Considerations for getting to v2.0

Considerations for getting to v2.x

Chaincode lifecycle

In this topic we'll cover recommendations for upgrading to the newest release from the previous release as well as from the most recent long term support (LTS) release.

The new chaincode lifecycle that debuts in v2.0 allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. For more information about the new chaincode lifecycle, check out Chaincode for operators.

Upgrading from 2.1 to 2.2

It is a best practice to upgrade all of the peers on a channel before enabling the `Channel` and `Application` capabilities that enable the new chaincode lifecycle (the `Channel` capability is not strictly required, but it makes sense to update it at this time). Note that any peers that are not at v2.0 will crash after enabling either capability, while any ordering nodes that are not at v2.0 will crash after the `Channel` capability has been enabled. This crashing behavior is intentional, as the peer or orderer cannot safely participate in the channel if it does not support the required capabilities.

The 2.1 and 2.2 releases of Fabric are stabilization releases, featuring bug fixes and other forms of code hardening. As such there are no particular considerations needed for upgrade, and no new capability levels requiring particular image versions or channel configuration updates.

After the `Application` capability has been updated to `V2_0` on a channel, you must use the v2.0 lifecycle procedures to package, install, approve, and commit new chaincodes on the channel. As a result, make sure to be prepared for the new lifecycle before updating the capability.

Upgrading to 2.2 from the 1.4.x long term support release

The new lifecycle defaults to using the endorsement policy configured in the channel config (e.g., a `MAJORITY` of orgs). Therefore this endorsement policy should be added to the channel configuration when enabling capabilities on the channel.

Before attempting to upgrade from v1.4.x to v2.2, make sure to consider the following:

For information about how to edit the relevant channel configurations to enable the new lifecycle by adding an endorsement policy for each organization, check out Enabling the new chaincode lifecycle.

Chaincode lifecycle

Chaincode shim changes (Go chaincode only)

The new chaincode lifecycle that debuted in v2.0 allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. For more information about the new chaincode lifecycle, check out Fabric chaincode lifecycle concept topic.

The recommended approach is to vendor the shim in your v1.4 Go chaincode before making upgrades to the peers and channels. If you do this, you do not need to make any additional changes to your chaincode.

It is a best practice to upgrade all of the peers on a channel before enabling the `Channel` and `Application` capabilities that enable the new chaincode lifecycle (the `Channel` capability is not strictly required, but it makes sense to update it at this time). Note that any peers that are not at v2.x will crash after enabling either capability, while any ordering nodes that are not at v2.x will crash after the `Channel` capability has been enabled. This crashing behavior is intentional, as the peer or orderer cannot safely participate in the channel if it does not support the required capabilities.

If you did not vendor the shim in your v1.4 chaincode, the old v1.4 chaincode images will still technically work after upgrade, but you are in a risky state. If the chaincode image gets deleted from your environment for whatever reason, the next invoke on v2.0 peer will try to rebuild the chaincode image and you'll get an error that the shim cannot be found.

After the `Application` capability has been updated to `V2_0` on a channel, you must use the v2.x lifecycle procedures to package, install, approve, and commit new chaincodes on the channel. As a result, make sure to be prepared for the new lifecycle before updating the capability.

At this point, you have two options:

The new lifecycle defaults to using the endorsement policy configured in the channel config (e.g., a `MAJORITY` of orgs). Therefore this endorsement policy should be added to the channel configuration when enabling capabilities on the channel.

1. If the entire channel is ready to upgrade chaincode, you can upgrade the chaincode on all peers and on the channel (using either the old or new lifecycle depending on the `Application` capability level you have enabled). The best practice at this point would be to vendor the new Go chaincode shim using modules.

For information about how to edit the relevant channel configurations to enable the new lifecycle by adding an endorsement policy for each organization, check out Enabling the new chaincode lifecycle.

1. If the entire channel is not yet ready to upgrade the chaincode, you can use peer environment variables to specify the v1.4 chaincode environment `ccenv` be used to rebuild the chaincode images. This v1.4 `ccenv` should still work with a v2.0 peer.

