Microsoft Cloud Workshop

Containers and DevOps Hackathon

Learner Guide

November 2017

Containers and DevOps hackathon

Overview

Fabrikam Medical Conferences provides conference web site services tailored to the medical community. They are refactoring their application code, which is based on node.js, so that it can run as a Docker application and want to implement a POC that will help them get familiar with the development process, lifecycle of deployment and critical aspects of the hosting environment. They will be deploying their applications to Azure Container Service based on Apache Mesos and want to be educated on how to deploy containers in a dynamically load-balanced manner, discover containers, and scale them on demand.

In this Hackathon, you will assist with completing this POC with a subset of the application code base. You will create a build agent based on Linux, and an Azure Container Service cluster for running deployed applications. You will be helping them to complete the Docker setup for their application, test locally, push to an image repository, deploy to the cluster, and test load-balancing and scale.

IMPORTANT: Most Azure resources require unique names. Throughout these steps you will see the word "SUFFIX" as part of resource names. You should replace this with your Microsoft email prefix to ensure the resource is uniquely named.

Requirements

- Microsoft Azure subscription must be pay-as-you-go or MSDN.
  - Trial subscriptions will *not* work.
- Local machine or a virtual machine configured with:
  - A browser, preferably Chrome for consistency with the hackathon implementation tests
  - Command prompt
- You will be asked to install other tools throughout the exercises.

VERY IMPORTANT: If any of your Git Bash commands do not execute as expected, it could be the result of copy and paste issues between this document and the command line. Please try typing the command if an issue occurs with copy and paste.

Exercise 0: Before the Hackathon

Duration: 30 minutes (possibly additional time if Azure provisioning is slower)

You should follow all of the steps provided in Exercise 0 *before* attending the hackathon.

Task 1: Resource Group

You will create an Azure Resource Group to hold most of the resources that you create in this hackathon. This approach will make it easier to clean up later. You will be instructed to create new resources in this Resource Group during the remaining exercises.

Tasks to complete

- Create a new Azure Resource Group. Name it something like "fabmedical - SUFFIX". Choose a region you will also use for future steps so that resources you create in Azure are all kept within the same region.

Exit criteria

- Your Resource Group is listed in the Azure Portal.

Task 2: Install Git tools

In this section you will install Git tools. This documentation assumes you are installing Git tools for Windows but you may use another platform to achieve the same results at your own discretion.

Tasks to complete

- Download Git tools for Windows from this URL: https://git-scm.com/download.
- During the installation choose the following settings:
  - Use Git from the Windows command prompt
  - Checkout Windows-style
  - Use MinTTY

Exit criteria

- At the end of these steps you should see the Git Bash command line window.

Task 3: Create an SSH key

In this section you will create an SSH key to securely access the VMs you create during the upcoming exercises.

Tasks to complete

- Open Git Bash and generate a new SSH key using the following command:

```
ssh-keygen –t RSA –b 2048 –C admin@fabmedical
```

Exit criteria

- You will see two files generated, a public and private key

Task 4: Create a build agent VM

In this section you will create a Linux VM to act as your build agent. You will be installing Docker to this VM once it is set up and you will use this VM during the hackathon to develop and deploy.

NOTE: You can set up your local machine with Docker however the setup varies for different versions of Windows. For the purpose of this hackathon the build agent approach simply allows for predictable setup.

Tasks to complete

- Create a new VM based on Ubuntu Server 16.04 LTS
- Use the following settings for the VM:
    - Provide a unique name such as "fabmedical-SUFFIX"
    - Leave the VM disk type as SDD.
    - Provide a VM username such as "adminfabmedical".
    - Leave the Authentication type as SSH.
    - Copy the public key portion of your SSH key pair, fabmedical.pub
    - Choose the same subscription you are using for all your work.
    - Choose the same resource group as you created previously.
    - Choose the same region that you did before.
    - Choose DS1_V2 Standard.

Exit criteria

- When the VM is provisioned you will see it in your list of resources belonging to the resource group you created previously.

Task 5: Connect securely to the build agent

In this section you will validate that you can connect to the new build agent VM.

Tasks to complete

- From the Azure Portal, navigate to the new VM you created in the previous task and find the IP address.
- Launch Git Bash on your local machine, and from the directory where the private key was created run the following command to connect to the VM:

```
ssh -i [PRIVATEKEYNAME] [BUILDAGENTUSERNAME]@[BUILDAGENTIP]
```

Exit criteria

- You will now be connected to the VM with a command prompt.

