DZone REFCARDZ

CONTENTS

Java Containerization

BY AMJAD AFANAH

WHAT IS A LINUX CONTAINER?

A Linux container is an operating system level virtualization technology that, unlike a virtual machine (VM), shares the host operating system kernel and makes use of the guest operating system system libraries for providing the required OS capabilities. Since there is no dedicated operating system, containers are more lightweight and start much faster than VMs.

In 2013, Docker was developed as an open platform for packaging, deploying, and running distributed applications. Docker uses its own Linux container library called libcontainer. It has become the most popular and widely used container management system.

This Refcard will focus on the design, deployment, service discovery and management of Java applications on Docker.

DOCKER'S ARCHITECTURE

Docker uses a typical client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon communicate via sockets or through a RESTful API.

REFERENCE	DESCRIPTION
Docker Images	Read-only templates that use union file systems to combine layers—making them very lightweight. Images are built from Dockerfiles.
Docker Registries	Store Docker images. Users can push (or publish) their images to a public registry (like Docker Hub) or to their own registry behind a firewall. Registries store the "tagged" images—allowing users to maintain different versions of the same image.
Docker Containers	Virtualized application environments that run on a Docker Host in isolation. Containers are launched from Docker images, adding a read-write layer on top of the image (using a union file system) as well as the network/bridge interface and IP address. When a container is launched, the process specified in the Dockerfile is executed and the logs are captured for auditing and diagnostics.
Dockerfiles	Composed of various commands (instructions) listed successively to automatically perform actions on a base image in order to create a new one. The instructions specify the operating system, application artifacts, data volumes, and exposed ports to be used, as well as the command (or script) to run when launching a Docker container.
Docker Host	A Linux host (either a physical/bare-metal server or a virtual machine) that is running a Docker daemon on which images can be built, pulled or pushed and containers can run in isolation.
Docker Client	Command-line utility or other tool that takes advantage of the Docker API (docs.docker.com/reference/api/docker_remote_api) to communicate with a Docker daemon

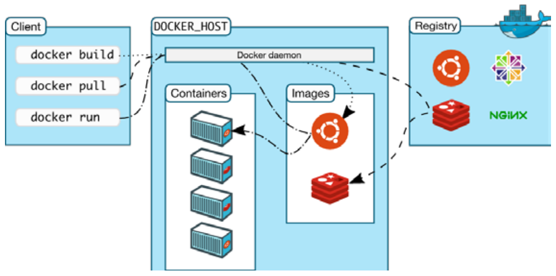


FIGURE 1: DOCKER ARCHITECTURE

HOW IS DOCKER DIFFERENT THAN JVM?

The JVM is Java's solution for application portability across different platforms—but Docker provides a different kind of virtualization that makes use of the guest operating system system libraries, and not just the Java application. When a Docker container is launched, a filesystem is allocated, along with the network/bridge interface and IP address. The command (or script) specified in the Dockerfile used to build the underlying Docker image is then executed and the resulting Linux process then runs in isolation.

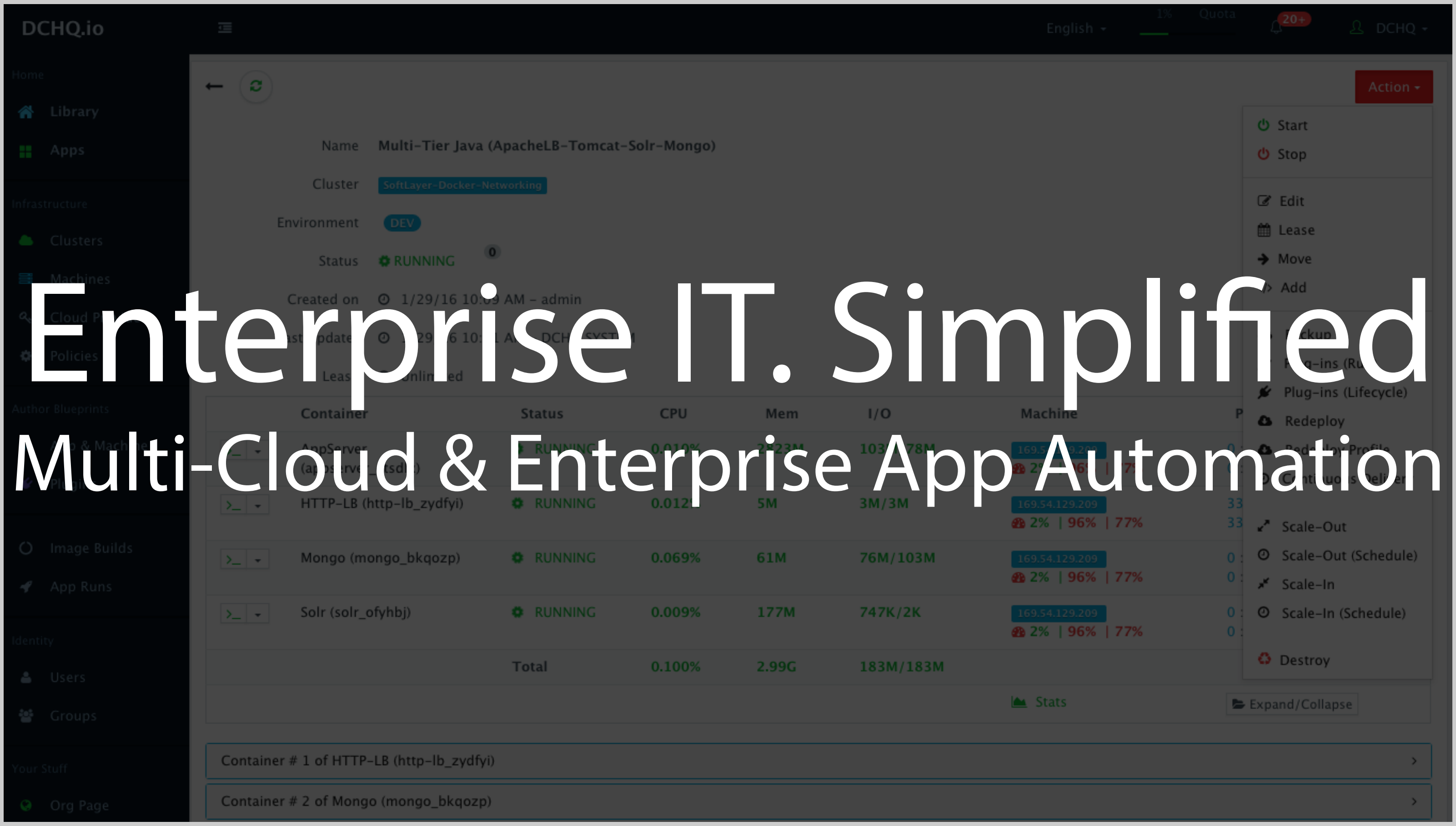
As a result, Docker can be used to package an entire JVM along with the JAR or WAR files and other parts of the application into a single container that can run on any Linux host consistently. This eliminates some of the challenges associated with making sure that the right JAR file version is used on the right JVM. Moreover, CPU and Memory resource controls can be used with Docker containers—allowing users to allocate maximum amounts of resources to allocate for an application.

