Western Digital POC Plan

v2.0 4/11/18

Introduction:

Western Digital is currently exploring container orchestration and big data platform for an edge-computing use case to be implemented globally. The end solution will consist of microservices leveraging certified data services (Kafka, Spark, HDFS) that DC/OS provides.

WHO - Who will be performing the DC/OS Evaluation?

Mesosphere	Western Digital
Account Executive: Justin Moayed Solutions Engineer: Alex Ly, Corbin Pacheco Mesosphere Tech Support as Needed	James Trier - Sr. Manager Engineering, Big Data Platform & Analytics Keehan Mallon - Software Developer Matt Bigelow - Sr. Big Data Engineer Jeremy Bolie - Solutions Architect Fayaz Syed - Sr. Manager Engineering, Big Data Platform & Analytics

Areas of Interest:

Use Case	Description	Success Criteria	References	Weighting
Container Orchestration	Containers (Docker/Mesos) should be deployed with self-healing, upgrading, scaling, and service-discovery capabilities to ensure High Availability and Performance of running application	Ability to recover datacenter workloads from application or infrastructure failures such as node shutdown, rescheduling, and load-balancing service discovery to ensure maximum uptime.	DC/OS - Deploying Services and Pods DC/OS Networking DC/OS - DNS Service Discovery DC/OS - Scaling a Service DC/OS - Rolling Upgrades	

		Zero downtime upgrades of microservices		
Kafka	Demonstrate the ability that Kafka users are able to deploy a Kafka cluster on-demand that is secure, scalable, highly available, and easily upgradeable	Single Click Deploy of distributed highly available architecture with no Single-Point-of-Failure for all critical system components including system configuration, meta data & running state Display automated setup of TLS Encryption (if required) Show non-disruptive upgrade of Kafka Framework Scale Kafka cluster within DC/OS Multiple versions	Apache Kafka Service Guide Apache Kafka - Security Apache Kafka - Connecting Clients	
Spark	Demonstrate the ability that Spark users are able to deploy a Spark cluster on-demand that is secure, scalable, highly available, and easily upgradeable	Spark user needs to be able to complete ETL jobs mapped to HDFS Multiple Versions	Apache Spark - Service Guide Apache Spark - Integration with HDFS	
HDFS	Demonstrate the ability that HDFS users are able to deploy a HDFS cluster on-demand that is secure, scalable, highly available, and easily	Single Click Deploy of distributed highly available architecture with no Single-Point-of-Failure for all critical system components including system	HDFS Service Guide HDFS - Security HDFS - Connecting Clients	

	upgradeable. Operation of service should be comparable to existing vendor solutions	configuration, meta data & running state Load test data and connect clients Scale HDFS Framework Multiple Versions		
Shared DB Services	DB user should be able to self-service deploy traditional stateful database workloads (i.e. MySQL) as well as modern distributed stateful databases (Cassandra, Kafka, Elastic) from one single platform	Traditional Data Services: Deployment of Traditional DB service (MySQL, Postgres) and demonstrate setup for HA leveraging Portworx or alternative storage solution Distributed Data Services: (See above for specific success criteria)	DC/OS - Service Docs DC/OS - Storage Run Portworx with Mesosphere DC/OS	
Active Directory Integration	Solution should integrate with existing Active Directory for simplified user and group management with centralized authentication	Display successful integration of DC/OS to existing Active Directory Ability to add local users and segregate them by group Permissions test on AD users with RBAC controls	DC/OS - Directory Based Authentication via LDAP	

Other Notes to Consider:

- Initial POC exercises will be started in AWS sandbox environment in parallel to ordering of baremetal production infrastructure
    - Lead time for procurement of physical infrastructure - ~6-8 weeks

- Autoscaling is not a current feature out-of-the-box for Marathon services but is in the DC/OS roadmap
  - An Autoscaling Tutorial is available to show how to do Marathon autoscaling with minor effort - Unsupported at the moment

Number of Servers (Mesosphere recommends at least 3 masters and 5 agent nodes and 2 public agent nodes running RHEL 7, CentOS 7 or CoreOS):

- Operating System:
- # of Master Nodes: 3
- # of Public Agent Nodes: 1
- # of Private Agent Nodes: 6

Location of Servers (data center location or public cloud vendor location):

Initial POC:

- Cloud Provider: AWS

Production deployment:

- On premises
- Locations: Bangalore, India & China Datacenters

Location of Customer Personnel:

- San Jose, CA

Location of Mesosphere Personnel:

- Justin Moayed - San Francisco, CA
- Alex Ly - San Francisco, CA
- Corbin Pacheco - San Francisco, CA
- Mesosphere Tech Support Personnel - San Francisco, CA and Hamburg, Germany

WHEN - Evaluation Start and End Dates:

Planned Start Date: AWS POC Kickoff

- Week of April 16th final review and planning
- Week of April 23rd planned start

Planned Finish Date:

- AWS POC:
  - Initial test plans completed by 5/7
  - Remaining 2 weeks for WD team to play with sandbox environment

Planned Evaluation Results Briefing Date:

- May - AWS POC Review and Briefing
- June/July - On-Premises Plan Review and Next Steps

Would like to go into production by:

Communication Cadence:

- Week of Kickoff - Lots of working sessions
- 30 minutes twice a week - Schedule TBD in following weeks
- Open Slack channel for direct communication

Multi-Phase Plan:

Phase I - AWS POC:

- Functional testing of critical must-have capabilities
  - Container Orchestration
  - Data Services
- Deploy and operate on AWS IaaS
- Develop familiarity with DC/OS Platform

Phase II - Pilot:

- Pilot implementation in predefined subset of Edge locations
- Pilot Architecture design and deploy
- Pilot base tech deployment (data services)
- Pilot app deployment/integration
- DC/OS Training

Phase III - Production:

- Production readiness and operations
- Monitoring, logging and security implementation
- Production rollout to remaining edge locations
- Production architecture review
- Post-pilot additional services/apps scope and plan

Planned Tests/Demos:

Container Orchestration:

- Test 1: Kill a running service task to watch Marathon rescheduling behavior
- Test 2: Kill a running application to watch Marathon rescheduling
- Test 3: Expose a service using Marathon-LB
- Test 4: Service Discovery using VIPs

Data Services:

- Test 1: Deploy HA Certified Data Service
  - GUI method
  - CLI method
- Test 2: Run a Spark HDFS Job
- Test 3: Upgrading Certified Data Service
- Test 4: Updating Data Service Configurations
- Test 5: Multiple Versions of Data Services
- Test 6: Traditional Database Services Using Local Persistent Drives

DC/OS Security:

