Chapter 14: Understanding Docker for Hyperledger Composer.

Return to Table of Contents

This is a high level tutorial on how HyperLedger Composer uses Docker. For a thorough introduction to Docker, I recommend The Docker Book.

HyperLedger Composer v0.16 (the current version of this tutorial), creates 4 docker containers to run the local dev environment, one each for your network, the orderer, the peer, and one for couchdb (the noSQL database). This happens each time you run the ./buildAndDeploy script, which calls deployNetwork.sh. During the execution of the deployNetwork script, the following information is displayed in your terminal window:

```
ARCH=$ARCH docker-compose -f
"${DIR}"/composer/docker-compose.yml down
Stopping peer0.org1.example.com ... done
Stopping ca.org1.example.com    ... done
Stopping orderer.example.com    ... done
Stopping couchdb                ... done
Removing peer0.org1.example.com ... done
Removing ca.org1.example.com    ... done
Removing orderer.example.com    ... done
Removing couchdb                ... done
Removing network composer_default
ARCH=$ARCH docker-compose -f
"${DIR}"/composer/docker-compose.yml up -d
Creating orderer.example.com ... done
Creating peer0.org1.example.com ... done
Creating couchdb ...
Creating ca.org1.example.com ...
Creating peer0.org1.example.com ...
```

Three things are happening here: (1) If the network is up, each of the 4 Docker containers running are stopped. (2) Then each container is removed from your computer (3) Then each container is restarted.

All of this is controlled by the docker-compose.yml file, which is located in the fabric-tools folder installed during the set up process. Specifically, the file is in the following path: `fabric-tools/fabric-scripts/hlfv1/composer`. The purpoe of the docker compose yaml file is to capture all of the commands and parameters required to start up a group of containers in a readable and consistent manner. While everything in the yaml file could be executed directly from the command line, the commands become very unwieldy, so Docker Compose was created to allow all of the parameters to be placed into a yaml file and allow all of the container starts to be in the same yaml file.

what containers are created?

You can discover this information by executing the following command: `docker ps -a`, which displays the following for our network:

```
CONTAINER ID        ... NAMES
0386e33b66c5        ... dev-peer0.org1.example.com-
zerotoblockchain-network-0.16.0
fd0187712cfc        ... peer0.org1.example.com
c6f532764faa        ... couchdb
cf3608360871        ... orderer.example.com
346f77c65284        ... ca.org1.example.com
```

Let's look at the contents of the docker-compose.yml file:

```
version: '2'

services:
```

The file starts out simply.

- version refers to the version of docker-compose being used, in this case it's version 2. This specifies both what can be put in the yaml file and how each term is to be grouped and specified.
- services refers to the set of docker containers which will be started by this yaml file.

Each service (container) has a similar structure and we will look at all four:

ca – Certificate Authority:

```
  ca.org1.example.com:
    image: hyperledger/fabric-ca:$ARCH-1.0.4
    environment:
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
      - FABRIC_CA_SERVER_CA_NAME=ca.org1.example.com
#      - FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/org1.example.com-cert.pem
#      - FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/a22daf356b2aab5792ea53e35f66fccef1d7f1aa2b3a2b92dbfbf96a448ea26a_sk

    ports:
      - "7054:7054"
    command: sh -c 'fabric-ca-server start --
```

```
ca.certfile /etc/hyperledger/fabric-ca-server-
config/ca.org1.example.com-cert.pem --ca.keyfile
/etc/hyperledger/fabric-ca-server-
config/19ab65abbb04807dad12e4c0a9aaa6649e70868e3abd02
17a322d89e47e1a6ae_sk -b admin:adminpw -d'
    volumes:
      - ./crypto-
config/peerOrganizations/org1.example.com/ca/:/etc/hy
perledger/fabric-ca-server-config
    container_name: ca.org1.example.com
```

- image refers to the docker container which is to be started. In this case, it's owned by 'hyperledger' and is shown here as the 'fabric-ca:$ARCH-1.0.4' image where $ARCH is the architecture of your local operating system, which is retrieved and set in the startFabric.sh script. On my OSX laptop ARCH is set to x86_64, so the image to be retrieved would be: fabric-ca:x86_64-1.0.4 where fabric-ca is the image name and x86-64-1.0.4 is the version identifier.
- ENVIRONMENT sets environment variables inside the container. These variables are not visible to applications running on your host operating system.
    - The '#' sign makes everything that follows it a comment
- PORTS: identifies a port inside the container (left number) and links it to a port visible to your host operating system.
- COMMAND lists the command to be executed when the container starts. In this case, the shell is called to execute fabric-ca-server start command with 3 options
- VOLUMES links one or more specific folders on your local computer to a specific path inside the container. The './' prefix means 'start here', where 'here' is defined by the location of the docker-compose.yml file (shown earlier). This allows local information to be shared in a persistent manner with a docker container and, as needed, changed without having to rebuild the container.