DOCKER'S BENEFITS

The main advantages of Docker are:

Application Portability—Docker containers run exactly the same on any Linux host. This eliminates the challenge of deploying

DCHQ.io
Enterprise IT. Simplified.
Cloud Automation
App Automation
Governance
PRIVATE
PUBLIC
JBOSS
Deploy & monitor your Java applications on any cloud
Download Now
http://dchq.co/dchq-on-premise.html



Name Multi-Tier Java (ApacheLB-Tomcat-Solr-Mongo)
Cluster SoftLayer-Docker-Networking
Environment DEV
Status RUNNING
Created on 1/29/16 10:09 AM - admin

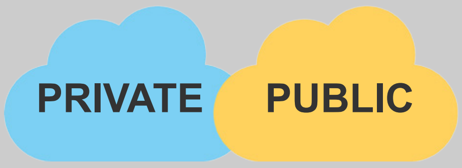
Container	Status	CPU	Mem	I/O	Machine
HTTP-LB (http-lb_zydfyi)	RUNNING	0.012%	5M	3M/3M	169.54.129.209 2% | 96% | 77%
Mongo (mongo_bkqozp)	RUNNING	0.069%	61M	76M/103M	169.54.129.209 2% | 96% | 77%
Solr (solr_ofyhbj)	RUNNING	0.009%	177M	747K/2K	169.54.129.209 2% | 96% | 77%
	Total	0.100%	2.99G	183M/183M	

Container # 1 of HTTP-LB (http-lb_zydfyi)
Container # 2 of Mongo (mongo_bkqozp)

Enterprise IT. Simplified

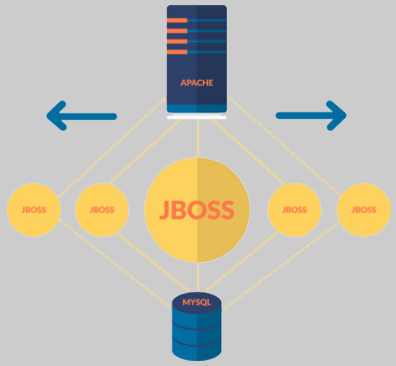
Multi-Cloud & Enterprise App Automation

Cloud Automation



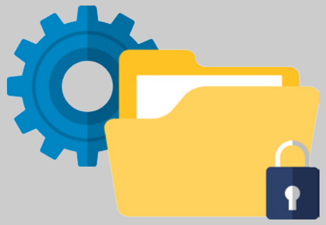
Infrastructure Provisioning
& Auto-Scaling
on 18+ Clouds &
Virtualization Platforms

App Automation



Model Anything, Deploy Anywhere
Compatible with Enterprise Apps
& Microservices
Infinitely Flexible Plug-in Framework

Governance



Role-based Access Controls,
Approvals, Quotas, Policies,
Monitoring & Alerts

Deploy & monitor your Java applications on any cloud

Download Now
http://dchq.co/dchq-on-premise.html

founders@dchq.io

http://dchq.co

DCHQ.io

applications across different compute infrastructure (e.g. local developer machine, vSphere VM or in the cloud).

Higher Server Density—The light-weight nature of containers allows developers to optimize server utilization for their application workloads by running more containers on the same host while achieving the same isolation and resource allocation benefits of virtual machines.

THE CHALLENGES WITH DOCKERIZING JAVA APPLICATIONS

Containerizing enterprise Java applications is still a challenge mostly because existing application composition frameworks do not address complex dependencies, service discovery or auto-scaling workflows post-provision. Moreover, the ephemeral design of containers meant that developers had to spin up new containers and re-create the complex dependencies & external integrations with every version update.

DCHQ, available in hosted and on-premise versions, addresses all of these challenges and simplifies the containerization of enterprise Java applications through an advanced application composition framework that extends Docker Compose with cross-image environment variable bindings, automatic container IP retrieval and injection, extensible plug-ins with lifecycle stages to handle service discovery use cases, and application clustering for high availability across multiple hosts or regions.

Once an application is provisioned, a user can monitor the CPU, Memory, & I/O of the running containers, get notifications & alerts, and get access to application backups, automatic scale in/out workflows, and plug-in execution workflows to update running containers. Moreover, out-of-box workflows that facilitate Continuous Delivery with Jenkins allow developers to refresh the Java WAR file of a running application without disrupting the existing dependencies & integrations.

DOCKER BASIC WORKFLOW

The typical workflow in Docker involves building an image from a Dockerfile or pulling an image from a registry (like Docker Hub).

Once the image is available on the Docker Host, a container can be launched as a runtime environment. Docker Hub has approximately 100 "official" images published by software vendors—eliminating the need to build a Tomcat or MySQL image from scratch in most cases. Once a container is running, it can be stopped, started or restarted using the CLI. If changes are made to the container, a user can commit the changes made into a new image with either the same tag (or version) or a different one. The new image can of course then be pushed to a registry (like Docker Hub).

Instead of listing all the supported workflows, this Refcard walks through Docker Java application examples—starting with basic use cases to more advanced ones.

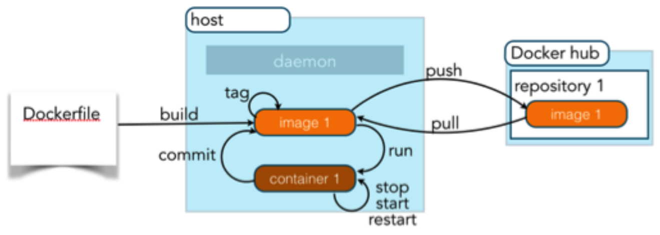


FIGURE 2: DOCKER BASIC WORKFLOW

SETTING UP DOCKER

INSTALLING DOCKER MANUALLY

You can refer to Docker's official documentation for detailed installation instructions. For Ubuntu for example, please refer to this document: docs.docker.com/engine/installation/ubuntulinux.