- Test 1: Role Based Access Control
- Demo 2: LDAP Integration

Container Orchestration Test Plans:

Test #1: Kill a running service task to watch Marathon rescheduling behavior

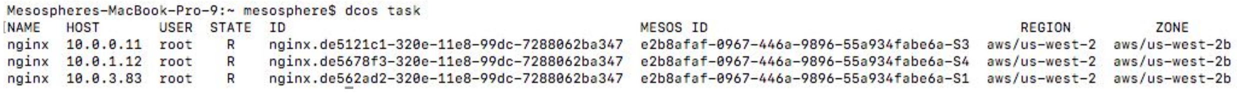
Step 1: Review and save Marathon App Definition nginx.json in Appendix A

Step 2: Deploy nginx.json application definition

```
dcos marathon app add nginx.json
```

Step 3: Use the DC/OS CLI to observe running tasks

```
dcos task
```



```
Mesospheres-MacBook-Pro-9:~ mesosphere$ dcos task
[NAME   HOST       USER  STATE  ID                                          MESOS ID                                    REGION        ZONE
 nginx  10.0.0.11  root    R    nginx.de5121c1-320e-11e8-99dc-7288062ba347  e2b8afaf-0967-446a-9896-55a934fabe6a-S3  aws/us-west-2  aws/us-west-2b
 nginx  10.0.1.12  root    R    nginx.de5678f3-320e-11e8-99dc-7288062ba347  e2b8afaf-0967-446a-9896-55a934fabe6a-S4  aws/us-west-2  aws/us-west-2b
 nginx  10.0.3.83  root    R    nginx.de562ad2-320e-11e8-99dc-7288062ba347  e2b8afaf-0967-446a-9896-55a934fabe6a-S1  aws/us-west-2  aws/us-west-2b
```

Step 4: Use the DC/OS CLI to kill a running container

```
dcos marathon task kill <ID>
```

Step 5: Observe rescheduling behavior on DC/OS UI

- Task reschedule to another node

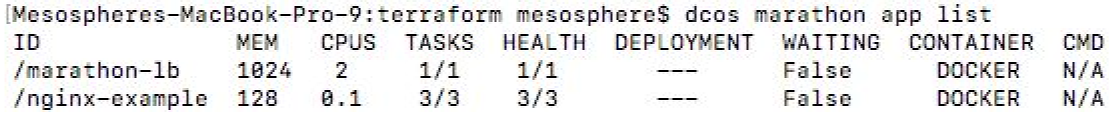
Test #2: Kill a running application to watch Marathon rescheduling

Step 1: Deploy Marathon App Definition nginx.json:

If not already deployed, deploy the nginx.json application definition from Test #1 (Appendix A)

Step 2: Use the DC/OS CLI to observe running applications

```
dcos marathon app list
```



```
[Mesospheres-MacBook-Pro-9:terraform mesosphere$ dcos marathon app list
 ID               MEM   CPUS  TASKS  HEALTH  DEPLOYMENT  WAITING  CONTAINER  CMD
 /marathon-lb     1024   2     1/1    1/1       ---       False     DOCKER    N/A
 /nginx-example   128   0.1    3/3    3/3       ---       False     DOCKER    N/A
```

Step 3: Use the DC/OS CLI to kill a running application

```
dcos marathon app kill nginx-example
```

Step 4: Observe rescheduling behavior in DC/OS UI

- All three instances will be killed simultaneously
- Observe reschedule by looking at the HOST IP in the GUI
- Containers are rescheduled to another available node

Test #3 Expose a service using Marathon-LB

Step 1: Add Labels to nginx.json Application Definition

```
{
  "labels": {
    "HAPROXY_GROUP": "external",
    "HAPROXY_0_VHOST": "<PUBLIC_NODE_IP>"
  },
```

Step 2: Re-Deploy service

```
dcos marathon app add nginx.json
```

Step 3: Access Service

```
http://<PUBLIC_NODE_IP>
```

Test #4 Service Discovery using VIPs

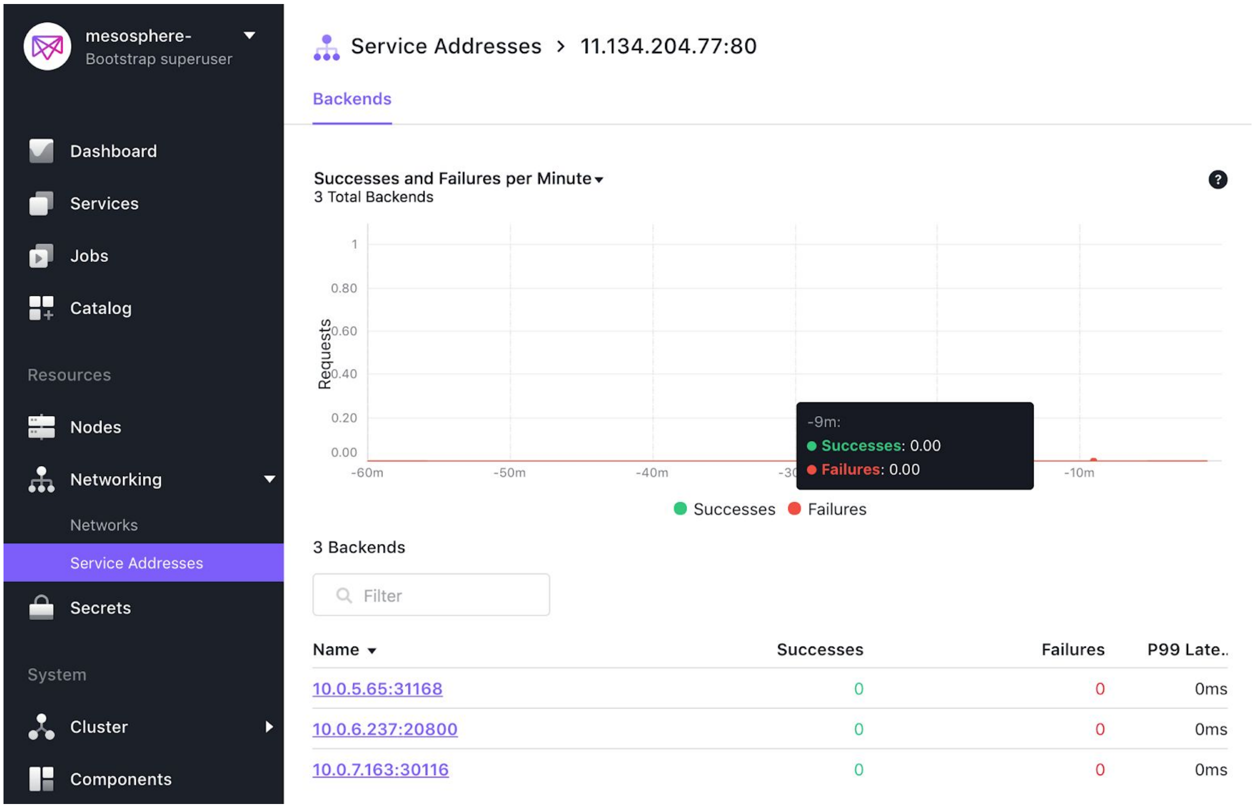
Step 1: Review and Deploy Marathon App Definition nginx.json:

If not already deployed, deploy the nginx.json application definition from Appendix A. Take a look under the “labels” parameter to see the usage of a Name-based VIP (Virtual IP Address)

```
"labels": {
          "VIP_0": "/nginx-example:80",
```

Step 2: View L4 Minuteman Service Addresses (Named-Based VIPs) in the UI

- Note that the existing # of service addresses correspond to the instance count in your nginx-example.json application definition



Service Addresses > 11.134.204.77:80

Backends

Successes and Failures per Minute
3 Total Backends

3 Backends

Filter

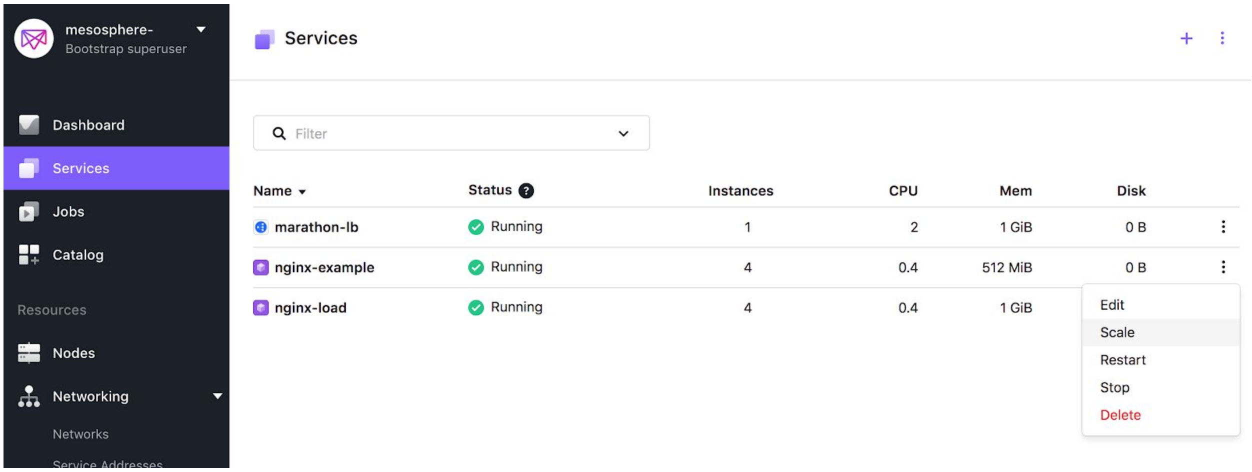
Name ▾	Successes	Failures	P99 Late..
10.0.5.65:31168	0	0	0ms
10.0.6.237:20800	0	0	0ms
10.0.7.163:30116	0	0	0ms

Step 3: Scale nginx-example service

CLI:

```
dcos marathon app update <APP_ID> instances=<TOTAL_DESIRED_INSTANCES>
```

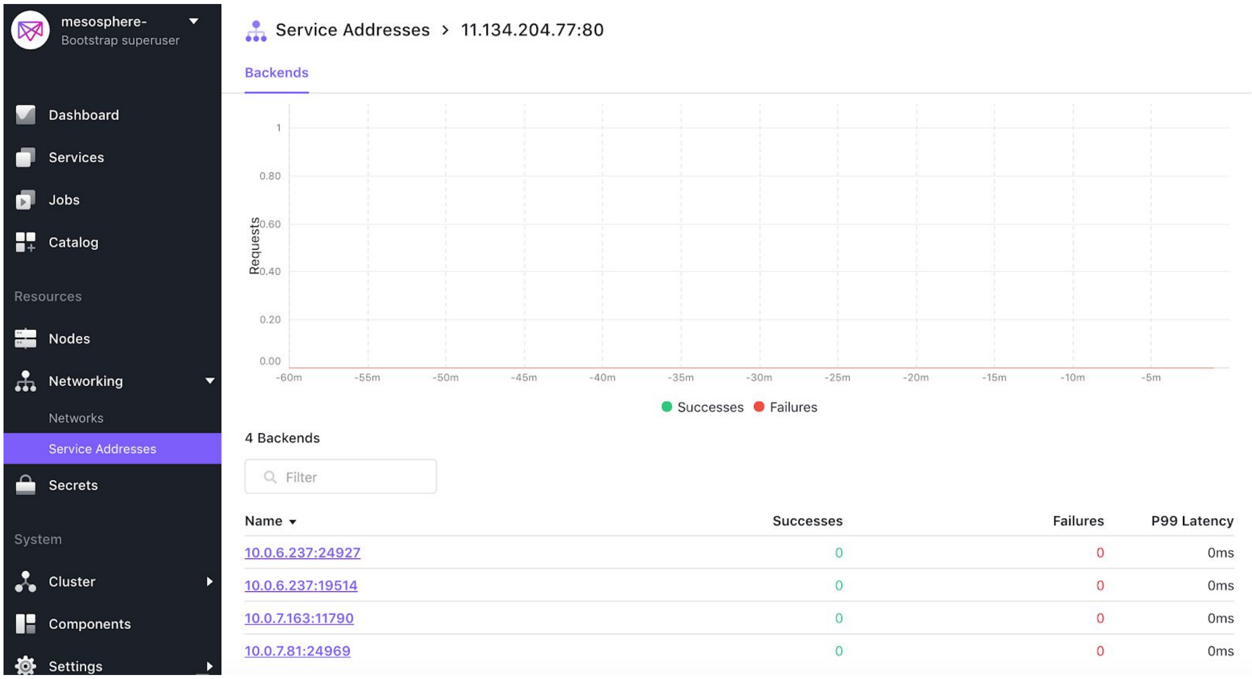
GUI:



Name	Status	Instances	CPU	Mem	Disk
marathon-lb	Running	1	2	1 GiB	0 B
nginx-example	Running	4	0.4	512 MiB	0 B
nginx-load	Running	4	0.4	1 GiB	

Step 4: Return to Service Addresses tab in the UI and observe Service Discovery

- nginx-example scaled from 3-4 instances and added to the backend pool
- If you followed the optional steps you should also see a load generated against these backends, load-balanced in round-robin