Task 6: Complete the build agent setup

In this task you will update the packages and install Docker engine.

Tasks to complete

- Using the connection to the build agent VM, run this command to update packages, and install Docker engine:

```
sudo apt-get update && sudo apt install docker.io curl nodejs npm
```

- Now, upgrade the Ubuntu packages to the latest version by typing the following in a single line command.

```
sudo apt-get upgrade
```

- Add your user to the docker group so that you do not have to elevate privileges with sudo for every command. You can ignore any errors you see in the output. Restart your SSH session after this step for the change to take effect.

```
sudo usermod –aG docker $USER
```

Exit criteria

- Run a few Docker commands to ensure things are running properly, you will not see any errors.

```
docker ps
docker ps -a
```

Task 7: Create an Azure Container Registry

Docker images are deployed from a registry. To complete the hackathon, you will need access to a registry that is accessible to the Azure Container Service cluster you are creating. In this task, you will create an Azure Container Registry (ACR) for this purpose, where you push images for deployment.

Tasks to complete

- In the Azure Portal, create an Azure Container Registry (ACR).

Exit criteria

- Navigate to your ACR account in the Azure Portal. As this is a new account, you will not see any repositories yet. You will create these during the hackathon.

Task 8: Create a Storage Account

In this section, you will create an Azure Storage Account for storing your ACR account credentials, for use later by DC/OS.

Tasks to complete

- In the Azure Portal, create a blob storage account.

Exit criteria

- You have a storage account with a blob container defined.

Task 9: Create an Azure Container Service cluster

In this task you will create your Azure Container Service cluster based on Apache Mesos. You will use the same SSH key you created previously to connect to this cluster in the next task.

NOTE: In this step, you will be asked to create a second Resource Group as part of the template for creating the Azure Container Service cluster. This is due to the current template not providing the option to select an existing Resource Group, and this may change in the future.

Tasks to complete

- From the Azure Portal, create a new Azure Container Service with the following settings:
  - Enter a username such as “adminfabmedical”.
  - Paste the same SSH public key you used for the agent VM previously.
  - Choose the same subscription and location as you are using for other resources.
  - Since the template does not support using an existing Resource Group, provide a new name such adding a “2” to the suffix, such as fabmedical-SUFFIX2.
  - Choose DC/OS for the Orchestrator configuration
  - Set the Agent count to 2.
  - Choose Standard DS1 for the Agent virtual machine size.
  - Set the Master count to 1.
  - Set the DNS prefix to something like “fabmedical-SUFFIX”.

Exit criteria

- You should see a successful deployment notification when the cluster is ready. It can take up to 10 minutes or more before your Azure Container Service cluster is listed in the Azure Portal.

  Note: If you experience errors related to lack of available cores, you may have to delete some other compute resources or request additional cores be added to your subscription and then try this again.

Exercise 1: Environment setup

Duration: 10 minutes

FabMedical has provided starter files for you. They have taken a copy of one of their web sites, for their customer Contoso Neuro, and refactored it from a single node.js site into a web site with a content API that serves up the speakers and sessions. This is a starting point to validate the containerization of their web sites. They have asked you to use this as the starting point for helping them complete a POC that helps them to validate the development workflow for running the web site and API as Docker containers, and managing them within the Azure Container Service DC/OS environment.

Task 1: Download the FabMedical starter files

- From Git Bash, connect to the build agent VM
- Download and unzip the starter file with the following commands:

```
curl -L -o FabMedical.tgz http://bit.ly/2k5aZdg

mkdir FabMedical

tar –C FabMedical -xzf FabMedical.tgz
```

NOTE: Keep this Git Bash window open as your build agent SSH connection. You will later open new Git Bash sessions to other machines.

Exit Criteria

- Navigate to FabMedical folder and list the contents. You'll see the listing includes two folders, one for the web site and another for the content API.

  /content-api

  /content-web

Exercise 2: Create and run a Docker application

Duration: 40 minutes

In this exercise, you will take the starter files and run the node.js application as a Docker application. You will create a Dockerfile, build Docker images and run containers to execute the application.

NOTE: These tasks will be completed from the Git Bash window with the build agent session.

Task 1: Test the application

The purpose of this task is to make sure you are able to run the application successfully before applying changes to run it as a Docker application.

