Application Framework

Developer Guide

Table of Contents

[1 Introduction 3](#_Toc517731123)

[2 Fundamentals of DCNM AFW 4](#_Toc517731124)

[2.1 Docker fundamentals 4](#_Toc517731125)

[3 Anatomy of Application Package 6](#_Toc517731126)

[3.1 Application Specification 6](#_Toc517731127)

[3.2 Docker Container Image 9](#_Toc517731128)

[3.3 Front End Package 13](#_Toc517731129)

[4 Life as a DCNM Application Container 14](#_Toc517731130)

[4.1 Environment 14](#_Toc517731131)

[4.2 Fabric Connectivity 14](#_Toc517731132)

[4.3 Service Discovery 15](#_Toc517731133)

[4.3.1 GET/afw/service/{serviceid} 15](#_Toc517731134)

[4.3.2 GET/afw/service/{serviceid}/{fabricid} 16](#_Toc517731135)

[4.4 Developing Front End 18](#_Toc517731136)

[4.4.1 Offloaded 18](#_Toc517731137)

[4.4.2 Integrated 18](#_Toc517731138)

[4.4.3 Writing and Serving Pages 19](#_Toc517731139)

[4.5 Clustering 19](#_Toc517731140)

[4.6 Storage 21](#_Toc517731141)

[4.6.1 Non-Persistent Storage 21](#_Toc517731142)

[4.6.2 Persistent Local Storage 22](#_Toc517731143)

[4.6.3 Persistent Shared Storage 22](#_Toc517731144)

[4.6.4 Handling Concurrency 22](#_Toc517731145)

[4.7 Application DCNM API Access 23](#_Toc517731146)

[4.7.1 Daemon-Token 23](#_Toc517731147)

[4.7.2 User-App Token 24](#_Toc517731148)

[4.7.3 Making API Call 24](#_Toc517731149)

# Introduction

Application frame Work for DCNM is an Infrastructure on DCNM which adds scalability and agility for DCNM application development. Developing applications on DCNM involved writing software that was tightly integrated with everything on DCNM. This resulted in having to wait for DCNM releases to provide a release vehicle for applications that are to be delivered to the customer. The new Frame work adds the ability to develop applications in a separate dev-ops kind of model.

An application can be built using any programming language and deployed onto a production DCNM.

An application can be upgraded anytime on a production DCNM with significantly smaller downtime.

DCNM will be scalable such that it can utilize Compute infra incrementally for newer applications

This document is categorized into following parts

1. Fundamentals of DCNM Application Frame work (AFW)

2. Anatomy of Application Package

3. Writing the application backend

4. Developing a front end

5. Network Attachment

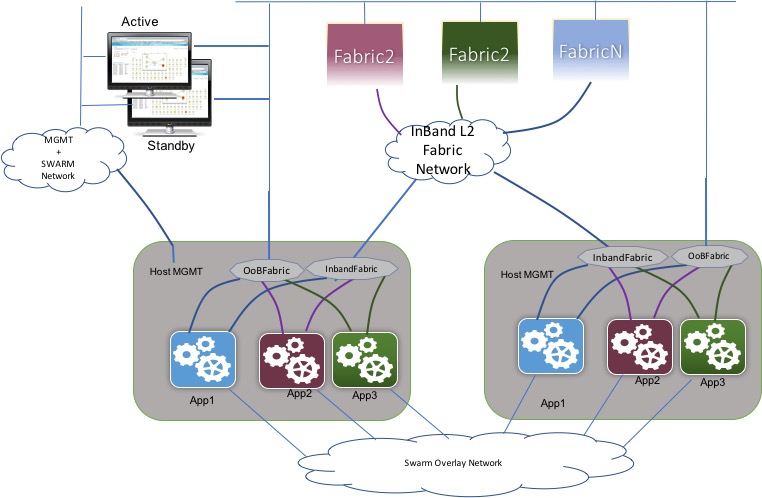
6. Storage Requirement

# Fundamentals of DCNM AFW

To achieve the primary goals as set out earlier DCNM application frame work is built with a notion of DCNM running in two roles

1. DCNM running as a master that oversees the infrastructure
2. DCNM running as a worker that is capable of running applications with required resources

DCNM together with its workers helps execute applications in a highly available manner. Workers can be added to scale number of applications or to scale an application itself. DCNM implements a Docker Swarm infrastructure. DCNM acts as a swarm manager, this is true for standby as well. DCNM will also act as a worker node. Applications can be deployed on both manager and worker as desired by the DCNM administrator. New worker nodes can be added to the infrastructure on demand. Here is a picture that describes functional architecture of the DCNM App frame work.