You can use these simple commands to install Docker:

```
apt-get update
apt-get -y install wget
wget -qO- https://get.docker.com/ | sh
```

PROVISION A DOCKER-ENABLED LINUX HOST ON ANY CLOUD

Sign Up on dchq.io for a free account.

- **Register a Cloud Provider**—navigate to Cloud Providers and register an end-point for one of the following: VMware vSphere, OpenStack, Cloudstack, AWS, Google Compute Engine, Microsoft Azure, Rackspace, DigitalOcean, IBM SoftLayer, and others.
- **Create a Cluster**—navigate to Clusters and create a new cluster with Docker networking selected
- **Provision a Docker-enabled Host**—navigate to Machines and provision a Docker-enabled Linux host on the cloud provider & cluster of your choosing.

You can refer to the detailed documentation here: dchq.co/docker-infrastructure-as-a-service.html.

DEPLOYING A CONTAINER

USING DOCKER CLI

In this example, we'll show you how to deploy a simple Tomcat container with a sample Java WAR file using the Docker CLI.

CREATE A DOCKER HUB ACCOUNT FOR STORING YOUR IMAGES

- Sign up a free account on Docker Hub: hub.docker.com
- Create a public repository called "tomcat"

BUILD A CUSTOM TOMCAT IMAGE WITH A SAMPLE JAVA WAR FILE ON YOUR LINUX MACHINE

Clone this GitHub project.

```
git clone https://github.com/dchqinc/basic-docker-tomcat-example.git
```

Build your own custom Tomcat image using the Dockerfile cloned from GitHub and push the image to your Docker Hub repository.

```
docker login
docker build -t <your username>/tomcat:latest .
docker push <your username>/tomcat:latest
```

Here's the basic Dockerfile used in this GitHub project.

```
FROM tomcat:8.0.21-jre8

COPY ./software/ /usr/local/tomcat/webapps/
```

The image is built using the Tomcat image with the tag 8.0.21-jre8 and copies the sample Java WAR file into the /usr/local/tomcat/webapps/ directory.

RUN THE CONTAINER USING THE DOCKER CLI

```
docker run -p 8080:8080 -d –name tomcat  <your-username>/
  tomcat:latest
```

ACCESS THE SAMPLE APPLICATION

You can access the sample application on this URL: http://host-ip:8080/sample

ACCESS THE LOGS

You can use this simple command to check the Catalina logs of the Tomcat container

```
docker logs tomcat
```

CHECK THE FILES INSIDE THE CONTAINER

You can run this command to enter the container and check the files under the webapps directory.

```
docker exec -it tomcat bash
ls -lrt /usr/local/tomcat/webapps
```

USING DCHQ

Here we'll show you how to deploy a simple Tomcat container with a sample Java WAR file using DCHQ

BUILD A CUSTOM TOMCAT IMAGE WITH A SAMPLE JAVA WAR FILE USING GITHUB AND DCHQ

- Sign Up on dchq.io for a free account
- Register your Docker Hub account by navigating to Cloud Providers and then selecting Docker Registries from the + dropdown
- Navigate to Image Builds and create new build by selecting

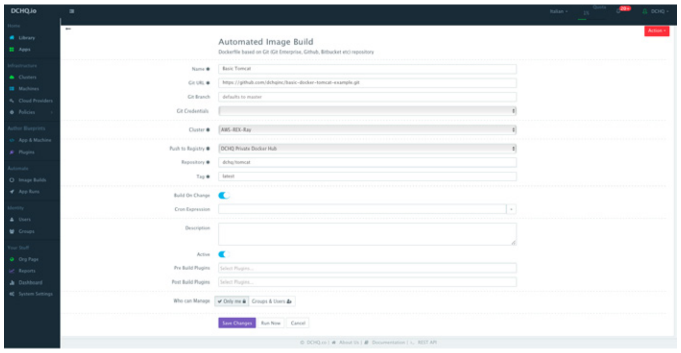

Automated Image Build
Dockerfile based on Git (Git Enterprise, Github, Bitbucket etc) repository
Name: Basic Tomcat
Git URL: https://github.com/dchqinc/basic-docker-tomcat-example.git
Git Branch: defaults to master
Git Credentials:
Cluster: AWS-REX-Ray
Push to Registry: DCHQ Private Docker Hub
Repository: dchq/tomcat
Tag: latest
Build On Change
Cron Expression
Description
Active
Pre Build Plugins: Select Plugins...
Post Build Plugins: Select Plugins...
Who can Manage: ✓ Only me | Groups & Users

FIGURE 4: DOCKER IMAGE BUILD

GitHub from the + dropdown

- Enter the this GitHub URL: github.com/dchqinc/basic-docker-tomcat-example.git
- Run the build using the "play" button

You can refer to the detailed documentation here: dchq.co/docker-compose.html

CREATE A DOCKER COMPOSE TEMPLATE FOR THE CUSTOM TOMCAT IMAGE

- Sign Up on dchq.io for a free account
- Navigate to App & Machine and select Docker Compose after clicking on the + button. Provide the following YAML file.

```
tomcat:
  image: your-username/tomcat:latest
  mem_min: 500m
  cpu_shares: 1
  publish_all: true
```

image: your-username/tomcat:latest - This is the Docker image that will be pulled from a registry to launch a container. By default, all images are pulled from Docker Hub. - In order to pull from a private repository, the registry_id parameter needs to be added. This should reference the ID of the Docker registry you would have registered. - To pull from an official repository (like MySQL), you can use simply enter image: mysql:latest - The tag name refers to the tagged images available in a repository.

mem_min: 500m - mem_min refers to the minimum amount of memory you would like to allocate to a container. In this case, the container will be allocated at least 50MB of memory and will continue using resources from the host based on the load.

cpu_shares: 1—cpu_shares refer to the amount of CPU allocated to the container

publish_all: true—If the value is true, this parameter will randomly bind all the exposed ports in the Dockerfile to a random port between 32000-59000 on the host. In this case, port 8080 is exposed in the Dockerfile—so a random port on the host will be bound to port 8080 in the container.

