create these certificates, the Fabric CA also creates MSP structures that are needed for components and organizations to be properly defined. If a user chooses to use a CA other than the Fabric CA, they will have to create the MSP folders themselves.

* One CA (or more, if you are using intermediate CAs — more on intermediate CAs below) is used to generate (through a process called “enrollment”) the certificates of the admin of an organization, the MSP of that organization, and any nodes owned by that organization. This CA will also generate the certificates for any additional users. Because of its role in “enrolling” identities, this CA is sometimes called the “enrollment CA” or the “ecert CA”.
* The other CA generates the certificates used to secure communications on Transport Layer Security (TLS). For this reason, this CA is often referred to as a “TLS CA”. These TLS certificates are attached to actions as a way of preventing “man in the middle” attacks. Note that the TLS CA is only used for issuing certificates for nodes and can be shut down when that activity is completed. Users have the option to use one way (client only) TLS as well as two way (server and client) TLS, with the latter also known as “mutual TLS”. Because specifying that your network will be using TLS (which is recommended) should be decided before deploying the “enrollment” CA (the YAML file specifying the configuration of this CA has a field for enabling TLS), you should deploy your TLS CA first and use its root certificate when bootstrapping your enrollment CA. This TLS certificate will also be used by the fabric-ca client when connecting to the enrollment CA to enroll identities for users and nodes.

While all of the non-TLS certificates associated with an organization can be created by a single “root” CA (that is, a CA that is its own root of trust), for added security organizations can decide to use “intermediate” CAs whose certificates are created by a root CA (or another intermediate CA that eventually leads back to a root CA). Because a compromise in the root CA leads to a collapse for its entire trust domain (the certs for the admins, nodes, and any CAs it has generated certificates for), intermediate CAs are a useful way to limit the exposure of the root CA. Whether you choose to use intermediate CAs will depend on the needs of your use case. They are not mandatory. Note that it is also possible to configure a Lightweight Directory Access Protocol (LDAP) to manage identities on a Fabric network for those enterprises that already have this implementation and do not want to add a layer of identity management to their existing infrastructure. The LDAP effectively pre registers all of the members of the directory and allows them to enroll based on the criteria given.

**In a production network, it is recommended to deploy at least one CA per organization for enrollment purposes and another for TLS.** For example, if you deploy three peers that are associated with one organization and an ordering node that is associated with an ordering organization, you will need at least four CAs. Two of the CAs will be for the peer organization (generating the enrollment and TLS certificates for the peer, admins, communications, and the folder structure of the MSP representing the organization) and the other two will be for the orderer organization. Note that users will generally only register and enroll with the enrollment CA, while nodes will register and enroll with both the enrollment CA (where the node will get its signing certificates that identify it when it attempts to sign its actions) and with the TLS CA (where it will get the TLS certificates it uses to authenticate its communications).

For an example of how to setup an organization CA and a TLS CA and enroll their admin identity, check out the [Fabric CA Deployment Guide](https://hyperledger-fabric-ca.readthedocs.io/en/latest/deployguide/ca-deploy.html). The deploy guide uses the Fabric CA client to register and enroll the identities that are required when setting up CAs.

**Step four: Use the CA to create identities and MSPs**

After you have created your CAs, you can use them to create the certificates for the identities and components related to your organization (which is represented by an MSP). For each organization, you will need to, at a minimum:

* **Register and enroll an admin identity and create an MSP**. After the CA that will be associated with an organization has been created, it can be used to first register an identity and then enroll it. In the first step, a username and password for the identity is assigned by the admin of the CA. Attributes and affiliations can also be given to the identity (for example, a role of admin, which is necessary for organization admins). After the identity has been registered, it can be enrolled by using the username and password. The CA will generate two certificates for this identity — a public certificate (also known as a “signcert” or “public cert”) known to the other members of the network, and the private key (stored in the keystore folder) used to sign actions taken by the identity. The CA will also generate a set of folders called an “MSP” containing the public certificate of the CA issuing the certificate and the root of trust for the CA (this may or may not be the same CA). This MSP can be thought of as defining the organization associated with the identity of the admin. In cases where the admin of the org will also be an admin of a node (which will be typical), **you must create the org admin identity before creating the local MSP of a node, since the certificate of the node admin must be used when creating the local MSP**.
* **Register and enroll node identities**. Just as an org admin identity is registered and enrolled, the identity of a node must be registered and enrolled with both an enrollment CA and a TLS CA (the latter generates certificates that are used to secure communications). Instead of giving a node a role of admin or user when registering it with the enrollment CA, give it a role of peer or orderer. As with the admin, attributes and affiliations for this identity can also be assigned. The MSP structure for a node is known as a “local MSP”, since the permissions assigned to the identities are only relevant at the local (node) level. This MSP is created when the node identity is created, and is used when bootstrapping the node.