Let’s get some fundamentals of Docker and swarm straight before going further.

## Docker fundamentals

Docker is a tool that helps create and run applications in a shell called container. Applications running in container will only see resources that are made available to it e.g., Network interfaces, directory filesystem, CPU shares, Memory Share etc. Docker supports creating these containers from the application in a very simplified approach, this is explained in the later sections. A Docker node is a machine which runs Docker daemon and can run these applications inside containers. Docker also built a clustering infrastructure which orchestrates these containers across a cluster of docker nodes. Docker calls the cluster as Swarm. There are similar technologies available e.g., Kubernetes, Mesas. App frame work is built on Docker Swarm infrastructure mainly because of

1. simplicity of deployment
2. simple scale that is required for DCNM

Docker Swarm cluster provides various features:

1. Applications can be created as a service and Docker Swarm ensures that these applications are always running, that is if an application exits Docker automatically restarts the container.
2. Applications can be deployed as multiple replicas, so certain work can be executed in a load balanced manner across multiple nodes or instances.
3. Applications can publish certain service via a TCP port and can be utilized by other applications or an external client from any of the nodes in the swarm cluster. Such service can also be load balanced across multiple instances of the same application service.
4. Applications can be attached to a variety of persistent storage options. They include some simple options like NFS and local Directory binding.

More information about Docker containers and their deep dive can be obtained from. Let’s now dive into understanding how an application can be developed inside an application framework.

# Anatomy of Application Package

A DCNM application is comprised of the following

1. An application specification
2. A Docker container image as a simple tar ball
3. A Frontend Package, if desired to be run on DCNM webserver
4. An icon that best depicts the application on the DCNM Catalog

## Application Specification

An application Spec describes to the Application frame work how the application should be deployed and what resources it requires to function correctly. Specification is written in YAML which is very to write and is readable much better than JSON. A full spec is described below, Mandatory fields are in Bold.

---

*# A Name for this application. It is suggested to choose a unique name for the application*

**Name**: APP Name

*# An AppId is App Name without spaces. It can contain ‘\_’ special character. Example (epl\_1)*

**AppId**: <App\_name>

*# Vendor is the app vendor who is managing and releasing the app. Example (Cisco etc)*

**Vendor**: <Vendor\_name>

*# Version can only have ‘.’ special character. Example (0.1 or 0.1.2)*

**Version**: <version>

*# An image tag. This should follow the format of <imageName>:<Version>. Version*

*# cannot be simply latest. It has to be a decimal denoting Min and Max version.*

*#****The imageName has to be unique across different applications***

**Tag**: <imageName>:<version>

**Description**: Template APP specification

*# The following Tag Describes Dependency images for this App to be runnable. In order to*

*# deploy the app following images need to be available on the DCNM*.

Depends:

- Tag1

- Tag2

 - Tag3

*# A Meaningful representation of the image category. Use one of Application or Infra*

**Category**: Application / Infra

*# How should this App be deployed?*

Deployment:

*# Should this be app be run on the installation of DCNM? If it is default then it will be*

*# started as soon DCNM is installed and brought up on the boot.*

Default: yes

*# Do we need an instance of this application per Fabric scope in the DCNM?*

FabricAware: true

*# How many instances of this application do we need? ‘\*’ indicates that it is a global service i.e. one container per compute.*

Replication:

StartCount: { \* | <integer : [ 1 - 16 ]> }

MaxCount: { \* | <integer : [ 1- 16]> }

*# The Resources this Application needs to correctly run. This section describes volumes.*

*# CPU Memory and Network. These values are editable on the DCNM after it is uploaded*

*# while the image is not deployed.*

**Resource**:

# The following two values are mandatory

**CPU**: {[1-9][0-9]\*[MmGg] : [256M – 2G]}

**MEMORY**: {[1-9][0-9]\*[MmGg] : [256M – 2G]}

# Specify a only if you need a fabric connection, or to reach this application

# from outside of DCNM. A browser connection is not treated as outside.

# A good example of outside of DCNM is a fabric switch trying to reach this

# application, e.g, switch telemetry.

NET: {[inbandfabricnetwork | oobfabricnetwork]}

VOL:

*# The following mount will be provisioned from NFS as it states Shared is true. It*

*# is strictly advised to use this for storing small amounts of data.*

/mnt/mount1:

**Shared**: true

**Persistent**: true

**Limit**: {[1-9][0-9]\*[MmGg] : [8M – 32M]}

*# The following mount will be provisioned on the host where the container will be*

*# instantiated. This is essentially a Directory inside the host.****This data is not***

***# moved to a different host in case the container of the application moves****.*

/mnt/mount2:

**Shared**: false

**Persistent**: true

**Limit**: {[1-9][0-9]\*[MmGg] : [32M – 4G]

*# What Services is this Application publishing. This will be available to interested applications*

*# when they query for it using the App Framework service discovery. The port number specified*

*# here is the port that application has actually used in its implementation. Service Names*

*# can be simple uniquely identifiable names as strings.*

Services:

**<serviceName1>:**

**Type**: {[internal, external]}

**Port**: <PortNumber [ 1 – 65535]>

**Proto:** {[udp, tcp]}

.