Tasks to complete

- Run the api and web applications in that order, by navigating to their respective folders and running these commands:

```
npm install

nodejs ./server.js &
```

Exit criteria

- Test the results by curling these URLs and observing the JSON responses:

  API Application:

```
curl http://localhost:3001/speakers
```

  Web Application:

```
curl http://localhost:3000
```

Task 2: Browse to the application

In this task, you will open a port range on the agent VM so that you can browse to the application for testing.

Tasks to complete

- From the Azure portal select the Network Security Group associated with the build agent VM. Create an inbound security rule to allow TCP calls over port range 3000-3010.
- Find the IP address for the VM and browse to that IP address with port 3000.

- Once you have seen the web site running, you can stop the running node processes with this command:

```
killall nodejs
```

Exit criteria

- You can browse to the web site and see the speakers and content pages, you are able to reach the web site from a browser and have correctly opened the port on the agent VM.

Task 3: Create a Dockerfile

In this task, you will create a new Dockerfile that will be used to run the API as a containerized application.

Tasks to complete

- From the content-api folder, using VIM create a new file named Dockerfile.
- Create a Dockerfile that is based on the node:argon image.
- Include instructions to copy the source files for the application to the image and to run npm install.
- Expose port 3001 and put the startup command npm

Exit criteria

- Verify the file contents, to ensure it saved as expected.

Task 4: Create Docker images

In this task, you will create Docker images for the application – one for the API and another for the web application. Each image will be created via Docker commands that rely on a Dockerfile.

Tasks to complete

- From the content-api directory, build a Docker image using the Dockerfile you created, and tag it for your ACR account as [ACRLOGINSERVER]/fabmedical/content-api.
- From the content-web directory, build a Docker image with the existing Dockerfile in the folder, and tag it for your ACR account as [ACRLOGINSERVER]/fabmedical/content-web.

Exit criteria

- Once the image is successfully built, you will see three images now exist when you run the Docker images command.

Task 5: Run a containerized application

The web application container will be calling endpoints exposed by the API application container. In this exercise, you will create a bridge network so that containers can communicate with one another, and then launch the images you created as containers in that network.

Tasks to complete

- On the build agent VM create a new Docker network named “fabmedical”.
- Start the api and web containers with these commands:

```
docker run –d –p 3001:3001 --name api --net fabmedical content-api

docker run –d –P --name web --net fabmedical content-web
```

Exit criteria

- Test the api application by curling the speakers URL once again.
- Test the web application by curling the site but using the dynamic port assigned to the container from the run command.

Task 6: Setup environment variables

In this task, you will configure the web container to communicate with the API container using an environment variable. You will modify the web application to read the URL from the environment variable, rebuild the Docker image, and then run the container again to test connectivity.

Tasks to complete

- From Git Bash, stop and remove the web container.
- Edit content-web\data-access\index.js and locate the TODO item and modify the code so that the contentApiUrl variable will be set to an environment variable.
- Edit the Dockerfile in content-web to add an environment variable CONTENT_API_URL matching http://localhost:3001.
- Rebuild the web Docker image.

- Create and start the image passing the correct URI to the api container as an environment variable, then check the port that the container is running on.

```
docker run –d –P --name web --net fabmedical –e
CONTENT_API_URL=http://api:3001 content-web
```

- Curl the speakers path again, using the port assigned to the web container.

Exit criteria

- Stop and restart the container so that it uses port 3000 and then browse to the URL at the build agent IP address with that port to view the web site.

Task 7: Push images to Azure Container Registry

To run containers in a remote environment you will typically push images to a Docker registry, where you can store and distribute images. Each service will have a repository that can be pushed to and pulled from with Docker commands. Azure Container Registry (ACR) is a managed Docker registry service based on the open-source Docker Registry 2.0.

In this task you will push images to your ACR account, version images with tagging, and run containers from an image in your ACR.

Tasks to complete

- Navigate to your ACR account in the Azure Portal.
- Push the api and web images to the ACR.
- Assign a new tag "v1" to both images and push the newly tagged version of the images to the ACR.

Exit criteria

- Verify the tagged images are created in the ACR.
- Perform a pull on each image to validate pulls are working to your build agent VM.

Exercise 3: Deploy the solution to Azure Container Service

Duration: 30 minutes

In this exercise, you will connect to the Azure Container Service cluster you created in Exercise 0 and deploy the Docker application to the cluster using DC/OS and Marathon.