For more conceptual information about identities and permissions in a Fabric-based blockchain network, see [Identity](https://hyperledger-fabric.readthedocs.io/en/latest/identity/identity.html) and [Membership Service Provider (MSP)](https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html).

For more information about how to use a CA to register and enroll identities, including sample commands, check out [Registering and enrolling identities with a CA](https://hyperledger-fabric-ca.readthedocs.io/en/latest/deployguide/use_CA.html).

**Step five: Deploy nodes**

Once you have gathered all of the certificates and MSPs you need, you’re almost ready to create a node. As discussed above, there are a number of valid ways to deploy nodes.

**Create a peer**

Before you can create a peer, you will need to customize the configuration file for the peer. In Fabric, this file is called core.yaml. You can find a sample core.yaml configuration file [in the sampleconfig directory of Hyperledger Fabric](https://github.com/hyperledger/fabric/blob/master/sampleconfig/core.yaml).

As you can see in the file, there are quite a number of parameters you either have the option to set or will need to set for your node to work properly. In general, if you do not have the need to change a tuning value, leave it alone. You will, however, likely need to adjust the various addresses, specify the database type you want to use, as well as to specify where the MSP for the node is located.

You have two main options for tuning your configuration.

1. Edit the YAML file bundled with the binaries.
2. Use environment variable overrides when deploying.
3. Specify flags on CLI commands.

Option 1 has the advantage of persisting your changes whenever you bring down and bring back up the node. The downside is that you will have to port the options you customized to the new YAML when upgrading to a new binary version (you should use the latest YAML when upgrading to a new version).

Either way, here are some values in core.yaml you must review.

* peer.localMspID: this is the name of the local MSP of your peer organization. This MSP is where your peer organization admins will be listed as well as the peer organization’s root CA and TLS CA certificates.
* peer.mspConfigPath: the place where the local MSP for the peer is located. Note that it is a best practice to mount this volume external to your container. This ensures that even if the container is stopped (for example, during a maintenance cycle) that the MSPs are not lost and have to be recreated.
* peer.address: represents the endpoint to other peers in the same organization, an important consideration when establishing gossip communication within an organization.
* peer.tls: When you set the enabled value to true (as should be done in a production network), you will have to specify the locations of the relevant TLS certificates. Note that all of the nodes in a network (both the peers and the ordering nodes) must either all have TLS enabled or not enabled. For production networks, it is highly recommended to enable TLS. As with your MSP, it is a best practice to mount this volume external to your container.
* ledger: users have a number of decisions to make about their ledger, including the state database type (LevelDB or CouchDB, for example), and its location (specified in fileSystemPath). Note that for CouchDB, it is a best practice to operate your state database external to the peer (for example, in a separate container), as you will be better able to allocate specific resources to the database this way. For latency and security reasons, it is a best practice to have the CouchDB container on the same server as the peer. Access to the CouchDB container should be restricted to only the peer container.
* gossip: there are a number of configuration options to think about when setting up [Gossip data dissemination protocol](https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html), including the externalEndpoint (which makes peers discoverable to peers owned by other organizations) as well as the bootstrap address (which identifies a peer in the peer’s own organization).
* chaincode.externalBuilders: this field is important to set when using [Chaincode as an external service](https://hyperledger-fabric.readthedocs.io/en/latest/cc_service.html).

When you’re comfortable with how your peer has been configured, how your volumes are mounted, and your backend configuration, you can run the command to launch the peer (this command will depend on your backend configuration).

