Docker and Kubernetes for Java EE Developers

v1.0, Jun 16, 2015

Table of Contents

1. Preface 2
2. Setup Environments 3
    2.1. Instructor 3
    2.2. Attendees (From Instructor's Machine) 3
    2.3. Attendees (From Internet) 8
3. Docker Basics 9
    3.1. Docker Machine 12
    3.2. Docker Client 12
    3.3. Verify Docker Configuration 13
4. Run Container 14
    4.1. Pull Image 14
    4.2. Run Container 16
    4.3. Run Container with Default Port 18
    4.4. Run Container with Specified Port 19
    4.5. Enabling WildFly Administration 20
    4.6. Stop and Remove Container 22
5. Deploy Java EE 7 Application (Pre-Built WAR) 23
6. Deploy Java EE 7 Application (Container Linking) 24
7. Build and Deploy Java EE 6 Application (Ticket Monster) 26
    7.1. Build Application 27
    7.2. Start Database Server 28
    7.3. Start Application Server 29
    7.4. Configure JBoss Developer Studio 30
    7.5. Deploy Application Using Shared Volumes 34
    7.6. Deploy Application Using CLI (OPTIONAL) 35
    7.7. Deploy Application Using Web Console (OPTIONAL) 36
    7.8. Deploy Application Using Management API (OPTIONAL) 37
8. Docker Maven Plugin 41
    8.1. Run Java EE Application 41
    8.2. Understand Plugin Configuration 41

9. Docker Tools in Eclipse 43
9.1. Install Docker Tools Plugins 43
9.2. Docker Explorer 45
9.3. Docker Images 49
9.4. Docker Containers 51
9.5. Details on Images and Containers 52
10. Test Java EE Applications on Docker 54
11. Multiple Containers Using Docker Compose 56
11.1. Configuration File 57
11.2. Start Services 58
11.3. Verify Application 60
11.4. Stop Services 61
11.5. Scale Services 62
12. Java EE Application on Mod Cluster 62
13. Java EE Application on Docker Swarm Cluster 66
14. Java EE Application on Kubernetes Cluster 72
14.1. Key Concepts 73
14.2. Start Kubernetes Cluster 74
14.3. Deploy Java EE Application 79
14.4. Access Java EE Application 84
14.5. Self-healing Pods 84
14.6. Application Logs 84
14.7. Delete Kubernetes Resources 85
14.8. Stop Kubernetes Cluster 85
14.9. View Logs 85
14.10. Debug Kubernetes Master (OPTIONAL) 86
15. Common Docker Commands 87
16. References 88

1. Preface

Containers are enabling developers to package their applications (and underlying dependencies) in new ways that are portable and work consistently everywhere? On your machine, in production, in your data center, and in the cloud. And Docker has become the de facto standard for those portable containers in the cloud.

Docker is the developer-friendly Linux container technology that enables creation of your stack: OS, JVM, app server, app, and all your custom configuration. So with all it offers, how comfortable are you and your team taking Docker from development to

production? Are you hearing developers say, “But it works on my machine!” when code breaks in production?

This lab offers developers an intro-level, hands-on session with Docker, from installation, to exploring Docker Hub, to crafting their own images, to adding Java apps and running custom containers. It will also explain how to use Swarm to orchesorchestrate these containers together. This is a BYOL (bring your own laptop) session, so bring your Windows, OSX, or Linux laptop and be ready to dig into a tool that promises to be at the forefront of our industry for some time to come.

Latest content of this lab is always at https://github.com/javaee-samples/docker-java

2. Setup Environments

This section describes the relevant steps for both attendees and instructors to setup the environments. Please follow the parts, that are appropriate for you.

2.1. Instructor

The instructor setup is designed to make the lab most reliable even with bad Internet connections. Most, if not all, of the software can be directly downloaded from the instructor’s machine. The machine is setup as *Docker Host* and also runs a *Docker Registry* and Nexus container.

Follow all the instructor setup instructions[1] at least a day before the lab. Make sure there is a decent Internet connection available.

2.2. Attendees (From Instructor’s Machine)

This lab is designed for a BYOL (Brying Your Own Laptop) style hands-on-lab. This section provide instructions to setup an attendee environment from an instructor’s machine.

The lab contents are at https://github.com/javaee-samples/docker-java/.

Hardware

1. CPU

[1] https://github.com/javaee-samples/docker-java/blob/master/instructor/readme.adoc

a. Mac: X64 (i5 or superior)

b. Linux / Windows: x64 (i5 and comparable)

2. Memory

a. At least 4GB, preferred 8 GB

Software

1. Operating System

a. Mac OS X (10.8 or later), Windows 7 (SP1), Fedora (21 or later)

2. Java: Oracle JDK 8u45[2]

3. Webbrowser

a. Chrome[3]

b. Firefox[4]

A Word About Licenses

This tutorial only uses software which is open source or at least free to use in development and/or education. Please refer to the individual products/tools used in this tutorial.

Configure Instructor Host

All downloads and relevant infrastructure is setup on instructor's machine. Configure the IP address of instructor's machine into the resolver configuration of your operating system.

Edit the `/etc/resolv.conf` (Mac OS / Linux)

```
nameserver  <INSTRUCTOR IP>
```

On Windows, configure Domain Suffixes or DNS Suffixes as explained at http://www.pc-freak.net/blog/configure-equivalent-linux-etcresolvconf-search-domaincom-ms-windows-dns-suffixes/.

[2] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[3] https://www.google.com/chrome/browser/desktop/
[4] http://www.getfirefox.com

Git Client

Install Git Client as explained at: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Windows client is available at http://classroom.example.com:8082/downloads/Git-1.9.5-preview20150319.exe.

Maven

1. Download Apache Maven from http://classroom.example.com:8082/downloads/apache-maven-3.3.3-bin.zip
2. Unzip to a folder of your choice and add the folder to your PATH environment variable.

VirtualBox

Docker currently runs natively on Linux, but you can use VirtualBox to run Docker in a virtual machine on your box, and get the best of both worlds. This is why Virtualbox is a requirement to have on your machine. Get the latest downloads from the instructur machine:

Downloads are available from Mac[5], Linux[6], Windows[7].

Linux Users

1. Have your kernel updated
2. Users should have the GNU compiler, build and header files for your current Linux kernel
3. Create a `/usr/src/linux` link to the current kernel source

Vagrant

1. Download Vagrant for Mac[8], Windows[9], Linux[10] or Debian[11] and install.

[5] http://classroom.example.com:8082/downloads/VirtualBox-4.3.26-98988-OSX.dmg
[6] http://classroom.example.com:8082/downloads/VirtualBox-4.3.26-98988-Linux_amd64.run
[7] http://classroom.example.com:8082/downloads/VirtualBox-4.3.26-98988-Win.exe
[8] http://classroom.example.com:8082/downloads/vagrant_1.7.2.dmg
[9] http://classroom.example.com:8082/downloads/vagrant_1.7.2.msi
[10] http://classroom.example.com:8082/downloads/vagrant_1.7.2_x86_64.rpm

Docker Machine

Download your binary from http://classroom.example.com:8082/downloads/

```
# MacOS
curl -L  http://classroom.example.com:8082/downloads/docker-
machine_darwin-amd64 > /usr/local/bin/docker-machine
chmod +x /usr/local/bin/docker-machine

# Linux
curl -L  http://classroom.example.com:8082/downloads/docker-machine_linux-
amd64 > /usr/local/bin/docker-machine
chmod +x /usr/local/bin/docker-machine

#Windows
curl http://classroom.example.com:8082/downloads/docker-machine_windows-
amd64.exe
```

On Windows copy the script into `C:\docker` directory and rename to: `docker-machine.exe`. Add `C:\docker` to your `PATH` variable.

Docker Client

Download your binary from http://classroom.example.com:8082/downloads/

```
# MacOS
curl -L  http://classroom.example.com:8082/downloads/docker-latest-mac > /
usr/local/bin/docker
chmod +x /usr/local/bin/docker

# Linux (other distros)
curl -L  http://classroom.example.com:8082/downloads/docker-latest-linux
 > /usr/local/bin/docker
chmod +x /usr/local/bin/docker

#Windows
curl http://classroom.example.com:8082/docker-1.6.0.exe
```

On Windows rename the file to `C:\docker\docker.exe`.

[11] http://classroom.example.com:8082/downloads/vagrant_1.7.2_x86_64.deb

Create Lab Docker Host

1. Create the Docker Host to be used in the lab:

```
docker-machine create --driver=virtualbox --engine-opt
 dns=<INSTRUCTOR IP> --virtualbox-boot2docker-url=http://
classroom.example.com:8082/downloads/boot2docker.iso --engine-insecure-
registry=classroom.example.com:5000 lab
eval "$(docker-machine env lab)"
```

   Substitute `<INSTRUCTOR_IP>` with the IP address of the instructor's machine.

2. To make it easier to access the containers, we add an entry into the host mapping table of your operating system. Add a host entry for this Docker Host running on your machine. Find out the IP address of your machine:

```
docker-machine ip lab
```

   Edit `/etc/hosts` (Mac OS or Linux) or `C:\Windows\System32\drivers\etc\hosts` (Windows) and add:

```
<OUTPUT OF DOCKER MACHINE COMMAND>  dockerhost
```

Kubernetes

1. Download Kubernetes (0.18.1) from http://classroom.example.com:8082/downloads/kubernetes.tar.gz
2. Install it by clicking on the archive.

WildFly

1. Download WildFly 8.2 from http://classroom.example.com:8082/downloads/wildfly-8.2.0.Final.zip
2. Install it by clicking on the archive.

JBoss Developer Studio 9 - Beta 2

To install JBoss Developer Studio stand-alone, complete the following steps:

1. Download http://classroom.example.com:8082/downloads/jboss-devstudio-9.0.0.Beta2-v20150609-1026-B3346-installer-standalone.jar

2. Start the installer as:

```
java -jar jboss-devstudio-9.0.0.Beta2-v20150609-1026-B3346-installer-standalone.jar
```

Follow the on-screen instructions to complete the installation process.

2.3. Attendees (From Internet)

1. Chrome or Firefox
2. Oracle JDK 8 u45[12]
3. Git client[13]
4. Maven 3.3.3[14]
5. Latest Virtual Box[15]
6. Vagrant[16]
7. Docker
   a. Docker Machine[17]
   b. Docker Client
      i. Mac: `curl https://get.docker.com/builds/Darwin/x86_64/docker-latest > /usr/local/bin/docker`
      ii. Windows: http://test.docker.com.s3.amazonaws.com/builds/Windows/x86_64/docker-1.6.0.exe
      iii. Linux: `apt-get install docker.io`
8. Kubernetes 0.18.1[18]
9. JBoss
   a. WildFly 8.2[19]
   b. JBoss Developer Studio 9 Nightly[20]

[12] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[13] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[14] https://maven.apache.org/download.cgi
[15] https://www.virtualbox.org/
[16] https://www.vagrantup.com/downloads.html
[17] https://docs.docker.com/machine/#installation
[18] https://github.com/GoogleCloudPlatform/kubernetes/releases/download/v0.18.1/kubernetes.tar.gz
[19] http://download.jboss.org/wildfly/8.2.0.Final/wildfly-8.2.0.Final.zip
[20] https://devstudio.redhat.com/9.0/snapshots/builds/devstudio.product_master/latest/installer/

3. Docker Basics

> Docker is a platform for developers and sysadmins to develop, ship, and run applications. Docker lets you quickly assemble applications from components and eliminates the friction that can come when shipping code. Docker lets you get your code tested and deployed into production as fast as possible.
>
> — docs.docker.com/

Docker simplifies software delivery by making it easy to build and share images that contain your application's entire environment, or *application operating system*.

What does it mean by an application operating system ?

Your application typically require a specific version of operating system, application server, JDK, database server, may require to tune the configuration files, and similarly multiple other dependencies. The application may need binding to specific ports and certain amount of memory. The components and configuration together required to run your application is what is referred to as application operating system.

You can certainly provide an installation script that will download and install these components. Docker simplifies this process by allowing to create an image that contains your application and infrastructure together, managed as one component. These images are then used to create Docker containers which run on the container virtualization platform, provided by Docker.

What can a Java Developer use Docker for?

1. **Faster delivery of your applications**: Docker helps you with the development lifecycle. Docker allows you to develop on local containers that contain your applications and services. It can then integrate into a continuous integration and deployment workflow.
   For example, you write code locally and share the development stack via Docker with colleagues. When everybody is ready, you push the code and the stack you all are developing onto a test environment and execute any required tests.

   From the testing environment, you can then push the Docker images into production and deploy your code.
2. **Deploying and scaling more easily**: Docker's container-based platform allows for portable workloads. Docker containers can run on a developer's local host, on physical or virtual machines in a data center, or in the Cloud.

Docker's portability and lightweight nature also make dynamically managing workloads easy. You can use Docker to quickly scale up or tear down applications and services. Docker is so fast, that scaling can be near real time.

How is it different from VM?

Docker is an open source container virtualization platform.

Docker has three main components:

1. *Images* are **build component** of Docker and a read-only template of application operating system.
2. *Containers* are **run component** of Docker, and created from, images.Containers can be run, started, stopped, moved, and deleted.
3. Images are stored, shared, and managed in a *registry*, the **distribution component** of Docker. The publically available registry is known as Docker Hub.

In order for these three components to work together, there is **Docker Daemon** that runs on a host machine and does the heavy lifting of building, running, and distributing Docker containers. In addition, there is **Client** that is a Docker binary which accepts commands from the user and communicates back and forth with the daemon.

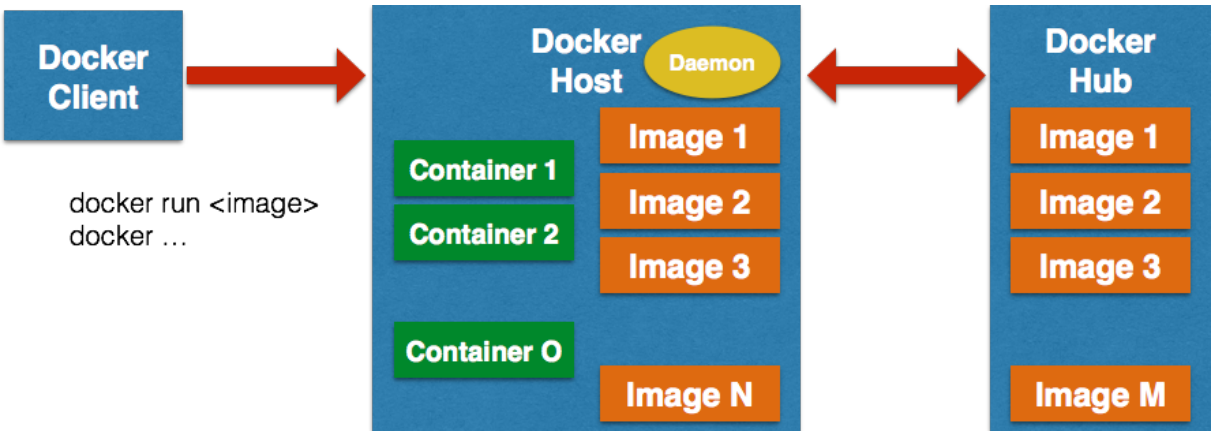


Figure 1. Docker architecture

Client communicates with Daemon, either co-located on the same host, or on a different host. It requests the Daemon to pull an image from the repository using `pull` command. The Daemon then downloads the image from Docker Hub, or whatever registry is configured. Multiple images can be downloaded from the registry and installed on Daemon host. Images are run using `run` command to create containers on demand.

How does a Docker Image work?

We've already seen that Docker images are read-only templates from which Docker containers are launched. Each image consists of a series of layers. Docker makes use of union file systems to combine these layers into a single image. Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.

One of the reasons Docker is so lightweight is because of these layers. When you change a Docker image—for example, update an application to a new version— a new layer gets built. Thus, rather than replacing the whole image or entirely rebuilding, as you may do with a virtual machine, only that layer is added or updated. Now you don't need to distribute a whole new image, just the update, making distributing Docker images faster and simpler.

Every image starts from a base image, for example `ubuntu`, a base Ubuntu image, or `fedora`, a base Fedora image. You can also use images of your own as the basis for a new image, for example if you have a base Apache image you could use this as the base of all your web application images.

> By default, Docker obtains these base images from Docker Hub.

Docker images are then built from these base images using a simple, descriptive set of steps we call instructions. Each instruction creates a new layer in our image. Instructions include actions like:

1. Run a command.
2. Add a file or directory.
3. Create an environment variable.
4. What process to run when launching a container from this image.

These instructions are stored in a file called a Dockerfile. Docker reads this Dockerfile when you request a build of an image, executes the instructions, and returns a final image.

How does a Container work?

A container consists of an operating system, user-added files, and meta-data. As we've seen, each container is built from an image. That image tells Docker what the container

holds, what process to run when the container is launched, and a variety of other configuration data. The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image (using a union file system as we saw earlier) in which your application can then run.

3.1. Docker Machine

Machine makes it really easy to create Docker hosts on your computer, on cloud providers and inside your own data center. It creates servers, installs Docker on them, then configures the Docker client to talk to them.

Once your Docker host has been created, it then has a number of commands for managing them:

1. Starting, stopping, restarting
2. Upgrading Docker
3. Configuring the Docker client to talk to your host

You used Docker Machine already during the attendee setup. We won't need it too much further on. But if you need to create hosts, it's a very handy tool to know about. From now on we're mostly going to use the docker client.

Find out more about the details at the Official Docker Machine Website[21]

Check if docker machine is working using the following command:

```
docker-machine -v
```

It shows the following output:

```
docker-machine version 0.3.0-rc3 (a5fd8cf)
```

3.2. Docker Client

The client communicates with the demon process on your host and let's you work with images and containers.

Check if your client is working using the following command:

[21] https://docs.docker.com/machine/

```
docker -v
```

It shows the following output:

```
Docker version 1.6.2, build 7c8fca2
```

The most important options you'll be using frequently are:

1. `run` - runs a container
2. `ps` - lists containers
3. `stop` - stops a container

Get a full list of available commands with

```
docker
```

3.3. Verify Docker Configuration

Check if your Docker Host is running:

```
docker-machine ls
```

You should see the output similar to:

```
NAME         ACTIVE   DRIVER       STATE     URL
 SWARM
lab                   virtualbox   Running   tcp://192.168.99.101:2376
```

If the machine state is stopped, start it with:

```
docker-machine start lab
```

After it is started you can find out IP address of your host with:

```
docker-machine ip lab
```

We already did this during the setup document, remember? So, this is a good chance to check, if you already added this IP to your hosts file.

Type:

```
ping dockerhost
```

and see if this resolves to the IP address that the docker-machine command printed out. You should see an output as:

```
> ping dockerhost
PING dockerhost (192.168.99.101): 56 data bytes
64 bytes from 192.168.99.101: icmp_seq=0 ttl=64 time=0.394 ms
64 bytes from 192.168.99.101: icmp_seq=1 ttl=64 time=0.387 ms
```

If it does, you're ready to start over with the lab. If it does not, make sure you've followed the steps to configure your host[22].

4. Run Container

The first step in running any application on Docker is to run an image. There are plenty of images available from the official Docker registry (aka Docker Hub[23]). To run any of them, you just have to ask the Docker Client to run it. The client will check if the image already exists on Docker Host. If it exists then it'll run it, otherwise the host will download the image and then run it.

4.1. Pull Image

Let's first check, if there are any images already available:

```
docker images
```

At first, this list is empty. Now, let's get a plain `jboss/wildfly` image from the instructor's registry:

```
docker pull dockerhost:5000/wildfly
```

[22] https://github.com/javaee-samples/docker-java/tree/master/attendees#configure-host
[23] https://hub.docker.com

By default, docker images are retrieved from Docker Hub[24]. This lab is pre-congfigured to run such that everything can be pulled from instructor's machine.

You can see, that Docker is downloading the image with it's different layers.

In a traditional Linux boot, the Kernel first mounts the root File System as read-only, checks its integrity, and then switches the whole rootfs volume to read-write mode. When Docker mounts the rootfs, it starts read-only, as in a traditional Linux boot, but then, instead of changing the file system to read-write mode, it takes advantage of a union mount to add a read-write file system over the read-only file system. In fact there may be multiple read-only file systems stacked on top of each other. Consider each one of these file systems as a layer.

At first, the top read-write layer has nothing in it, but any time a process creates a file, this happens in the top layer. And if something needs to update an existing file in a lower layer, then the file gets copied to the upper layer and changes go into the copy. The version of the file on the lower layer cannot be seen by the applications anymore, but it is there, unchanged.

We call the union of the read-write layer and all the read-only layers a *union file system*.

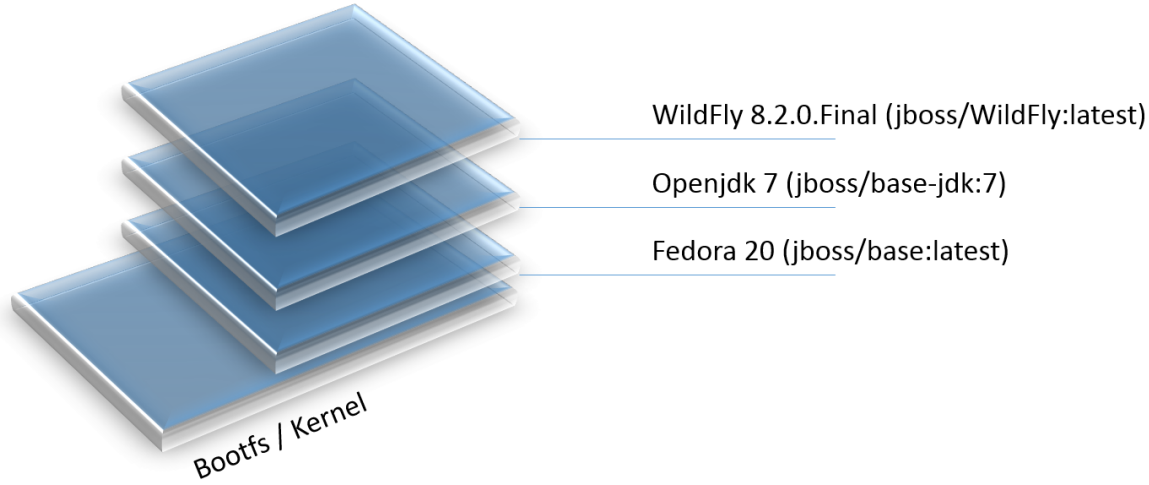


Figure 2. Docker Layers

[24] https://hub.docker.com/

In our particular case, the jboss/wildfly[25] image extends the jboss/base-jdk:7[26] image which adds the OpenJDK distribution on top of the jboss/base[27] image. The base image is used for all JBoss community images. It provides a base layer that includes:

1. A jboss user (uid/gid 1000) with home directory set to `/opt/jboss`
2. A few tools that may be useful when extending the image or installing software, like unzip.