[illegible] iRDFRER_ENERAL_LO 0E0E0E0K □CarRger/fabric-oio$AR -

- CONTAINER_NAME is the name that the container will display when docker ps is used.

You will observe an identical structure in each of the following services:

Create the orderer container

```
    orderer.example.com:
     container_name: orderer.example.com
     image: hyperledger/fabric-orderer:$ARCH-1.0.4
     environment:
       - ORDERER_GENERAL_LO 0E0E0E0E□ea□rd□ -r □ rr
    container_name: peer0.org1.example.com
    image: hyperledger/fabric-peer:$ARCH-1.0.4
    environment:
      - CORE_LOGGING_PEER=critical
      - CORE_CHAINCODE_LOGGING_LEVEL=critical
      -
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer0.org1.example.com
      - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
      -
CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=composer_default

      - CORE_PEER_LOCALMSPID=Org1MSP
      -
CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/peer/msp
      - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
      -
CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchd
b:5984
    working_dir:
/opt/gopath/src/github.com/hyperledger/fabric
    command: peer node start --peer-
defaultchain=false
    ports:
      - 7051:7051
      - 7053:7053
    volumes:
        - /var/run/:/host/var/run/
        - ./:/etc/hyperledger/configtx
        - ./crypto-
config/peerOrganizations/org1.example.com/peers/peer0
.org1.example.com/msp:/etc/hyperledger/peer/msp
        - ./crypto-
config/peerOrganizations/org1.example.com/users:/etc/
hyperledger/msp/users
    depends_on:
      - orderer.example.com
      - couchdb
```

Here you see an additional parameter:

- DEPENDS_ON which, as you might guess, lists docker container names which must be running before this container will successfully start.

Create the couchdb container:

```
  couchdb:
   container_name: couchdb
   image: hyperledger/fabric-couchdb:$ARCH-1.0.4
   ports:
     - 5984:5984
   environment:
     DB_URL: http://localhost:5984/member_db
```

OK, so now we have the four base containers for our network, but we don't yet have the network. These docker-compose commands are executed when the startFabric script runs:

```
ARCH=$ARCH docker-compose -f
"${DIR}"/composer/docker-compose.yml down
ARCH=$ARCH docker-compose -f
"${DIR}"/composer/docker-compose.yml up -d

# wait for Hyperledger Fabric to start
# incase of errors when running later commands, issue
export FABRIC_START_TIMEOUT=<larger number>
echo ${FABRIC_START_TIMEOUT}
sleep ${FABRIC_START_TIMEOUT}

# Create the channel
```

```
docker exec peer0.org1.example.com peer channel
create -o orderer.example.com:7050 -c composerchannel
-f /etc/hyperledger/configtx/composer-channel.tx

# Join peer0.org1.example.com to the channel.
docker exec -e
"CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/A
dmin@org1.example.com/msp" peer0.org1.example.com
peer channel join -b composerchannel.block
```

The script first takes down the network and then restarts it in detached (that's the -d) mode. The script then pauses for 15 seconds (timeout default) and then creates the peer0 channel (remember the depends_on) which connects peer0 to the orderer and records that connection in the couchdb database. We are missing a container. The one with our network and the first one in the docker ps -a list. That's created by the deployNetwork.sh script when the

```
composer network start -c PeerAdmin@hlfv1 -A admin -S
adminpw -a $NETWORK_NAME.bna --file networkadmin.card
```

script is executed.

So where does the oher network come from? The one with our business network? That is created when the deployNetwork.sh script is run at the end of buildAndDeploy. Each time you run the buildAndDeploy script, it stops, removes, and rebuilds all 5 containers. The first 4 are built based on the docker-compose.yml file, the last one, with your network, is created in the deploy script. You could do the same thing in the admin console by using the create and install network services. If you do that, please be sure to use a different network name.