.

.

.

**<serviceNameN>:**

**Type**: {[internal, external]}

**Port**: <PortNumber [ 1 – 65535]>

**Proto:** {[udp, tcp]}

*# If this application has GUI then they are listed below.*

GUI:

#App needs to be specify atleast one of Integrated or Offloaded Option.

*# An integrated GUI essentially means that the APP has its own webserver and content*

*# to serve. DCNM will plumb a dynamic port into the App. The plumbing is only*

*# between DCNM Server and Compute. This port is not externally visible through*

*# any worker nodes. Only one such port is supported per application. This value does*

*# not have to be unique across multiple applications.*

**Integrated**:

**Port**: 80

**Protocol**: http

*# Offloaded GUI package. These are war files that will be packaged with the application*

*# image. These files are extracted from the image and saved into DCNM, so the GUI can*

*# be served from the DCNM Server itself.*

**Offloaded**:

*# A File name inside the application image. The name mentioned here should*

*#****exactly match****with the file packaged inside the application image.*

**File**: {.zip file}

*# A filename with the path inside the zip that needs to be treated as the home*

*# page for this frontend. If none specified it is assumed as index.html*

Default: configure.html

## Docker Container Image

An application may be developed in various programming languages. While some developers are convenient with Python, some developers prefer Go or Java. Application frame work expects the application to be pre-packaged as a docker container. This is the right thing to do as the application developers have a better knowledge of the application’s requirement. Building a Docker container once the application is developed is actually a trivial thing. A Docker container is usually built using a Dockerfile. Following link [https://docs.docker.com/engine/reference/builder/#environment-replacement](https://mail.cisco.com/owa/UrlBlockedError.aspx) describes in detail various constructs  of a Dockerfile and how a docker image can be built.

One very important criteria of an image packaging is that it should be as compact as possible. Developing in Java requires that the Open JDK be packaged in the container. Open JDK containers tend to be bigger in size and then adding application on top also increases the container size. Python has issue with having to figure out all the packages required and ensure that they are packaged into the final application package container.

Go is a very micro service friendly language, once developed and compiled into binary it does not have any other dependency, no shared object or run time package dependency.

One other important consideration is what image to start Docker container from, we strongly suggest that base container be from an alpine Linux. With a GO based application using an alpine container the overhead is not more than 4 – 5 Meg.

Below is an example script that can be used to containerize your application.  Usage of the script is

./buildInfraGlobalApp.sh foo:0.1 Apso

The following

#!/bin/bash

name=`echo $1 | cut -d ":" -f 1`

echo $name

title=$2

mkdir -p packDir

sed "s/NAME/$1/" htmlTemplate  > index.html

echo "

---

Name: \"$name\"

AppId: \"$appid\"

Vendor: \"$vendor\"

Version: \"$version\"

Tag: \"$1\"

Description: \"Test Application Generated\"

Type: \"Fabric aware application\"

Deployment:

  FabricAware: True

  Replication:

     StartCount: \"\*\"

     MaxCount: \"\*\"

  Default: false

Category: Infra

Resource:

  CPU: 256M

  MEMORY: 512M

  NET: \"InBandFabricNetwork EastWestCluster\"

GUI:

  Integrated:

    Port: 80

    Protocol: http

" > packDir/spec.yml

cp AppIcon.png packDir/

echo "Creating Docker Image"

docker build -t $1 .

docker save $1 -o image.tar

chown $USER:$USER image.tar

docker rmi $1

echo "Pack DIR has been built, Now creating Docker Image"

mv image.tar packDir/

tar -czvf $name.tar.gz packDir/

echo "Image $name has been built"

Following is the Dockerfile used by docker to build the application. In this case Application is developed in Go and serves a simple index.html at port 80 within itself.

