Platform as a Service and Container Clouds using IBM Bluemix and Docker for Cloud Migrations
Agenda

- Why is Migration to Cloud Enabled not Enough?
- A Overview of Bluemix Platform as a Service
  - Cloud Foundry Platform
  - Available Services
  - Container Cloud
- Docker Overview
  - What are Containers?
  - Virtual Machines vs. Containers
  - Creating Images
  - Deploying to the Bluemix Container Cloud
One Form of Migration to Cloud is Moving What You Have
Traditional Migration Types

Map
Provision
Migrate

Discover/Analyze

Fit for Scenario

Analysis determines approach

Workload/Application Migration

Image Migration

Image Copy
Image Fix-up

Tooling

Image Fix-up

Fix-up

App OS
x series
App OS
App OS

Analysis determines approach

Discover/Analyze

Test

user

Workload/Application Migration

System z
System p

Cloud
Migration to Cloud Enabled

- Image Migration and Workload Migration don’t take full advantage of the Cloud
- You have the same number of VM’s as you had Physical Servers
- Which means you need to manage the same number of servers in the end
- …so what did you really save?

```
=  VM  VM  VM  VM  VM
```

```
=  Cloud
```
Transformation to Cloud Native

What if you could Transformation
What you Have to Cloud Native?
You Should Take Advantage of Cloud Services
Some Cloud Platform Services

- Heroku
- IBM Bluemix™
- Engine Yard™
- App Engine
- OpenShift
- AppScale
- Docker
An Introduction To:

IBM Bluemix
What is Bluemix?

- Platform Environments Based on Cloud Foundry
- Container Cloud Based on Docker
- Plain Ole’ Virtual Machines Based on OpenStack
Advantages of Platform as a Service

- Minimal set-up time to get coding
- Developers can concentrate on the application and not the infrastructure
- Large number of services to take advantage of:
  - Database
  - Messaging
  - Analytics
  - Mobile
  - etc...
- Continuous deployment pipeline into the cloud
- Very easy to scale with growth
- Delete it if it doesn’t work out and pay nothing (or very little)
Example of Binding a Service
Example of Development Pipeline

Pipeline: All Stages

**Build Stage**
- **Stage Passed**
- **Last Input**: Git URL
- **Jobs**: Build Succeeded 3 min ago
- **Last Execution Result**: Build 6

**Deploy Stage**
- **Stage Passed**
- **Last Input**: Stage: Build Stage / Job: Build 6
- **Jobs**: Deploy to dev Succeeded last now
- **Last Execution Result**: nyu-demo1
  - [Link to runtime log]
  - Build 6
DEMO OF IBM BLUEMIX
An Introduction To: docker
What is Docker?

- Docker is a light-weight container service that runs on Linux
  - File system overlay
  - One Process Space
  - One Network Interface

- Shares the Linux kernel with other containers and processes
  - Uses LXC

- Containers can encapsulate a run-time environment
  - They can contain code, libraries, package manager, data, etc.

- Almost no overhead
  - Containers spin up in seconds not minutes like VMs
  - Native performance because there is no emulation
  - Package only what you need

- Open Source (GitHub public repository + issue tracked)
What is the Docker Architecture?
So how does Docker work?

- You can build Docker images that hold your applications.
- You can create Docker containers from those Docker images to run your applications.
- You can share those Docker images via Docker Hub or your own registry.
- You can pull those images from the Docker registry to deploy them as Containers on a server running Docker
- You can even deploy those containers in the Bluemix Container Cloud!