OK, so, now we want to test and debug. The part of the docker environment where our code will run is in the 5th container - the one named: **dev-peer0.org1.example.com-zerotoblockchain-network-0.16.0**

In much of the code we've written, we will often use console.log to display what's going on. This works in all of our nodeJS code EXCEPT for the code in the Chapterxx/network/lib folder. For some reason, those console.log messaegs never print out. Well, actually, they do.

We're going to start by going to any chapter between Chapter5 and Chapter 12 where you've completed your coding. We'll run the buildAndDeploy script to create a clean instance of the network and then we're going to start following the log for our business network. We'll need two terminal windows open at the same time. One will show the log output in real time from our business network container, the other will show the log output from our nodejs application. The examples in this script were created using Chapter 6.

(1) go to `ZeroToBlockChain\Chaper06` (2) run `./buildAndDeploy` (3) run `docker ps -a` which will return something like the following:

```
CONTAINER ID        ... NAMES
0386e33b66c5        ... dev-peer0.org1.example.com-
zerotoblockchain-network-0.16.0
fd0187712cfc        ... peer0.org1.example.com
c6f532764faa        ... couchdb
cf3608360871        ... orderer.example.com
346f77c65284        ... ca.org1.example.com
```

(4) run `docker logs dev-peer0.org1.example.com-zerotoblockchain-network-0.16.0 -f` (5) open a new terminal window and go to `ZeroToBlockChain\Chaper06` (6) run ```npm start``

```
> zerotoblockchain-network@0.1.6 start
/Users/rddill/Documents/GitHub/Z2B_Master/Chapter06
```

```
> node index

Listening locally on port 6002
```

(7) go to your browser and connect to localhost:PORT where PORT is the number displayed after npm start - 6002 in this example (8) Select Admin --> Preload Network

- When you do this, you'll see messages starting to scroll in both windows. You'll see that everytime the ERROR message appears in the npm start window, a similar message appears in the docker logs window. OK, boring, but nice to see the consistency.

Now let's add a console message to the sample.js file in Chapter06/network/lib - one we've tried before and never been able to see.

(9) press CTRL-c in both terminal windows to stop both the logging process and the nodejs server (10) In your favorite editor, open the sample.js file and update the CreateOrder function by adding this line to it:

```javascript
  console.log('Order for '+purchase.buyer+' created wih amount: '+purchase.amount);
```

~~~
 - your CreateOrder function should now look like this:
```javascript
function CreateOrder(purchase) {
    purchase.order.buyer = purchase.buyer;
    purchase.order.amount = purchase.amount;
    purchase.order.financeCo = purchase.financeCo;
~~~

```
    purchase.order.created = new
Date().toISOString();
    purchase.order.status =
JSON.stringify(orderStatus.Created);
    console.log('Order for '+purchase.buyer+' created
wih amount: '+purchase.amount);
    return
getAssetRegistry('org.acme.Z2BTestNetwork.Order')
        .then(function (assetRegistry) {
            return
assetRegistry.update(purchase.order);
        });
}
```

(11) Save your updated sample.js file (12) run ./buildAndDeploy (13) when complete, run docker logs dev-peer0.org1.example.com-zerotoblockchain-network-0.16.0 -f (14) go to your 2nd terminal window and run npm start (15) go to your browser and load/reload the browser page for localhost:PORT

- please note, the port number changes sometimes (16) select Admin --> Preload Network and watch the logs.
- ***WOOHOO*** Our messages now show up in the log listing from docker:
    - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#eric@bornonthecloudinc.com} created wih amount: 6055
    - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#yes@softwaresolutionsinc.com} created wih amount: 9125
    - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#yes@softwaresolutionsinc.com} created wih amount: 38800
    - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#yes@softwaresolutionsi

nc.com} created wih amount: 6055

  - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#yes@softwaresolutionsinc.com} created wih amount: 9125
  - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#yes@softwaresolutionsinc.com} created wih amount: 6055
  - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#eric@bornonthecloudinc.com} created wih amount: 38800
  - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#eric@bornonthecloudinc.com} created wih amount: 9125
  - Order for Resource {id=org.acme.Z2BTestNetwork.Buyer#eric@bornonthecloudinc.com} created wih amount: 38800

Congratulations

You now have the ability to debug not only your code in your browser and in the nodejs server, you now, also, have the ability to trace and debug your chaincode running in the docker container!