FROM alpine:latest

MAINTAINER Rajesh Nataraja <rajenata@cisco.com>

# Make sure to Modify the Proxy Server values if required

COPY demoApp.bin /bin

RUN mkdir -p /var/www

COPY index.html /var/www/

ENTRYPOINT ["/bin/demoApp.bin"]

Following is an example of Docker file used to build a Java based application, In this particular case it is elasticsearch.

FROM openjdk:8-jre-alpine

# ensure elasticsearch user exists

RUN addgroup -S elasticsearch && adduser -S -G elasticsearch elasticsearch

# grab su-exec for easy step-down from root

# and bash for "bin/elasticsearch" among others

RUN apk add --no-cache 'su-exec>=0.2' bash

# https://artifacts.elastic.co/GPG-KEY-elasticsearch

ENV GPG\_KEY 46095ACC8548582C1A2699A9D27D666CD88E42B4

WORKDIR /usr/share/elasticsearch

ENV PATH /usr/share/elasticsearch/bin:$PATH

ENV ELASTICSEARCH\_VERSION 2.4.5

ENV ELASTICSEARCH\_TARBALL\_SHA1="180353a1a65995f5e4533ff0a58f18e1e85f28ae"

COPY elasticsearch.tar.gz.asc /usr/share/elasticsearch/

COPY elasticsearch.tar.gz /usr/share/elasticsearch/

RUN set -ex; \

\

apk add --no-cache --virtual .fetch-deps \

ca-certificates \

gnupg \

openssl \

tar \

; \

pwd; \

\

tar -xf elasticsearch.tar.gz --strip-components=1; \

rm elasticsearch.tar.gz; \

\

apk del .fetch-deps; \

\

mkdir -p ./plugins; \

for path in \

./data \

./logs \

./config \

./config/scripts \

; do \

mkdir -p "$path"; \

chown -R elasticsearch:elasticsearch "$path"; \

done;

COPY config ./config

COPY esInit.sh /usr/bin/

COPY esCluster /usr/bin/

VOLUME /usr/share/elasticsearch/data

COPY docker-entrypoint.sh /

WORKDIR /usr/bin/

EXPOSE 9200 9300

ENTRYPOINT ["/docker-entrypoint.sh"]

CMD ["esInit.sh"]

In the above Dockerfile, container was based off of “openjdk:8-jre-alpine”. Python also has a Docker image that is based on alpine, e.g., python:3.4.6-alpine . For the earlier Go based application it was directly based on alpine:latest directly as go builds a static binary. The only trick to building GO code for alpine is to avoid lab dependency by compiling as follows:

setenv CGO\_ENABLED 0

setenv GOOS linux

go build -a -installsuffix cgo -o eplc.bin eplcmain.go

Here is an output of Docker images and their size comparison:

RAJENATA-MAC$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE

python              3.4.6-alpine        37f0ecae85a6        8 days ago          83.1 MB

openjdk             8-jre-alpine        c017141bdaa8        2 months ago        81.4 MB

alpine              latest              4a415e366388        3 months ago        3.99 MB

## Front End Package

Front end package contains all the elements required to serve the application’s front-end on the browser. Application frontend is only available if the user is logged in from DCNM. There is no way to view the application front-end while not being logged in. As described in the Application Spec, frame work needs to be provided information about the applications frontend. For an Offloaded application, Specified zip fill will be extracted and placed into the DCNM Server. A separate section later describes developing Front End for an application in the DCNM Application Frame work.

# Life as a DCNM Application Container

## Environment

All containers will be provided with information about how to reach DCNM via the environment variables. Following are the environment variables exported to the container by the application framework.

# The following value provides IP address to make DCNM API calls to

DCNM\_MGMT\_VIP=10.1.0.2

#Name of the file containing token file which is in the /run/secrets directory

APP\_DAEMON\_TOKEN\_FILE=compliance\_Cisco\_Fab2\_sct

#The Fabric Scope for which the application is being spawned. Please note that an application can be

# instantiated once per fabric it is specified as Fabric Aware.

DCNM\_FABRIC\_ID=Fab1

## Fabric Connectivity

Many of the applications developed inside the DCNM will need a way to connect to the switches in the fabric for various reasons, some listed below:

1. Be a neighbor and talk to a protocol on the Switch
2. Receive data from the switch via telemetry streaming
3. Receive REST requests from the Switch (e.g., PMN if containerized)

As noted in the application specification, an application can specify what networks it needs to be connected to. There are three kinds of networks available

1. InbandFabricNetwork
2. OoBFabricNetwork
3. EastWestcluster

Application is always connected to EastWestCluster irrespective of the spec information. Application can choose to be connected with one of InBandFabric or OoBFabric . For each network that is specified, each container of the application will get an IP address. The IP address is unique across all the applications. Eastwest cluster is only visible behind the swarm and not available to be reached from or by the fabric. All the containers talk to each other via the east west cluster. Every application also gets a VIP for all its containers in the eastwest cluster.