RUN THE CUSTOM TOMCAT IMAGE USING DCHQ

- Navigate to the Library and click Customize on the applications you would like to run (e.g. Basic Tomcat).
- Select the Cluster of your choosing and then click Run

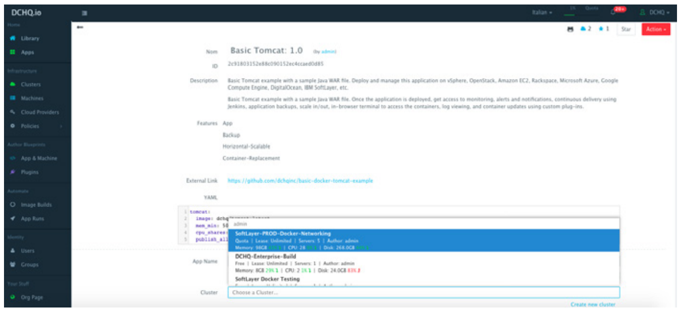

Nom: Basic Tomcat: 1.0 (by admin)
ID: 2c91803152e88c090152ec4ccaed0d85
Description: Basic Tomcat example with a sample Java WAR file. Deploy and manage this application on vSphere, OpenStack, Amazon EC2, Rackspace, Microsoft Azure, Google Compute Engine, DigitalOcean, IBM SoftLayer, etc.
Basic Tomcat example with a sample Java WAR file. Once the application is deployed, get access to monitoring, alerts and notifications, continuous delivery using Jenkins, application backups, scale in/out, in-browser terminal to access the containers, log viewing, and container updates using custom plug-ins.
Features: App, Backup, Horizontal-Scalable, Container-Replacement
External Link: https://github.com/dchqinc/basic-docker-tomcat-example
YAML
SoftLayer-PROD-Docker-Networking
Quota | Lease: Unlimited | Servers: 5 | Author: admin
DCHQ-Enterprise-Build
Free | Lease: Unlimited | Servers: 1 | Author: admin
Memory: 8GB 29% | CPU: 2 1% | Disk: 24.0GB 83%
SoftLayer Docker Testing
Cluster: Choose a Cluster...

FIGURE 5: DOCKER APP REQUEST

DEPLOY A MULTI-TIER NAMES DIRECTORY JAVA APPLICATION

CONFIGURING THE WEB.XML AND WEBAPP-CONFIG.XML FILES IN THE JAVA APPLICATION

You can clone this sample "Names Directory" Java application from GitHub.

```
git clone https://github.com/dchqinc/dchq-docker-java-example.git
```

This is the most important step in "Dockerizing" your Java application. In order to leverage the environment variables you can pass when running containers, you will need to make sure that your application is configured in a way that will allow you to change certain properties at request time—like:

- The database driver you would like to use
- The database URL
- The database credentials
- Any other parameters that you would like to change at request time (e.g. the min/max connection pool size, idle timeout, etc.)

To achieve this, you need pass environment variables in the context file storing your JNDI datasource connection details. Instead of hard-coding the database information, this file should have environment variables that can be overridden at request time. Here is the documentation on setting up JNDI connection details in Tomcat: tomcat.apache.org/tomcat-6.0-doc/jndi-datasource-examples-howto.html

Here's documentation on defining a context in Tomcat: tomcat.apache.org/tomcat-6.0-doc/config/context.html#Defining_a_context

In this example, we will configure web.xml to use the bootstrap Servlet to start up the Spring context.

github.com/dchqinc/dchq-docker-java-example/blob/master/src/main/webapp/WEB-INF/web.xml

```
<servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.
  DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/webapp-config.xml</param-
  value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

You will notice that the contextConfigLocation is referencing /WEB-INF/spring/webapp-config.xml

Next, we will need to configure parameters in the webapp-config.xml file to reference host environment variables that will be passed at request time.

github.com/dchqinc/dchq-docker-java-example/blob/master/src/main/webapp/WEB-INF/spring/webapp-config.xml

```
<bean id="dataSource" class="snaq.db.DBPoolDataSource" destroy-
  method="release">
    <property name="driverClassName" value="${database_
  driverClassName}"/>
    <property name="url" value="${database_url}"/>
    <property name="user" value="${database_username}"/>
    <property name="password" value="${database_password}"/>
    <property name="minPool" value="1"/>
    <property name="maxPool" value="10"/>
    <property name="maxSize" value="10"/>
    <property name="idleTimeout" value="60"/>
</bean>
```

You will notice that specific dataSource properties are referencing the following environment variables that will be passed on at request time:

- **database_driverClassName**
- **database_url**
- **database_username**
- **database_password**

If you are unable to change the context files in your Java application, then you can use DCHQ's plug-in framework to execute custom scripts to search for hard-coded parameter values and replace them with the right database connection details. DCHQ automatically retrieves information about the container IP, port and environment variable values for the connected database and allows you to inject this information inside Tomcat or other application servers that may need to connect to it.

USING THE LIQUIBASE BEAN TO INITIALIZE THE CONNECTED DATABASE

We typically recommend initializing the database schema as part of the Java application deployment itself. This way, you don't have to worry about maintaining separate SQL files that need to be executed on the database separately.

However, if you already have those SQL files and you still prefer executing them on the database separately—then DCHQ can help you automate this process through its plug-in framework. You can refer to this section for more information.

In order to include the SQL scripts for creating the database tables in the Java application, you will need to configure webapp-config.xml file to use Liquibase bean that checks the dataSource and runs new statements from upgrade.sql. Liquibase tracks which changelog statements have run against each database.

github.com/dchqinc/dchq-docker-java-example/blob/master/src/main/webapp/WEB-INF/spring/webapp-config.xml

```
<bean id="liquibase" class="liquibase.integration.spring.
  SpringLiquibase">
    <property name="dataSource" ref="dataSource" />
    <property name="changeLog" value="/WEB-INF/upgrade/upgrade.sql"
  />
</bean>
```

Here's the actual upgrade.sql file with the SQL statements for initializing the schema on the connected MySQL, PostgreSQL or Oracle database: github.com/dchqinc/dchq-docker-java-example/blob/master/src/main/webapp/WEB-INF/upgrade/upgrade.sql

DEPLOY A MULTI-TIER JAVA APPLICATION USING THE DOCKER CLI

BUILDING A DOCKER IMAGE FOR TOMCAT WITH THE JAVA WAR FILE USING THE DOCKER CLI

You can create a very simple Dockerfile that simply copies the Java WAR file into the /usr/local/tomcat/webapps directroy.

First, you can simply wget the actual Java WAR file from the GitHub project.