Chaincode shim changes (Go chaincode only)

Chaincode logger (Go chaincode only)

The recommended approach is to vendor the shim in your v1.4 Go chaincode before making upgrades to the peers and channels. If you do this, you do not need to make any additional changes to your chaincode.

Support for user chaincodes to utilize the chaincode shim's logger via `NewLogger()` has been removed. Chaincodes that used the shim's `NewLogger()` must now shift to their own preferred logging mechanism.

If you did not vendor the shim in your v1.4 chaincode, the old v1.4 chaincode images will still technically work after upgrade, but you are in a risky state. If the chaincode image gets deleted from your environment for whatever reason, the next invoke on v2.x peer will try to rebuild the chaincode image and you'll get an error that the shim cannot be found.

For more information, check out Logging control.

At this point, you have two options:

Peer databases upgrade

1. If the entire channel is ready to upgrade chaincode, you can upgrade the chaincode on all peers and on the channel (using either the old or new lifecycle depending on the `Application` capability level you have enabled). The best practice at this point would be to vendor the new Go chaincode shim using modules.

The databases of all peers (which include not just the state database but the history database and other internal databases for the peer) must be rebuilt using the v2.0 data format as part of the upgrade to v2.0. To trigger the rebuild, the databases must be dropped before the peer is started.

1. If the entire channel is not yet ready to upgrade the chaincode, you can use peer environment variables to specify the v1.4 chaincode environment `ccenv` be used to rebuild the chaincode images. This v1.4 `ccenv` should still work with a v2.x peer.

For information about how to upgrade peers, check out our documentation on upgrading components. During the process for upgrading your peers, you will need to pass a `peer node upgrade-dbs` command to drop the databases of the peer.

Chaincode logger (Go chaincode only)

Follow the commands to upgrade a peer until the `docker run` command you see to launch the new peer container (you can skip the step where you set an `IMAGE_TAG`, since the `upgrade dbs` command is for the v2.0 release of Fabric only, but you will need to set the `PEER_CONTAINER` and `LEDGERS_BACKUP` environment variables). Instead of the `docker run` command to launch the peer, run this one instead:

Support for user chaincodes to utilize the chaincode shim's logger via `NewLogger()` has been removed. Chaincodes that used the shim's `NewLogger()` must now shift to their own preferred logging mechanism.

```
docker run --rm -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:2.0 peer node upgrade-dbs
```

For more information, check out Logging control.

This will drop the databases of the peer. Then issue this command to start the peer using the `2.0` tag:

Peer databases upgrade

```
docker run -d -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:2.0 peer node start
```

For information about how to upgrade peers, check out our documentation on upgrading components. During the process for upgrading your peers, you will need to perform one additional step to upgrade the peer databases. The databases of all peers (which include not just the state database but the history database and other internal databases for the peer) must be rebuilt using the v2.x data format as part of the upgrade to v2.x. To trigger the rebuild, the databases must be dropped before the peer is started. The instructions below utilize the `peer node upgrade-dbs` command to drop the local databases managed by the peer and prepare them for upgrade, so that they can be rebuilt the first time the v2.x peer starts. If you are using CouchDB as the state database, the peer has support to automatically drop this database as of v2.2. To leverage the support, you must configure the peer with CouchDB as the state database and start CouchDB before running the `upgrade-dbs` command. In v2.0 and v2.1, the peer does not automatically drop the CouchDB state database; therefore you must drop it yourself.

Because rebuilding the databases can be a lengthy process (several hours, depending on the size of your databases), monitor the peer logs to check the status of the rebuild. Every 1000th block you will see a message like `[lockbasedtxmgr] CommitLostBlock -> INFO 041 Recommitting block [1000] to state database` indicating the rebuild is ongoing.