The Inband and OoB fabric networks are used to access the fabric underlay and management networks respectively without having to go through the NAT. These IP addresses are reachable from the fabric in case switch needs to reach the container. There is no concept of VIP for these networks at this time, may be supported in the future if required. At this time there is no consistent naming that can be used to identify which network belongs what interface inside the application’s container. But Default route always points towards the Fabric Network when it is connected. If the application is not connected to fabric then the default route is NATed to the outside world. If an application wants to extract its external IP for setting up reachability, it should be done as follows:

Extract Environment Variable APP\_EXTERNAL\_IP and use it, this is only available when the container is NATed out to the external world. If the Container is indeed connected to external fabric using the default route interface IP address is the best way to do it.

## Service Discovery

Applications will need to use certain services e.g., Post Gres Database, Elasticsearch database etc. An application publishing a service needs to be specify all of its services via the Service section the application spec as described in the earlier sections. This is an example of elasticsearch specification publishing its API port.

Services:

    elasticsearch-2.3.3:

       IP: Dynamic

       Port: 9200

It is a good idea to embed version into the service name, but it should ideally be different from container version of that particular application. Importantly because APIs mostly are compatible among certain versions. Another way is to list all compatible versions against the same port number here.

When AFW notices the application-spec contains services section it will make an entry of all the services with relevant information in its database. This information is served directly to the application that is trying to discovery this service. Following REST API, as described by swagger, can be used for service discovery, please note that all the variables of the request are available in the environment of a requesting container.

### GET/afw/service/{serviceid}

Returns Service Information for the service ID

***4.3.1.1 Parameters***

|  |  |
| --- | --- |
| **Name** | **Description** |
| serviceid \*  string  (path) | ID of the service to discover |

***4.3.1.2 Responses***

Response content type

|  |  |
| --- | --- |
| Code | Description |
| 200 | successful operation  Example Value  Model  {  "ServiceName":"string",  "FabricId":"string",  "ImageTag":"string",  "IPAddress":"string",  "Port":**0**,  "PublicIP":"string",  "PublishedPort":**0**,  "Protocol":"string"  } |
| 400 | Invalid ID supplied |
| 404 | Service Not Found |

### GET/afw/service/{serviceid}/{fabricid}

Returns Service Information for the service ID

#### Parameters

|  |  |
| --- | --- |
| **Name** | **Description** |
| fabricid \*  string  (path) | Fabric Scope in which this service discovery is being done. |
| serviceid \*  string  (path) | ID of the service to discover |

#### Responses

Response content type

|  |  |
| --- | --- |
| Code | Description |
| 200 | successful operation  Example Value  Model  {  "ServiceName":"string",  "FabricId":"string",  "ImageTag":"string",  "IPAddress":"string",  "Port":**0**,  "PublicIP":"string",  "PublishedPort":**0**,  "Protocol":"string"  } |
| 400 | Invalid ID supplied |
| 404 | Service Not Found |

It is not possible for App Framework to ensure that applications providing a service are completely up, and hence it is strongly suggested that applications ensure that service is functional through some means, e.g., issuing a GET before starting to save state or do more meaningful calls. It is also possible that the service is not published yet, if a service discovery API fails it may succeed after sometime. So applications should retry to the best of their design to perform a successful serice discovery. Just like an application can do service discovery from its container. An application can do service discovery from its front-end via the browser as well. Reason for the need to perform service discovery from the browser is explained in the Front-End Development section. Here is a GO example for performing service discovery.

func getServiceInfo(dcnmIP string, serviceName string) (serviceDiscoveryResp, error) {

var svc serviceDiscoveryResp

fmt.Println("http://" + dcnmIP + ":35000" + "/afw/service/" + serviceName)

resp, err := http.Get("http://" + dcnmIP + ":35000" + "/afw/service/" + serviceName)

if err != nil {

fmt.Println("Error in Get: ", err.Error())

return svc, err

}

err = json.NewDecoder(resp.Body).Decode(&svc)

if err != nil {

fmt.Println("Error in decode: ", err.Error())

return svc, err

}

return svc, nil

}

.

.

.

fmt.Printf("Perform Service Discovery for Elastic Search")

for {

svc, err = getServiceInfo(dcnmIP, "elasticsearch-2.3.3")

if err != nil {

fmt.Printf("Waiting for Elastic Search Discovery to be available")

time.Sleep(5 \* time.Second)

continue

}

time.Sleep(2 \* time.Second)

break

}

esIP = svc.ServiceIP

esPort = svc.Port

.