The ``jboss/base-jdk:7" image adds:

1. Latest OpenJDK distribution
2. Adds a JAVA_HOME environment variable

When the download is done, you can list the images again and will see the following:

```
docker images

REPOSITORY               TAG      IMAGE ID       CREATED        VIRTUAL SIZE
dockerhost:5000/wildfly  latest   2ac466861ca1   10 weeks ago   951.3 MB
```

4.2. Run Container

Interactive Container

Run WildFly container in an interactive mode.

```
docker run -it dockerhost:5000/wildfly
```

This will show the output as:

```
> docker run -it 192.168.99.100:5000/wildfly
=========================================================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /opt/jboss/wildfly
```

[25] https://github.com/jboss-dockerfiles/wildfly/blob/master/Dockerfile
[26] https://github.com/jboss-dockerfiles/base/blob/master/Dockerfile
[27] https://github.com/jboss-dockerfiles/base/blob/master/Dockerfile

```
  JAVA: /usr/lib/jvm/java/bin/java

  JAVA_OPTS:  -server -Xms64m -Xmx512m -XX:MaxPermSize=256m -
Djava.net.preferIPv4Stack=true -
Djboss.modules.system.pkgs=org.jboss.byteman -Djava.awt.headless=true

=========================================================================

17:58:58,353 INFO  [org.jboss.modules] (main) JBoss Modules version
 1.3.3.Final
17:58:58,891 INFO  [org.jboss.msc] (main) JBoss MSC version 1.2.2.Final
17:58:59,056 INFO  [org.jboss.as] (MSC service thread 1-2) JBAS015899:
 WildFly 8.2.0.Final "Tweek" starting

. . .

17:59:03,211 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015961:
 Http management interface listening on http://127.0.0.1:9990/management
17:59:03,212 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951:
 Admin console listening on http://127.0.0.1:9990
17:59:03,213 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874:
 WildFly 8.2.0.Final "Tweek" started in 5310ms - Started 184 of 234
 services (82 services are lazy, passive or on-demand)
```

This shows that the server started correctly, congratulations!

By default, Docker runs in the foreground. `-i` allows to interact with the STDIN and `-t` attach a TTY to the process. Switches can be combined together and used as `-it`.

Hit Ctrl+C to stop the container.

Detached Container

Restart the container in detached mode:

```
> docker run -d 192.168.99.100:5000/wildfly
972f51cc8422eec0a7ea9a804a55a2827b5537c00a6bfd45f8646cb764bc002a
```

`-d` runs the container in detached mode.

The output is the unique id assigned to the container. Check the logs as:

```
> docker logs
 972f51cc8422eec0a7ea9a804a55a2827b5537c00a6bfd45f8646cb764bc002a
```

```
=========================================================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /opt/jboss/wildfly

. . .
```

We can check it by issuing the `docker ps` command which retrieves the images process which are running and the ports engaged by the process:

```
> docker ps
CONTAINER ID        IMAGE                                      COMMAND
         CREATED             STATUS              PORTS
 NAMES
0bc123a8ece0        192.168.99.100:5000/wildfly:latest     "/opt/jboss/
wildfly/   4 seconds ago       Up 4 seconds        8080/tcp
  tender_wozniak
```

Also try `docker ps -a` to see all the containers on this machine.

4.3. Run Container with Default Port

Startup log of the server shows that the server is located in the `/opt/jboss/wildfly`. It also shows that the public interfaces are bound to the `0.0.0.0` address while the admin interfaces are bound just to `localhost`. This information will be useful to learn how to customize the server.

`docker-machine ip <machine-name>` gives us the Docker Host IP address and this was already added to the hosts file. So, we can give it another try by accessing: http://dockerhost:8080. However, this will not work either.

If you want containers to accept incoming connections, you will need to provide special options when invoking `docker run`. The container, we just started, can't be accessed by our browser. We need to stop it again and restart with different options.

```
docker stop 0bc123a8ece0
```

Restart the container as:

```
> docker ps
```

```
CONTAINER ID        IMAGE                                          COMMAND
        CREATED             STATUS              PORTS
 NAMES
4545ced66242        192.168.99.100:5000/wildfly:latest     "/opt/jboss/
wildfly/    3 seconds ago       Up 3 seconds        0.0.0.0:32768->8080/tcp
   suspicious_wozniak
```

`-P` flag map any network ports inside the image it to a random high port from the range 49153 to 65535 on Docker host.

The port mapping is shown in the `PORTS` column. Access the WildFly server at http://dockerhost:32768:8080. Make sure to use the correct port number as shown in your case.

4.4. Run Container with Specified Port

Lets stop the previously running container as:

```
docker stop 4545ced66242
```

Restart the container as:

```
docker run -it -p 8080:8080 dockerhost:5000/wildfly
```

The format is `-p hostPort:containerPort`. This option maps container ports to host ports and allows other containers on our host to access them.

Docker Port Mapping

> Port exposure and mapping are the keys to successful work with Docker. See more about networking on the Docker website Advanced Networking[28]

Now we're ready to test http://dockerhost:8080 again. This works with the exposed port, as expected.

[28] https://docs.docker.com/articles/networking/

Figure 3. Welcome WildFly

4.5. Enabling WildFly Administration

Default WildFly image exposes only port 8080 and thus is not available for administration using either the CLI or Admin Console.

Default Port Mapping

The following command will override the default command in Docker file, explicitly starting WildFly, and binding application and management port to all network interfaces.

```
docker run -P -d dockerhost:5000/wildfly /opt/jboss/wildfly/bin/
standalone.sh -b 0.0.0.0 -bmanagement 0.0.0.0
```

Accessing WildFly Administration Console require a user in administration realm. A pre-created image, with appropriate username/password credentials, is used to start WildFly as:

```
docker run -P -d dockerhost:5000/wildfly-management
```

`-P` flag map any network ports inside the image it to a random high port from the range 49153 to 65535 on Docker host.

Look at the exposed ports as:

```
 docker ps
CONTAINER ID        IMAGE
 COMMAND                  CREATED             STATUS              PORTS
                                        NAMES
6f610b310a46        dockerhost:5000/wildfly-management:latest   "/bin/sh -
c '/opt/jb   6 seconds ago       Up 6 seconds        0.0.0.0:32769->8080/
tcp, 0.0.0.0:32770->9990/tcp   determined_darwin
```

Look for the host port that is mapped in the container, `32770` in this case. Access the admin console at http://dockerhost:32770.

The username/password credentials are:

Field	Value
Username	admin
Password	docker#admin

Additional Ways To Find Port Mapping

The exact mapped port can also be found as:

1. Using `docker inspect`:

   ```
   docker inspect --format='{{(index (index .NetworkSettings.Ports "9990/
   tcp") 0).HostPort}}' 6f610b310a46
   ```

2. Using `docker port`:

   ```
   docker port 6f610b310a46
   ```

   to see the output as:

```
0.0.0.0:32769->8080/tcp
0.0.0.0:32770->9990/tcp
```

Fixed Port Mapping

This management image can also be started with a pre-defined port mapping as:

```
docker run -p 8080:8080 -p 9990:9990 -d 192.168.99.100:5000/wildfly-
management
```

In this case, Docker port mapping will be shown as:

```
8080/tcp -> 0.0.0.0:8080
9990/tcp -> 0.0.0.0:9990
```

4.6. Stop and Remove Container

Stop Container

1. Stop a specific container:

```
docker stop 0bc123a8ece0
```

2. Stop all the running containers

```
docker rm $(docker stop $(docker ps -q))
```

3. Stop only the exited containers

```
docker ps -a -f "exited=-1"
```

Remove Container

1. Remove a specific container:

```
docker rm 0bc123a8ece0
```

2. Containers meeting a regular expression

```
docker ps -a | grep wildfly | awk '{print $1}' | xargs docker rm
```

3. All running containers, without any criteria

```
docker rm $(docker ps -aq)
```

5. Deploy Java EE 7 Application (Pre-Built WAR)

Java EE 7 Hands-on Lab[29] has been delivered all around the world and is a pretty standard application that shows design patterns and anti-patterns for a typical Java EE 7 application.

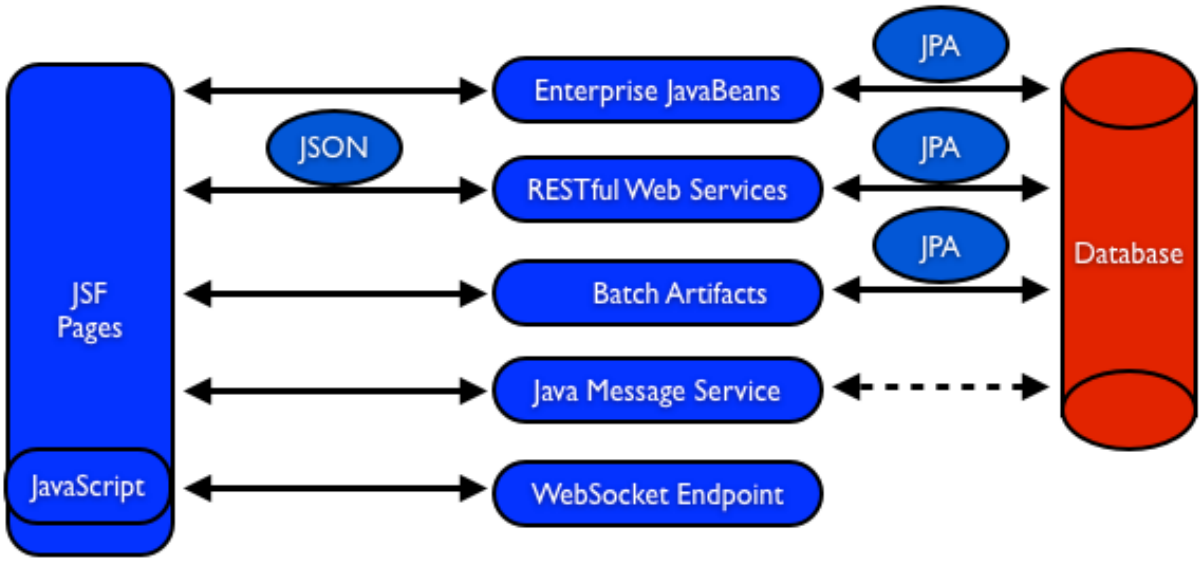


Figure 4. Java EE 7 Application Architecture

Pull the Docker image that contains WildFly and pre-built Java EE 7 application WAR file as shown:

```
docker pull classroom.example.com:5000/javaee7-hol
```

The javaee7-hol Dockerfile[30] is based on `jboss/wildfly` and adds the movieplex7 application as war file.

Run it as:

[29] https://github.com/javaee-samples/javaee7-hol
[30] https://github.com/arun-gupta/docker-images/blob/master/javaee7-hol/Dockerfile

```
docker run -it -p 8080:8080 classroom.example.com:5000/javaee7-hol
```

See the application in action at http://dockerhost:8080/movieplex7/.

This uses an in-memory database with WildFly application server as shown in the image:

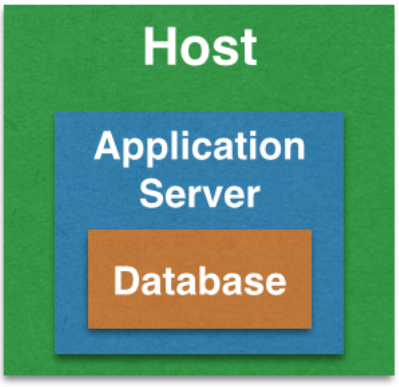


Figure 5. In-memory Database

Only two changes are required to the standard `jboss/wildfly` image:

1. Start WildFly in full platform:

```
CMD ["/opt/jboss/wildfly/bin/standalone.sh", "-c", "standalone-
full.xml", "-b", "0.0.0.0"]
```

2. WAR file is copied to `standalone/deployments` directory as:

```
RUN curl -L https://github.com/javaee-samples/javaee7-hol/blob/jrebel/
solution/movieplex7-1.0-SNAPSHOT.war?raw=true -o /opt/jboss/wildfly/
standalone/deployments/movieplex7-1.0-SNAPSHOT.war
```

6. Deploy Java EE 7 Application (Container Linking)

Section 5, “Deploy Java EE 7 Application (Pre-Built WAR)” explained how to use an in-memory database with the application server. This gets you started rather quickly but becomes a bottleneck soon as the database is only in-memory. This means that any changes made to your schema and data are lost after the application server shuts down. In this case, you need to use a database server that resides outside the application server. For example, MySQL as the database server and WildFly as the application server.

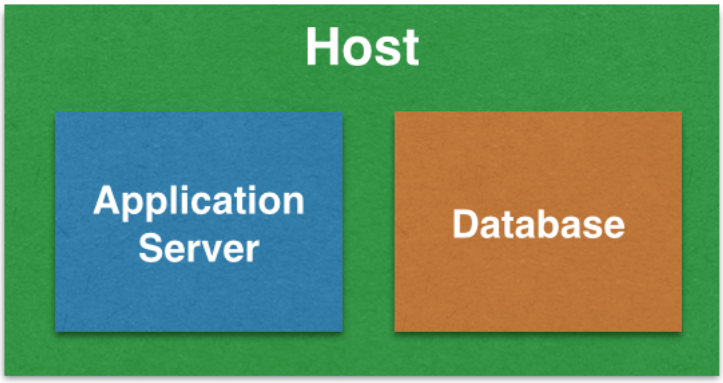


Figure 6. Two Containers On Same Docker Host

This section will show how Docker Container Linking[31] can be used to connect to a service running inside a Docker container via a network port.

1. Start MySQL server as:

```
docker run --name mysqldb -e MYSQL_USER=mysql -e MYSQL_PASSWORD=mysql -
e MYSQL_DATABASE=sample -e MYSQL_ROOT_PASSWORD=supersecret -p 3306:3306
 -d mysql
```

   `-e` define environment variables that are read by the database at startup and allow us to access the database with this user and password.

2. Start WildFly and deploy Java EE 7 application as:

```
docker run -it --name mywildfly --link mysqldb:db -p 8080:8080
 arungupta/wildfly-mysql-javaee7
```

   `--link` takes two parameters - first is name of the container we're linking to and second is the alias for the link name.

Container Linking

> Creating a link between two containers creates a conduit between a source container and a target container and securely transfer information about source container to target container.
>
> In our case, target container (WildFly) can see information about source container (MySQL). When containers are linked, information about a source container can be sent to a recipient container. This allows the recipient to see

[31] https://docs.docker.com/userguide/dockerlinks/

> selected data describing aspects of the source container. For example, IP address of MySQL server is expoed at $DB_PORT_3306_TCP_ADDR and port of MySQL server is exposed at $DB_PORT_3306_TCP_PORT. These are then used to create the JDBC resource.
>
> See more about container communication on the Docker website Linking Containers Together[32]

3. See the output as:

```
> curl http://$(docker-machine ip lab):8080/employees/resources/
employees
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
><collection><employee><id>1</id><name>Penny</name></
employee><employee><id>2</id><name>Sheldon</name></
employee><employee><id>3</id><name>Amy</name></
employee><employee><id>4</id><name>Leonard</name></
employee><employee><id>5</id><name>Bernadette</
name></employee><employee><id>6</id><name>Raj</name></
employee><employee><id>7</id><name>Howard</name></
employee><employee><id>8</id><name>Priya</name></employee></collection>
```

7. Build and Deploy Java EE 6 Application (Ticket Monster)

Ticket Monster is an example application that focuses on Java EE6 - JPA 2, CDI, EJB 3.1 and JAX-RS along with HTML5 and jQuery Mobile. It is a moderately complex application that demonstrates how to build modern web applications optimized for mobile & desktop. TicketMonster is representative of an online ticketing broker - providing access to events (e.g. concerts, shows, etc) with an online booking application.

Apart from being a demo, TicketMonster provides an already existing application structure that you can use as a starting point for your app. You could try out your use cases, test your own ideas, or, contribute improvements back to the community.

[32] https://docs.docker.com/userguide/dockerlinks/

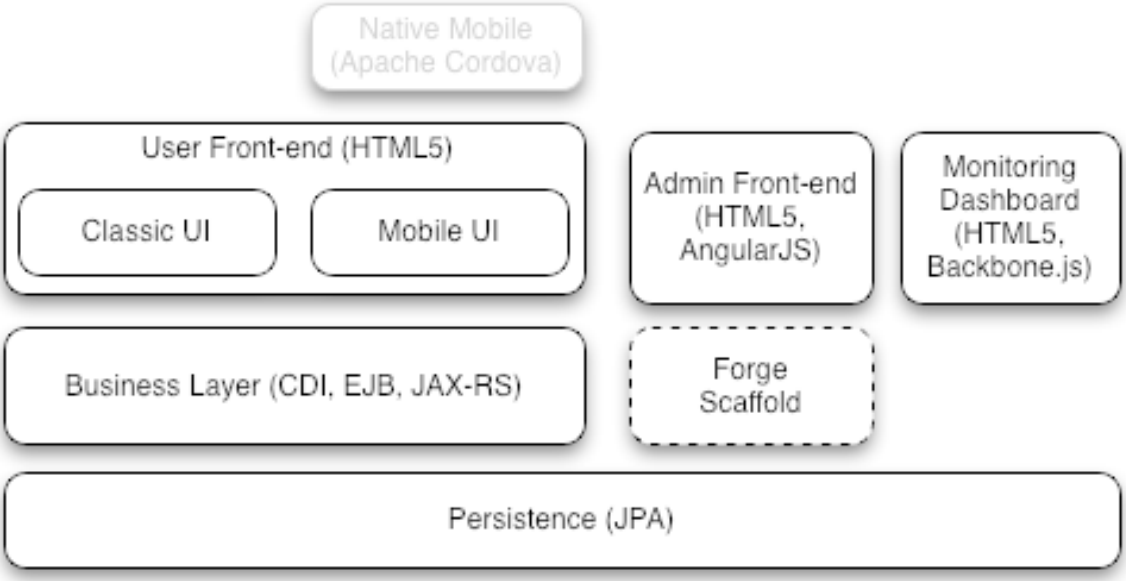


Figure 7. TicketMonster architecture

The application uses Java EE 6 services to provide business logic and persistence, utilizing technologies such as CDI, EJB 3.1 and JAX-RS, JPA 2. These services back the user-facing booking process, which is implemented using HTML5 and JavaScript, with support for mobile devices through jQuery Mobile.

The administration site is centered around CRUD use cases, so instead of writing everything manually, the business layer and UI are generated by Forge, using EJB 3.1, CDI and JAX-RS. For a better user experience, Twitter Bootstrap is used.

Monitoring sales requires staying in touch with the latest changes on the server side, so this part of the application will be developed in HTML5 and JavaScript using a polling solution.

7.1. Build Application

First thing, you're going to do is to build the application from source. Create a directory for the source and change to it:

```
mkdir docker-java/
cd docker-java/
```

And checkout the sources from the instructor's git repository.

```
git clone -b WildFly-docker-test http://
root:dockeradmin@classroom.example.com:10080/root/ticket-monster.git
```

`-b WildFly-docker-test` is a branch of Ticket Monster that contains a ``docker-test'' profile to run Arquillian Cube test. More on this later.

You're free to explore the application. Open it with with the favorite IDE of your choice. Find more background about the use-cases and how the application is designed at Ticket Monster Website[33].

Copy the Maven `lab-settings.xml` file that you have downloaded from the instructor machine and place it inside `docker-java` directory.

When you're ready, it is time to build the application. Switch to the checkout directory and run maven package.

```
cd docker-java/
mvn -s lab-settings.xml -f ticket-monster/demo/pom.xml -Ppostgresql clean
 package
```

Congratulations! You just build the applications war file. Let's deploy it!

7.2. Start Database Server

The application require an application server and a database server. This lab will use WildFly and Postgres for them respectively.

Start Postgres database as:

```
docker run --name db -d -p 5432:5432 -e POSTGRES_USER=ticketmonster -
e POSTGRES_PASSWORD=ticketmonster-docker classroom.example.com:5000/
postgres
```

This command starts a container named `db'' from the image in your instructor's registry `classroom.example.com:5000/postgres`. As this will not be present locally, it needs to be downloaded first. But you'll have a very quick connection to the instructor registry and this shouldn't take long.

The two `-e` options define environment variables which are read by the db at startup and allow us to access the database with this user and password.

[33] http://www.jboss.org/ticket-monster/whatisticketmonster/

Finally, the `-d` option tells docker to start a demon process. Which means, that the console window, you're running this command in, will be available again after it is issued. If you skip this parameter, the console will be directly showing the output from the process.

`-p` option maps container ports to host ports and allows other containers on our host to access them.

This starts the database container. It can be confirmed as:

```
> docker ps
CONTAINER ID        IMAGE
 COMMAND                  CREATED             STATUS              PORTS
                                        NAMES
047bab6a86fe        192.168.99.100:5000/postgres:latest             "/
docker-entrypoint.    42 seconds ago      Up 3 seconds        0.0.0.0:5432-
>5432/tcp                               db
```

Server logs can be viewed as:

```
docker logs -f db
```

The `-f` flag keeps refreshing the logs and pushes new events directly out to the console.

7.3. Start Application Server

Start WildFly server as:

```
docker run -d --name wildfly -p 8080:8080 --link db:db -v /Users/
youruser/tmp/deployments:/opt/jboss/wildfly/standalone/deployments/:rw
 classroom.example.com:5000/wildfly
```

Make sure to replace `/Users/youruser/tmp/deployments` to a directory on your local machine. Also, make sure this directory already exists.

This command starts a container named `` `wildfly''. `--link `` takes two parameters - first is name of the container we're linking to and second is the alias for the link name.

Container Linking

> Creating a link between two containers creates a conduit between a source container and a target container and securely transfer information about source container to target container.
>
> In our case, target container (WildFly) can see information about source container (Postgres). When containers are linked, information about a source container can be sent to a recipient container. This allows the recipient to see selected data describing aspects of the source container.
>
> See more about container communication on the Docker website Linking Containers Together[34]

The `-v` flag maps a directory from the host into the container. This will be the directory to put the deployments. `rw` ensures that the Docker container can write to it.

> Windows users, please make sure to use `-v /c/Users/` notation for drive letters.

Check logs to verify if the server has started.

```
docker logs -f wildfly
```

And access the http://dockerhost:8080 with your webbrowser to make sure the instance is up and running.

Now you're ready to deploy the application for the first time. Let's use JBoss Developer Studio for this.