Task 1: Create package for your ACR credentials

In this task, you will package up your ACR credentials, so they can be used by DC/OS to authenticate against your ACR repositories.

Tasks to complete

- Package your ACR login credentials in a gzip package, by executing the following command in a new Git Bash window.

```
cd ~
tar czf docker.tar.gz .docker
```

- Copy the gzipped file to your local user home folder using the following SCP command.

```
scp -i [PRIVATEKEY] [BUILDAGENTUSERNAME]@[BUILDAGENTIP]:docker.tar.gz
C:/[LOCALPATH]/
```

Exit criteria

- A copy of docker.tar.gz is present on your local drive, in the location specified above.

Task 2: Upload the credential package to Azure Storage

In this task, you will upload the credential package in the Azure Storage account created in Exercise 0, Task 8, for use by DC/OS.

Tasks to complete

- Upload the gzipped file from the previous task into the storage account you created in Exercise 0, Task 8.

Exit criteria

- The docker.tar.gz file exists in the blob storage container.

Task 3: Tunnel in to the Azure Container Service cluster

In this task, you will gather the information you need about your Azure Container Service cluster in order to connect to the cluster, and execute commands to connect to the DC/OS management UI from your local machine.

Tasks to complete

- From the Azure Portal, find the MASTERFQDN and LINUXADMINUSERNAME for the Azure Container Service deployment.
- Open Git Bash and created an SSH tunnel linking a local port on your machine to port 80 on the management node of the cluster. Execute the command below replacing the values accordingly:

```
ssh -L [LOCALPORT]:localhost:80 -N -i [PRIVATEKEY]
[LINUXADMINUSERNAME]@[MASTERFQDN] -p 2200
```

  NOTE: The command will not show any output and will not return a prompt until the tunnel is terminated. This is normal.

- Open a browser and navigate to the port you used for the tunnel. Leave this running for the tasks to follow.

  http://localhost:[LOCALPORT]

Exit criteria

- If the tunnel is successful, you will see the DC/OS management dashboard.

Task 4: Deploy the Api application using DC/OS

In this task, you will deploy the API application to the Azure Container Service cluster using DC/OS UI.

Tasks to complete

- From the DC/OS dashboard, click Services on the left navigation bar.
  - Set the ID to “api”
  - Use “Bridged” networking
  - Use [ACRLOGINSERVER]/fabmedical/content-api for the image
  - Use “/speakers” as the health check endpoint
  - Use the JSON Editor to add the following fetch node for retrieving your ACR credentials from the gzipped package placed in Azure Storage.

```
"fetch": [
  {
      "uri":
"https://[STORAGEACCOUNTNAME].blob.core.windows.net/[CONTAINERNAME]/
docker.tar.gz",
      "extract": true,
      "executable": false,
      "cache": false,
      "outputFile": "dockerConfig.tar.gz"
    }
],
```

- From JSON Mode, create a portMappings section with the following settings:

```
"portMappings": [
{
  "containerPort": 3001,
  "hostPort": 3001,
  "protocol": "tcp",
  "name": null,
  "labels": null
}
```

Exit criteria

- After the deployment is successful it will transition to a Running state.

Task 5: Deploy the Web application using Marathon REST API

In this task, you will make HTTP requests through the SSH tunnel you have open and connected to the cluster. These requests will deploy the Web application using the REST Api.

Tasks to complete

- Open a text editor, such as Notepad, and copy the following text into that editor. Save the file as marathon-web.json, and keep it open to edit entries. For now, update the following:
  - [STORAGEACCOUNTNAME]: Set to the storage account name you created in Exercise 0, Task 8.
  - [CONTAINERNAME]: Set to the name of the blob storage container you created in the storage account above.
  - [ACRLOGINSERVER]: entry to match the name of your ACR account. You can retrieve this value by selecting Access Keys after navigating the your Container Registry in the Azure Portal.

```
{
  "id": "/web",
  "cmd": null,
  "cpus": 1,
  "mem": 128,
  "disk": 0,
  "instances": 1,
  "fetch": [
    {
      "uri": "
https://[STORAGEACCOUNTNAME].blob.core.windows.net/[CONTAINERNAME]/d
ocker.tar.gz",
      "extract": true,
      "outputFile": "dockerConfig.tar.gz"
    }
  ],
  "acceptedResourceRoles": [
    "slave_public"
  ],
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "[ACRLOGINSERVER]/fabmedical/content-web",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80,
          "protocol": "tcp",
          "labels": {}
        }
      ],
      "privileged": false,
      "parameters": [],
      "forcePullImage": true
    }
  },
  "env": {
    "CONTENT_API_URL": "http://api.marathon.mesos:3001"
  },
  "healthChecks": [
    {
      "path": "/",
      "protocol": "HTTP",
      "portIndex": 0,
      "gracePeriodSeconds": 300,
      "intervalSeconds": 60,
      "timeoutSeconds": 20,
      "maxConsecutiveFailures": 3,
      "ignoreHttp1xx": false
    }
  ],
  "portDefinitions": [
    {
      "port": 10001,
      "protocol": "tcp",
      "labels": {}
    }
  ]
}
```