.

.

Following is an example of How to perform Service discovery from the front-end using Javascript.

      <script src="/js/dcnm/util/AfwApi.js"> </script>

.

.

.

    esLocation = AfwDiscoverService("elasticsearch-2.3.3", "fabrica");

    if (esLocation.length == 0) {

        alert("Failure Discoverying Elastic Search Service");

        return null;

    }

## Developing Front End

Every Application in the AFW can have a front end which can only be served via the DCNM UI iframe. AFW will redirect the application to the front-end via a click on the catalog icon while its running. Following are the simple rules to keep in mind while developing frontend:

1. Logging into DCNM is mandatory to access the frontend.

2. Frontend is accessible only if the application is in the running state.

3. For a fabric aware application frontend should be per fabric.

4. The only way an application can reach its frontend pages is to go via DCNM iframe.

Application Framework supports two types of Front end.

### Offloaded

An offloaded method of Frontend is used by applications that do not run webserver with in them. In this method, AFW will deploy the zip file and serve it directly from DCNM Server. AFW will also ensure that the catalog icon URL has appropriate URL, so that it is redirected to the correct frontend. This approach is better if application intends to be of a tiny foot print and comprises of HTML / Java script. This method also renders slightly faster due to not requiring an extra hop into the application container. Developers can create a zip of their entire front-end directory and provide that as a file in the image package, DCNM will extract the zip file and load it in the relevant location for DCNM to serve.

### Integrated

In this method application hosts its own webserver. It may be of any form, like an nginx or apache running inside container to as simple as an application hosting its own port to serve simple information. Sometimes the integrated webserver can also be a few REST apis that the application requires, e.g. EPL, see below:

http.HandleFunc("/about", aboutEndPointLocator)

http.HandleFunc("/status", endPointLocatorStatus)

http.HandleFunc("/start", startEndPointLocator)

http.HandleFunc("/rrStatus", retrieveRRStatus)

http.HandleFunc("/stop", stopEndPointLocator)

fmt.Printf("Now starting REST Listen\n")

http.ListenAndServe(":80", nil)

To achieve an integrated frontend an application spec has to specify in the GUI an integrated section with port and protocol that it uses to serve the content. The port specified in the Integrated section is the port that container is listening to for serving the pages. The specified port is not available outside the container, AFW will publish a different port for use from outside the swarm cluster. AFW will discover the frontend of the application and redirect to the appropriate URL.

### Writing and Serving Pages

While developing web pages for AFW is not much different than normal web development. But there are some important details to follow so that front-end is realized correctly. Application can never refer to their sub-pages using an absolute path. Sub Pages could be Java Script files, stylesheets or other HTML hrefs. E.g,

*<a href=”./foo/bar.html”> foo < /a>* is a right way to specify the reference to foo/bar.html inside applications index.html. But assuming root path and using as *<a href=”/foo/bar.html”> foo < /a>* is not correct and the iframe will fail to locate page.

In some use cases it will be required to access applications front end from a totally random location and specifying relative path may be difficult. In those scenarios a Javascript library can provide the URL of an application’s front end. The API can be called as follows

AfwGetFrontEndUrl(appId)

Where appId is <vendorId>:<AppId>:<Version> as specified by the application in its spec. The URL returned by the above API can be concatenated with the application’s required URL for use in its frontend logic. AFW Java Script API can be imported as follows

*<script src="/js/dcnm/util/AfwApi.js"> </script>*

## Clustering

A cool feature provided by the application framework is to be able to instantiate multiple containers of the same application. This is very useful when an application is designed to collectively handle large computation. Such applications are called replicated applications.

An app can specify its replication through the application spec as shown in the previous section. All Network attachments are individually performed on each of the container. There are two modes of replicated applications

1. Count Replicated Mode
2. Global Replicated Mode

A Count Replicated application initially starts of at a particular count and can be replicated from the GUI up to a Max Count specified in its application spec. A Global mode application behaves a bit differently, in this case an application gets one container created on each of the Compute node of the DCNM. Global Mode applications cannot be replicated from the GUI.

It will be required for application containers to know each other when operating collectively, such information is exported, via a read only configuration file: “/etc/MyClyster.json”, into every container of the application. The file is in JSON format and lists down information about each container that is running for the application. Application needs to build resiliency in their design to cater for containers getting removed, added and being unreachable in the cluster list. Application framework does not ensure that a container listed in the cluster file is actually working as expected, it will be listed if the information is known to application frame work. Following is the format of the cluster file, Pls note that comments are only for readability

{

/\* Virtual IP address assigned to all the containers. This Ip address is

* For use by a different application. A container in the same application
* Using this will result into reaching its self.