7.4. Configure JBoss Developer Studio

Start JBoss Developer Studio, if not already started.

1. Create a server adapter

[34] https://docs.docker.com/userguide/dockerlinks/

Figure 8. Server adapter

2. Assign or create a WildFly 8.x runtime (changed properties are highlighted.)

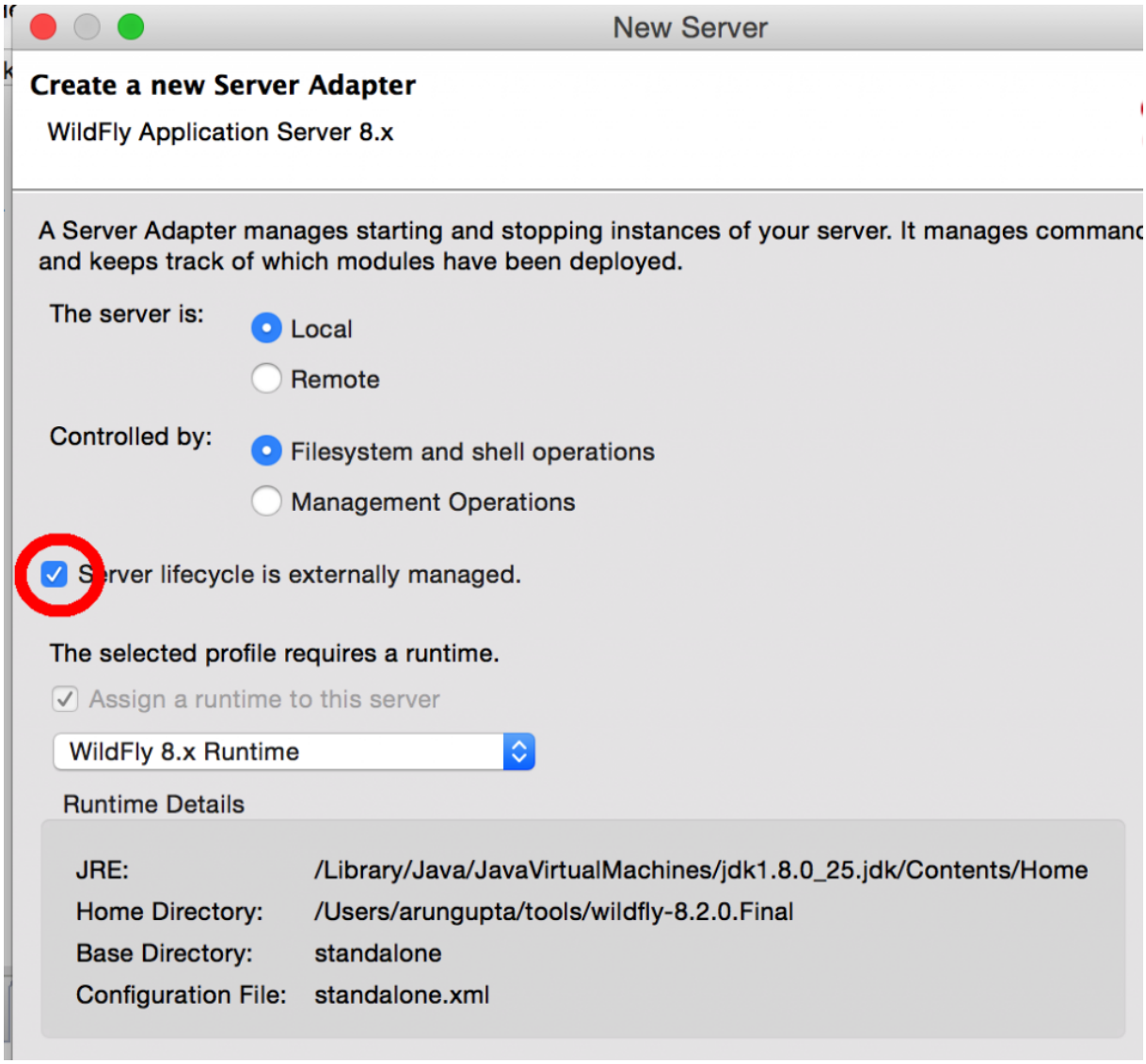


Figure 9. WildFly Runtime Properties

3. Setup server properties as shown in the following image.
   Two properties on the left are automatically propagated from the previous dialog. Additional two properties on the right side are required to disable to keep deployment scanners in sync with the server.

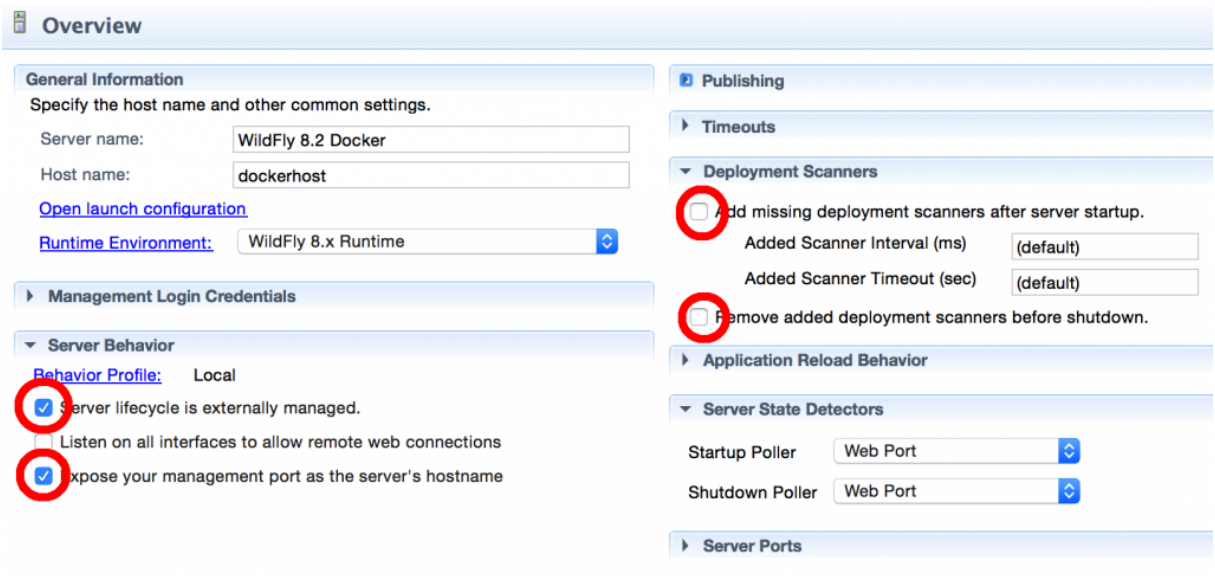


Figure 10. Server properties

4. Specify a custom deployment folder on Deployment tab of Server Editor

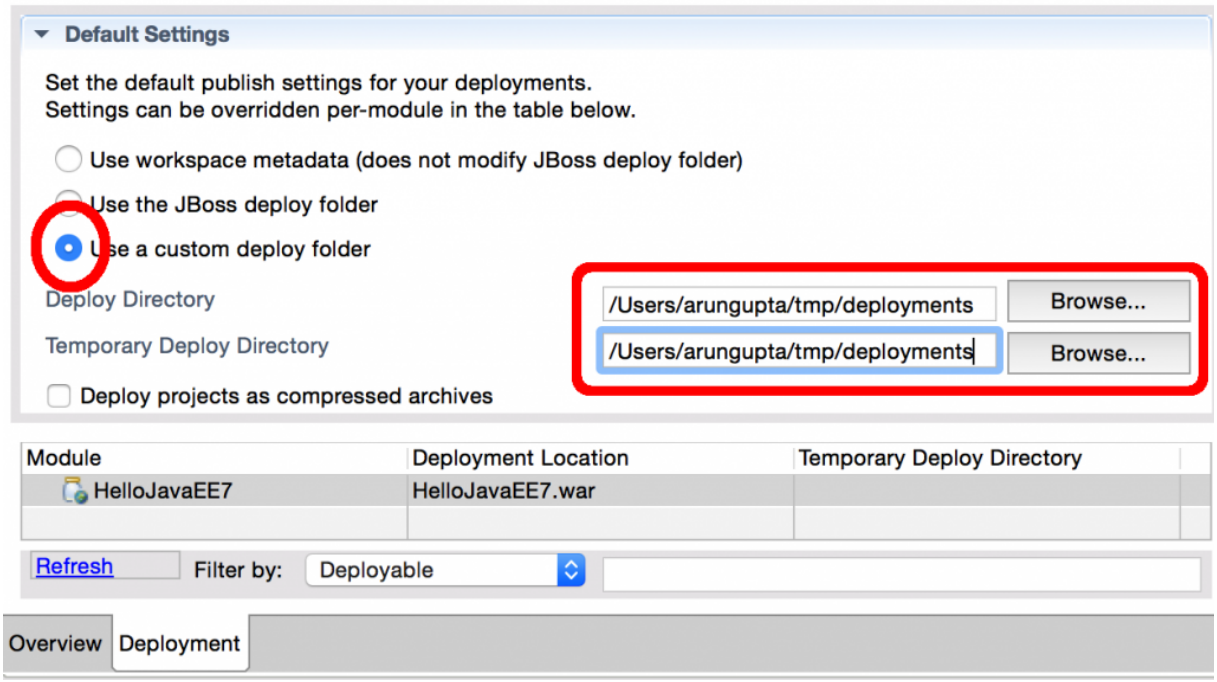


Figure 11. Server Editor

5. Right-click on the newly created server adapter and click ``Start''.

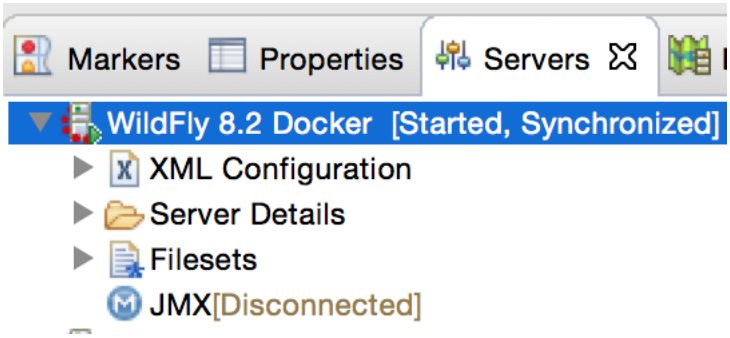


Figure 12. Start Server

7.5. Deploy Application Using Shared Volumes

Open Ticket Monster application source code. Right-click on the project, select ``Run on Server'' and chose the previously created server.

The project runs and displays the start page of Ticket Monster application.

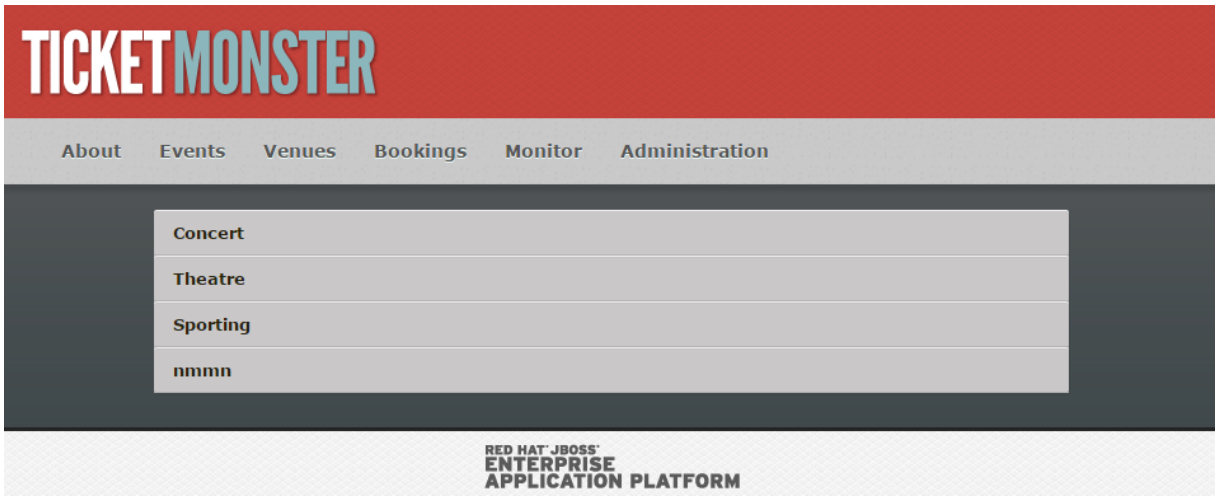


Figure 13. Start Server

Congratulations! You've just deployed your first application to WildFly running in a Docker container from JBoss Developer Studio.

Stop WildFly container when you're done.

```
docker stop wildfly
```

7.6. Deploy Application Using CLI (OPTIONAL)

The Command Line Interface (CLI) is a tool for connecting to WildFly instances to manage all tasks from command line environment. Some of the tasks that you can do using the CLI are:

1. Deploy/Undeploy web application in standalone/Domain Mode.
2. View all information about the deployed application on runtime.
3. Start/Stop/Restart Nodes in respective mode i.e. Standalone/Domain.
4. Adding/Deleting resource or subsystems to servers.

Lets use the CLI to deploy Ticket Monster to WildFly running in the container.

1. CLI needs to be locally installed and comes as part of WildFly. Download WildFly 8.2 from http://classroom.example.com:8082/downloads/wildfly-8.2.0.Final.zip. Unzip into a folder of your choice (e.g. `/Users/arungupta/tools/`). This will create `wildfly-8.2.0.Final` directory here. This folder is named $WIDLFY_HOME from here on. Make sure to add the `/Users/arungupta/tools/wildfly-8.2.0.Final/bin` to your $PATH.

```
# Windows Example
set PATH=%PATH%;%WILDFLY_HOME%/bin
```

2. Run the ``wildfly-management" image with fixed port mapping as explained in the section called “Fixed Port Mapping”.
3. Run the `jboss-cli` command and connect to the WildFly instance.

```
cd %WIDLFY_HOME%/bin
./jboss-cli.sh --controller=dockerhost:9990  -u=admin -p=docker#admin -
c
```

   This will show the output as:

```
[standalone@dockerhost:9990 /]
```

4. Deploy the application as:

```
deploy <TICKET_MONSTER_PATH>/ticket-monster.war --force
```

Now you’ve sucessfully used the CLI to remote deploy the Ticket Monster application to WildFly running as docker container.

And again, keep the container running, we're going to look into the last deployment option you have.

7.7. Deploy Application Using Web Console (OPTIONAL)

WildFly comes with a web-based administration console. It also relies on the same management APIs that we've already been using via JBoss Developer Tools and the CLI. It does provide a nice web-based way to administrate your instance and if you've already exposed the container ports, you can simply access it via the URL: http://dockerhost:9990 in your web browser.

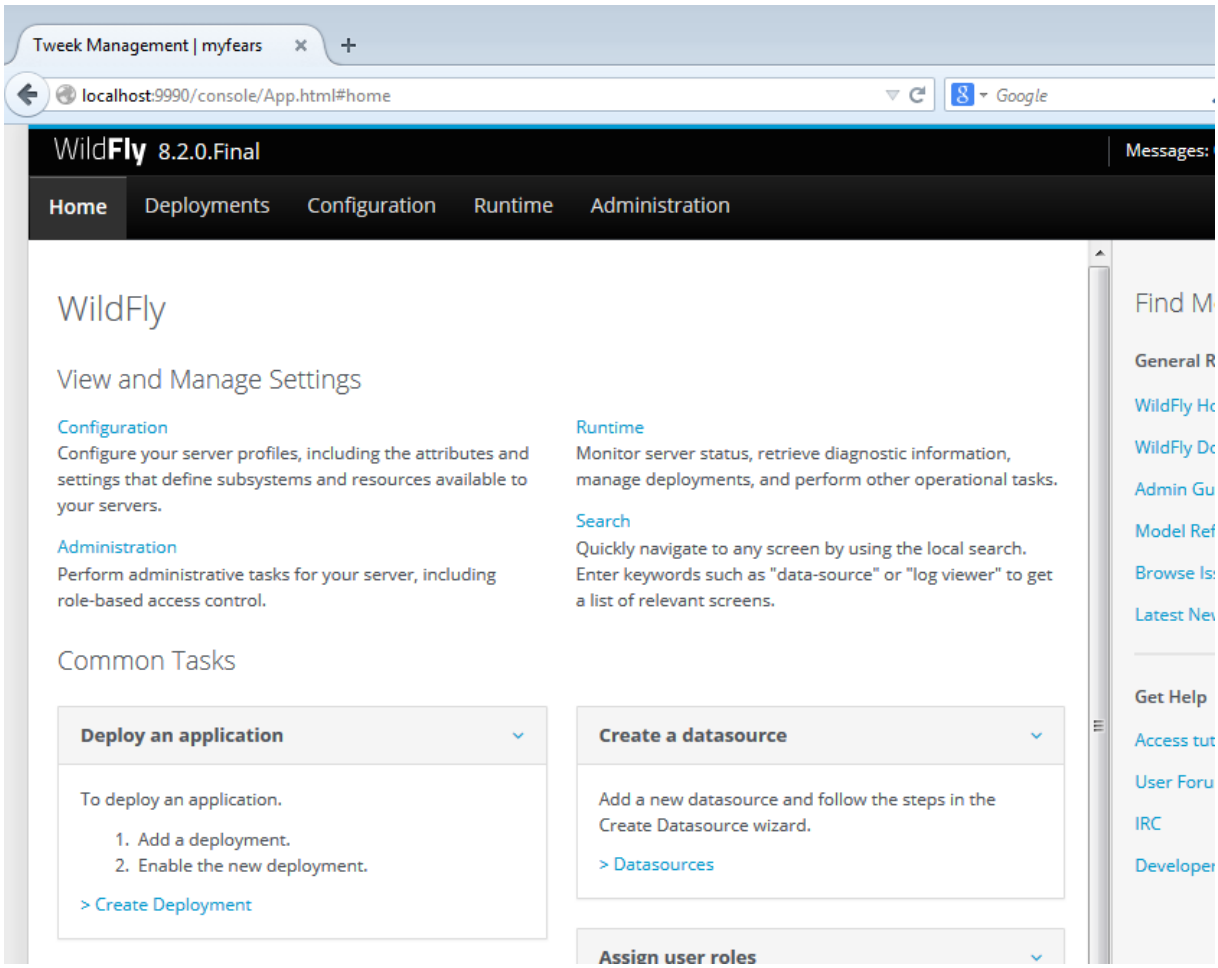


Figure 14. WildFly Web Console

Username and password credentials are shown in ???. Now navigate through the console and execute the following steps to deploy the application:

1. Go to the ``Deployments'' tab.
2. Click on ``Add'' button.

3. On `Step 1/2: Deployment Selection'' screen, select the <TICKET_MONSTER_PATH>/ticket-monster.war file on your computer and click` Next". This would be `ticket-monster/demo/target/ticket-monster.war` from Section 7.1, “Build Application”.
4. On the `Step 2/2: Verify Deployment Names'' screen, select` Enable" checkbox, and click on ``Save".

This will complete the deployment of Ticket Monster using Admin Console.

7.8. Deploy Application Using Management API (OPTIONAL)

A standalone WildFly process, process can be configured to listen for remote management requests using its ``native management interface". The CLI tool that comes with the application server uses this interface, and user can develop custom clients that use it as well. In order to use this, WildFly management interface listen IP needs to be changed from 127.0.0.1 to 0.0.0.0 which basically means, that it is not only listening on the localhost but also on all publicly assigned IP addresses.

1. Start another WildFly instance again:

```
docker run -d --name wildflymngm -p 8080:8080 -p 9990:9990 --link db:db
 classroom.example.com:5000/wildfly-management
```

   There is no mapped volume in this case but an additional port exposed. The WildFly image that is used makes it easier for you to play around with the deployment via the management API. It has a tweaked start script which changes the management interface according to the behavior described in the first sentence.