Follow the commands to upgrade a peer until the `docker run` command to launch the new peer container (you can skip the step where you set an `IMAGE_TAG`, since the `upgrade-dbs` command is for the v2.x release of Fabric only, but you will need to set the `PEER_CONTAINER` and `LEDGERS_BACKUP` environment variables). Instead of the `docker run` command to launch the peer, run this command instead to drop and prepare the local databases managed by the peer (substitute `2.1` for `2.0` in these commands if you are upgrading to that binary version from the 1.4.x LTS):

If the database is not dropped as part of the upgrade process, the peer start will return an error message stating that its databases are in the old format and must be dropped using the `peer node upgrade-dbs` command above. The node will then need to be restarted.

```
docker run --rm -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:2.0 peer node upgrade-dbs
```

Capabilities

In v2.0 and v2.1, if you are using CouchDB as the state database, also drop the CouchDB database. This can be done by removing the CouchDB /data volume directory.

As can be expected for a 2.0 release, there is a full complement of new capabilities for 2.0.

Then issue this command to start the peer using the `2.0` tag:

- **Application** `V2_0`: enables the new chaincode lifecycle as described in Chaincode for Operators.

```
docker run -d -v /opt/backup/$PEER_CONTAINER/:/var/hyperledger/production/ \
            -v /opt/msp/:/etc/hyperledger/fabric/msp/ \
            --env-file ./env<name of node>.list \
            --name $PEER_CONTAINER \
            hyperledger/fabric-peer:2.0 peer node start
```

- **Channel** `V2_0`: this capability has no changes, but is used for consistency with the application and orderer capability levels.

The peer will rebuild the databases using the v2.x data format the first time it starts. Because rebuilding the databases can be a lengthy process (several hours, depending on the size of your databases), monitor the peer logs to check the status of the rebuild. Every 1000th block you will see a message like `[lockbasedtxmgr] CommitLostBlock -> INFO 041 Recommitting block [1000] to state database` indicating the rebuild is ongoing.

- **Orderer** `V2_0`: controls `UseChannelCreationPolicyAsAdmins`, changing the way that channel creation transactions are validated. When combined with the `-baseProfile` option of configtxgen, values which were previously inherited from the orderer system channel may now be overridden.

If the database is not dropped as part of the upgrade process, the peer start will return an error message stating that its databases are in the old format and must be dropped using the `peer node upgrade-dbs` command above (or dropped manually if using CouchDB state database). The node will then need to be restarted again.

As with any update of the capability levels, make sure to upgrade your peer binaries before updating the `Application` and `Channel` capabilities, and make sure to upgrade your orderer binaries before updating the `Orderer` and `Channel` capabilities.

Capabilities

For information about how to set new capabilities, check out Updating the capability level of a channel.

The 2.0 release featured three new capabilities.

Define ordering node endpoint per org (recommend)

- **Application** `V2_0`: enables the new chaincode lifecycle as described in Fabric chaincode lifecycle concept topic.

Starting with version v1.4.2, it was recommended to define orderer endpoints in both the system channel and in all application channels at the organization level by adding a new `OrdererEndpoints` stanza within the channel configuration of an organization, replacing the the global `OrdererAddresses` section of channel configuration. If at least one organization has an ordering service endpoint defined at an organizational level, all orderers and peers will ignore the channel level endpoints when connecting to ordering nodes.

- **Channel** `V2_0`: this capability has no changes, but is used for consistency with the application and orderer capability levels.

Utilizing organization level orderer endpoints is required when using service discovery with ordering nodes provided by multiple organizations. This allows clients to provide the correct organization TLS certificates.

- **Orderer** `V2_0`: controls `UseChannelCreationPolicyAsAdmins`, changing the way that channel creation transactions are validated. When combined with the `-baseProfile` option of configtxgen, values which were previously inherited from the orderer system channel may now be overridden.

If your channel configuration does not yet include `OrdererEndpoints` per org, you will need to perform a channel configuration update to add them to the config. First, create a JSON file that includes the new configuration stanza.

As with any update of the capability levels, make sure to upgrade your peer binaries before updating the `Application` and `Channel` capabilities, and make sure to upgrade your orderer binaries before updating the `Orderer` and `Channel` capabilities.