4 Backends

Name	Successes	Failures	P99 Latency
10.0.6.237:24927	0	0	0ms
10.0.6.237:19514	0	0	0ms
10.0.7.163:11790	0	0	0ms
10.0.7.81:24969	0	0	0ms

Test #5 Deploy and Expose WD specific container

Step 1: (Western Digital to Provide containerized application in .JSON format)

Step 2: Deploy WD.json application definition

```
dcos marathon app add WD.json
```

Step 3: Use the DC/OS CLI to observe running tasks

```
dcos task
```

Step 4: Expose Service

- Depending on application requirements, follow Test 3 or Marathon-LB Quickstart

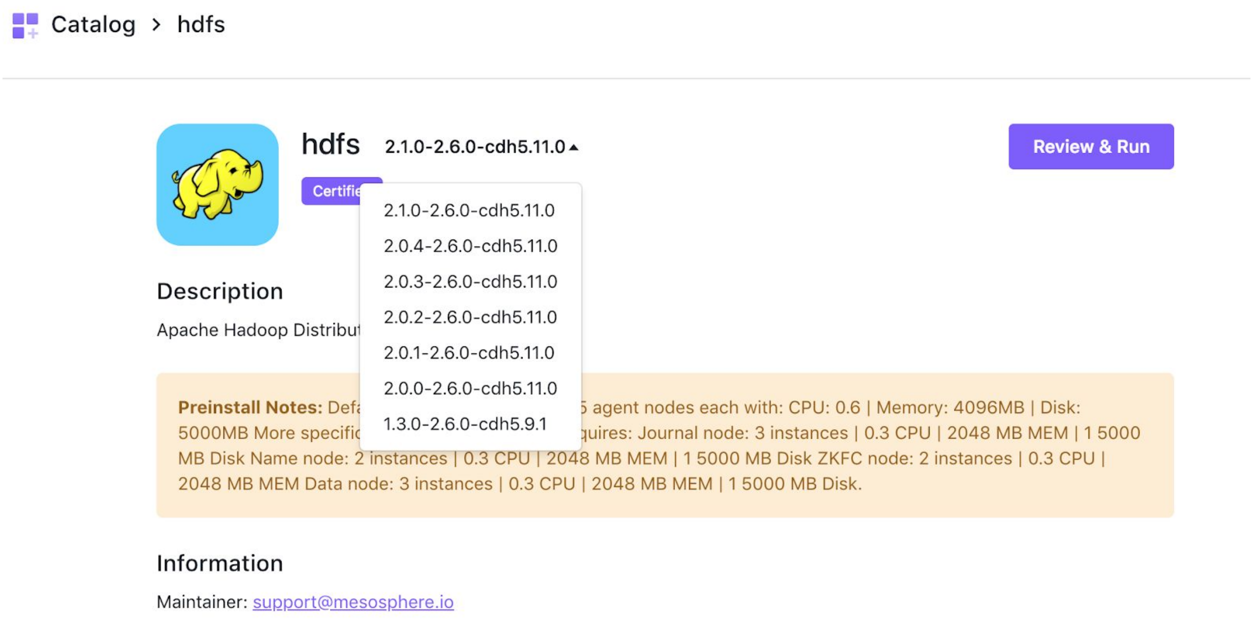
Data Services Test Plans:

Test #1 Deploy HA Certified Data Service

- Mesosphere Certified Catalog packages are built to be highly available and production ready by default
    - See Catalog Packages for a full list of existing Certified/Community packages

GUI Method:

Step 1: Navigate to the Catalog → HDFS Service → Select HDFS version → Review & Run