```
wget https://github.com/dchqinc/dchq-docker-java-example/raw/master/
  dbconnect.war
```

Then create the Dockerfile

```
FROM tomcat:8.0.21-jre8
RUN ["rm", "-rf", "/usr/local/tomcat/webapps/ROOT"] COPY dbconnect.
  war /usr/local/tomcat/webapps/ROOT.war
CMD ["catalina.sh", "run"]
```

Finally, build your "Names Directory" Tomcat image using the Dockerfile and push the image to your Docker Hub repository.

```
docker login
docker build -t your-username/tomcat-names:latest .
docker push your-username/tomcat-names:latest
```

RUN THE TWO-TIER JAVA APPLICATION USING THE DOCKER CLI

First, you can run the MySQL container.

```
docker run -d -e MYSQL_USER=root -e MYSQL_DATABASE=names -e MYSQL_
  ROOT_PASSWORD=password -name mysql mysql:latest
```

Then you can run the Tomcat container, which already contains the Names Directory Java WAR file. The Tomcat container will be linked to MySQL.

```
docker run -d -p 8080:8080 -name names-directory -link mysql:mysql
  -e database_driverClassName=com.mysql.jdbc.Driver -e database_
  url=jdbc:mysql://mysql:3306/names -e database_username=root -e
  database_password=password your-username/tomcat-names
```

You can access the Names Directory application on this URL: http://host-ip:8080

DEPLOY A MULTI-TIER JAVA APPLICATION USING DCHQ

CREATING DOCKER COMPOSE APPLICATION TEMPLATES THAT CAN RE-USED ON ANY LINUX HOST RUNNING ANYWHERE

Once logged in to DCHQ (either the hosted DCHQ.io or on-premise version), a user can navigate to App & Machine and then click on the + button to create a new Docker Compose template.

We have created 52 application templates using the official images from Docker Hub for the same "Names Directory" Java application—but for different application servers and databases: github.com/dchqinc/dchq-docker-java-example github.com/dchqinc/dchq-docker-java-solr-mongo-cassandra-example

The templates include examples of the following application stacks (for the same Java application): Apache HTTP Server (httpd) & NGINX for load balancing—Tomcat, Jetty, WebSphere and JBoss for the application servers—Solr for the full-text search—and MySQL, MariaDB, PostgreSQL, Oracle XE, Mongo and Cassandra for the databases.

DOCKER SERVICE DISCOVERY USING PLUG-IN LIFECYCLE STAGES

Across all these application templates, you will notice that some of the containers are invoking BASH, Perl, Python, or Ruby script plug-ins in order to configure the container at different life-cycle stages.

These plug-ins can be created by navigating to Plug-ins. Once the script is provided, the DCHQ agent will execute this script inside the container. A user can specify arguments that can be overridden at request time and post-provision. For a BASH script, anything preceded by the $ sign is considered an argument—for example:

file_url can be an argument that allows developers to specify the download URL for a WAR file. This can be overridden at request time and post-provision when a user wants to refresh the Java WAR file on a running container.

The plug-in ID needs to be provided when defining the YAML-based application template. For example, to invoke a BASH script plug-in for NGINX, we would reference the plug-in ID as follows:

```
LB:
  image: nginx:latest
  publish_all: true
  mem_min: 50m
  host: host1
  plugins:
    - !plugin
    id: 0H1Nk
    restart: true
    lifecycle: >
      on_create,
      post_scale_out:AppServer,
      post_scale_in:AppServer,
      post_start:AppServer,
      post_stop:AppServer
    arguments:
      # Use container_private_ip if you're using Docker networking
      - servers=server {{AppServer | container_private_ip}}:8080;
      # Use container_hostname if you're using Weave networking
      #- servers=server {{AppServer | container_hostname}}:8080;
```

The service discovery framework in DCHQ provides event-driven life-cycle stages that executes custom scripts to re-configure application components. This is critical when scaling out clusters for which a load balancer may need to be re-configured or a replica set may need to be re-balanced.

You will notice that the NGINX plug-in is getting executed during these different stages or events:

- When the NGINX container is created—in this case, the container IP's of the application servers are injected into the default configuration file to facilitate the load balancing to the right services
- When the application server cluster is scaled in or scale out—in this case, the updated container IP's of the application servers are injected into the default configuration file to facilitate the load balancing to the right services
- When the application servers are stopped or started—in this case, the updated container IP's of the application servers are injected into the default configuration file to facilitate the load balancing to the right services

So the service discovery framework here is doing both service registration (by keeping track of the container IP's and environment variable values) and service discovery (by executing the right scripts during certain events or stages).

Here are the parameters supported when invoking a plugin: -

- **id** — this is the ID of the plug-in. This can be retrieved from Manage > Plugins and then clicking Edit on your plugin of choice.
- **restart** — this is a Boolean parameter. If set to true, then the container is restarted after executing the plugin.
- **arguments** — you can override the arguments specified in the plugin here. The arguments can be overridden when creating the template, when deploying the application and post-provision.

The lifecycle parameter in plug-ins allows you to specify the exact stage or event to execute the plug-in. If no lifecycle is specified, then by default, the plug-in will execute on_create. Here are the supported lifecycle stages:

- **on_create**—executes the plug-in when creating the container
- **on_start**—executes the plug-in after a container starts
- **on_stop**—executes the plug-in before a container stops
- **on_destroy**—executes the plug-in before destroying a container
- **post_create**—executes the plug-in after the container is created and running
- **post_start[:Node]**—executes the plug-in after another container starts
- **post_stop[:Node]**—executes the plug-in after another container stops
- **post_destroy[:Node]**—executes the plug-in after another container is destroyed
- **post_scale_out[:Node]**—executes the plug-in after another cluster of containers is scaled out
- **post_scale_in[:Node]**—executes the plug-in after another cluster of containers is scaled in

The application servers (Tomcat, Jetty, JBoss and WebSphere) are also invoking a BASH script plug-in to deploy the Java WAR file from the accessible GitHub URL: github.com/dchqinc/dchq-docker-java-example/raw/master/dbconnect.war

USING PLUG-INS AND THE HOST PARAMETER TO DEPLOY HIGHLY-AVAILABLE DOCKER JAVA APPLICATIONS

You will notice that the cluster_size parameter allows you to specify the number of containers to launch (with the same application dependencies). This allows you to deploy a cluster of application servers for example.

The host parameter allows you to specify the host you would like to use for container deployments. This is possible if you have selected Weave as the networking layer when creating your clusters. That way you can ensure high-availability for your application server clusters across different hosts (or regions) and you can comply with affinity rules to ensure that the database runs on a separate host for example. Here are the values supported for the host parameter:

- **host1, host2, host3, etc.**—selects a host randomly within a data-center (or cluster) for container deployments
- **IP Address 1, IP Address 2, etc.**—allows a user to specify the actual IP addresses to use for container deployments
- **Hostname 1, Hostname 2, etc.**—allows a user to specify the actual hostnames to use for container deployments
- **Wildcards (e.g. "db-*", or "app-srv-*")**—to specify the wildcards to use within a hostname

ENVIRONMENT VARIABLE BINDINGS ACROSS IMAGES

Additionally, a user can create cross-image environment variable bindings by making a reference to another image's environment variable. In this case, we have made several bindings—including database_url=jdbc:mysql://{{MySQL|container_ip}}:3306/{{MySQL|MYSQL_DATABASE}}—in which the database container IP is resolved dynamically at request time and is used to ensure that the application servers can establish a connection with the database.

Here is a list of supported environment variable values:

- **{{alphanumeric | 8}}**—creates a random 8-character alphanumeric string. This is most useful for creating random passwords.
- **{{Image Name | ip}}**—allows you to enter the host IP address of a container as a value for an environment variable. This is most useful for allowing the middleware tier to establish a connection with the database.
- **{{Image Name | container_ip}}**—allows you to enter the name of a container as a value for an environment variable. This is most useful for allowing the middleware tier to establish a secure connection with the database (without exposing the database port).
- **{{Image Name | container_private_ip}}**—allows you to enter the internal IP of a container as a value for an environment variable. This is most useful for allowing the middleware tier to establish a secure connection with the database (without exposing the database port).
- **{{Image Name | port_Port Number}}**—allows you to enter the Port number of a container as a value for an environment variable. This is most useful for allowing the middleware tier to establish a connection with the database. In this case, the port number specified needs to be the internal port number—i.e. not the external port that is allocated to the container. For example, {{PostgreSQL | port_5432}} will be translated to the actual external port that will allow the middleware tier to establish a connection with the database.
- **{{Image Name | Environment Variable Name}}**—allows you to enter the value an image's environment variable into another image's environment variable. The use cases here are endless—as most multi-tier applications will have cross-image dependencies.

Here is one example template, but you can check out the GitHub projects for 50 more examples at github.com/dchqinc/dchq-docker-java-example and github.com/dchqinc/dchq-docker-java-solr-mongo-cassandra-example

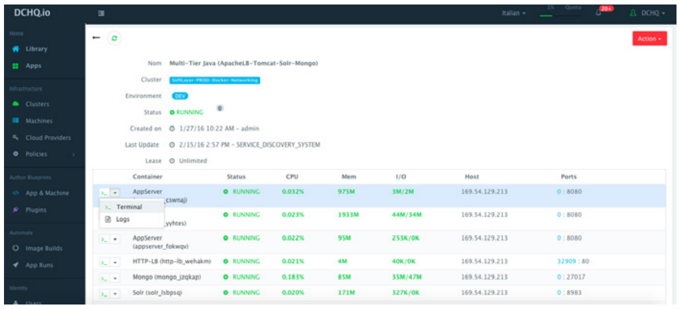
3-TIER JAVA (NGINX—TOMCAT—MYSQL)