In this example, we will create a stanza for a single org called `OrdererOrg`. Note that if you have multiple ordering service organizations, they will all have to be updated to include endpoints. Let's call our JSON file `orglevelEndpoints.json`.

For information about how to set new capabilities, check out Updating the capability level of a channel.

```
{
  "OrdererOrgEndpoint": {
      "Endpoints": {
          "mod_policy": "Admins",
          "value": {
              "addresses": [
                 "127.0.0.1:30000"
              ]
          }
      }
    }
}
```

Define ordering node endpoint per org (recommend)

Then, export the following environment variables:

Starting with version v1.4.2, it was recommended to define orderer endpoints in both the system channel and in all application channels at the organization level by adding a new `OrdererEndpoints` stanza within the channel configuration of an organization, replacing the global `OrdererAddresses` section of channel configuration. If at least one organization has an ordering service endpoint defined at an organizational level, all orderers and peers will ignore the channel level endpoints when connecting to ordering nodes.

- `CH_NAME`: the name of the channel being updated. Note that all system channels and application channels should contain organization endpoints for ordering nodes.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `TLS_ROOT_CA`: the absolute path to the root CA certificate of the organization proposing the system channel update.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
- `ORGNAME`: The name of the organization you are currently updating. For example, `OrdererOrg`.

Utilizing organization level orderer endpoints is required when using service discovery with ordering nodes provided by multiple organizations. This allows clients to provide the correct organization TLS certificates.

Once you have set the environment variables, navigate to Step 1: Pull and translate the config.

If your channel configuration does not yet include `OrdererEndpoints` per org, you will need to perform a channel configuration update to add them to the config. First, create a JSON file that includes the new configuration stanza.

Once you have a `modified_config.json`, add the lifecycle organization policy (as listed in `orglevelEndpoints.json`) using this command:

In this example, we will create a stanza for a single org called `OrdererOrg`. Note that if you have multiple ordering service organizations, they will all have to be updated to include endpoints. Let's call our JSON file `orglevelEndpoints.json`.

```
jq -s ".[0] * {\"channel_group\":{\"groups\":{\"Orderer\": {\"groups\": {\"$ORGNAME\": {\"values\
```

```
{
  "OrdererOrgEndpoint": {
      "Endpoints": {
          "mod_policy": "Admins",
          "value": {
              "addresses": [
                 "127.0.0.1:30000"
              ]
          }
      }
   }
}
```

Then, follow the steps at Step 3: Re-encode and submit the config.

Then, export the following environment variables:

If every ordering service organization performs their own channel edit, they can edit the configuration without needing further signatures (by default, the only signature needed to edit parameters within an organization is an admin of that organization). If a different organization proposes the update, then the organization being edited will need to sign the channel update request.

- `CH_NAME`: the name of the channel being updated. Note that all system channels and application channels should contain organization endpoints for ordering nodes.
- `CORE_PEER_LOCALMSPID`: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
- `CORE_PEER_MSPCONFIGPATH`: the absolute path to the MSP representing your organization.
- `TLS_ROOT_CA`: the absolute path to the root CA certificate of the organization proposing the system channel update.
- `ORDERER_CONTAINER`: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
- `ORGNAME`: The name of the organization you are currently updating. For example, `OrdererOrg`.

Once you have set the environment variables, navigate to Step 1: Pull and translate the config.

Then, add the lifecycle organization policy (as listed in `orglevelEndpoints.json`) to a file called `modified_config.json` using this command:

```
jq -s ".[0] * {\"channel_group\":{\"groups\":{\"Orderer\": {\"groups\": {\"$ORGNAME\": {\"values\
```

Then, follow the steps at Step 3: Re-encode and submit the config.

If every ordering service organization performs their own channel edit, they can edit the configuration without needing further signatures (by default, the only signature needed to edit parameters within an organization is an admin of that organization). If a different organization proposes the update, then the organization being edited will need to sign the channel update request.