**Create an ordering node**

Unlike the creation of a peer, you will need to create a genesis block (or reference a block that has already been created, if adding an ordering node to an existing ordering service) and specify the path to it before launching the ordering node.

In Fabric, this configuration file for ordering nodes is called orderer.yaml. You can find a sample orderer.yaml configuration file [in the sampleconfig directory of Hyperledger Fabric](https://github.com/hyperledger/fabric/blob/master/sampleconfig/orderer.yaml). Note that orderer.yaml is different than the “genesis block” of an ordering service. This block, which includes the initial configuration of the orderer system channel, must be created before an ordering node is created because it is used to bootstrap the node.

As with the peer, you will see that there are quite a number of parameters you either have the option to set or will need to set for your node to work properly. In general, if you do not have the need to change a tuning value, leave it alone.

You have two main options for tuning your configuration.

1. Edit the YAML file bundled with the binaries.
2. Use environment variable overrides when deploying.
3. Specify flags on CLI commands.

Option 1 has the advantage of persisting your changes whenever you bring down and bring back up the node. The downside is that you will have to port the options you customized to the new YAML when upgrading to a new binary version (you should use the latest YAML when upgrading to a new version).

Either way, here are some values in orderer.yaml you must review. You will notice that some of these fields are the same as those in core.yaml only with different names.

* General.LocalMSPID: this is the name of the local MSP, generated by your CA, of your orderer organization.
* General.LocalMSPDir: the place where the local MSP for the ordering node is located. Note that it is a best practice to mount this volume external to your container.
* General.ListenAddress and General.ListenPort: represents the endpoint to other ordering nodes in the same organization.
* FileLedger: although ordering nodes do not have a state database, they still all carry copies of the blockchain, as this allows them to verify permissions using the latest config block. Therefore the ledger fields should be customized with the correct file path.
* Cluster: these values are important for ordering service nodes that communicate with other ordering nodes, such as in a Raft based ordering service.
* General.BootstrapFile: this is the name of the configuration block used to bootstrap an ordering node. If this node is the first node generated in an ordering service, this file will have to be generated and is known as the “genesis block”.
* General.BootstrapMethod: the method by which the bootstrap block is given. For now, this can only be file, in which the file in the BootstrapFile is specified. Starting in 2.0, you can specify none to simply start the orderer without bootstrapping.
* Consensus: determines the key/value pairs allowed by the consensus plugin (Raft ordering services are supported and recommended) for the Write Ahead Logs (WALDir) and Snapshots (SnapDir).

When you’re comfortable with how your ordering node has been configured, how your volumes are mounted, and your backend configuration, you can run the command to launch the ordering node (this command will depend on your backend configuration).

**Next steps**

Blockchain networks are all about connection, so once you’ve deployed nodes, you’ll obviously want to connect them to other nodes! If you have a peer organization and a peer, you’ll want to join your organization to a consortium and join or [Channels](https://hyperledger-fabric.readthedocs.io/en/latest/channels.html). If you have an ordering node, you will want to add peer organizations to your consortium. You’ll also want to learn how to develop chaincode, which you can learn about in the topics [The scenario](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/scenario.html) and [Writing Your First Chaincode](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html).

Part of the process of connecting nodes and creating channels will involve modifying policies to fit the use cases of business networks. For more information about policies, check out [Policies](https://hyperledger-fabric.readthedocs.io/en/latest/policies/policies.html).

One of the common tasks in a Fabric will be the editing of existing channels. For a tutorial about that process, check out [Updating a channel configuration](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html). One popular channel update is to add an org to an existing channel. For a tutorial about that specific process, check out [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/latest/channel_update_tutorial.html). For information about upgrading nodes after they have been deployed, check out [Upgrading your components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html).

**Deploying a Production CA**

* [Planning for a CA](https://hyperledger-fabric-ca.readthedocs.io/en/latest/deployguide/ca-deploy-topology.html)
* [Checklist for a production CA server](https://hyperledger-fabric-ca.readthedocs.io/en/latest/deployguide/ca-config.html)
* [CA deployment steps](https://hyperledger-fabric-ca.readthedocs.io/en/latest/deployguide/cadeploy.html)