```
LB:
  image: nginx:latest
  publish_all: true
  mem_min: 50m
  host: host1
  plugins:
    - !plugin
      id: 0H1Nk
      restart: true
      lifecycle: >
        on_create,
        post_scale_out:AppServer,
        post_scale_in:AppServer,
        post_stop:AppServer,
        post_start:AppServer
      arguments:
        # Use container_private_ip if you're using Docker networking
        - servers=server {{AppServer | container_private_ip}}:8080;
        # Use container_hostname if you're using Weave networking
        #- servers=server {{AppServer | container_hostname}}:8080;
AppServer:
  image: tomcat:8.0.21-jre8
  mem_min: 600m
  host: host1
  cluster_size: 1
  environment:
    - database_driverClassName=com.mysql.jdbc.Driver
    - database_url=jdbc:mysql://{{MySQL|container_hostname}}:3306/
  {{MySQL|MYSQL_DATABASE}}
    - database_username={{MySQL|MYSQL_USER}}
    - database_password={{MySQL|MYSQL_ROOT_PASSWORD}}
  plugins:
    - !plugin
      id: oncXN
      restart: true
      arguments:
        - file_url=https://github.com/dchqinc/dchq-docker-java-
  example/raw/master/dbconnect.war
        - dir=/usr/local/tomcat/webapps/ROOT.war
        - delete_dir=/usr/local/tomcat/webapps/ROOT
MySQL:
  image: mysql:latest
  host: host1
  mem_min: 400m
  environment:
    - MYSQL_USER=root
    - MYSQL_DATABASE=names
    - MYSQL_ROOT_PASSWORD={{alphanumeric|8}}
```

ACCESSING THE IN-BROWSER TERMINAL FOR THE RUNNING CONTAINERS

A command prompt icon should be available next to the containers' names on the Live Apps page. This allows users to enter the container using a secure communication protocol through the agent message queue. A white list of commands can be defined by the Tenant Admin to ensure that users do not make any harmful changes on the running containers.

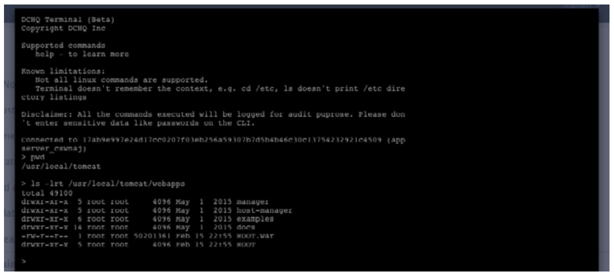
For the Tomcat deployment for example, you can use the command prompt to make sure that the Java WAR file was deployed under the /usr/local/tomcat/webapps/ directory.



Nom Multi-Tier Java (ApacheLB-Tomcat-Solr-Mongo)
Cluster SoftLayer-PROD-Docker-Networking
Environment DEV
Status RUNNING
Created on 1/27/16 10:22 AM - admin
Last Update 2/15/16 2:57 PM - SERVICE_DISCOVERY_SYSTEM
Lease Unlimited

Container	Status	CPU	Mem	I/O	Host	Ports
AppServer (_cswnaj)	RUNNING	0.032%	975M	3M/2M	169.54.129.213	0 : 8080
(_yyhtes)	RUNNING	0.023%	1933M	44M/34M	169.54.129.213	0 : 8080
AppServer (appserver_fokwqv)	RUNNING	0.022%	95M	253K/0K	169.54.129.213	0 : 8080
HTTP-LB (http-lb_wehakm)	RUNNING	0.021%	4M	40K/0K	169.54.129.213	32909 : 80
Mongo (mongo_jzqkap)	RUNNING	0.183%	85M	35M/47M	169.54.129.213	0 : 27017
Solr (solr_lsbpsq)	RUNNING	0.020%	171M	327K/0K	169.54.129.213	0 : 8983

FIGURE 6: CONTAINER TERMINAL REQUEST