\*/

"ApplicationIP": "10.1.0.2",

/\* List of all the Members in this application. \*/

"MemberList": [{

"MemberName": "foo.1", /\* Just a Name, no functionality attached \*/

/\* All the IP Addresses attached to the container \*/

"InbandFabricIP": "",

"OoBFabricIP": "",

"EastWestIP": "10.1.0.3"

}, {

"MemberName": "foo.2",

"InbandFabricIP": "",

"OoBFabricIP": "",

"EastWestIP": "10.1.0.4"

}]

}

Here is an example of how an elastic search database can be deployed in a cluster. The snippets of code below give an idea of how to use the cluster file to manage the application cluster.

func monitorFileChange() {

for {

time.Sleep(5 \* time.Second)

stat, err := os.Stat(“/etc/MyCluster.json”)

if err != nil {

/\* There is no cluster file yet \*/

continue

}

if stat.ModTime() != initialStat.ModTime() {

/\* Cluster file has changed we need to handle any changes \*/

initialStat = stat

break

}

}

}

func main() {

/\* Snip \*/

for {

monitorFileChange()

elasticHandleFileChange()

}

/\* Snip \*/

}

Sometimes in a clustered solution, a cluster manager may be built as a separate container / application. But the cluster file is only available to those applications that are in the application that specified the specification. In the app framework, a simple way to implement the cluster manager solution is to create a service in the spec of the cluster manager which can be used by the clustered containers to report their information. Cluster Manager service can be dynamically discovered by the clustered containers using the App framework service discovery.

## Storage

Applications in the AFW have the following options for storage

1. Non – Persistent Storage
2. Persistent Shared Storage
3. Persistent Local Storage

When a persistent storage is required by an application it needs to be explicitly specified in the Application’s YAML spec as described in the previous section. A persistent storage is not mandatory for an application, it’s the choice of an application to have such storage.

### Non-Persistent Storage

A Non – Persistent storage is implicitly available to the container and does not have to be specified in the application spec. Any application desiring to have a mount point for a temporary storage can do so by simply specifying MKDIR command inside the Dockerfile.

This storage data is not available across the reload of a coantiner, it is temporary and cleared up.

### Persistent Local Storage

This type of storage is only available inside the node where the container of the application is running. A disk image on the local compute is mounted into the specified path of the application’s container. The following two rules are very important

1. If container restarts in a different location due to a node failure, storage data is not migrated to the new node.
2. When a replicated container uses this method, it needs to be aware that the same storage is also accessed by a different container if its running in the same node. This is most likely to happen if the number of replications are more than the number of nodes available. It is applications responsibility to handles such concurrent access.
3. If an application cannot manage concurrency then it has an option choose global replication mode.

In general, the limit of a local storage can be very high, this makes it a good candidate for storing database content or large log files.

### Persistent Shared Storage

The method of accessing the storage is not much different from the previous method, but in this scenario storage is mounted via an NFS server. The same NFS server is available on all the computes, and hence date written in one node is available at all the nodes. If a container gets re-created in a different node its data is not lost. This type of storage is very useful to applications for storing their persistent configuration, which needs to be available in case the container restarts. It is strongly discouraged to rely on this storage for storing gigabytes of data. The issue of concurrency applies to this storage as well, if multiple containers running in the application. Application needs to be handle such concurrency.

### Handling Concurrency

One easier way of handling concurrency is to create individual directories for each of the replicated instance of the container. This can be done by looking at the “/etc/MyCluster.json” and matching any of the available interfaces with the member list. Once matched the Member name can be used as a sub directory inside the mount point. Other applications like Elastic Search create a proprietary concept of node id where in each container gets a node id. The node ID is used as a sub directory as shown below from the snapshot of elastic search application.

[root@rajenata-8-212 nodes]# ls -l

total 4

drwxr-xr-x. 3 100 101 4096 Jun 9 10:56 0

drwxr-xr-x. 3 100 101 4096 Jun 9 10:57 1

[root@rajenata-8-212 nodes]# pwd

/var/lib/dcnm/afw/vols/data/afwelastic\_0\_1\_\_afw/usr/share/elasticsearch/data/afw\_elastic\_cluster/nodes

[root@rajenata-8-212 nodes]#

## Application DCNM API Access

An app running the application hosting framework can make calls to DCNM without having to log into DCNM. This basically avoids having to provide a particular user credentials to the app for making the calls. There are at present two kinds of tokens available to the application based on the kind of call it makes.