Description

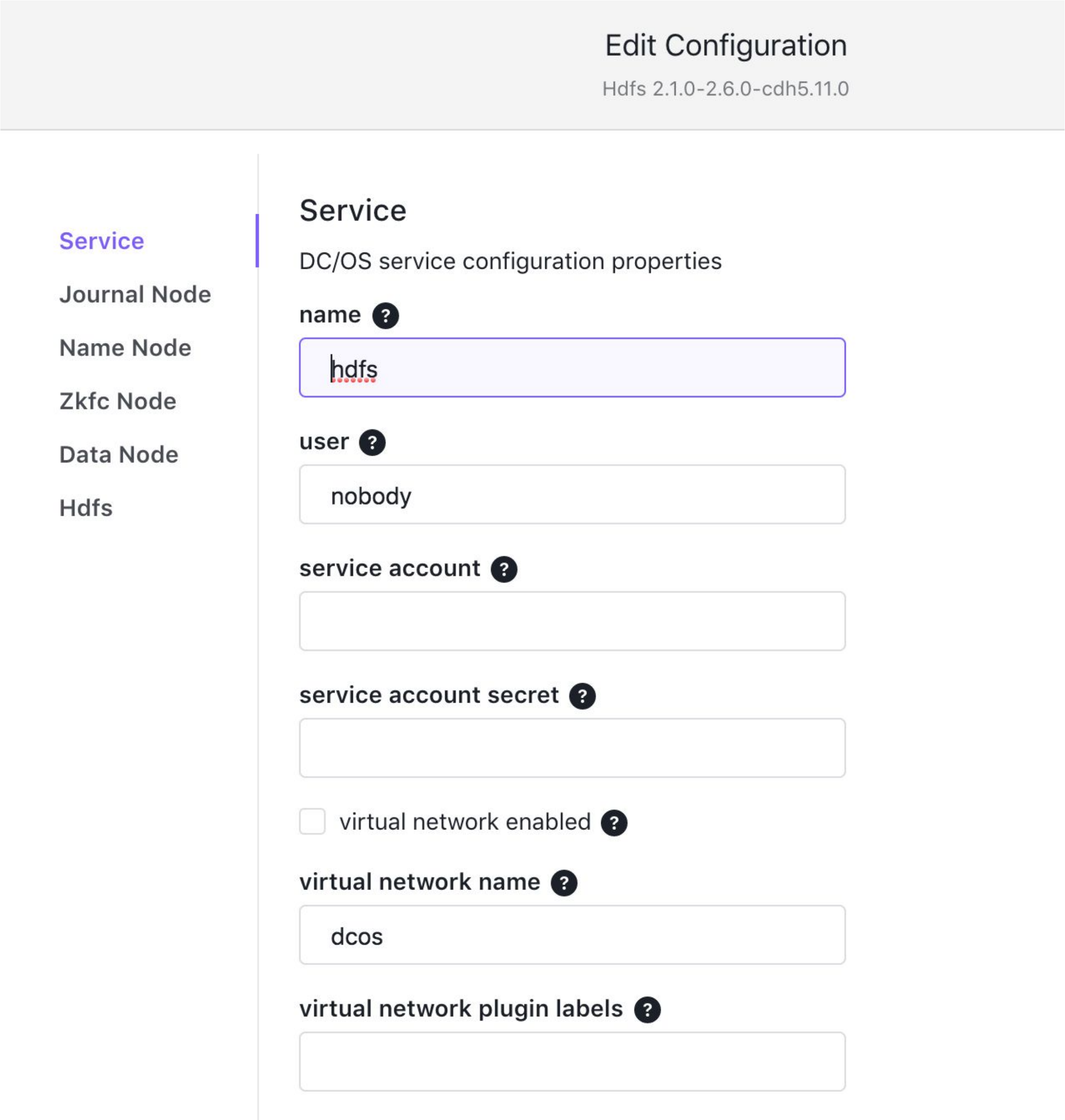
Apache Hadoop Distribut

Preinstall Notes: Defa 5 agent nodes each with: CPU: 0.6 | Memory: 4096MB | Disk: 5000MB More specific quires: Journal node: 3 instances | 0.3 CPU | 2048 MB MEM | 1 5000 MB Disk Name node: 2 instances | 0.3 CPU | 2048 MB MEM | 1 5000 MB Disk ZKFC node: 2 instances | 0.3 CPU | 2048 MB MEM Data node: 3 instances | 0.3 CPU | 2048 MB MEM | 1 5000 MB Disk.

Information

Maintainer: support@mesosphere.io

Step 2: Review default HDFS configuration and make any parameter changes necessary (i.e. storage, node count, CPU, memory, HDFS-specific config) → Review & Install



Edit Configuration

Hdfs 2.1.0-2.6.0-cdh5.11.0

Service

DC/OS service configuration properties

name ?

hdfs

user ?

nobody

service account ?

service account secret ?

□ virtual network enabled ?

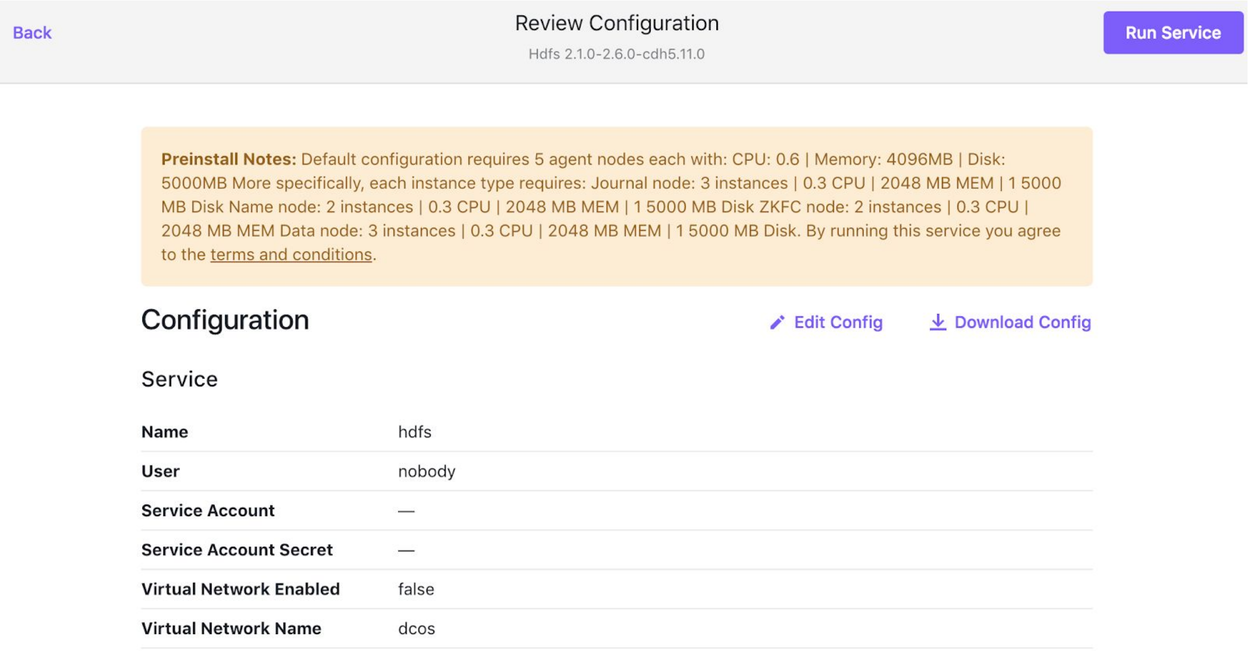
virtual network name ?

dcos

virtual network plugin labels ?

Step 3: Review Configuration and Run Service

- Note that you can also download any custom config for future re-use



Review Configuration

Hdfs 2.1.0-2.6.0-cdh5.11.0

Preinstall Notes: Default configuration requires 5 agent nodes each with: CPU: 0.6 | Memory: 4096MB | Disk: 5000MB More specifically, each instance type requires: Journal node: 3 instances | 0.3 CPU | 2048 MB MEM | 1 5000 MB Disk Name node: 2 instances | 0.3 CPU | 2048 MB MEM | 1 5000 MB Disk ZKFC node: 2 instances | 0.3 CPU | 2048 MB MEM Data node: 3 instances | 0.3 CPU | 2048 MB MEM | 1 5000 MB Disk. By running this service you agree to the terms and conditions.

Configuration

Service

Name	hdfs
User	nobody
Service Account	—
Service Account Secret	—
Virtual Network Enabled	false
Virtual Network Name	dcos

Step 4: View deployment in the GUI

CLI Method:

Step 1: Use DC/OS CLI to search for the HDFS package

```
dcos package search hdfs
```

Step 2: Install HDFS Package using DC/OS CLI

```
dcos package install hdfs --package-version=<package_version>

Note: It is possible to pass a custom configuration by using the
--options=<options.json> flag
```

Step 3: View deployment in the GUI

Test #2 Run a Spark HDFS Job

Access SMACK stack Github repo
[Github: SMACK Stack Tutorial]

- Full tutorial with step by step instructions are provided in PDF
- Deployment of HDFS + Spark + Kafka tutorial guides a reader through a simple example of running a Spark job that reads a file from the HDFS service and from a Kafka queue.

NOTE: This tutorial will require at least 10 private agent nodes (m4.xlarge) to complete

Test #3 Upgrading Certified Data Service

Prerequisites:

- Enterprise DC/OS 1.10 or newer
- A DC/OS SDK-based Service with a version greater than 2.0.0-x
- The DC/OS CLI installed and available
- The service's subcommand available and installed on your local machine
    - You can install just the subcommand CLI by running dcos package install --cli <service-name>.