… and that’s how we migrate to the cloud using Docker ;-}
Virtual Machines vs. Containers

- Virtual Machines are heavy-weight emulations of real hardware

<table>
<thead>
<tr>
<th>App</th>
<th>App</th>
<th>App</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
<td>OS</td>
</tr>
<tr>
<td>VM</td>
<td>VM</td>
<td>VM</td>
</tr>
</tbody>
</table>

Hypervisor

Host OS

Virtual Machines

- Containers are light-weight like a process

<table>
<thead>
<tr>
<th>App</th>
<th>App</th>
<th>App</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Docker

Host OS (CoreOS)

Containers

The app looks like it's running on the Host OS
Containers Are Created From Images

- Docker Images are templates for creating containers
- Best practice is to make containers stateless
  - All state is maintained elsewhere (e.g., host OS, object stores, etc.)
- Images can be created manually or from Dockerfiles
- Using Dockerfiles allows you to treat images as code
  - Check them into GitHub and use version control
- Dockerfiles are similar to Vagrantfiles we used in previous lectures
Create a Docker Image Manually

1. `docker run ubuntu bash`
2. `apt-get install blah, blah, blah`
3. `docker commit <containerid> <imagename>`
4. `docker run <imagename> bash`
5. `git clone git://github.com/mycode.git`
6. `pip install –r requirements.txt`
7. `docker commit <containerid> <imagename>`
8. `repeat steps 4-7 as necessary`
9. `docker tag <imagename> <user/image>`
10. `docker push <user/image>`
Create a Docker Image from a Dockerfile

1. FROM ubuntu:trusty
2. MAINTAINER John Rofran "rofrano@us.ibm.com"
3. RUN apt-get update --fix-missing
4. RUN apt-get install -y --fix-missing \\ software-properties-common \\ python-software-properties \\ build-essential \\ python-dev \\ python-pip
5. RUN pip install elasticsearch elasticsearch-dsl kafka-python pykafka
6. VOLUME [ "/var/log/va_image/" ]
7. ADD . /opt/va_image/
8. ENTRYPOINT [ "/opt/va_image/report_image_va.py" ]

`docker build -t rofrano/image-reporter .`
The Docker Hub

- You can go to https://hub.docker.com to get lots of premade images
Create a Docker Image from another Image

- You can create a docker image by starting with an existing image:
  1. FROM nginx
  2. ADD contents /usr/share/nginx/html

    
    docker build –t mynginx .

- You can develop using a Docker image (just like with Vagrant)
  1. FROM nginx
  2. VOLUME ["/usr/share/nginx/html"

    
    docker build –t mynginx .
    docker run –d –v /Users/rofrano/web:/usr/share/nginx/html mynginx

- You can copy data into a container or externalize it as a volume so that it can vary
  1. FROM nginx
  2. COPY content /usr/share/nginx/html
  3. COPY conf /etc/nginx
  4. VOLUME /var/log/nginx/log
Installing Docker Locally on your Computer

- Installing on Mac OS X or Windows
  - Get Docker Toolbox from the Docker web site and install it:
    - [https://www.docker.com/docker-toolbox](https://www.docker.com/docker-toolbox)

- Installing on Ubuntu Trusty 14.04 (LTS)
  - `sudo apt-get update`
  - `sudo apt-get install docker-engine`

- Installing on Other OS
  - See Docker installation guide:
    - [https://docs.docker.com/installation/](https://docs.docker.com/installation/)

© 2015 IBM Corporation
Using the Docker Toolkit on your local computer

You should see this when Docker is loaded as a shell:

docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

iotia:nyu-demol rofrano$
Creating a Container from an Image

Run `docker ps` to see that there are no containers running:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                CREATED       STATUS         PORT                     NAMES
```

Run `docker images` to see what images we have or pull one from the docker registry with `docker pull`:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
postgres            latest              77cad3350a1e        10 days ago         265.1 MB
ruby                2.2.0               51473a2975de        9 months ago        774.7 MB
```

Run `docker run` to create a container from the postgres image:

```
$ docker run -d --name mypg postgres
bcc5ea77c04cf5dc51c5a3075926ea4a76121789c9cb6c4d41c66349a3db416c
```