```
DCHQ Terminal (Beta)
Copyright DCHQ Inc

Supported commands
   help - to learn more

Known limitations:
   Not all linux commands are supported.
   Terminal doesn't remember the context, e.g. cd /etc, ls doesn't print /etc dire
ctory listings

Disclaimer: All the commands executed will be logged for audit puprose. Please don
't enter sensitive data like passwords on the CLI.

connected to 17ab9e997e24d17cc0207f03eb256a59307b7d5b4b46c30c13754232921c4509 (app
server_cswnaj)
> pwd
/usr/local/tomcat

> ls -lrt /usr/local/tomcat/webapps
total 49100
drwxr-xr-x  5 root root     4096 May  1  2015 manager
drwxr-xr-x  5 root root     4096 May  1  2015 host-manager
drwxr-xr-x  6 root root     4096 May  1  2015 examples
drwxr-xr-x 14 root root     4096 May  1  2015 docs
-rw-r--r--  1 root root 50201361 Feb 15 22:55 ROOT.war
drwxr-xr-x  5 root root     4096 Feb 15 22:55 ROOT

>
```

FIGURE 7: CONTAINER TERMINAL

MONITORING THE CPU, MEMORY & I/O UTILIZATION OF THE RUNNING CONTAINERS

Once the application is up and running, a user can monitor the CPU, Memory, & I/O of the running containers to get alerts when these metrics exceed a pre-defined threshold. This is especially useful when developers are performing functional & load testing.

A user can perform historical monitoring analysis and correlate issues to container updates or build deployments. This can be done by clicking on the *Stats** button. A custom date range can be selected to view CPU, Memory and I/O historically.

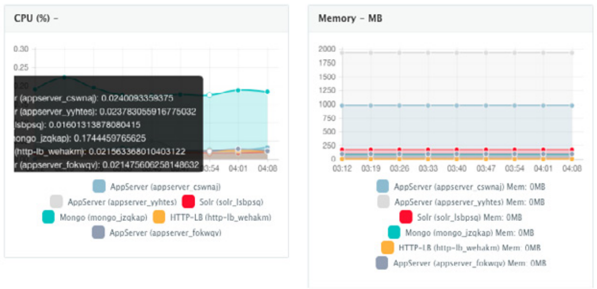
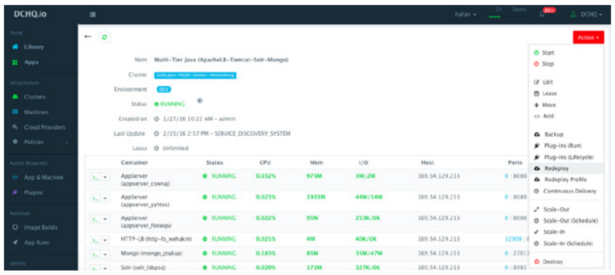


FIGURE 8: CONTAINER MONITORING

REDEPLOYING CONTAINERS WHEN A NEW IMAGE IS PUSHED INTO A DOCKER REGISTRY

A user can set up a container "re-deployment" policy that can be triggered when a new image is pushed into a Docker registry. This allows users to create continuous delivery workflows based on Docker image builds. This can be done by clicking on the Actions menu of the running application and then selecting Redeploy. A user can then select the Registry and the name of the Repository.



Nom: Multi-Tier Java (ApacheLB-Tomcat-Solr-Mongo)
Environment: DEV
Status: RUNNING
Created on: 1/27/16 10:22 AM - admin
Last Update: 2/15/16 2:57 PM - SERVICE_DISCOVERY_SYSTEM
Lease: Unlimited

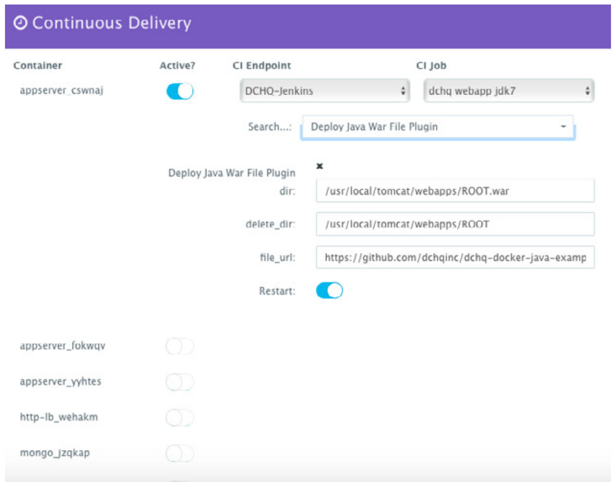
Container	Status	CPU	Mem	I/O	Host	Ports
AppServer (appserver_cswnaj)	RUNNING	0.032%	975M	3M/2M	169.54.129.213	0 : 8080
AppServer (appserver_yyhtes)	RUNNING	0.023%	1933M	44M/34M	169.54.129.213	0 : 8080
AppServer (appserver_fokwqv)	RUNNING	0.022%	95M	253K/0K	169.54.129.213	0 : 8080
HTTP-LB (http-lb_wehakm)	RUNNING	0.021%	4M	40K/0K	169.54.129.213	12909 :
Mongo (mongo_jzqkap)	RUNNING	0.183%	85M	35M/47M	169.54.129.213	0 : 2701
Solr (solr_lsbpsq)	RUNNING	0.020%	171M	327K/0K	169.54.129.213	0 : 8983

FIGURE 9: CONTAINER REDEPLOY

ENABLING THE CONTINUOUS DELIVERY WORKFLOW WITH JENKINS

UPDATE THE WAR FILE OF THE RUNNING APPLICATION WHEN A BUILD IS TRIGGERED

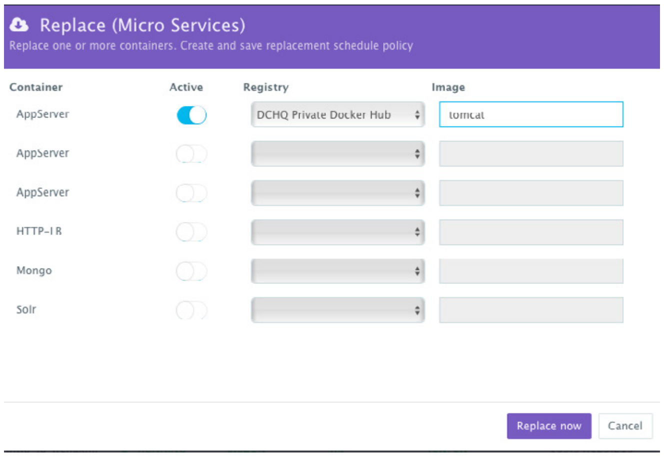