- Open a new Git Bash window and deploy the JSON configuration using this command:

```
curl -X POST -d @marathon-web.json -H "Content-type: application/json"
http://localhost:[LOCALPORT]/marathon/v2/apps
```

- Return to Services in the browser and observe the health of the api and web services as the deployment is executed. The web service will not deploy immediately.

Exit criteria

- On the Services page, you will see two services deployed, one healthy, one Deploying or Unhealthy depending on how long you wait.

Task 6: Explore service instance logs and resolve an issue

In this task, you will determine why the web application deployment is unhealthy and learn how to explore logs in DC/OS and Marathon to troubleshoot the issue. You will then fix the problem and redeploy the application.

Tasks to complete

- From the DC/OS dashboard, navigate to services and observe the tasks. Navigate to Log the api and web application and view the logs from the Logs tab. You may see some killed tasks if the web application task was restarted by this time due to a failed deployment.

- From Marathon UI navigate to the web application Configuration, and edit it to change the containerPort to 3000. Deploy the update.

Exit criteria

- After a few minutes, the state of the applications list will show the web application in a Deploying state. A few minutes later it will show as a healthy Running application.

Task 7: Test the application in a browser

In this task, you will verify that you can browse to the web application you have deployed, and view the speaker and content information exposed by the API service.

Tasks to complete

- From the Azure portal, find the AGENTFQDN value for the Azure Container Service deployment. Navigate to this URL.

Exit criteria

- You will see the application in your browser and be able to click on the Speakers and Content links to view those pages without errors.

Exercise 4: Scale the application and test HA

Duration: 20 minutes

At this point you have deployed a single instance of the web and api service containers. In this exercise, you will increase the number of container instances for the web service and scale the front end on the existing cluster.

Task 1: Increase api instances from the Marathon UI

In this task, you will increase the number of instances for the api service container in the DC/OS UI. While it is deploying, you will observe the changing status.

Tasks to complete

- Scale the api service to 2 instances. Observe the status of the deployment, active tasks and health of the service as the scale action is completed.
- Note that when the deployment is completed, two healthy tasks are running.

Exit criteria

- Navigate to the web service from the browser again. The site should still work without errors as you navigate to Speakers and Sessions pages.
- Navigate to the /stats.html page. You'll see information about the environment.

Task 2: Increase api instances beyond available resources

In this task, you will try to increase the number of instances for the api service container beyond available resources in the cluster. You will observe how Marathon handles this condition and correct the problem.

Tasks to complete

- Scale the api service to 3 instances. Observe the status of the deployment, and that it shows as Waiting and Unscheduled.
- Rollback the deployment.

Exit criteria

- Almost immediately you'll be able to see the deployment cancelled and if you navigate to the api service you'll see 2 healthy running instances again.

Task 3: Restart containers and test HA

In this task, you will restart containers and validate that the restart does not impact the running service. First you will restart the api service, followed by the web service. You will also note that there are issues with HA restarts when there is only a single instance.

Tasks to complete

- Open a browser and navigate to the web application /stats.html page. Refresh this page repeatedly as you initiate a restart of the api service. Note the changes to the api task identifier in the UI as the page reaches one of the existing instances.
- Scale the instances of the api service to 1. Continue to refresh that stats page to observe the task identifier, which is eventually only 1 instance.
- Restart the service again and observe that a new instance identifier is shown.

Exit criteria

- Return to Marathon and view the api service tasks, noting that there is only 1 task now.

Exercise 5: Setup load balancing and service discovery

Duration: 45 minutes

Thus far there have been some restrictions to the scale properties of the service. In this exercise you will add marathon-lb, a load balancer that integrates directly with Marathon.

Marathon is able to discover the ports assigned to each task allowing you to run multiple instances of a task on the same agent node – something that is not possible when a specific, static port (such as 3001 for the api) is configured.