Step 1: If you are running an older version of the subcommand CLI that doesn't have the update command, uninstall and reinstall your CLI.

```
dcos package uninstall --cli <service-name>
dcos package install --cli <service-name>
```

Step 2: View available Upgrade/Downgrade version options

```
dcos <service-name> update package-versions
```

Step 3: Update CLI subcommand to new version

```
dcos package uninstall --cli <service-name>

dcos package install --cli <service-name>
--package-version="<package-version>"
```

Step 4: Initiate upgrade

```
dcos <service-name> update start
--package-version="<package-version>"
```

NOTE: If you are missing mandatory configuration parameters, the update command will return an error.

Step 5: Monitor Upgrade status

```
dcos <service> --name=<service-name> update status
```

Test #4 Updating Data Service Configurations

Step 1: Fetch full configuration of a service

```
dcos <service-name> describe > options.json
```

Step 2: Make any configuration changes

- Scaling example: Increase Kafka default broker count from default 3 → 4

Step 3: Update Configuration

```
dcos <service-name> update start --options=options.json
```

Step 4: Monitor Update status

```
dcos <service-name> update status
```

See Advanced Update Actions for more useful update commands reference

Test #5 Multiple Versions of Data Services

Step 1: Install Cassandra 2.1.0-3.0.16

```
dcos package install cassandra --package-version=2.1.0-3.0.16
--options=cassandra-config-2.1.0.json --yes
```

- See **Appendix B** for cassandra-config-2.1.0.json
    - DC/OS defaults to ‘Cassandra’ as the deployment name so an <options.json> is required to distinguish multiple IDs

    - <options.json> also allows for separate Cassandra configuration needs

Step 2: Install Cassandra CLI

```
dcos package install cassandra --cli --yes
```

Step 3: Monitor deployment

```
dcos cassandra plan status deploy --name=cassandra-2.1.0
```

Step 4: Install Cassandra 2.0.3-3.0.14

```
dcos package install cassandra --package-version=2.0.3-3.0.14
--options=cassandra-config-2.0.3.json --yes
```

- See **Appendix C** for cassandra-config-2.1.0.json
    - DC/OS defaults to ‘Cassandra’ as the deployment name so an <options.json> is required to distinguish multiple IDs
    - <options.json> also allows for separate Cassandra configuration needs

Step 5: Monitor deployment

```
dcos cassandra plan status deploy --name=cassandra-2.0.3
```

Step 6: Display Services

```
dcos package list
```

Step 7: Uninstall Services

```
dcos package uninstall cassandra --app-id=cassandra-2.1.0
dcos package uninstall cassandra --app-id=cassandra-2.0.3
```

Test #6 Traditional Database Services Using Local Persistent Drives

Step 1: Review postgres.json Application Definition (Appendix D)

- Notice the volumes field, which declares the persistent volume for Postgres to use for its data. Even if the task dies and restarts, it will get that volume back and data will not be lost.

Step 2: Add Postgres service to the DC/OS cluster

```
dcos marathon app add postgres.json
```

Step #3: Stop service

```
dcos marathon app stop postgres
```

This command scales the instances count down to 0 and kills all running tasks. If you inspect the tasks list again, you will notice that the task is still there. The list provides information about which agent it was placed on and which persistent volume it had attached, but without a startedAt value. This allows you to restart the service with the same metadata.

Step #4: Restart the Service

```
dcos marathon app start postgres
```

The metadata of the previous postgres task is used to launch a new task that takes over the reservations and volumes of the previously stopped service. Inspect the running task again by repeating the command from the previous step. You will see that the running service task is using the same data as the previous one.

See DC/OS Storage: Local Persistent Volumes for a more extensive list of options regarding Local Persistent Volumes

DC/OS Security Test Plan:

Test #1 Role Based Access Control

Step 1: Make sure DC/OS Enterprise CLI is installed

```
dcos package install dcos-enterprise-cli --cli --yes
```

Step 2: Create group a and add users 1 & 2 using the DC/OS CLI

```
dcos security org groups create groupa
dcos security org users create -d User1 -p User1 User1
dcos security org users create -d User2 -p User2 User2
dcos security org groups add_user groupa User1
dcos security org groups add_user groupa User2
```

Step 3: Create group b and add users 3 & 4

```
dcos security org groups create groupb
dcos security org users create -d User3 -p User3 User3
dcos security org users create -d User4 -p User4 User4
dcos security org groups add_user groupb User3
dcos security org groups add_user groupb User4
```

Step 4: Create permission to access native Marathon instance using API method

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":""}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:service:marathon
```

Step 5: Give permission to native Marathon instance

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission to groups"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:service:marathon/grou
ps/groupa/full
```

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission to groups"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:service:marathon/grou
ps/groupb/full
```

Step 6: Create permission to the Mesos agent UI and API

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Create permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:ops:slave
```

Step 7: Give permission to Mesos agent UI and API

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:ops:slave/groups/grou
pa/full

curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:ops:slave/groups/grou
pb/full
```

Step 8: Create permission to launch DC/OS services

NOTE: groupa and groupb only have access to launch services in their respective team group folder (e.g. /groupa/postgres)

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Create permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:service:marathon:marathon:service
s:groupa

curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Create permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:service:marathon:marathon:service
s:groupb
```

Step 9: Give permission to launch DC/OS services

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:service:marathon:marathon:service
s:groupa/groups/groupa/full
```

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:service:marathon:marathon:service
s:groupb/groups/groupb/full
```

Step 10: Create permission to launch packages from the DC/OS Universe
Note: groupa and groupb only have access to launch services in their respective team group folder (e.g. /Grouop_A/postgres)

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Create permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:package
```

Step 11: Give permission to launch packages from the DC/OS Universe

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:package/groups/groupa
/full
```

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:package/groups/groupb
/full
```

Step 12: Create permission to the Mesos master UI and API

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
```

```
'{"description":"Create permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:ops:mesos
```

Step 13: Give permission to the Mesos master UI and API

```
curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:ops:mesos/groups/grou
pa/full

curl -X PUT -k -H "Authorization: token=$(dcos config show
core.dcos_acs_token)" -H "Content-Type: application/json" -d
'{"description":"Give permission"}' $(dcos config show
core.dcos_url)/acs/api/v1/acls/dcos:adminrouter:ops:mesos/groups/grou
pb/full
```

Walkthrough Workflow:

1. Show Superuser full view
2. Show locked-down user view
3. Login to groupa/groupb personas and test deploy nginx-example.json into root Marathon folder and watch it fail.
4. Retry the deployment into the group (i.e. /groupa/nginx-example.json) folder and watch it deploy successfully
5. Test deployment of catalog package into root folder and watch it fail
6. Retry the deployment into the group (i.e. /groupa/kafka) folder and watch it deploy successfully

Demo #2 LDAP Integration

- We will demo this integration using our own AD server to show functionality
- If time permits we can explore this further after initial few weeks of tackling tasks above

Appendix:

Appendix A: nginx.json Marathon application definition

```
{
  "id": "/nginx-example",
  "backoffFactor": 1.15,
  "backoffSeconds": 1,
  "container": {
    "portMappings": [
      {
        "containerPort": 80,
        "hostPort": 0,
        "labels": {
          "VIP_0": "/nginx-example:80"
        },
        "protocol": "tcp",
        "servicePort": 10101,
        "name": "nginx-example"
      }
    ],
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "nginx",
      "forcePullImage": false,
      "privileged": false,
      "parameters": []
    }
  },
  "cpus": 0.1,
  "disk": 0,
  "healthChecks": [
    {
      "gracePeriodSeconds": 5,
      "intervalSeconds": 10,
      "maxConsecutiveFailures": 2,
      "portIndex": 0,
      "timeoutSeconds": 10,
      "delaySeconds": 5,
      "protocol": "MESOS_TCP",
      "portName": "nginx-example"
    }
  ],
  "instances": 3,
  "maxLaunchDelaySeconds": 30,
  "mem": 128,
  "gpus": 0,
  "networks": [
    {
      "mode": "container/bridge"
    }
  ],
  "requirePorts": false,
  "upgradeStrategy": {
    "maximumOverCapacity": 1,
    "minimumHealthCapacity": 1
  },
  "killSelection": "YOUNGEST_FIRST",
  "unreachableStrategy": {
    "inactiveAfterSeconds": 1,
    "expungeAfterSeconds": 5
  },
  "fetch": [],
  "constraints": []
}
```

Appendix B: cassandra-config-2.1.0.json application definition

```
{
  "service": {
    "name": "cassandra-2.1.0",
    "user": "nobody",
    "service_account": "",
    "service_account_secret": "",
    "virtual_network_enabled": false,
    "virtual_network_name": "dcos",
    "virtual_network_plugin_labels": "",
    "mesos_api_version": "V1",
    "log_level": "INFO",
    "data_center": "datacenter1",
    "remote_seeds": "",
    "backup_restore_strategy": "serial",
    "security": {
      "transport_encryption": {
        "enabled": false,
        "allow_plaintext": false
      }
    }
  },
  "nodes": {
    "count": 3,
    "cpus": 0.5,
    "mem": 4096,
    "disk": 10240,
    "disk_type": "ROOT",
    "placement_constraint": "[[\"hostname\", \"MAX_PER\", \"1\"]]",
    "heap": {
      "size": 2048,
      "new": 100,
      "gc": "CMS"
    }
  },
  "cassandra": {
    "cluster_name": "cassandra",
    "authenticator": "AllowAllAuthenticator",
    "authorizer": "AllowAllAuthorizer",
    "jmx_port": 7199,
    "num_tokens": 256,
    "hinted_handoff_enabled": true,
    "max_hint_window_in_ms": 10800000,
    "hinted_handoff_throttle_in_kb": 1024,
    "max_hints_delivery_threads": 2,
    "batchlog_replay_throttle_in_kb": 1024,
    "partitioner": "org.apache.cassandra.dht.Murmur3Partitioner",
    "key_cache_save_period": 14400,
    "row_cache_size_in_mb": 0,
    "row_cache_save_period": 0,
    "commitlog_sync_period_in_ms": 10000,
    "commitlog_segment_size_in_mb": 32,
    "concurrent_reads": 16,
    "concurrent_writes": 32,
    "concurrent_counter_writes": 16,
    "memtable_allocation_type": "heap_buffers",
    "index_summary_resize_interval_in_minutes": 60,
    "storage_port": 7000,
    "ssl_storage_port": 7001,
    "start_native_transport": true,
    "native_transport_port": 9042,
    "start_rpc": false,
    "rpc_port": 9160,
    "rpc_keepalive": true,
    "thrift_framed_transport_size_in_mb": 15,
    "tombstone_warn_threshold": 1000,
    "tombstone_failure_threshold": 100000,
    "column_index_size_in_kb": 64,
    "batch_size_warn_threshold_in_kb": 5,
    "batch_size_fail_threshold_in_kb": 50,
    "compaction_throughput_mb_per_sec": 16,
    "sstable_preemptive_open_interval_in_mb": 50,
    "read_request_timeout_in_ms": 5000,
    "range_request_timeout_in_ms": 10000,
    "write_request_timeout_in_ms": 2000,
    "counter_write_request_timeout_in_ms": 5000,
    "cas_contention_timeout_in_ms": 1000,
    "truncate_request_timeout_in_ms": 60000,
    "request_timeout_in_ms": 10000,
    "dynamic_snitch_update_interval_in_ms": 100,
    "dynamic_snitch_reset_interval_in_ms": 600000,
    "dynamic_snitch_badness_threshold": 0.1,
    "internode_compression": "all",
    "max_hints_file_size_in_mb": 128,
    "hints_flush_period_in_ms": 10000,
    "concurrent_materialized_view_writes": 32,
    "commitlog_total_space_in_mb": 8192,
    "auto_snapshot": true,
    "roles_update_interval_in_ms": 1000,
    "permissions_update_interval_in_ms": 1000,
    "key_cache_keys_to_save": 100,
    "row_cache_keys_to_save": 100,
    "counter_cache_keys_to_save": 100,
    "file_cache_size_in_mb": 512,
    "memtable_heap_space_in_mb": 2048,
    "memtable_offheap_space_in_mb": 2048,
    "memtable_cleanup_threshold": 0.11,
    "memtable_flush_writers": 8,
    "listen_on_broadcast_address": false,
    "internode_authenticator":
"org.apache.cassandra.auth.AllowAllInternodeAuthenticator",
    "native_transport_max_threads": 128,
    "native_transport_max_frame_size_in_mb": 256,
    "native_transport_max_concurrent_connections": -1,
    "native_transport_max_concurrent_connections_per_ip": -1,
    "rpc_min_threads": 16,
    "rpc_max_threads": 2048,
    "rpc_send_buff_size_in_bytes": 16384,
    "rpc_recv_buff_size_in_bytes": 16384,
    "concurrent_compactors": 1,
    "stream_throughput_outbound_megabits_per_sec": 200,
    "inter_dc_stream_throughput_outbound_megabits_per_sec": 200,
    "streaming_socket_timeout_in_ms": 86400000,
    "phi_convict_threshold": 8,
    "buffer_pool_use_heap_if_exhausted": true,
    "disk_optimization_strategy": "ssd",
    "max_value_size_in_mb": 256,
    "otc_coalescing_strategy": "TIMEHORIZON"
  }
}
```