Run `docker ps` again to see we have a running container called “mypg”:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                CREATED       STATUS         PORT                     NAMES
bcc5ea77c04c   postgres   */docker-entrypoint.   3 days ago    Up 18 seconds 0.0.0.0:5432->5432/tcp   mypg
```
Creating a container from your own Image
(or just enough Docker to be Dangerous!)

- Create a Dockerfile for your image
  - `vi Dockerfile`

- Build an image with:
  - `docker build -t `<your_image_name>` .`

- Run a container with:
  - `docker run -d --name `<your_container_name>` `<your_image_name>``

- Show the containers logs with:
  - `docker logs `<your_container_name>``

- Stop the container with:
  - `docker stop `<your_container_name>``

- Remove the container with:
  - `docker rm `<your_container_name>``

- Remove the image with:
  - `docker rmi `<your_image_name>``
Advanced Docker Options

- **Linking** (`--link <other>`)  
  - You can link containers so that they don’t need to know each other’s IP addresses

- **Volumes** (`-v <host>:<cont>`)  
  - You can mount volumes so that data can remain outside of the container and keep the containers stateless

- **Ports** (`-P` or `-p <host_port>:<cont_port>`)  
  - You can map ports to expose them outside of the container `<host>:<container>`

- **Hostname** (`-h <name>`)  
  - Assigns a hostname to the container

- **Daemon** (`-d`)  
  - Run the container as a daemon process

- **Fault Tolerance** (`--always-restart`)  
  - Tells Docker to restart the container if the main process ever fails

- **Environment Variables** (`-e ENV_VAR=xxx`)  
  - Passes environment variables into a container at startup
Several Containers can be Composed

- Docker-compose will allow you to create and link multiple containers at once:

```yaml
mq:
  image: mikaelhg/docker-rabbitmq
  restart: always
  ports:
    - "5672:5672"
    - "15672:15672"

db:
  image: postgres
  restart: always
  volumes:
    - /var/lib/postgresql/data:/var/lib/postgresql/data
  ports:
    - "5432:5432"

app:
  build: .
  restart: always
  volumes:
    - /var/log/rails:/app/log
  ports:
    - "80:80"
    - "443:443"
  links:
    - db
    - mq
  environment:
    RAILS_ENV: production
```
Creating a Container in Bluemix

DOCKER DEMO TIME !!!!
Homework Assignment #2 – Platform as a Service Hands-On

- **Cloud Foundry:** Deploy an application in Bluemix using Cloud Foundry (60%)
  - Go to: [https://github.com/rofrano/nyu_hw2](https://github.com/rofrano/nyu_hw2) and clone this Ruby on Rails application called: “NYU Cloud Blog” (the README.md file will have instructions)
  - Use the Vagrantfile in the project you just cloned to create a local Ruby on Rails environment and run the NYU Cloud Blog application (10%)
    - Note that the local application is using the MySQL database in a Docker container locally. This is important when you go to deploy into Bluemix!
    - Show us the app running on your laptop
  - Deploy the NYU Cloud Blog on Bluemix using Cloud Foundry Ruby Services (50%)
    - You will need to create both a Ruby environment and a Database Service for MySQL (I recommend using ClearDB)
    - Steps 4 and 6 in this tutorial will give you hints on how to deploy (but they are using DB2 so don’t follow exactly): [http://www.ibm.com/developerworks/cloud/library/cl-blograils-app/](http://www.ibm.com/developerworks/cloud/library/cl-blograils-app/)
    - Show us the app running at a URL in the Bluemix cloud

- **Container Cloud:** Deploy a web site in Bluemix using Containers (40%)
  - Create a Docker image from nginx and add at least one web page to it and run a container locally on your laptop (20%)
    - Show us your Dockerfile or steps to reproduce if created manually
    - Show us the app running at a URL on your laptop
  - Push your nginx Docker image to Bluemix and run it in the Container Cloud (20%)
    - Show us the steps you took to push your image to Bluemix and deploy it
    - Show us the app running at a URL in the Bluemix cloud