Task 1: Scale the Azure Container Service cluster

In order to successfully work with Marathon Load Balancer (marathon-lb) you will need at least 3 agent nodes in the Azure Container Service cluster. In this exercise, you will scale your cluster from the Azure Portal to accommodate this.

Tasks to complete

- From the Azure Portal redeploy the Azure Container Service template adjusting the agent nodes (AGENTCOUNT) so they are increased from 2 to 3. Be sure to select the same Resource Group as you originally used when creating the cluster.

  NOTE: This selection is very important to ensure you redeploy to the same cluster to update the number of agent nodes.

Exit criteria

- Navigate to the DC/OS UI and on the dashboard you should see 4 healthy, connected nodes.
- Note that the services previously deployed to the cluster are still running.

Task 2: Install a public load balancer for the Web application

In this task, you will install the default Marathon Load Balancer (marathon-lb) to the Azure Container Service cluster to support dynamic discovery of web service instances.

Tasks to complete

- Destroy the api service.

- Install the marathon-lb package using the default installation.
- Observe that this deployment will be held in a waiting state. Edit the configuration for the service and change CPUs to 0.2 from 2. Redeploy.

Exit criteria

- The service update will deploy and show as healthy and running in the service list when the deployment is complete.

Task 3: Redeploy the web application

In this task, you will redeploy the web service with dynamically assigned ports and configure it for use with marathon-lb.

Tasks to complete

- From the Azure Portal, find the IP address for the public agent node of the Azure Container Service cluster.
- From Marathon UI create a new service and past the following JSON into the editor. Replace the following with your own values:
    - [STORAGEACCOUNTNAME]: Set to the storage account name you created in Exercise 0, Task 8.
    - [CONTAINERNAME]: Set to the name of the blob storage container you created in the storage account above.
    - [ACRLOGINSERVER]: entry to match the name of your ACR account. You can retrieve this value by selecting Access Keys after navigating the your Container Registry in the Azure Portal.
    - [PUBLICAGENTIP]: replace with the address from step 2.

```
{
  "id": "/web",
  "cmd": null,
  "cpus": 0.2,
  "mem": 128,
  "disk": 0,
  "instances": 1,
  "fetch": [
    {
      "uri":
"https://[STORAGEACCOUNTNAME].blob.core.windows.net/[CONTAINERNAME]/
docker.tar.gz",
      "extract": true,
      "outputFile": "dockerConfig.tar.gz"
    }
  ],
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "[ACRLOGINSERVER]/fabmedical/content-web",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 3000,
          "hostPort": 0,
          "servicePort" : 10001,
          "protocol": "tcp",
          "labels": {}
        }
      ],
      "privileged": false,
      "parameters": [],
      "forcePullImage": true
    }
  },
  "env": {
    "CONTENT_API_URL": "http://api.marathon.mesos:3001"
  },
  "healthChecks": [
    {
      "path": "/",
      "protocol": "HTTP",
      "portIndex": 0,
      "gracePeriodSeconds": 300,
      "intervalSeconds": 60,
      "timeoutSeconds": 20,
      "maxConsecutiveFailures": 3,
      "ignoreHttp1xx": false
    }
  ],
  "labels" : {
    "HAPROXY_GROUP" : "external",
    "HAPROXY_0_VHOST" : "[PUBLICAGENTIP]"

  },
    "upgradeStrategy": {
    "maximumOverCapacity": 1,
    "minimumHealthCapacity": 0
  }
}
```

- Navigate to the web site in the browser at the public IP and note that it still works. Scale the web service to 4 instances. Note additional tasks are created at random ports.

Exit criteria

- Browse to the web application at the public agent IP address and navigate to the /stats.html page. Refresh it many time and take note of the webTaskId value. It will cycle between the 4 tasks listed for the web service.

Task 4: Install an internal load balancer for the Api application

In this task, you will install an internal marathon-lb to the Azure Container Service cluster to support dynamic discovery of api instances.

Tasks to complete

- Suspend the api application.
- Install another instance of the marathon-lb package, this time with Advance Installation using the following changes to the settings:
    - Deselect bind-http-https.
    - Set cpus to 1 instead of 2.
    - Set haproxy-group to internal instead of external.
    - Deselect haproxy-map.
    - Enter 2 for instances.