### Daemon-Token

Every application can make calls using a Token that is made available to it when the container is created. This token is called a Daemon Token and this is available for the life of the application. The Daemon-Token has a privilege of a DCNM network-operator. The location of the Daemon-token inside the application’s container can be obtained by using the environment variable “APP\_DAEMON\_TOKEN\_FILE”. Token file is stored at “/run/secrets/${APP\_DAEMON\_TOKEN\_FILE}”.

Here is an example of how compliance app token file is stored.

*/run/secrets # ls*

*compliance\_Cisco\_CoreSpine\_sct*

And the contents of the token as are below

/run/secrets # cat compliance\_Cisco\_CoreSpine\_sct

{"token":"eyJhbGciOiJSU0EtT0FFUC0yNTYiLCJ0eXAiOiJKV1QiLCJlbmMiOiJBMTI4R0NNIn0=.hWrz17Q8vaILZNg-yzoLamzPzCr15TqT9BbSRkmrINlJ0zX\_K0CwOf1KGjfikXAAfWdxTtaW8BhoUinjAShAXP\_OhXvMwBUvEI3jcj8d7vNsBCpyUDYAH\_TJjop3MV219icgG-cs1HMPJPEEujocWhgAfVTPR5axJB8KX3ZCeSarwe1BVnErx7WNPIMrjw8CjpYB12ZyMY3Z46dsqxtqfggGWiqohF5gB\_kg-ynAq-xb53-n-FF44mgQCi\_I1nzXVT1iWq3uK\_X5A3Be0jOQwW8tLFEtFX2sEleoYLv4\_V9jIkEx9VooSStekaVwaRZ0dEZihYzQv1Wnpr5z6Yi\_Bw==.uIuZrQhkd-6bMqIK4V33MQ==.gFXOx2Q5\_IQwH9NIRxkIIfKT41s-rPzmuE1ML0N5iokM3eYdAb-6DOA8hDn5egAmeOdiHmxEtFJrweaxgRC\_2qCthUJmr2axbxNrqkyKSfTTqjifjQkjyqdpe-tteUu-DVfeTXStPRGHQbfriNereYHjBaKTd0EKG-L6ws9B7NhrPTq8vkp4mUH20To1bEUHUAX-Wm-Gl8lA4P31isSrcxmavTTeOc\_KDP84xp2BTbvdQl3v4PNuGA==.UePwktG275yWWqSfm6i9Cw==","timestamp":1527704710428,"afwTokenType":"APP\_TOKEN","appid":"compliance","role":"network-operator"}

An application making calls to DCNM using daemon-token will need to extract the token values as shown in the above file and pass them in the REST call to DCNM.

### User-App Token

In some scenarios applications need to make calls that are attached with certain user’s credentials or role. An application when being driven through the DCNM frontend is under the control of that user’s session. This user’s session is passed all the way to the application when the frontend makes API calls to application. The session headers contain a token and we call this token as user-app-token. For example, if the user “admin” is using the application frontend, a REST call from the DCNM frontend browser will pass the user-app-token all the way to the application. In this case user-app-token is associated with the user “admin”’s credentials.

The REST call received by the application contains a header name “uat”. This value can be used as a token while making calls to the application. Here is an example code that extracts the uat.

*uat := reqCtx.Header.Get("uat")*

*fmt.Printf("Received UAT %s\n", uat)*

This token is only valid as long as the user is in session, when the user logs out this token is invalid.

### Making API Call

Either of the tokens described above can be supplied as the header fields in the REST call made to DCNM. The token used depends on the type of the call made. Here is an example in golang that describes how to make the call.

*req, err = http.NewRequest("GET", urlStr, nil)*

*req.Header.Set("app-token", MyTokenOfChoice)*

*req.Header.Set("Content-Type", "appplication/json")*

No calls can be made if an app does not know the IP address where the call needs to be made. An application can basically query the environment variable “DCNM\_MGMT\_VIP” which contains the IP address of the DCNM it can make the API calls to.

All of the method described here does not necessarily have to be implemented by an application. The following git hub (internal to cisco) contains basic packages in python/golang/java that can make API calls by just invoking the package function as required.

https://{{YOUR [USERID}}@bitbucket-eng-sjc1.cisco.com/bitbucket/scm/dca/dcnm-afw-sdk.git](mailto:USERID%7d%7d@bitbucket-eng-sjc1.cisco.com/bitbucket/scm/dca/dcnm-afw-sdk.git)

PS: These packages may have modifications that need to be committed back.