Appendix C: cassandra-config-2.0.3.json application definition

```
{
  "service": {
    "name": "cassandra-2.0.3",
    "user": "nobody",
    "service_account": "",
    "service_account_secret": "",
    "virtual_network_enabled": false,
    "virtual_network_name": "dcos",
    "virtual_network_plugin_labels": "",
    "mesos_api_version": "V1",
    "log_level": "INFO",
    "data_center": "datacenter1",
    "remote_seeds": "",
    "backup_restore_strategy": "serial",
    "security": {
      "transport_encryption": {
        "enabled": false,
        "allow_plaintext": false
      }
    }
  },
  "nodes": {
    "count": 3,
    "cpus": 0.5,
    "mem": 4096,
    "disk": 10240,
    "disk_type": "ROOT",
    "placement_constraint": "[[\"hostname\", \"MAX_PER\", \"1\"]]",
    "heap": {
      "size": 2048,
      "new": 100,
      "gc": "CMS"
    }
  },
  "cassandra": {
    "cluster_name": "cassandra",
    "authenticator": "AllowAllAuthenticator",
    "authorizer": "AllowAllAuthorizer",
    "jmx_port": 7199,
    "num_tokens": 256,
    "hinted_handoff_enabled": true,
    "max_hint_window_in_ms": 10800000,
    "hinted_handoff_throttle_in_kb": 1024,
    "max_hints_delivery_threads": 2,
    "batchlog_replay_throttle_in_kb": 1024,
    "partitioner": "org.apache.cassandra.dht.Murmur3Partitioner",
    "key_cache_save_period": 14400,
    "row_cache_size_in_mb": 0,
    "row_cache_save_period": 0,
    "commitlog_sync_period_in_ms": 10000,
    "commitlog_segment_size_in_mb": 32,
    "concurrent_reads": 16,
    "concurrent_writes": 32,
    "concurrent_counter_writes": 16,
    "memtable_allocation_type": "heap_buffers",
    "index_summary_resize_interval_in_minutes": 60,
    "storage_port": 7000,
    "ssl_storage_port": 7001,
    "start_native_transport": true,
    "native_transport_port": 9042,
    "start_rpc": false,
    "rpc_port": 9160,
    "rpc_keepalive": true,
    "thrift_framed_transport_size_in_mb": 15,
    "tombstone_warn_threshold": 1000,
    "tombstone_failure_threshold": 100000,
    "column_index_size_in_kb": 64,
    "batch_size_warn_threshold_in_kb": 5,
    "batch_size_fail_threshold_in_kb": 50,
    "compaction_throughput_mb_per_sec": 16,
    "sstable_preemptive_open_interval_in_mb": 50,
    "read_request_timeout_in_ms": 5000,
    "range_request_timeout_in_ms": 10000,
    "write_request_timeout_in_ms": 2000,
    "counter_write_request_timeout_in_ms": 5000,
    "cas_contention_timeout_in_ms": 1000,
    "truncate_request_timeout_in_ms": 60000,
    "request_timeout_in_ms": 10000,
    "dynamic_snitch_update_interval_in_ms": 100,
    "dynamic_snitch_reset_interval_in_ms": 600000,
    "dynamic_snitch_badness_threshold": 0.1,
    "internode_compression": "all",
    "max_hints_file_size_in_mb": 128,
    "hints_flush_period_in_ms": 10000,
    "concurrent_materialized_view_writes": 32,
    "commitlog_total_space_in_mb": 8192,
    "auto_snapshot": true,
    "roles_update_interval_in_ms": 1000,
    "permissions_update_interval_in_ms": 1000,
    "key_cache_keys_to_save": 100,
    "row_cache_keys_to_save": 100,
    "counter_cache_keys_to_save": 100,
    "file_cache_size_in_mb": 512,
    "memtable_heap_space_in_mb": 2048,
    "memtable_offheap_space_in_mb": 2048,
    "memtable_cleanup_threshold": 0.11,
    "memtable_flush_writers": 8,
    "listen_on_broadcast_address": false,
    "internode_authenticator":
"org.apache.cassandra.auth.AllowAllInternodeAuthenticator",
    "native_transport_max_threads": 128,
    "native_transport_max_frame_size_in_mb": 256,
    "native_transport_max_concurrent_connections": -1,
    "native_transport_max_concurrent_connections_per_ip": -1,
    "rpc_min_threads": 16,
    "rpc_max_threads": 2048,
    "rpc_send_buff_size_in_bytes": 16384,
    "rpc_recv_buff_size_in_bytes": 16384,
    "concurrent_compactors": 1,
    "stream_throughput_outbound_megabits_per_sec": 200,
    "inter_dc_stream_throughput_outbound_megabits_per_sec": 200,
    "streaming_socket_timeout_in_ms": 86400000,
    "phi_convict_threshold": 8,
    "buffer_pool_use_heap_if_exhausted": true,
    "disk_optimization_strategy": "ssd",
    "max_value_size_in_mb": 256,
    "otc_coalescing_strategy": "TIMEHORIZON"
  }
}
```

Appendix D: postgres.json application definition

```
{
  "id": "/postgres",
  "cpus": 1,
  "mem": 1024,
  "instances": 1,
  "networks": [
    { "mode": "container/bridge" }
  ],
  "container": {
    "type": "DOCKER",
    "volumes": [
      {
        "containerPath": "pgdata",
        "mode": "RW",
        "persistent": {
          "size": 100
        }
      }
    ],
    "docker": {
      "image": "postgres:9.5"
    },
    "portMappings": [
      {
        "containerPort": 5432,
        "hostPort": 0,
        "protocol": "tcp",
        "labels": {
          "VIP_0": "5.4.3.2:5432"
        }
      }
    ]
  },
  "env": {
    "POSTGRES_PASSWORD": "DC/OS_ROCKS",
    "PGDATA": "/mnt/mesos/sandbox/pgdata"
  },
  "healthChecks": [
    {
      "protocol": "TCP",
      "portIndex": 0,
      "gracePeriodSeconds": 300,
      "intervalSeconds": 60,
      "timeoutSeconds": 20,
      "maxConsecutiveFailures": 3,
      "ignoreHttp1xx": false
    }
  ],
  "upgradeStrategy": {
    "maximumOverCapacity": 0,
    "minimumHealthCapacity": 0
  }
}
```