- Set the name to mlb-internal instead of marathon-lb to avoid conflicts with the previous marathon-lb installation.
- Remove the role setting and leave it blank, instead of slave_public.

NOTE: Despite clearing this setting it appears to be defaulting to slave_public so you will have to fix that after you install.

Exit criteria

- You will see a dialog indicating successful installation. After this you can navigate to Services and you'll see the marathon-lb service named mlb-internal is in a Waiting state.

Task 5: Validate the internal marathon-lb deployment

In this task, you will check the status of the marathon-lb deployment from the previous task. You will find that it is not successfully deploying due to a configuration error. You will fix this error and redeploy the load balancer.

Tasks to complete

- Go to Marathon UI and note that the internal load balancer hasn't deployed – it is in a waiting state.
  NOTE: If you see that the service has deployed and is listed, it could be that the bug that caused this problem is fixed. You will not see the deployment in progress in this case but you can instead navigate to the service view and look at the configuration to double check the settings mentioned in the upcoming steps are correct.

- Edit the configuration and clear the Resource Roles setting so that it is set to *, not slave_public. Also verify that the CPU setting is set to 0.2 instead of 1. Deploy this change and note the 2 tasks are deployed.

Exit criteria

- When this deployment is complete you'll be able to refresh the Services list and see mlb-internal is now listed there.

Task 6: Redeploy the Api application for load balancing

In this task, you will edit the suspended api service so that it will be called through the internal load balancer. As a result, you will be able to scale beyond the number of agent nodes.

Tasks to complete

- From Marathon UI, edit the api service in JSON Mode.
  - Set cpus to 0.2 from 1
  - Set hostPort to 0
  - Add a label named HAPROXY_GROUP with the value “internal”
- Deploy this change and then scale the service to 4 instances.

Exit criteria

- You will see 4 healthy of the api tasks running.
- Browse to the web site at the agent public IP address. Browse to the /stats.html page and you’ll see it is not working. You will fix this in the next step.

Task 7: Update the web service to call the internal load balancer

In this task, you will update the configuration for the web service so that it calls the api service at the internal load balancer address using the api service’s servicePort.

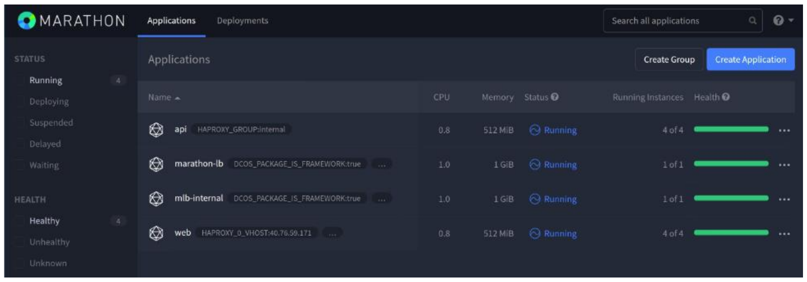
Tasks to complete

- Find the service port on the api service configuration.
- Edit the web service configuration.
- Modify the CONTENT_API_URL environment variable to match the following:

```
"env": {
   "CONTENT_API_URL": "http://mlb-internal.marathon.mesos:<api service port>"
 }
```

Exit criteria

- Once the web service deployment is completed, you will see the following applications running in the Services list.



Applications

Name	CPU	Memory	Status	Running Instances	Health
api HAPROXY_GROUP:internal	0.8	512 MiB	Running	4 of 4	🟩
marathon-lb DCOS_PACKAGE_IS_FRAMEWORK:true ...	1.0	1 GiB	Running	1 of 1	🟩
mlb-internal DCOS_PACKAGE_IS_FRAMEWORK:true ...	1.0	1 GiB	Running	1 of 1	🟩
web HAPROXY_0_VHOST:40.76.59.171 ...	0.8	512 MiB	Running	4 of 4	🟩

- Browse to the web site at the agent public IP address and navigate to the /stats.html page. Refresh the page and notice that the web and api task identifiers rotate between their respective 4 instances over time as they are requested through the load balancer.

YOU DID IT!

Exercise 6: Cleanup

Duration: 10 mins

Synopsis: In this exercise, attendees will deprovision any Azure resources that were created in support of the lab.

1. Delete both of the Resource Groups in which you placed all of your Azure resources.
    a. From the Portal, navigate to the blade of your Resource Group and click Delete in the command bar at the top.
    b. Confirm the deletion by re-typing the resource group name and clicking Delete.