Many developers may wish to update the running application server containers with the latest Java WAR file instead of re-deploying containers. This may be a common practice in DEV/TEST environments. For that, DCHQ allows developers to enable a continuous delivery workflow with Jenkins. This can be done by clicking on the Actions menu of the running application and then selecting Continuous Delivery. A user can select a Jenkins instance that has already been registered with DCHQ, the actual Job on Jenkins that will produce the latest WAR file, and then a plug-in to grab this build and deploy it on a running application server. Once this policy is saved, DCHQ will grab the latest WAR file from Jenkins any time a build is triggered and deploy it on the running application server.



Container	Active?	CI Endpoint	CI Job
appserver_cswnaj	on	DCHQ-Jenkins	dchq webapp jdk7
appserver_fokwqv	off		
appserver_yyhtes	off		
http-lb_wehakm	off		
mongo_jzqkap	off		

Search...: Deploy Java War File Plugin
Deploy Java War File Plugin
dir: /usr/local/tomcat/webapps/ROOT.war
delete_dir: /usr/local/tomcat/webapps/ROOT
file_url: https://github.com/dchqinc/dchq-docker-java-examp
Restart: on

FIGURE 10: CONTINUOUS DELIVERY


Replace (Micro Services)
Replace one or more containers. Create and save replacement schedule policy
Replace now
Cancel

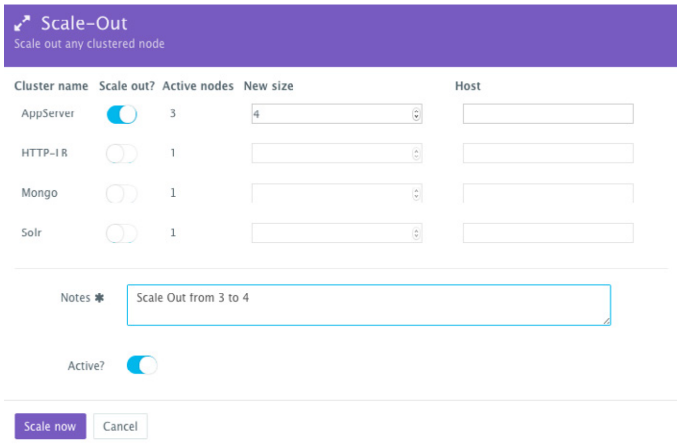
Container	Active	Registry	Image
AppServer	on	DCHQ Private Docker Hub	tomcat
AppServer	off		
AppServer	off		
HTTP-LB	off		
Mongo	off		
Solr	off		

FIGURE 11: CONTAINER REDEPLOY DETAILS.PNG

SCALING OUT THE TOMCAT APPLICATION SERVER CLUSTER

If the running application becomes resource constrained, a user can to scale out the application to meet the increasing load. Moreover, a user can schedule the scale out during business hours and the scale in during weekends for example.

To scale out the cluster of Tomcat servers from 1 to 2, a user can click on the Actions menu of the running application and then select Scale Out. A user can then specify the new size for the cluster and then click on Run Now.



Scale-Out
Scale out any clustered node

Cluster name	Scale out?	Active nodes	New size	Host
AppServer	on	3	4	
HTTP-LB	off	1		
Mongo	off	1		
Solr	off	1		

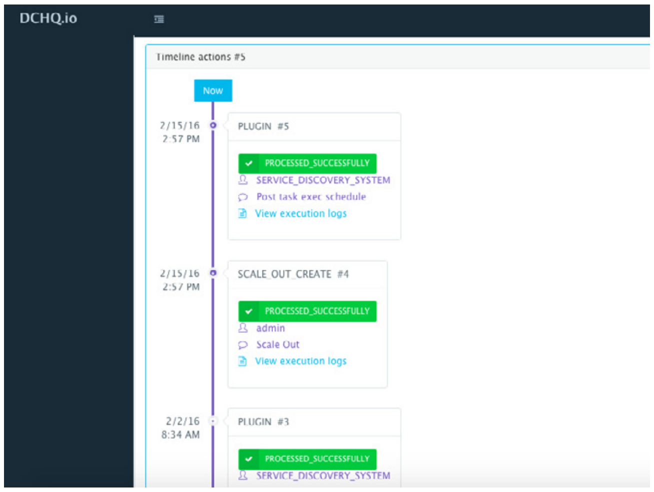
Notes ✱ Scale Out from 3 to 4

Active? on

FIGURE 12: SCALE OUT

As the scale out is executed, the Service Discovery framework will be used to update the load balancer. For Apache HTTP Server, for example, a plug-in updates httpd.conf file to inject the application server container IP's to ensure that the load balancer is routing traffic to the new application server containers added as part of the scale out.

An application time-line is available to track every change made to the application for auditing and diagnostics. This can be accessed from the expandable menu at the bottom of the page of a running application.



DCHQ.io

Timeline actions #5

Now

2/15/16 2:57 PM
PLUGIN #5
PROCESSED_SUCCESSFULLY
SERVICE_DISCOVERY_SYSTEM
Post task exec schedule
View execution logs

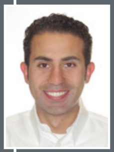
2/15/16 2:57 PM
SCALE OUT CREATE #4
PROCESSED_SUCCESSFULLY
admin
Scale Out
View execution logs

2/2/16 8:34 AM
PLUGIN #3
PROCESSED_SUCCESSFULLY
SERVICE_DISCOVERY_SYSTEM

FIGURE 13: APP TIMELINE

Alerts and notifications are also available for when containers or hosts are down or when the CPU & Memory Utilization of either hosts or containers exceed a defined threshold.

ABOUT THE AUTHOR



AMJAD AFANAH is the Founder of DCHQ, a governance, deployment automation and lifecycle management platform for Docker-based applications. He previously worked at VMware and Oracle.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0