2. Create another new server adapter in JBoss Developer Studio.

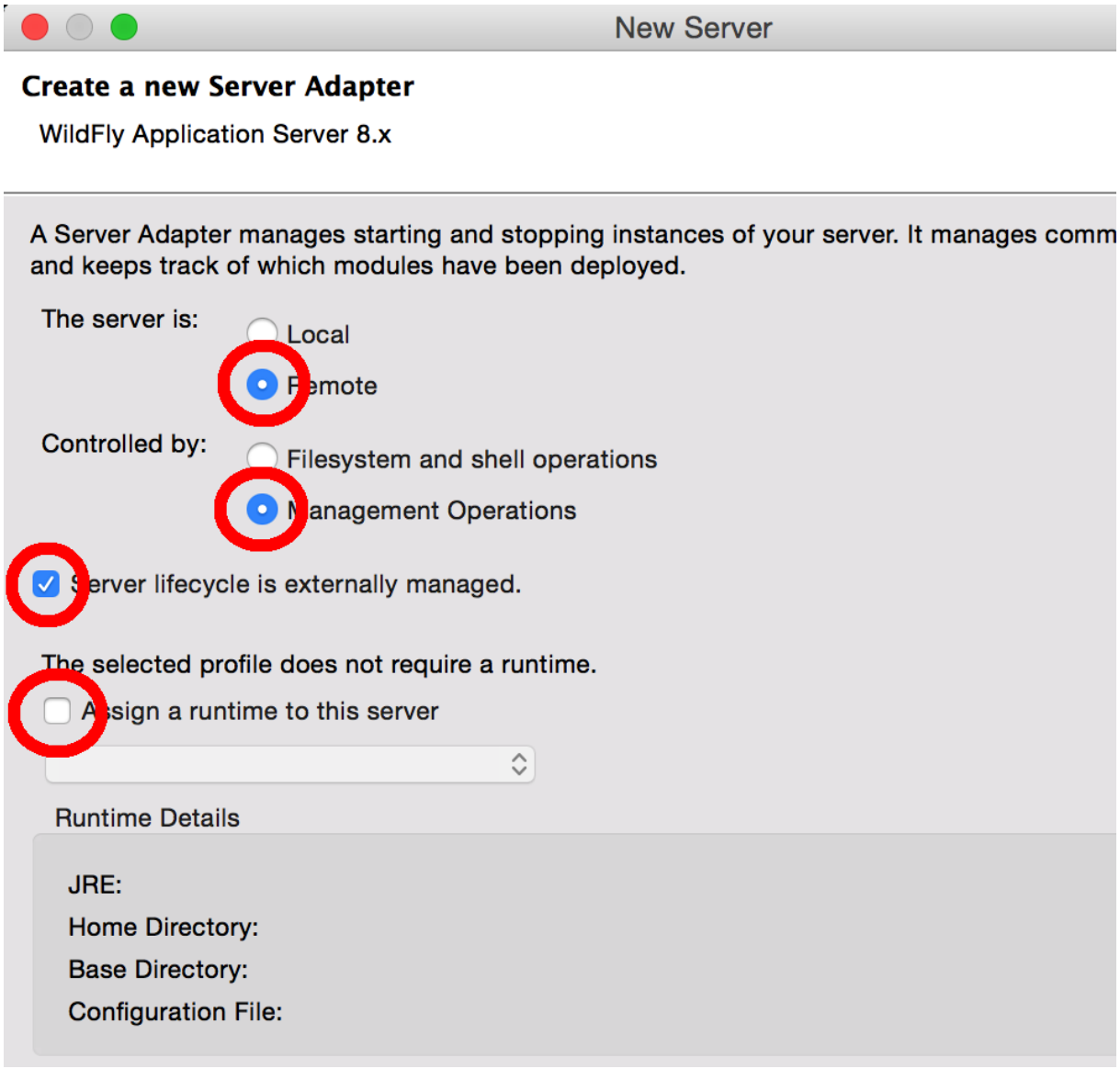


Figure 15. Create New Server Adapter

3. Keep the defaults in the adapter properties.

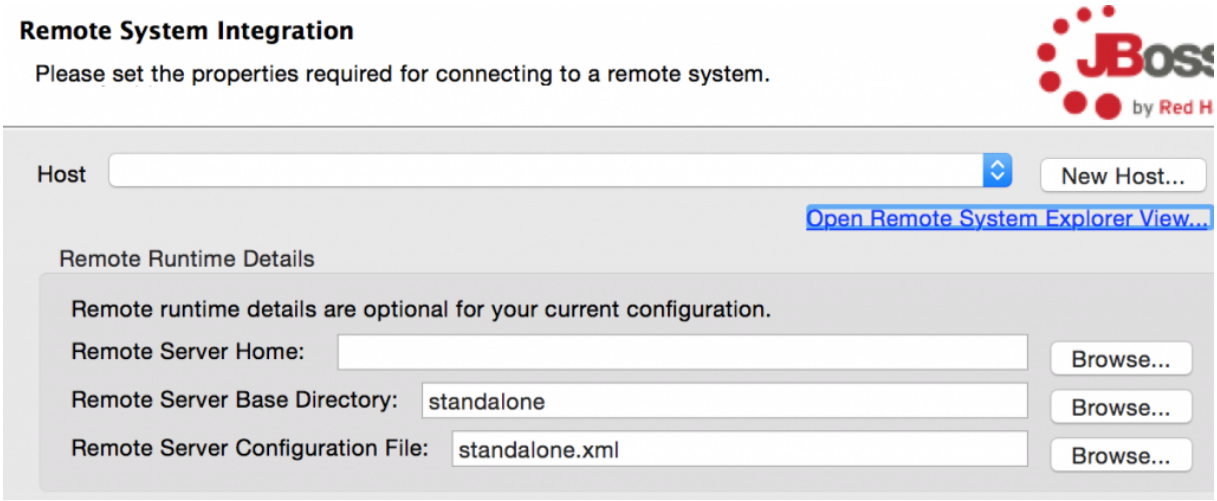


Figure 16. Adapter Properties

4. Set up server properties by specifying the admin credentials (docker#admin). Note, you need to delete the existing password and use this instead:

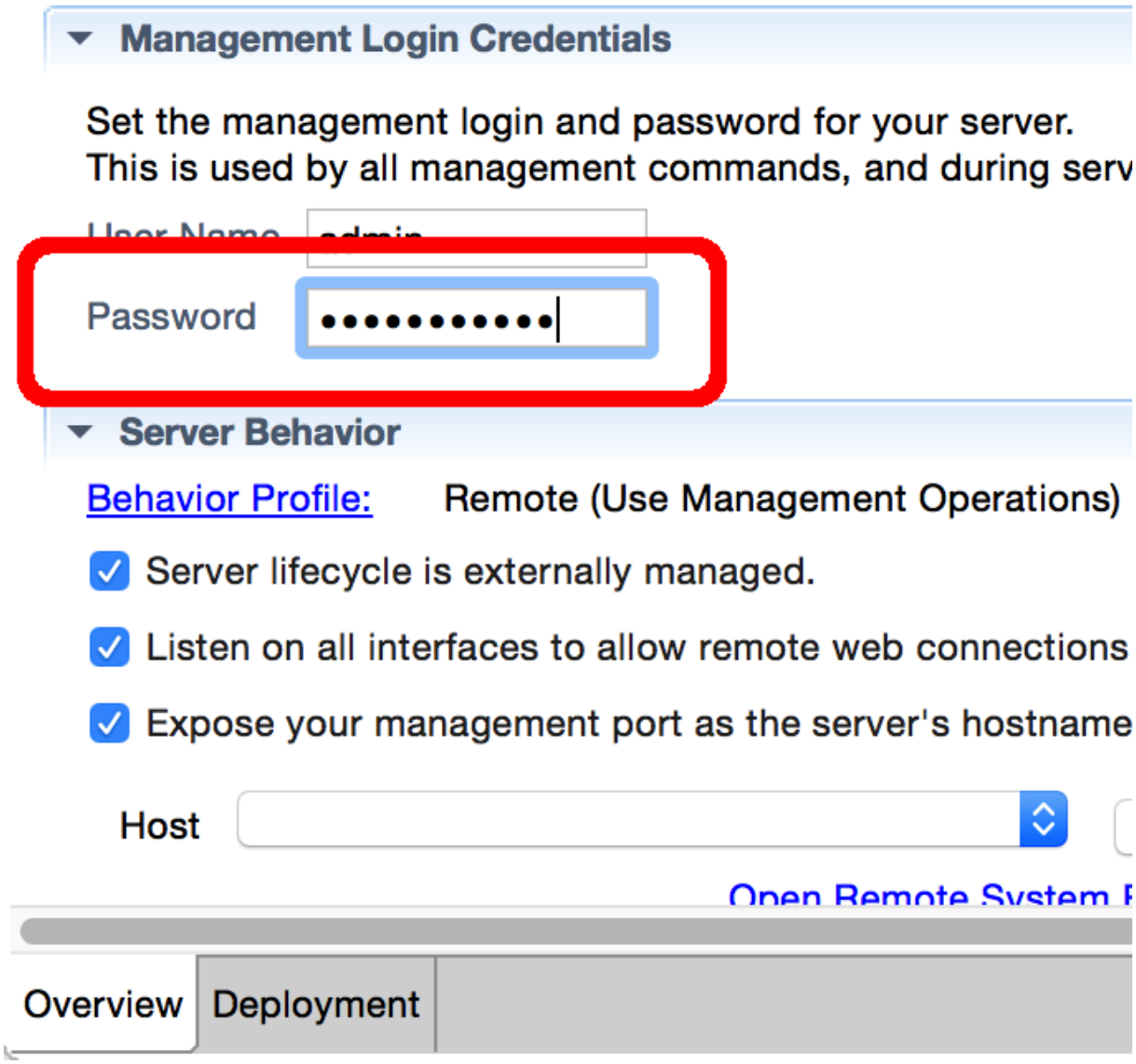


Figure 17. Management Login Credentials

5. Right-click on the newly created server adapter and click `Start''. Status quickly changes to` Started, Synchronized" as shown.

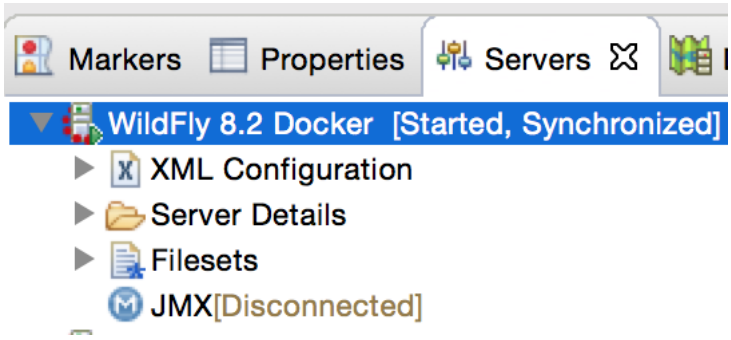


Figure 18. Synchronized WildFly Server

6. Right-click on the Ticket Monster project, select ``Run on Server'' and choose this server. The project runs and displays the start page of ticket-monster.
7. Stop WildFly when you're done.

```
docker stop wildflymngm
```

8. Docker Maven Plugin

Maven plugin allows you to manage Docker images and containers from `pom.xml`. It comes with predefined goals:

Goal	Description
`docker:start`	Create and start containers
`docker:stop`	Stop and destroy containers
`docker:build`	Build images
`docker:push`	Push images to a registry
`docker:remove`	Remove images from local docker host
`docker:logs`	Show container logs

8.1. Run Java EE Application

1. Clone the workspace as:

```
git clone https://github.com/javaee-samples/javaee7-docker-maven.git
```

2. Build the image as:

```
mvn package -Pdocker
```

3. Run the container as:

```
mvn install -Pdocker
```

4. Access your application at http://dockerhost:8080/javaee7-docker-maven/resources/persons.

8.2. Understand Plugin Configuration

`pom.xml` is updated to include docker-maven-plugin as:

```
<plugin>
```

```
    <groupId>org.jolokia</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.11.5</version>
    <configuration>
        <images>
            <image>
                <alias>user</alias>
                <name>arungupta/javaee7-docker-maven</name>
                <build>
                    <from>arungupta/wildfly:8.2</from>
                    <assembly>
                        <descriptor>assembly.xml</descriptor>
                        <basedir>/</basedir>
                    </assembly>
                    <ports>
                        <port>8080</port>
                    </ports>
                </build>
                <run>
                    <ports>
                        <port>8080:8080</port>
                    </ports>
                </run>
            </image>
        </images>
    </configuration>
    <executions>
        <execution>
            <id>docker:build</id>
            <phase>package</phase>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
        <execution>
            <id>docker:start</id>
            <phase>install</phase>
            <goals>
                <goal>start</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Each image configuration has three parts:

1. Image name and alias
2. `<build>` that defines how the image is created. Base image, build artifacts and their dependencies, ports to be exposed, etc to be included in the image are specified here. Assembly descriptor format[35] is used to specify the artifacts to be included and is defined in the `src/main/docker` directory. `assembly.xml` in our case looks like:

```
<assembly . . .>
  <id>javaee7-docker-maven</id>
  <dependencySets>
    <dependencySet>
      <includes>
        <include>org.javaee7.sample:javaee7-docker-maven</include>
      </includes>
      <outputDirectory>/opt/jboss/wildfly/standalone/deployments/</
outputDirectory>
      <outputFileNameMapping>javaee7-docker-maven.war</
outputFileNameMapping>
    </dependencySet>
  </dependencySets>
</assembly>
```

3. `<run>` that defines how the container is run. Ports that need to be exposed are specified here.

In addition, `package` phase is tied to `docker:build` goal and `install` phase is tied to `docker:start` goal.

9. Docker Tools in Eclipse

The Docker tooling is aimed at providing at minimum the same basic level features as the command-line interface, but also provide some advantages by having access to a full fledged UI.

9.1. Install Docker Tools Plugins

As this is still in early access stage, you will have to install it first:

[35] http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html

1. Download and Install JBoss Developer Studio 9.0 Beta 2[36], take defaults through out the installation.
   Alternatively, download Eclipse Mars latest build[37] and configure JBoss Tools plugin from the update site http://download.jboss.org/jbosstools/updates/nightly/mars/.

2. Open JBoss Developer Studio 9.0 Nightly

3. Add a new site using the menu items: `Help'' >` Install New Software…" > `Add… ''. Specify the` Name:" as `Docker Nightly'' and` Location:" as http://download.eclipse.org/linuxtools/updates-docker-nightly/.

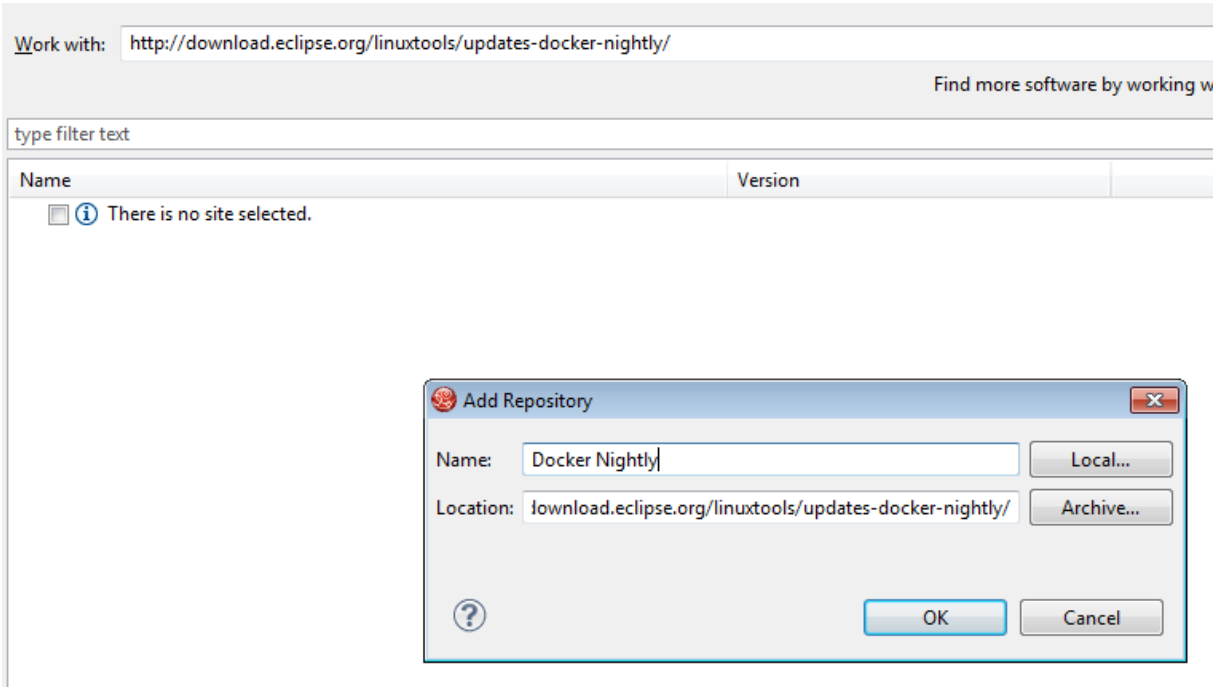


Figure 19. Add Docker Tooling To JBoss Developer Studio

4. Expand Linux Tools, select `Docker Client'' and` Docker Tooling".

[36] http://classroom.example.com:8082/downloads/jboss-devstudio-9.0.0.Beta2-v20150609-1026-B3346-installer-standalone.jar
[37] http://www.eclipse.org/downloads/index-developer-default.php

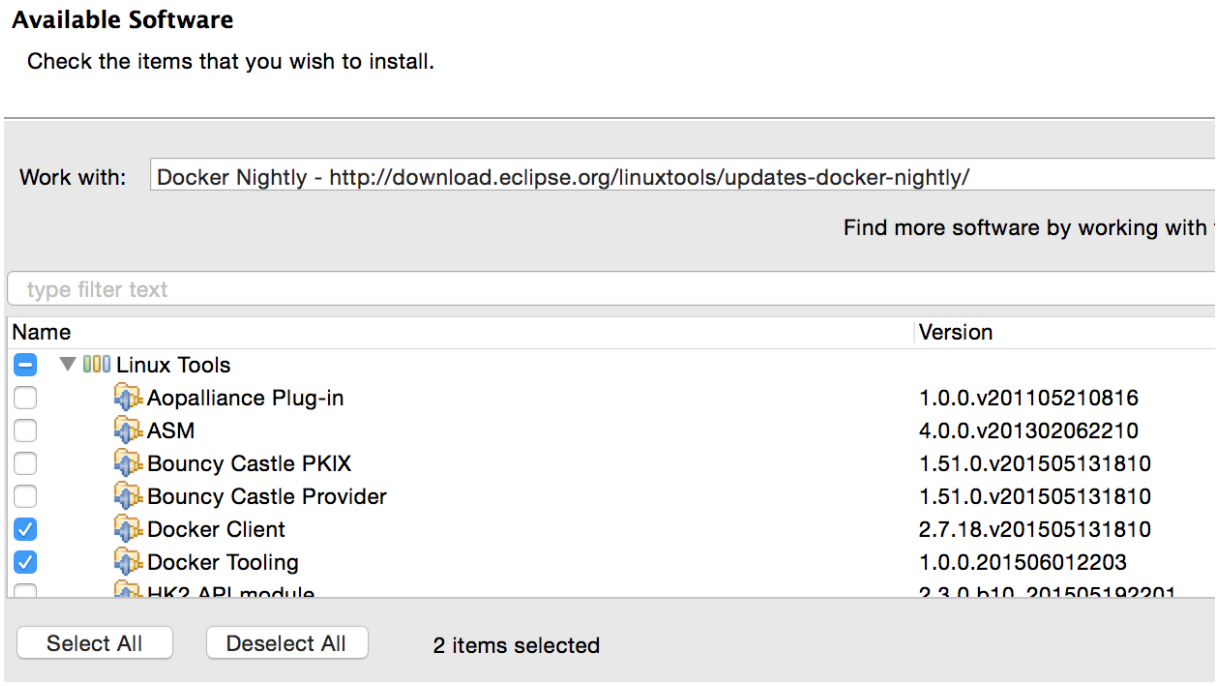


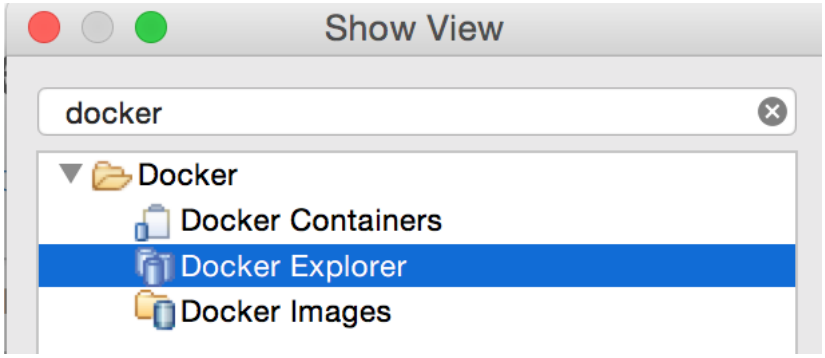
Figure 20. Add Docker Tooling

5. Click on `Next >'', ` Next >", accept the terms of the license agreement, and click on ``Finish". This will complete the installation of plugins.
   Restart the IDE for the changes to take effect.

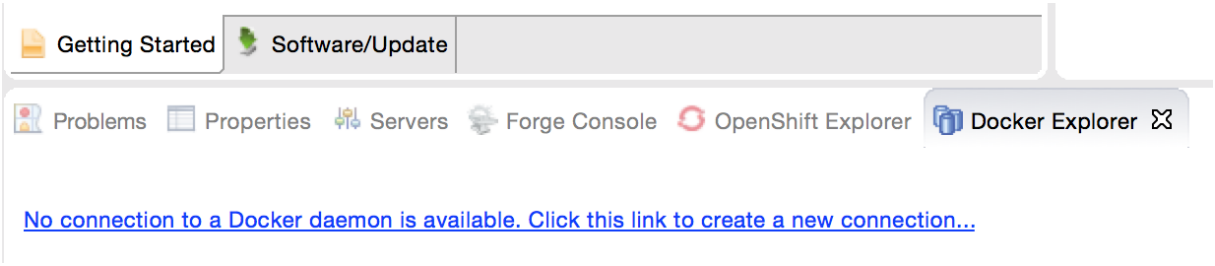
9.2. Docker Explorer

The Docker Explorer provides a wizard to establish a new connection to a Docker daemon. This wizard can detect default settings if the user's machine runs Docker natively (such as in Linux) or in a VM using Boot2Docker (such as in Mac or Windows). Both Unix sockets on Linux machines and the REST API on other OSes are detected and supported. The wizard also allows remote connections using custom settings.

1. Use the menu `Window'', ` Show View", `Other…''. Type ` docker" to see the output as:



2. Select ``Docker Explorer'' to open Docker Explorer.



3. Click on the link in this window to create a connection to Docker Host. Specify the settings as shown:

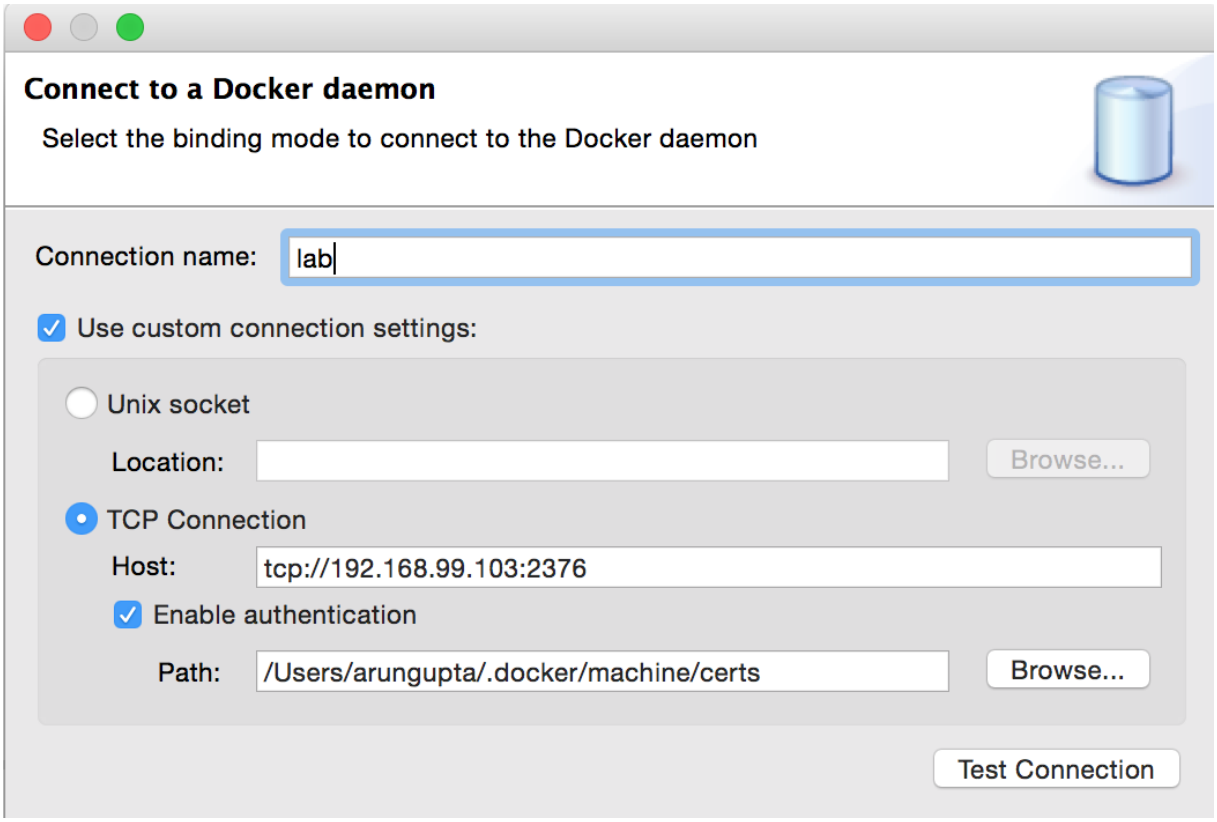


Figure 21. Docker Explorer

Make sure to get IP address of the Docker Host as:

```
docker-machine ip lab
```

Also, make sure to specify the correct directory for `.docker` on your machine.

4. Click on ``Test Connection'' to check the connection. This should show the output as:

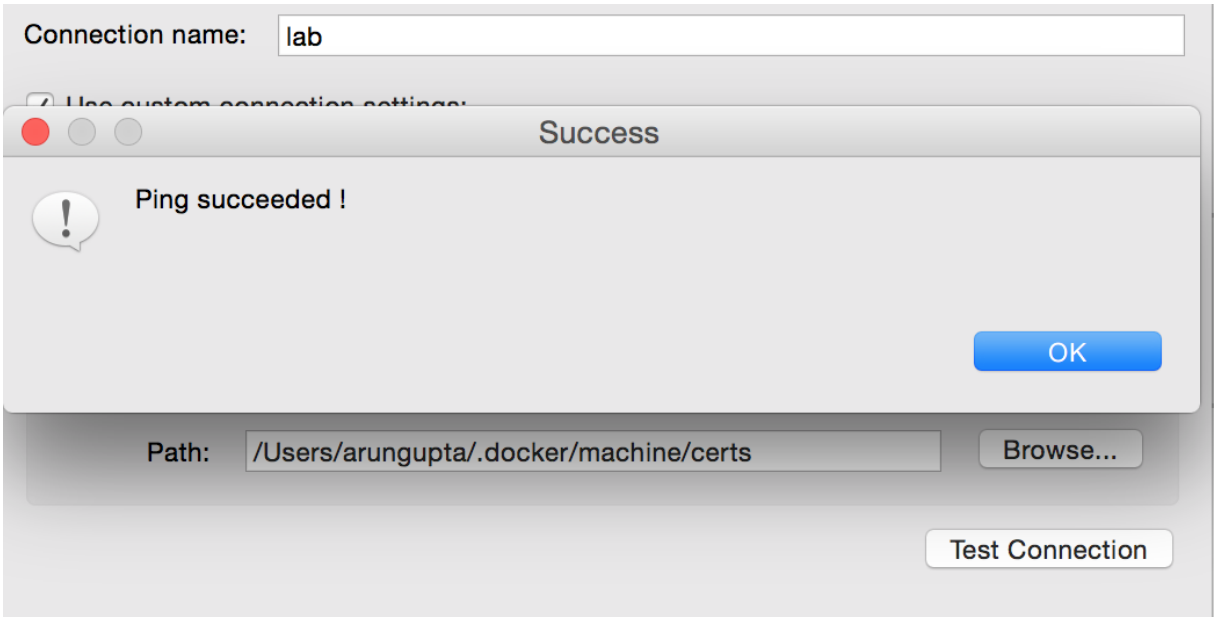


Figure 22. Docker Explorer

Click on `OK'' and` Finish" to exit out of the wizard.

5. Docker Explorer itself is a tree view that handles multiple connections and provides users with quick overview of the existing images and containers.

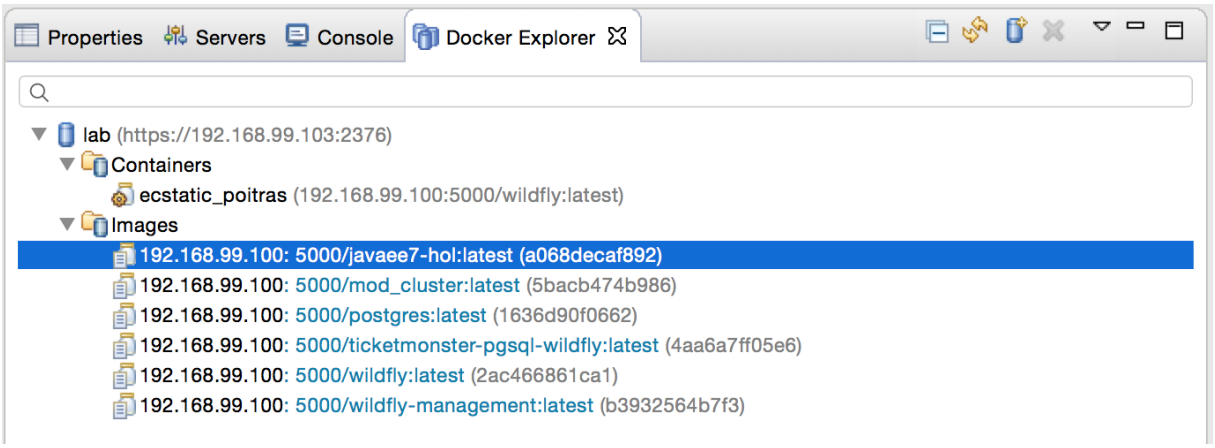


Figure 23. Docker Explorer Tree View

6. Customize the view by clicking on the arrow in toolbar:

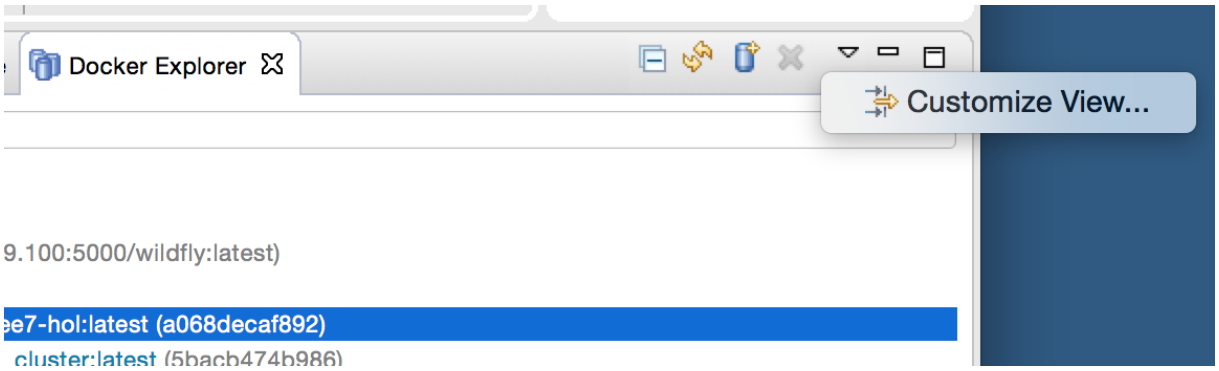


Figure 24. Docker Explorer Customize View

Built-in filters can show/hide intermediate and `dangling' images, as well as stopped containers.

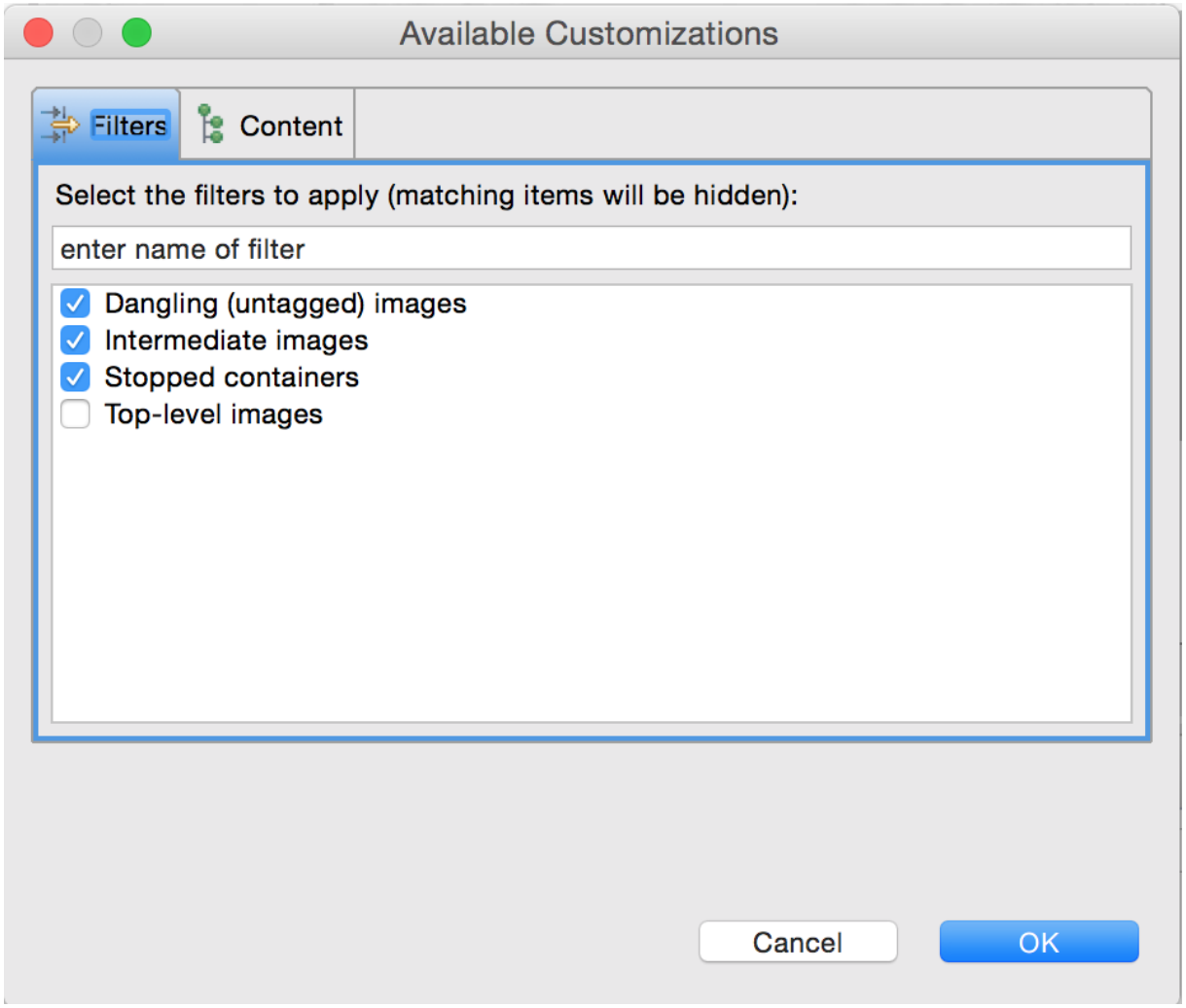


Figure 25. Docker Explorer Customize View Wizard

9.3. Docker Images

The Docker Images view lists all images in the Docker host selected in the Docker Explorer view. This view allows user to manage images, including:

1. Pull/push images from/to the Docker Hub Registry (other registries will be supported as well, #469306[38])
2. Build images from a Dockerfile
3. Create a container from an image

Lets take a look at it.

1. Use the menu `Window'', Show View", `Other…'', select` Docker Images". It shows the list of images on Docker Host:

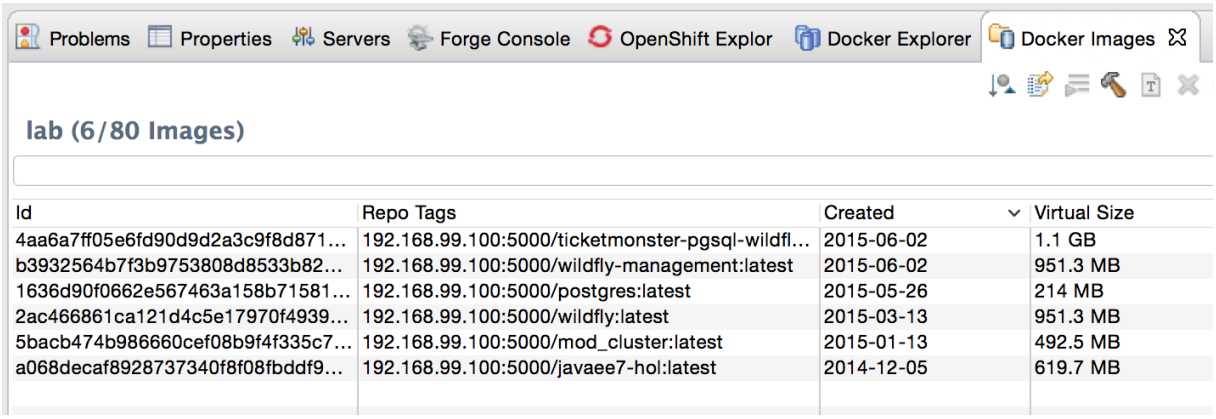


Figure 26. Docker Images View

2. Right-click on the image ending with ``wildfly:latest" and click on the green arrow in the toolbar. This will show the following wizard:

[38] https://bugs.eclipse.org/bugs/show_bug.cgi?id=469306

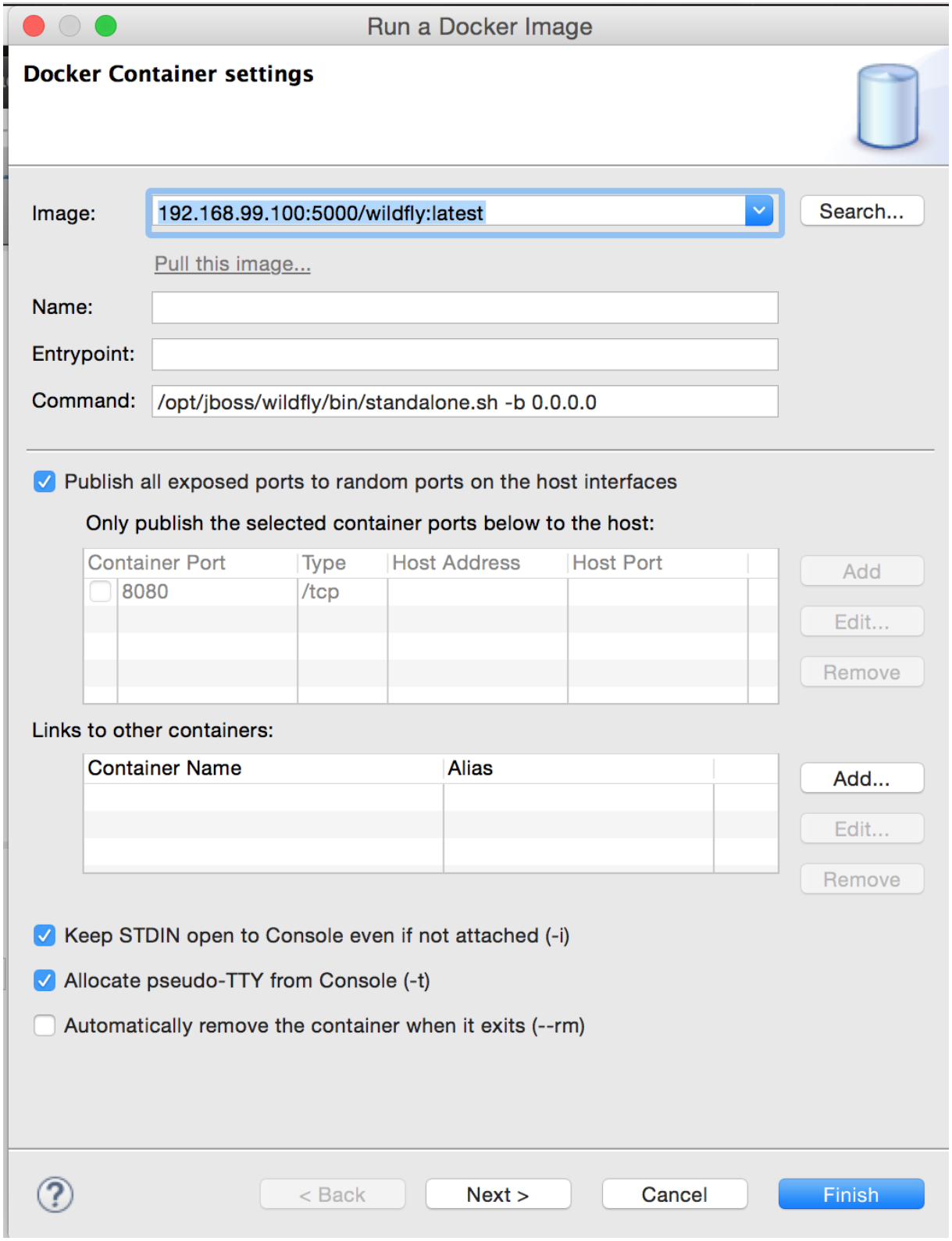


Figure 27. Docker Run Container Wizard

By default, all exports ports from the image are mapped to random ports on the host interface. This setting can be changed by unselecting the first checkbox and specify exact port mapping.

Click on ``Finish'' to start the container.

3. When the container is started, all logs are streamed into Eclipse Console:

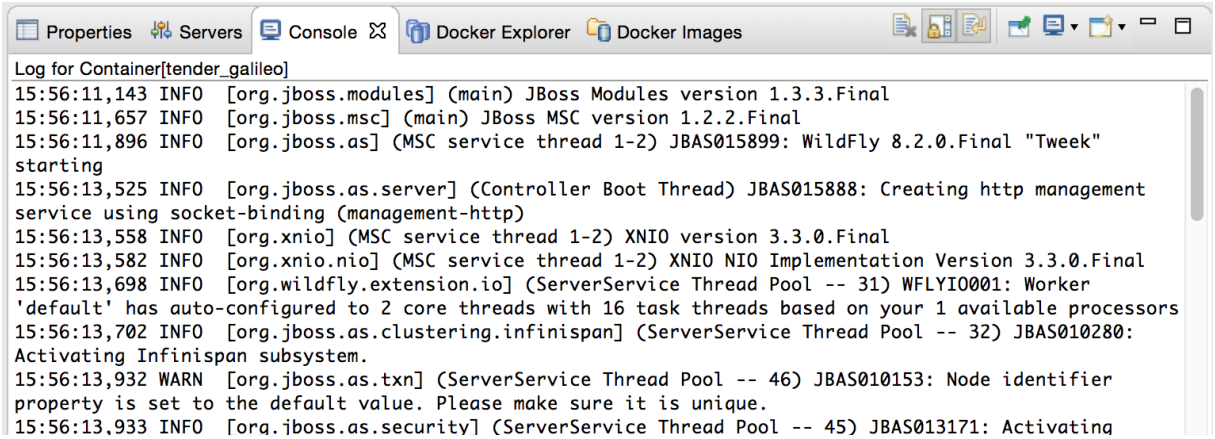


Figure 28. Docker Container Logs

9.4. Docker Containers

Docker Containers view lets the user manage the containers. The view toolbar provides commands to start, stop, pause, unpause, display the logs and kill containers.

1. Use the menu `Window''` Show View", `Other…''`, `select` Docker Containers". It shows the list of running containers on Docker Host:

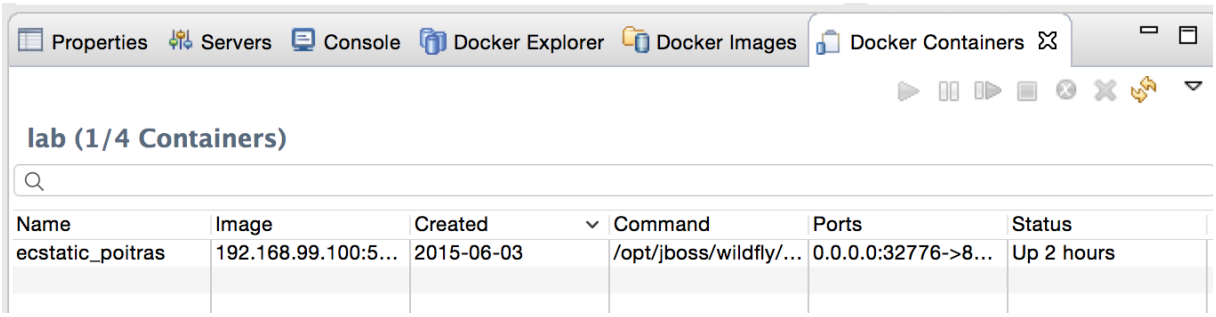


Figure 29. Docker Containers View

2. Pause the container by clicking on the `pause'' button in the toolbar (#469310`[39]`). Show the complete list of containers by clicking on the` View Menu", ``Show all containers".

[39] https://bugs.eclipse.org/bugs/show_bug.cgi?id=469310

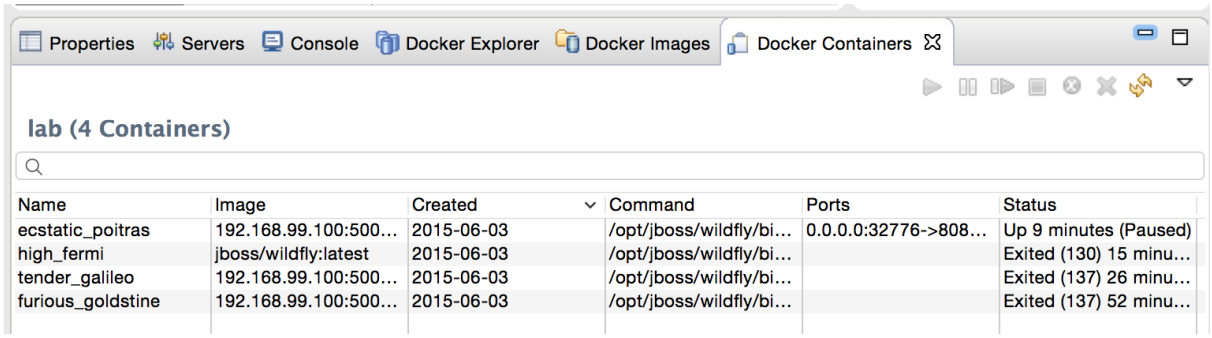


Figure 30. All Docker Containers

3. Select the paused container, and click on the green arrow in the toolbar to restart the container.
4. Right-click on any running container and select ``Display Log'' to view the log for this container.

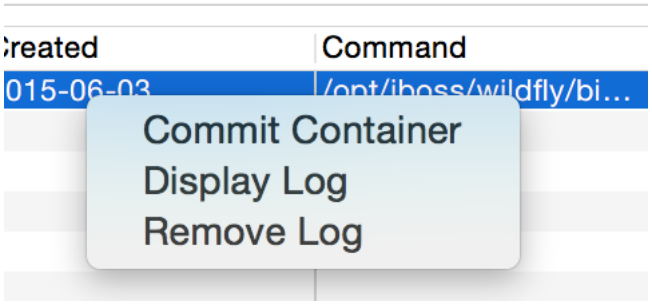


Figure 31. Eclipse Properties View

TODO: Users can also attach an Eclipse console to a running Docker container to follow the logs and use the STDIN to interact with it.

9.5. Details on Images and Containers

Eclipse Properties view is used to provide more information about the containers and images.

1. Just open the Properties View and click on a Connection, Container, or Image in any of the Docker Explorer View, Docker Containers View, or Docker Images View. This will fill in data in the Properties view.
   Info view is shown as:

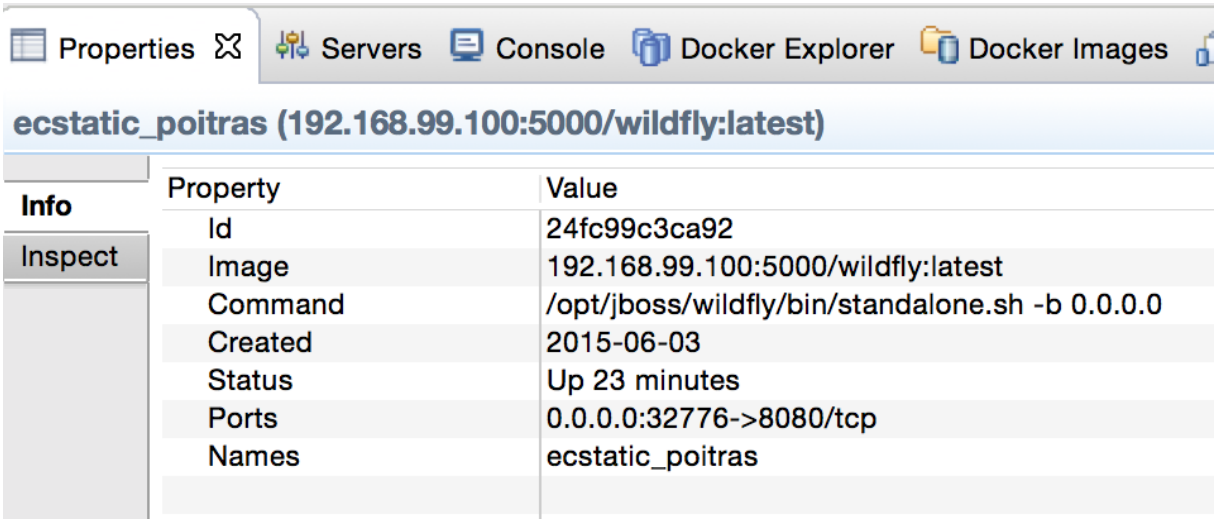


Figure 32. Docker Container Properties View Info

Inspect view is shown as:

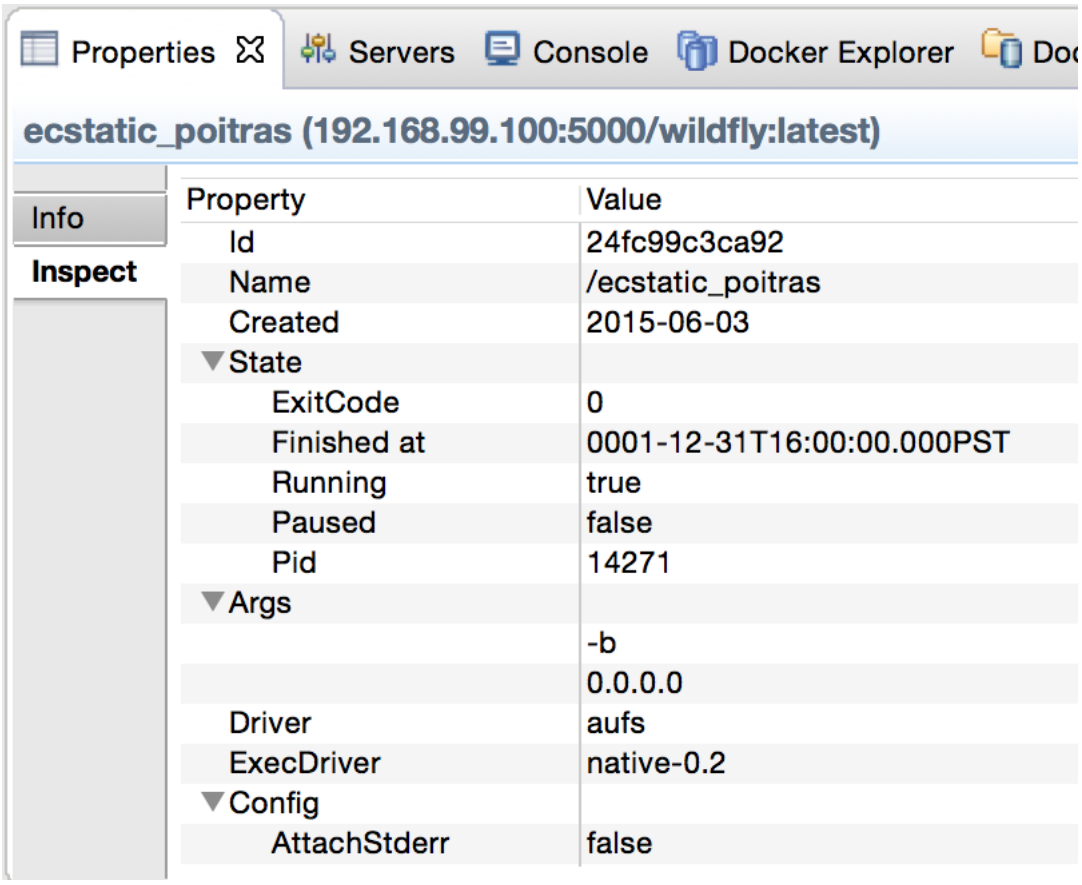


Figure 33. Docker Container Properties View Inspect

10. Test Java EE Applications on Docker

Testing Java EE applications is a very important aspect. Especially when it comes to in-container tests, JBoss Arquillian[40] is well known to make this very easy for Java EE application servers. Picking up where unit tests leave off, Arquillian handles all the plumbing of container management, deployment and framework initialization so you can focus on the task at hand, writing your tests.

With Arquillian, you can use WildFly remote container adapter[41] and connect to any WildFly instance running in a Docker container. But this wouldn't help with the Docker container lifycycle management.

Arquillian Cube[42], an extension of Arquillian, allows you to control the lifecycle of Docker images as part of the test lifecyle, either automatically or manually. This extension allows to start a Docker container with a server installed, deploy the required deployable file within it and execute Arquillian tests.

The key point here is that if Docker is used as deployable platform in production, your tests are executed in a the same container as it will be in production, so your tests are even more real than before.

1. Check out the workspace:

```
git clone http://github.com/javaee-samples/javaee-arquillian-cube
```

2. Edit `src/test/resources/arquillian.xml` file and change the IP address specified in `serverUri` property value to point to your Docker host's IP. This can be found out as:

```
docker-machine ip lab
```

3. Run the tests as:

```
mvn test
```

   This will create a container using the image defined in `src/test/resources/wildfly/Dockerfile`. The container qualifier in `arquillian.xml` defines the directory name in `src/test/resources` directory.

[40] http://www.arquillian.org
[41] http://arquillian.org/modules/wildfly-arquillian-wildfly-remote-container-adapter/
[42] http://arquillian.org/modules/cube-extension/

A pre-built image can be used by specifying:

```
wildfly:
  image: jboss/wildfly
```

instead of

```
wildfly:
  buildImage:
    dockerfileLocation: src/test/resources/wildfly
```

By default, the ``cube" profile is activated and this includes all the required dependencies.

The result is shown as:

```
Running org.javaee7.sample.PersonDatabaseTest
Jun 16, 2015 9:23:04 AM org.jboss.arquillian.container.impl.MapObject
 populate
WARNING: Configuration contain properties not supported by the backing
 object
 org.jboss.as.arquillian.container.remote.RemoteContainerConfiguration
Unused property entries: {target=wildfly:8.1.0.Final:remote}
Supported property names: [managementAddress, password, managementPort,
 managementProtocol, username]
Jun 16, 2015 9:23:13 AM org.xnio.Xnio <clinit>
INFO: XNIO version 3.2.0.Beta4
Jun 16, 2015 9:23:13 AM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.2.0.Beta4
Jun 16, 2015 9:23:13 AM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version (unknown)
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 16.406
 sec - in org.javaee7.sample.PersonDatabaseTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
 ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

```
[INFO]
------------------------------------------------------------------------
```

4. In `arquillian.xml`, add the following property:

```
<property name="connectionMode">STARTORCONNECT</property>
```

This bypasses the create/start Cube commands if a Docker Container with the same name is already running on the target system.

This allows you to prestart the containers manually during development and just connect to them to avoid the extra cost of starting the Docker Containers for each test run. This assumes you are not changing the actual definition of the Docker Container itself.

11. Multiple Containers Using Docker Compose

> Docker Compose is a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.
>
> — github.com/docker/compose

An application using Docker containers will typically consist of multiple containers. With Docker Compose, there is no need to write shell scripts to start your containers. All the containers are defined in a configuration file using *services*, and then `docker-compose` script is used to start, stop, and restart the application and all the services in that application, and all the containers within that service. The complete list of commands is:

Command	Purpose
`build`	Build or rebuild services
`help`	Get help on a command
`kill`	Kill containers
`logs`	View output from containers
`port`	Print the public port for a port binding
`ps`	List containers

Command	Purpose
`pull`	Pulls service images
`restart`	Restart services
`rm`	Remove stopped containers
`run`	Run a one-off command
`scale`	Set number of containers for a service
`start`	Start services
`stop`	Stop services
`up`	Create and start containers

Docker Compose script is only available for OSX and Linux. https://github.com/arun-gupta/docker-java/issues/3 is used for tracking Docker Compose on Windows.

11.1. Configuration File

1. Entry point to Compose is `docker-compose.yml`. Lets use the following file:

```
mysqldb:
  image: classroom.example.com:5000/mysql
  environment:
    MYSQL_DATABASE: sample
    MYSQL_USER: mysql
    MYSQL_PASSWORD: mysql
    MYSQL_ROOT_PASSWORD: supersecret
mywildfly:
  image: classroom.example.com:5000/wildfly-mysql-javaee7
  links:
    - mysqldb:db
  ports:
    - 8080:8080
```

   This file is available in ../attendees/docker-compose.yml and shows:

   a. Two services are defined by the name `mysqldb` and `mywildfly`
   b. Image name for each service is defined using `image`
   c. Environment variables for the MySQL container are defined in `environment`
   d. MySQL container is linked with WildFly container using `links`
   e. Port forwarding is achieved using `ports`

11.2. Start Services

1. All the services can be started, in detached mode, by giving the command:

```
docker-compose up -d
```

And this shows the output as:

```
Creating attendees_mysqldb_1...
Creating attendees_mywildfly_1...
```

An alternate compose file name can be specified using `-f`.

An alternate directory where the compose file exists can be specified using `-p`.

2. Started services can be verified as:

```
> docker-compose ps
         Name                        Command               State
        Ports
-------------------------------------------------------------------------------------
attendees_mysqldb_1     /entrypoint.sh mysqld            Up      3306/
tcp
attendees_mywildfly_1   /opt/jboss/wildfly/customi ...   Up
 0.0.0.0:8080->8080/tcp, 9990/tcp
```

This provides a consolidated view of all the services started, and containers within them.

Alternatively, the containers in this application, and any additional containers running on this Docker host can be verified by using the usual `docker ps` command:

```
> docker ps
CONTAINER ID        IMAGE                                      COMMAND
             CREATED             STATUS              PORTS
                 NAMES
3598e545bd2f        arungupta/wildfly-mysql-javaee7:latest   "/opt/
jboss/wildfly/   59 seconds ago      Up 58 seconds       0.0.0.0:8080-
>8080/tcp, 9990/tcp   attendees_mywildfly_1
```

```
b8cf6a3d518b        mysql:latest                                "/
entrypoint.sh mysq   2 minutes ago       Up 2 minutes        3306/tcp
                         attendees_mysqldb_1
```

3. Service logs can be seen as:

```
> docker-compose logs
Attaching to attendees_mywildfly_1, attendees_mysqldb_1
mywildfly_1 | => Starting WildFly server
mywildfly_1 | => Waiting for the server to boot
mywildfly_1 |
 =========================================================================
mywildfly_1 |
mywildfly_1 |   JBoss Bootstrap Environment
mywildfly_1 |
mywildfly_1 |   JBOSS_HOME: /opt/jboss/wildfly
mywildfly_1 |
mywildfly_1 |   JAVA: /usr/lib/jvm/java/bin/java
mywildfly_1 |
mywildfly_1 |   JAVA_OPTS:  -server -Xms64m -Xmx512m -
XX:MaxPermSize=256m -Djava.net.preferIPv4Stack=true -
Djboss.modules.system.pkgs=org.jboss.byteman -Djava.awt.headless=true
mywildfly_1 |

. . .

mywildfly_1 | 15:40:20,866 INFO
 [org.jboss.resteasy.spi.ResteasyDeployment] (MSC service
 thread 1-2) Deploying javax.ws.rs.core.Application: class
 org.javaee7.samples.employees.MyApplication
mywildfly_1 | 15:40:20,914 INFO  [org.wildfly.extension.undertow] (MSC
 service thread 1-2) JBAS017534: Registered web context: /employees
mywildfly_1 | 15:40:21,032 INFO  [org.jboss.as.server] (ServerService
 Thread Pool -- 28) JBAS018559: Deployed "employees.war" (runtime-
name : "employees.war")
mywildfly_1 | 15:40:21,077 INFO  [org.jboss.as] (Controller
 Boot Thread) JBAS015961: Http management interface listening on
 http://127.0.0.1:9990/management
mywildfly_1 | 15:40:21,077 INFO  [org.jboss.as] (Controller Boot
 Thread) JBAS015951: Admin console listening on http://127.0.0.1:9990
mywildfly_1 | 15:40:21,077 INFO  [org.jboss.as] (Controller Boot
 Thread) JBAS015874: WildFly 8.2.0.Final "Tweek" started in 9572ms
 - Started 280 of 334 services (92 services are lazy, passive or on-
demand)
mysqldb_1   | Running mysql_install_db
```

```
mysqldb_1   | 2015-06-05 15:38:31 0 [Note] /usr/sbin/mysqld (mysqld
 5.6.25) starting as process 27 ...
mysqldb_1   | 2015-06-05 15:38:31 27 [Note] InnoDB: Using atomics to
 ref count buffer pool pages

. . .

mysqldb_1   | 2015-06-05 15:38:40 1 [Note] Event Scheduler: Loaded 0
 events
mysqldb_1   | 2015-06-05 15:38:40 1 [Note] mysqld: ready for
 connections.
mysqldb_1   | Version: '5.6.25'  socket: '/var/run/mysqld/mysqld.sock'
 port: 3306  MySQL Community Server (GPL)
mysqldb_1   | 2015-06-05 15:40:18 1 [Warning] IP address '172.17.0.24'
 could not be resolved: Name or service not known
```

11.3. Verify Application

1. Access the application at http://dockerhost:8080/employees/resources/employees/. This is shown in the browser as:

This XML file does not appear to have any style information associate


```
▼<collection>
  ▼<employee>
     <id>1</id>
     <name>Penny</name>
   </employee>
  ▼<employee>
     <id>2</id>
     <name>Sheldon</name>
   </employee>
  ▼<employee>
     <id>3</id>
     <name>Amy</name>
   </employee>
  ▼<employee>
     <id>4</id>
     <name>Leonard</name>
   </employee>
  ▼<employee>
     <id>5</id>
     <name>Bernadette</name>
   </employee>
  ▼<employee>
     <id>6</id>
     <name>Raj</name>
   </employee>
  ▼<employee>
     <id>7</id>
     <name>Howard</name>
   </employee>
  ▼<employee>
     <id>8</id>
     <name>Priya</name>
   </employee>
 </collection>
```


Figure 34. Output From Servers Run Using Docker Compose

11.4. Stop Services

1. Stop the services as:

```
> docker-compose stop
Stopping attendees_mywildfly_1...
Stopping attendees_mysqldb_1...
```

11.5. Scale Services

https://github.com/arun-gupta/docker-java/issues/51

12. Java EE Application on Mod Cluster

A frequent requirement for Java EE based applications is running them on a cluster of application server. While setup and test can be complicated on developer machines, this is where Docker can play to it's full potential. With the help of images and automatic port mapping, we're ready to test Ticket Monster on a couple of WildFly instances and add and remove them randomly.

The diagram below shows what will be achieved in this section:

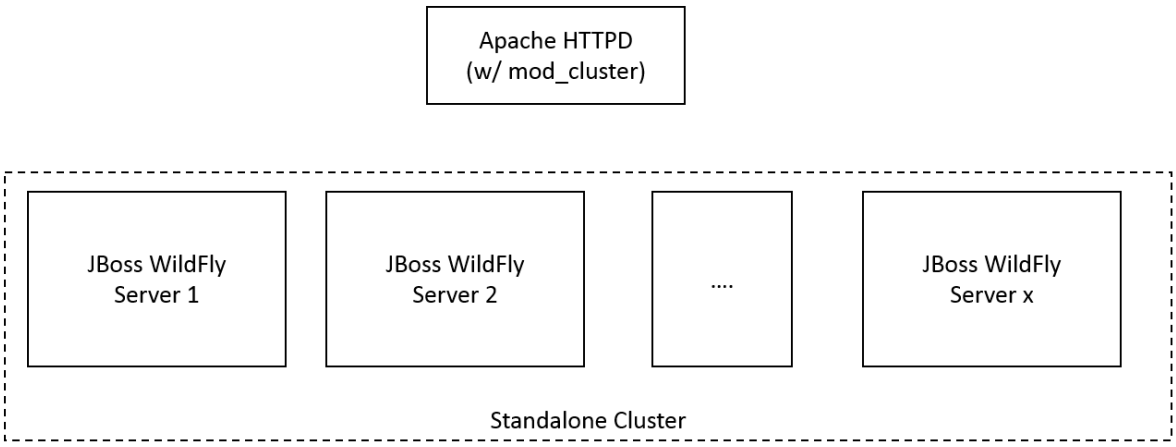


Figure 35. Standalone Cluster with WildFly and mod_cluster

1. Start Apache HTTPD server

```
docker run -d --name modcluster -p 80:80 classroom.example.com:5000/
mod_cluster
```

2. Open http://dockerhost/mod_cluster_manager in your browser to see the empty console as:

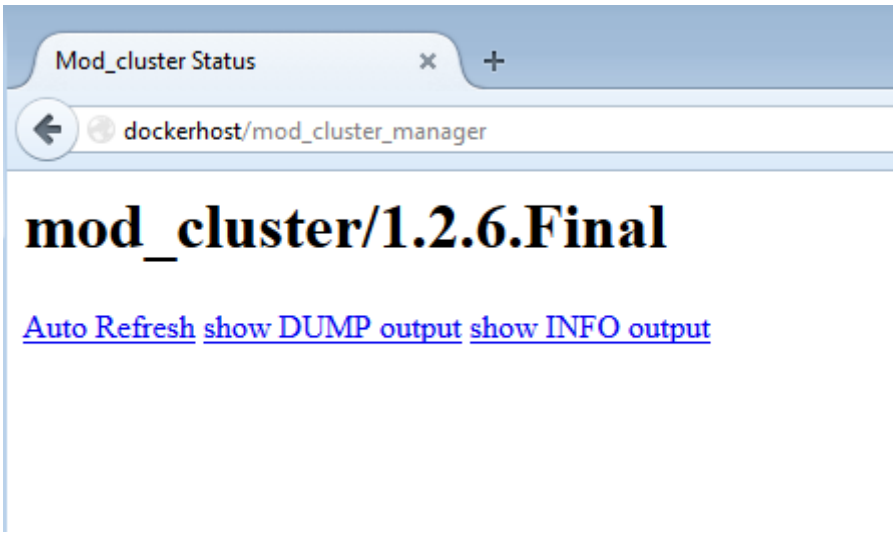


Figure 36. Apache HTTPD runing mod_cluster_manager interface

3. Start Postgres database container, if not already running:

```
docker run --name db -d -p 5432:5432 -e POSTGRES_USER=ticketmonster -
e POSTGRES_PASSWORD=ticketmonster-docker classroom.example.com:5000/
postgres
```

4. Start the first WildFly instance:

```
docker run -d --name server1 --link db:db --link modcluster:modcluster
 classroom.example.com:5000/ticketmonster-pgsql-wildfly
```

Besides linking the database container using `--link db:db`, we also link the ``modcluster'' container. This should be done rather quickly and if you now revisit the mod_cluster_manager[43] in your browser, then you can see that the first server was registered with the loadbalancer:

[43] http://dockerhost/mod_cluster_manager/

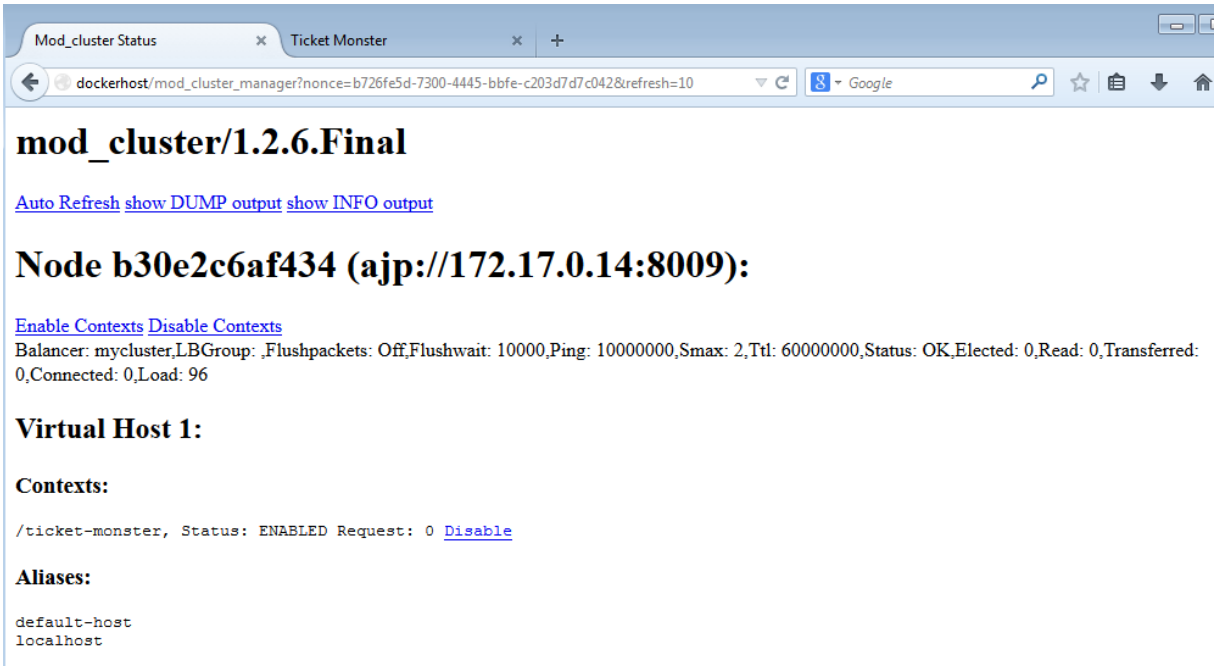


Figure 37. First WildFly instance registered with Load Balancer

5. To make sure the Ticket Monster application is also running just visit http://dockerhost/ticket-monster and you will be presented with the Ticket Monster welcome screen.

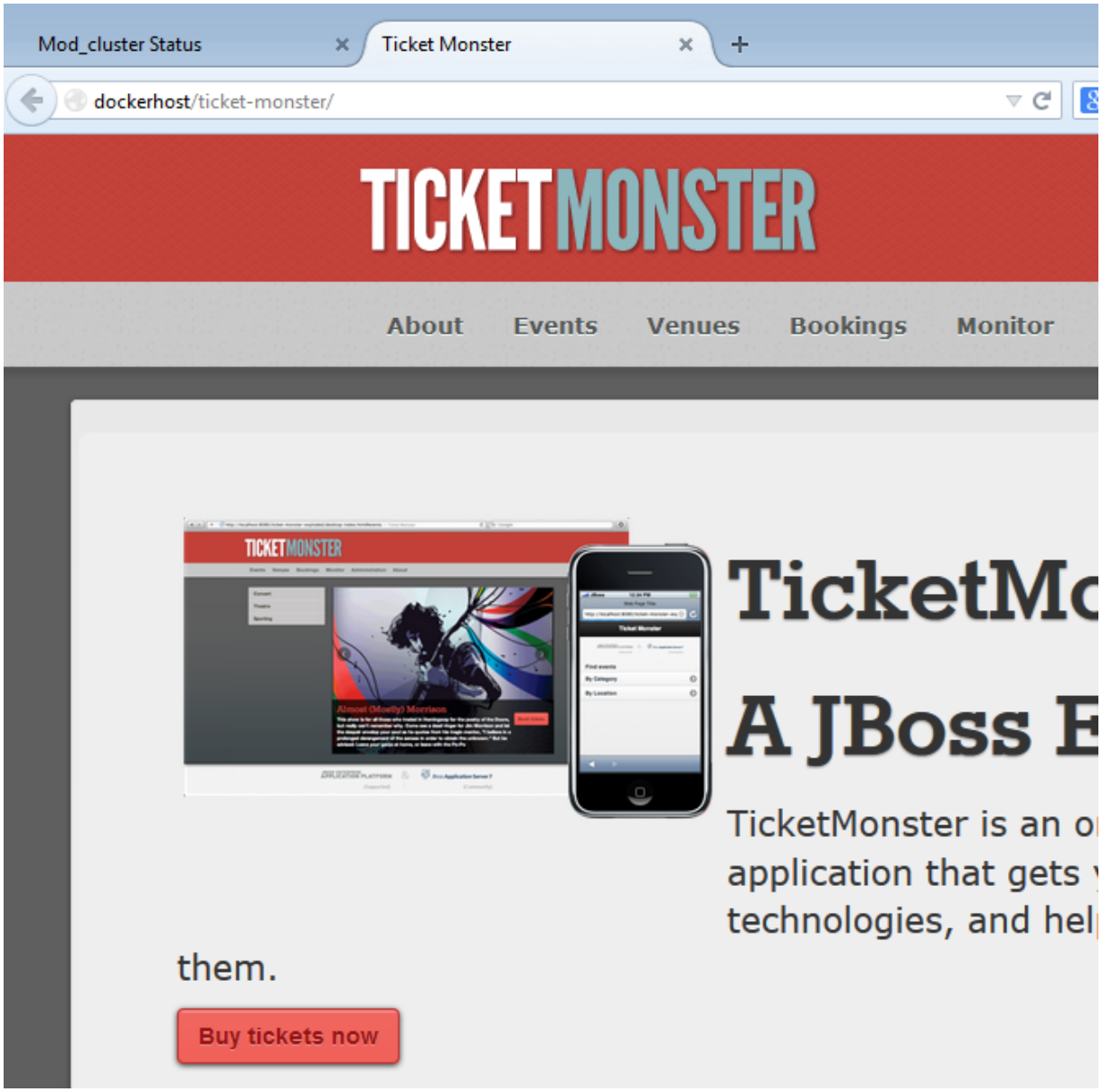


Figure 38. Clustered Ticket Monster Application

6. Start as many WildFly instances as you want (and your computer memory can handle):

```
docker run -d --name server2 --link db:db --link modcluster:modcluster
 classroom.example.com:5000/ticketmonster-pgsql-wildfly
docker run -d --name server3 --link db:db --link modcluster:modcluster
 classroom.example.com:5000/ticketmonster-pgsql-wildfly
docker run -d --name server4 --link db:db --link modcluster:modcluster
 classroom.example.com:5000/ticketmonster-pgsql-wildfly
```

7. Stop some servers:

```
docker stop server1
```

```
docker stop server3
```

Ensure that the application is still accessible at http://dockerhost/ticket-monster.

13. Java EE Application on Docker Swarm Cluster

Docker Swarm solves one of the fundamental limitations of Docker where the containers could only run on a single Docker host. Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual host.

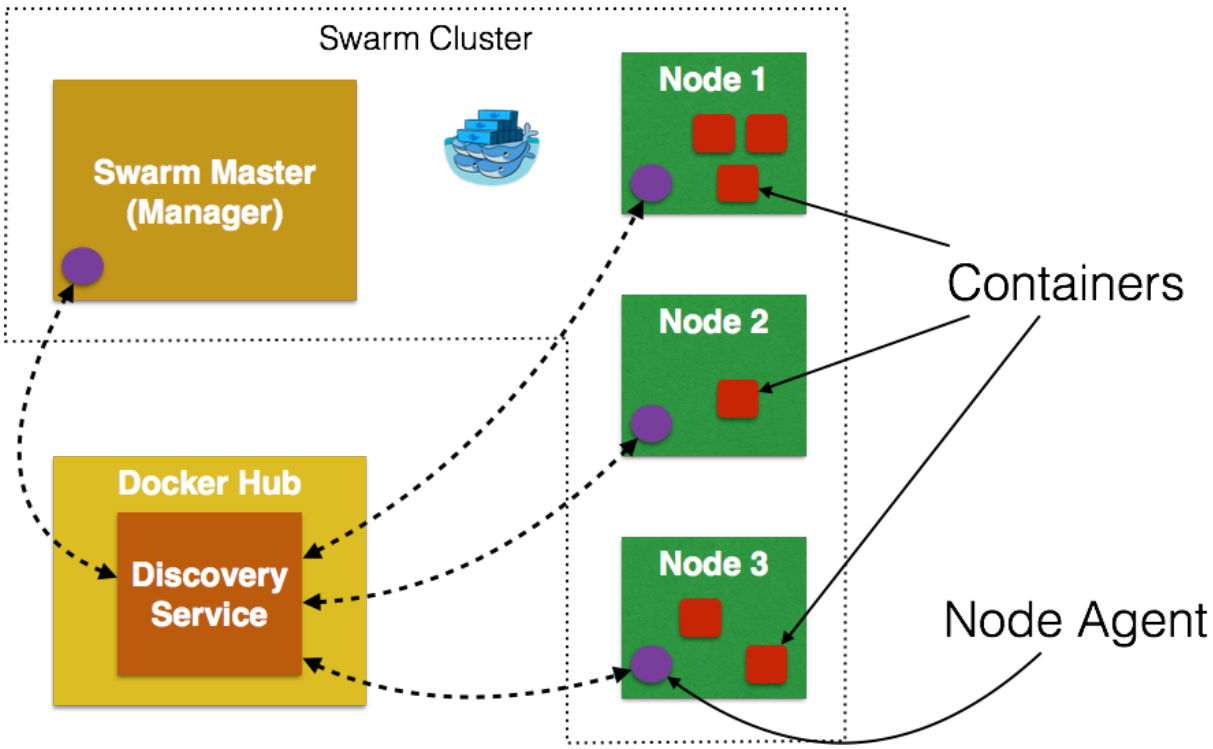


Figure 39. Key Components of Docker Swarm

Swarm Manager: Docker Swarm has a Master or Manager, that is a pre-defined Docker Host, and is a single point for all administration. Currently only a single instance of manager is allowed in the cluster. This is a SPOF for high availability architectures and additional managers will be allowed in a future version of Swarm with #598. TODO: ADD LINK.

Swarm Nodes: The containers are deployed on Nodes that are additional Docker Hosts. Each Swarm Node must be accessible by the manager, each node must listen to the same network interface (TCP port). Each node runs a node agent that registers the referenced Docker daemon, monitors it, and updates the discovery backend with the node's status. The containers run on a node.

Scheduler Strategy: Different scheduler strategies (`binpack'', ` spread" (default), and ``random") can be applied to pick the best node to run your container. The default strategy optimizes the node for least number of running containers. There are multiple kinds of filters, such as constraints and affinity. This should allow for a decent scheduling algorithm.

Node Discovery Service: By default, Swarm uses hosted discovery service, based on Docker Hub, using tokens to discover nodes that are part of a cluster. However etcd, consul, and zookeeper can be also be used for service discovery as well. This is particularly useful if there is no access to Internet, or you are running the setup in a closed network. A new discovery backend can be created as explained here. It would be useful to have the hosted Discovery Service inside the firewall and #660 will discuss this.

Standard Docker API: Docker Swarm serves the standard Docker API and thus any tool that talks to a single Docker host will seamlessly scale to multiple hosts now. That means if you were using shell scripts using Docker CLI to configure multiple Docker hosts, the same CLI would can now talk to Swarm cluster and Docker Swarm will then act as proxy and run it on the cluster.

There are lots of other concepts but these are the main ones.

1. Create a Swarm cluster. The easiest way of using Swarm is, by using the official Docker image:

   ```
   docker run swarm create
   ```

   This command returns a <TOKEN> and is the unique cluster id. It will be used when creating master and nodes later. This cluster id is returned by the hosted discovery service on Docker Hub.

   Make sure to note this cluster id now as there is no means to list it later.

2. Swarm is fully integrated with Docker Machine, and so is the easiest way to get started. Let’s create a Swarm Master next:

   ```
   docker-machine create -d virtualbox --swarm --swarm-master --swarm-
   discovery token://<TOKEN> swarm-master
   ```

   Replace `<TOKEN>` with the cluster id obtained in the previous step.

`--swarm` configures the machine with Swarm, `--swarm-master` configures the created machine to be Swarm master. Swarm master creation talks to the hosted service on Docker Hub and informs that a master is created in the cluster.

3. Connect to this newly created master and find some more information about it:

```
eval "$(docker-machine env swarm-master)"
docker info
```

If you're on Windows, use the `docker-machine env swarm-master` command only and copy the output into an editor to replace all appearances of EXPORT with SET and issue the three commands at your command prompt, remove the quotes and all duplicate appearences of "/".

This will show the output as:

```
> docker info
Containers: 2
Images: 7
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 11
 Dirperm1 Supported: true
Execution Driver: native-0.2
Kernel Version: 4.0.3-boot2docker
Operating System: Boot2Docker 1.6.2 (TCL 5.4); master : 4534e65 - Wed
 May 13 21:24:28 UTC 2015
CPUs: 1
Total Memory: 998.1 MiB
Name: swarm-master
ID: USSA:35LS:WVRN:GKAZ:MLD4:XMQF:P7PL:VQ5D:7V4K:2QAH:5D2L:HC4K
Debug mode (server): true
Debug mode (client): false
Fds: 24
Goroutines: 37
System Time: Wed Jun 10 03:40:00 UTC 2015
EventsListeners: 1
Init SHA1: 7f9c6798b022e64f04d2aff8c75cbf38a2779493
Init Path: /usr/local/bin/docker
Docker Root Dir: /mnt/sda1/var/lib/docker
Username: arungupta
Registry: [https://index.docker.io/v1/]
```

```
Labels:
 provider=virtualbox
```

4. Create Swarm nodes.

```
docker-machine create -d virtualbox --swarm --swarm-discovery token://
<TOKEN> swarm-node-01
```

Replace `<TOKEN>` with the cluster id obtained in the previous step.

Node creation talks to the hosted service at Docker Hub and joins the previously created cluster. This is specified by `--swarm-discovery token://…` and specifying the cluster id obtained earlier.

5. To make it a real cluster, let's create a second node:

```
docker-machine create -d virtualbox --swarm --swarm-discovery token://
<TOKEN> swarm-node-02
```

Replace `<TOKEN>` with the cluster id obtained in the previous step.

6. List all the nodes / Docker machines, that has been created so far.

```
docker-machine ls
```

This shows the output as:

```
NAME             ACTIVE   DRIVER       STATE     URL
     SWARM
lab                       virtualbox   Running
 tcp://192.168.99.103:2376
swarm-master     *        virtualbox   Running
 tcp://192.168.99.107:2376   swarm-master (master)
swarm-node-01             virtualbox   Running
 tcp://192.168.99.108:2376   swarm-master
swarm-node-02             virtualbox   Running
 tcp://192.168.99.109:2376   swarm-master
```

The machines that are part of the cluster have the cluster's name in the SWARM column, blank otherwise. For example, `lab'' is a standalone machine where as all other machines are part of the` swarm-master'' cluster. The Swarm master is also identified by (master) in the SWARM column.

7. Connect to the Swarm cluster and find some information about it:

```
eval "$(docker-machine env --swarm swarm-master)"
docker info
```

This shows the output as:

```
> docker info
Containers: 4
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-master: 192.168.99.107:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
 swarm-node-01: 192.168.99.108:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
 swarm-node-02: 192.168.99.109:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
```

There are 3 nodes – one Swarm master and 2 Swarm nodes. There is a total of 4 containers running in this cluster – one Swarm agent on master and each node, and there is an additional swarm-agent-master running on the master. This can be verified by connecting to the master and listing all the containers:

8. List nodes in the cluster with the following command:

```
docker run swarm list token://<TOKEN>
```

This shows the output as:

```
> docker run swarm list token://b9d9da9198c0facbeeae302242fb65a5
192.168.99.109:2376
192.168.99.108:2376
192.168.99.107:2376
```

The complete cluster is in place now, and we need to deploy the Java EE application to it.

Swarm takes care for the distribution of the deployments across the nodes. The only thing, we need to do is to deploy the application as already explained in Section 6, “Deploy Java EE 7 Application (Container Linking)”.

1. Start MySQL server as:

```
docker run --name mysqldb -e MYSQL_USER=mysql -e MYSQL_PASSWORD=mysql -
e MYSQL_DATABASE=sample -e MYSQL_ROOT_PASSWORD=supersecret -p 3306:3306
 -d mysql
```

   `-e` define environment variables that are read by the database at startup and allow us to access the database with this user and password.

2. Start WildFly and deploy Java EE 7 application as:

```
docker run -d --name mywildfly --link mysqldb:db -p 8080:8080
 arungupta/wildfly-mysql-javaee7
```

   This is using the Docker Container Linking[44] explained earlier.

3. Check the state of the cluster as:

```
> docker info
Containers: 7
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-master: 192.168.99.107:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
 swarm-node-01: 192.168.99.108:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
 swarm-node-02: 192.168.99.109:2376
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
```

   ``swarm-node-02'' is running three containers and so lets look at the list of containers running there:

[44] https://docs.docker.com/userguide/dockerlinks/

```
> eval "$(docker-machine env swarm-node-02)"
> docker ps -a
CONTAINER ID        IMAGE                                        COMMAND
             CREATED              STATUS              PORTS
        NAMES
f8022254703d        arungupta/wildfly-mysql-javaee7:latest   "/opt/
jboss/wildfly/   About a minute ago   Up About a minute   0.0.0.0:8080-
>8080/tcp   mywildfly
7b6e9324c735        mysql:latest                                 "/
entrypoint.sh mysq   7 minutes ago        Up 7 minutes
 0.0.0.0:3306->3306/tcp   mysqldb
6ed4c35c943b        swarm:latest                                 "/swarm
 join --addr    12 minutes ago       Up 12 minutes       2375/tcp
           swarm-agent
```

4. Access the application as:

```
curl http://$(docker-machine ip swarm-node-02):8080/employees/
resources/employees
```

to see the output as:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
><collection><employee><id>1</id><name>Penny</name></
employee><employee><id>2</id><name>Sheldon</name></
employee><employee><id>3</id><name>Amy</name></
employee><employee><id>4</id><name>Leonard</name></
employee><employee><id>5</id><name>Bernadette</
name></employee><employee><id>6</id><name>Raj</name></
employee><employee><id>7</id><name>Howard</name></
employee><employee><id>8</id><name>Priya</name></employee></collection>
```

TODO: https://github.com/javaee-samples/docker-java/issues/55

14. Java EE Application on Kubernetes Cluster

> Kubernetes is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications.
>
> — github.com/GoogleCloudPlatform/kubernetes/

Kubernetes, or ``k8s'' in short, allows the user to provide declarative primitives for the desired state, for example “need 5 WildFly servers and 1 MySQL server running”. Kubernetes self-healing mechanisms, such as auto-restarting, re-scheduling, and replicating containers then ensure this state is met. The user just define the state and Kubernetes ensures that the state is met at all times on the cluster.

How is it related to Docker?

Docker provides the lifecycle management of containers. A Docker image defines a build time representation of the runtime containers. There are commands to start, stop, restart, link, and perform other lifecycle methods on these containers. Kubernetes uses Docker to package, instantiate, and run containerized applications.

How does Kubernetes simplify containerized application deployment?

A typical application would have a cluster of containers across multiple hosts. For example, your web tier (for example Undertow) might run on a set of containers. Similarly, your application tier (for example, WildFly) would run on a different set of containers. The web tier would need to delegate the request to application tier. In some cases, or at least to begin with, you may have your web and application server packaged together in the same set of containers. The database tier would generally run on a separate tier anyway. These containers would need to talk to each other. Using any of the solutions mentioned above would require scripting to start the containers, and monitoring/bouncing if something goes down. Kubernetes does all of that for the user after the application state has been defined.

14.1. Key Concepts

At a very high level, there are three key concepts:

1. **Pods** are the smallest deployable units that can be created, scheduled, and managed. Its a logical collection of containers that belong to an application.
2. **Master** is the central control point that provides a unified view of the cluster. There is a single master node that control multiple minions.
3. **Minion** is a worker node that run tasks as delegated by the master. Minions can run one or more pods. It provides an application-specific “virtual host” in a containerized environment.

A picture is always worth a thousand words and so this is a high-level logical block diagram for Kubernetes:

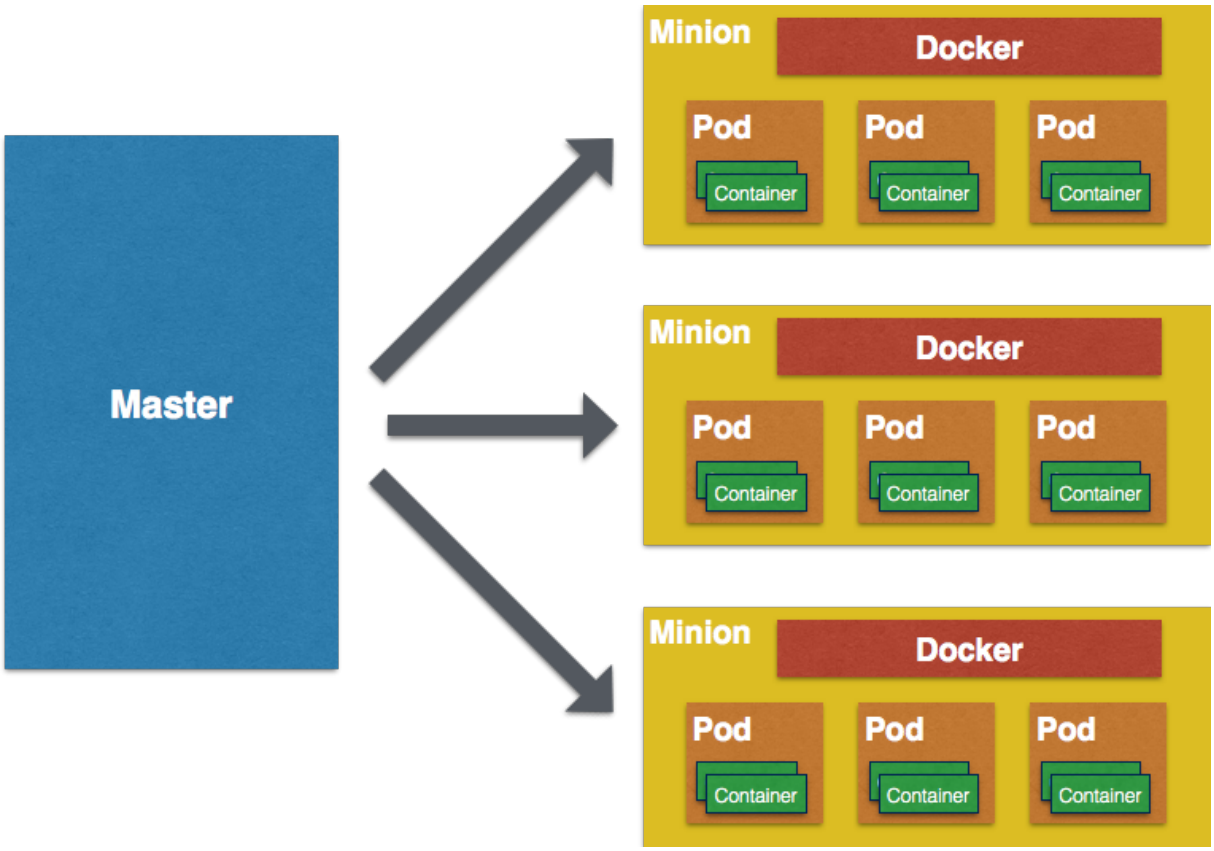


Figure 40. Kubernetes Key Concepts

After the 50,000 feet view, lets fly a little lower at 30,000 feet and take a look at how Kubernetes make all of this happen. There are a few key components at Master and Minion that make this happen.

1. **Replication Controller** is a resource at Master that ensures that requested number of pods are running on minions at all times.
2. **Service** is an object on master that provides load balancing across a replicated group of pods. Label is an arbitrary key/value pair in a distributed watchable storage that the Replication Controller uses for service discovery.
3. **Kubelet** Each minion runs services to run containers and be managed from the master. In addition to Docker, Kubelet is another key service installed there. It reads container manifests as YAML files that describes a pod. Kubelet ensures that the containers defined in the pods are started and continue running.
4. Master serves **RESTful Kubernetes API** that validate and configure Pod, Service, and Replication Controller.

14.2. Start Kubernetes Cluster

1. In ``kubernetes'' directory, setup a cluster as:

```
cd kubernetes

export KUBERNETES_PROVIDER=vagrant
./cluster/kube-up.sh
```

The `KUBERNETES_PROVIDER` environment variable tells all of the various cluster management scripts which variant to use.

This will take a few minutes, so be patience! Vagrant will provision each machine in the cluster with all the necessary components to run Kubernetes.

It shows the output as:

```
Starting cluster using provider: vagrant
... calling verify-prereqs
... calling kube-up
Using credentials: vagrant:vagrant

. . .

Cluster validation succeeded
Done, listing cluster services:

Kubernetes master is running at https://10.245.1.2
KubeDNS is running at https://10.245.1.2/api/v1beta3/proxy/namespaces/
default/services/kube-dns
```

Note down the address for Kubernetes master, `https://10.245.1.2` in this case.

2. Verify the Kubernetes cluster as:

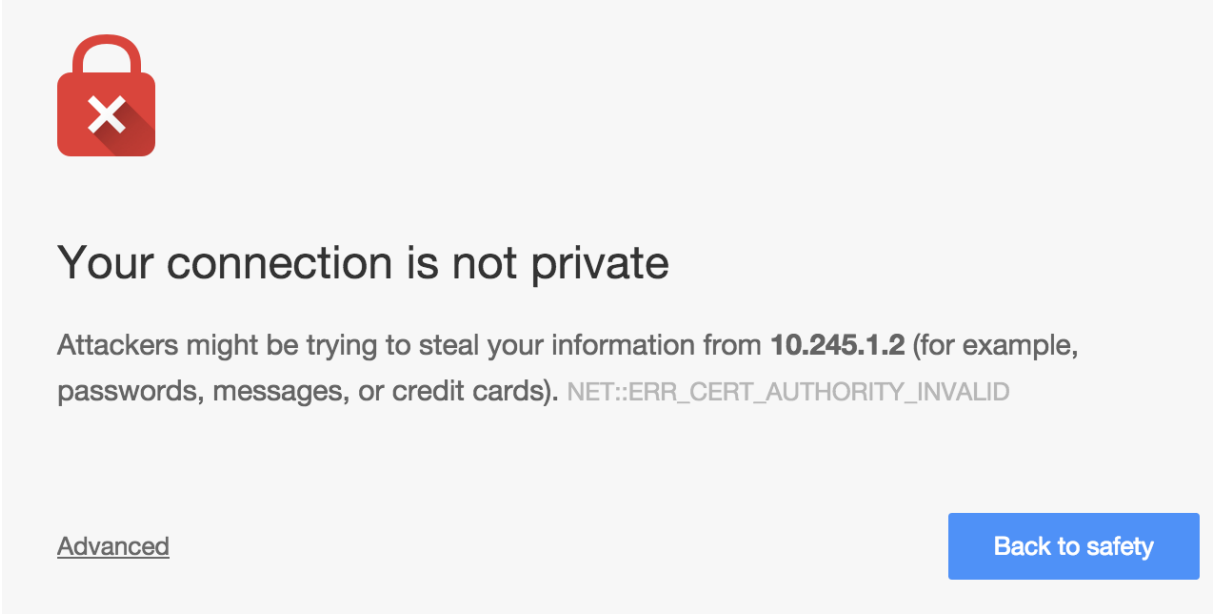
```
kubernetes> vagrant status
Current machine states:

master                    running (virtualbox)
minion-1                  running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

By default, the Vagrant setup will create a single kubernetes-master and 1 kubernetes-minion. Each VM will take 1 GB, so make sure you have at least 2GB to 4GB of free memory (plus appropriate free disk space).

> By default, only one minion is created. This can be manipulated by setting an environment variable NUM_MINIONS variable to an integer before invoking `kube-up.sh` script.

By default, each VM in the cluster is running Fedora, Kubelet is installed into ``systemd'', and all other Kubernetes services are running as containers on Master.

3. Access https://10.245.1.2 (or whatever IP address is assigned to your kubernetes cluster start up log). This may present the warning as shown below:



Click on `Advanced'' and then on` Proceed to 10.245.1.2'' to see the output as:

Figure 41. Kubernetes Output from Master

Use `vagrant'' as the username and` vagrant" as the password.

Check the list of nodes as:

```
> ./cluster/kubectl.sh get nodes
NAME          LABELS                                   STATUS
10.245.1.3    kubernetes.io/hostname=10.245.1.3    Ready
```

4. Check the list of pods:

```
kubernetes> ./cluster/kubectl.sh get po
POD                                 IP         CONTAINER(S)      IMAGE(S)
                                                                      HOST

                LABELS
          STATUS     CREATED      MESSAGE
etcd-server-kubernetes-master

 kubernetes-master/    <none>
                Running    2 minutes
```

```
                                          etcd-container   gcr.io/
google_containers/etcd:2.0.9

            Running   2 minutes
kube-apiserver-kubernetes-master

 kubernetes-master/   <none>
                Running   2 minutes
                                          kube-apiserver   gcr.io/
google_containers/kube-apiserver:465b93ab80b30057f9c2ef12f30450c3

            Running   2 minutes
kube-dns-v1-lxdof

      10.245.1.3/          k8s-app=kube-dns,kubernetes.io/cluster-
service=true,version=v1   Pending   2 minutes
                                          etcd             gcr.io/
google_containers/etcd:2.0.9
                                          kube2sky         gcr.io/
google_containers/kube2sky:1.7
                                          skydns           gcr.io/
google_containers/skydns:2015-03-11-001
kube-scheduler-kubernetes-master

 kubernetes-master/   <none>
                Running   2 minutes
                                          kube-scheduler   gcr.io/
google_containers/kube-scheduler:d1f640dfb379f64daf3ae44286014821

            Running   2 minutes
```

5. Check the list of services running:

```
> ./cluster/kubectl.sh get se
NAME            LABELS
                          SELECTOR           IP(S)          PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-
service=true,kubernetes.io/name=KubeDNS   k8s-app=kube-dns
 10.247.0.10   53/UDP

                                                           53/TCP
kubernetes      component=apiserver,provider=kubernetes
                          <none>             10.247.0.2     443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes
                          <none>             10.247.0.1     80/TCP
```

6. Check the list of replication controllers:

```
> ./cluster/kubectl.sh get rc
CONTROLLER    CONTAINER(S)    IMAGE(S)
       SELECTOR                        REPLICAS
kube-dns-v1   etcd            gcr.io/google_containers/etcd:2.0.9
       k8s-app=kube-dns,version=v1   1
              kube2sky        gcr.io/google_containers/kube2sky:1.7
              skydns          gcr.io/google_containers/
skydns:2015-03-11-001
```

14.3. Deploy Java EE Application

Pods, and the IP addresses assigned to them, are ephemeral. If a pod dies then Kubernetes will recreate that pod because of its self-healing features, but it might recreate it on a different host. Even if it is on the same host, a different IP address could be assigned to it. And so any application cannot rely upon the IP address of the pod.

Kubernetes services is an abstraction which defines a logical set of pods. A service is typically back-ended by one or more physical pods (associated using labels), and it has a permanent IP address that can be used by other pods/applications. For example, WildFly pod can not directly connect to a MySQL pod but can connect to MySQL service. In essence, Kubernetes service offers clients an IP and port pair which, when accessed, redirects to the appropriate backends.

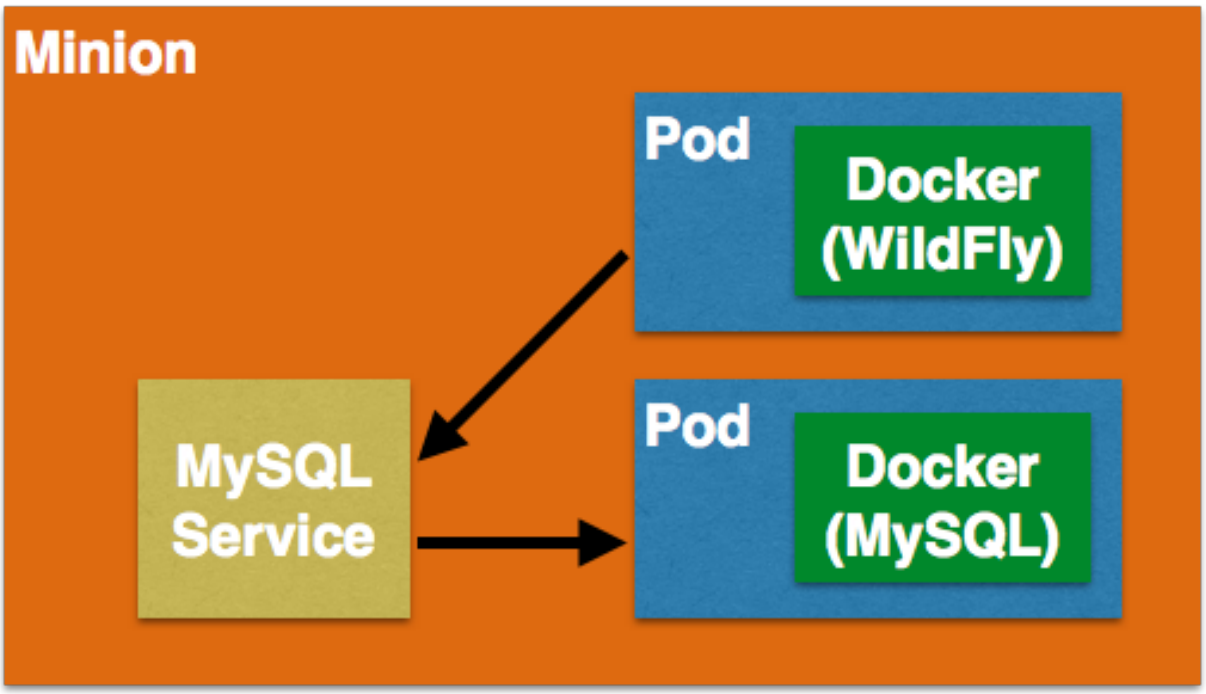


Figure 42. Kubernetes Services

In this case, all the pods are running on a single minion. This is because, that is the default number for a Kubernetes cluster. The pod can very be on another minion if more minions exist in the cluster.

Any Service that a Pod wants to access must be created before the Pod itself, or else the environment variables will not be populated.

Start MySQL Service

1. Start MySQL service as:

```
./cluster/kubectl.sh create -f ../../attendees/kubernetes/mysql-
service.yaml
```

   It uses the following configuration file:

```
id: mysql
kind: Service
apiVersion: v1beta3
metadata:
  name: mysql
  labels:
    name: mysql
    context: docker-k8s-lab
spec:
  ports:
    - port: 3306
  selector:
    name: mysql
  labels:
    name: mysql
```

2. Check that the service is created:

```
> ./cluster/kubectl.sh get se -l context=docker-k8s-lab
NAME      LABELS                                SELECTOR     IP(S)
     PORT(S)
mysql     context=docker-k8s-lab,name=mysql   name=mysql
 10.247.141.208   3306/TCP
```

   Note that the label used during the creation is used to query the service.

   When a Pod is run on a node, the kubelet adds a set of environment variables for each active Service.

It supports both Docker links compatible variables and simpler `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores.

Our service name is `` `mysql'' `` and so `` `MYSQL_SERVICE_HOST `` and `MYSQL_SERVICE_PORT` variables are available to other pods.

Send a Pull Request for #62[45].

Start MySQL Replication Controller

1. Start MySQL replication controller as:

```
> ./cluster/kubectl.sh --v=5 create -f ../../attendees/kubernetes/
mysql.yaml
I0616 19:41:55.441461    8346 defaults.go:174] creating security
 context for container mysql
replicationcontrollers/mysql
```

It uses the following configuration file:

```
kind: ReplicationController
apiVersion: v1beta3
metadata:
  name: mysql
  labels:
    name: mysql
    context: docker-k8s-lab
spec:
  replicas: 1
  selector:
    name: mysql
  template:
    metadata:
      labels:
        name: mysql
        context: docker-k8s-lab
    spec:
      containers:
        - name: mysql
          image: mysql:latest
```

[45] https://github.com/javaee-samples/docker-java/issues/62

```
    env:
      - name: MYSQL_USER
        value: mysql
      - name: MYSQL_PASSWORD
        value: mysql
      - name: MYSQL_DATABASE
        value: sample
      - name: MYSQL_ROOT_PASSWORD
        value: supersecret
    ports:
      - containerPort: 3306
        hostPort: 3306
```

Once again, the ``docker-k8s-lab" label is used. This simplifies querying the created pods later on.

2. Verify MySQL replication controller as:

```
> ./cluster/kubectl.sh get rc -l context=docker-k8s-lab
CONTROLLER   CONTAINER(S)   IMAGE(S)       SELECTOR     REPLICAS
mysql        mysql          mysql:latest   name=mysql   1
```

3. Check the status of MySQL pod as:

```
> ./cluster/kubectl.sh get po -l context=docker-k8s-lab
POD           IP        CONTAINER(S)   IMAGE(S)       HOST
 LABELS                                STATUS    CREATED          MESSAGE
mysql-7lq67                                            10.245.1.3/
 context=docker-k8s-lab,name=mysql   Pending   About a minute
                       mysql          mysql:latest
```

Start WildFly Replication Controller

1. Start WildFly replication controller as:

```
> ./cluster/kubectl.sh --v=5 create -f ../../attendees/kubernetes/
wildfly.yaml
I0616 18:59:00.563099    7849 defaults.go:174] creating security
 context for container wildfly
replicationcontrollers/wildfly
```

It uses the following configuration file:

```
kind: ReplicationController
```

```
apiVersion: v1beta3
metadata:
  name: wildfly
  labels:
    name: wildfly
    context: docker-k8s-lab
spec:
  replicas: 1
  selector:
    name: wildfly-server
  template:
    metadata:
      labels:
        name: wildfly-server
        context: docker-k8s-lab
    spec:
      containers:
        - name: wildfly
          image: arungupta/wildfly-mysql-javaee7:k8s
          ports:
            - containerPort: 8080
              hostPort: 8080
```

2. Verify WildFly replication controller using ``docker-k8s-lab" label as:

```
> ./cluster/kubectl.sh get rc -l context=docker-k8s-lab
CONTROLLER   CONTAINER(S)   IMAGE(S)
 SELECTOR              REPLICAS
mysql        mysql          mysql:latest
 name=mysql            1
wildfly      wildfly        arungupta/wildfly-mysql-javaee7:k8s
 name=wildfly-server   1
```

3. Check the status of WildFly pod as:

```
> ./cluster/kubectl.sh get pod -l context=docker-k8s-lab
POD             IP        CONTAINER(S)   IMAGE(S)
           HOST          LABELS
 STATUS    CREATED      MESSAGE
mysql-7lq67
           10.245.1.3/   context=docker-k8s-lab,name=mysql
 Pending   3 minutes
                          mysql          mysql:latest
wildfly-o0nw6
           10.245.1.3/   context=docker-k8s-lab,name=wildfly-server
 Pending   45 seconds
```

```
                                wildfly         arungupta/wildfly-mysql-
    javaee7:k8s
```

Make sure the status of both WildFly and MySQL pod is changed to ``running''. It will look like:

```
> ./cluster/kubectl.sh get pod -l context=docker-k8s-lab
POD             IP              CONTAINER(S)    IMAGE(S)
          HOST                      LABELS
     STATUS     CREATED       MESSAGE
mysql-7lq67     172.17.0.9
          10.245.1.3/10.245.1.3    context=docker-k8s-lab,name=mysql
     Running    14 minutes
                                mysql           mysql:latest

     Running    10 minutes
wildfly-o0nw6   172.17.0.10
             10.245.1.3/10.245.1.3    context=docker-k8s-lab,name=wildfly-
server    Running    11 minutes
                                wildfly         arungupta/wildfly-mysql-
javaee7:k8s
          Running    26 seconds
```

Takes a while for all the pods to start. It took ~25 minutes on a 16 GB, i7 Mac OS X.

14.4. Access Java EE Application

http://<pod_ip>:8080/employees/resources/employees

14.5. Self-healing Pods

1. Delete the WildFly pod
2. Wait for k8s to restart the pod because of RC

14.6. Application Logs

1. Login to Minion-1 VM:

```
> vagrant ssh minion-1
Last login: Fri Jun  5 23:01:36 2015 from 10.0.2.2
[vagrant@kubernetes-minion-1 ~]$
```

2. Log in as root:

```
[vagrant@kubernetes-minion-1 ~]$ su -
Password:
[root@kubernetes-minion-1 ~]#
```

Default root password for VM images created by Vagrant is ``vagrant''.

3. See the list of Docker containers running on this VM:

```
docker ps
```

4. View WildFly log as:

```
docker logs $(docker ps | grep arungupta/wildfly | awk '{print $1}')
```

5. View MySQL log as:

```
docker logs <CID>
```

14.7. Delete Kubernetes Resources

Individual resources (service, replication controller, or pod) can be deleted by using `delete` command instead of `create` command. Alternatively, all services and replication controllers can be deleted using a label as:

```
kubectl delete -l se,po context=docker-k8s-lab
```

14.8. Stop Kubernetes Cluster

```
> ./cluster/kube-down.sh
Bringing down cluster using provider: vagrant
==> minion-1: Forcing shutdown of VM...
==> minion-1: Destroying VM and associated drives...
==> master: Forcing shutdown of VM...
==> master: Destroying VM and associated drives...
Done
```

14.9. View Logs

Viewing logs is causing https://github.com/GoogleCloudPlatform/kubernetes/issues/9888.

14.10. Debug Kubernetes Master (OPTIONAL)

1. Log in to the master as:

```
> vagrant ssh master
Last login: Thu Jun  4 19:30:04 2015 from 10.0.2.2
[vagrant@kubernetes-master ~]$
```

2. Log in as root:

```
[vagrant@kubernetes-master ~]$ su -
Password:
Last login: Thu Jun  4 19:25:41 UTC 2015
[root@kubernetes-master ~]
```

   Default root password for VM images created by Vagrant is ``vagrant".

3. Check the containers running on master:

```
CONTAINER ID        IMAGE
                                    COMMAND                CREATED
     STATUS              PORTS               NAMES
2b92c80630d5        gcr.io/google_containers/etcd:2.0.9
                                           "/usr/local/bin/
etcd   5 hours ago         Up 5 hours
        k8s_etcd-container.ec4297e5_etcd-server-kubernetes-
master_default_3595ac402f3a17c29dab95f3e0f64c76_56fa3dce
64c375f8030b        gcr.io/google_containers/kube-
apiserver:465b93ab80b30057f9c2ef12f30450c3            "/bin/sh -
c '/usr/lo   5 hours ago         Up 5 hours
          k8s_kube-apiserver.f4e485e1_kube-apiserver-kubernetes-
master_default_c6b19d563bdbcfb0af80b57377ee905c_2f16c239
d7d9d40bd479        gcr.io/google_containers/kube-controller-
manager:572696d43ca87cd1fe0c774bac3a5f4b   "/bin/sh -c '/usr/lo   5
 hours ago         Up 5 hours                                 k8s_kube-
controller-manager.70259e73_kube-controller-manager-kubernetes-
master_default_8f8db766ebc90a00a99244c362284cf1_6eff7640
13251c4df211        gcr.io/google_containers/kube-
scheduler:d1f640dfb379f64daf3ae44286014821            "/bin/sh -
c '/usr/lo   5 hours ago         Up 5 hours
          k8s_kube-scheduler.f53b6329_kube-scheduler-kubernetes-
master_default_1f3b1657f7f1af67ce9f929d78c64695_de632a80
b1809bdabd9c        gcr.io/google_containers/pause:0.8.0
                                                  "/pause"
             5 hours ago         Up 5 hours
```

```
                k8s_POD.e4cc795_kube-apiserver-kubernetes-
master_default_c6b19d563bdbcfb0af80b57377ee905c_767dadb1
280baf845b00        gcr.io/google_containers/pause:0.8.0
                                                  "/pause"
             5 hours ago         Up 5 hours
                k8s_POD.e4cc795_kube-scheduler-kubernetes-
master_default_1f3b1657f7f1af67ce9f929d78c64695_52a4ca74
615a314a35bf        gcr.io/google_containers/pause:0.8.0
                                                 "/pause"
          5 hours ago         Up 5 hours
          k8s_POD.e4cc795_kube-controller-manager-kubernetes-
master_default_8f8db766ebc90a00a99244c362284cf1_97cc1739
7a554eea05f3        gcr.io/google_containers/pause:0.8.0
                                                   "/pause"
               5 hours ago         Up 5 hours
                  k8s_POD.e4cc795_etcd-server-kubernetes-
master_default_3595ac402f3a17c29dab95f3e0f64c76_593b9807
```

15. Common Docker Commands

Here is the list of commonly used Docker commands:

Purpose	Command
Build an image	`docker build --rm=true .`
Install an image	`docker pull ${IMAGE}`
List of installed images	`docker images`
List of installed images (detailed listing)	`docker images --no-trunc`
Remove an image	`docker rmi ${IMAGE_ID}`
Remove all untagged images	`docker rmi $(docker images | grep "^" | awk "{print $3}")`
Remove all images	`docker rm $(docker ps -aq)`
Run a container	`docker run`
List containers	`docker ps`
Stop a container	`docker stop ${CID}`
Stop all running containers	`docker stop  docker ps -q`
Find IP address of the container	`docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${CID}`

Purpose	Command
Attach to a container	`docker attach ${CID}`
Remove a container	`docker rm ${CID}`
Remove all containers	`docker rm $(docker ps -aq)`
Get container id for an image by regular expression	`docker ps | grep arungupta/wildfly | awk '{print $1}'`

16. References

1. JBoss and Docker: http://www.jboss.org/docker/