O'REILLY®

Compliments of NGINX

Docker for Java Developers

Package, Deploy, and Scale with Ease



Arun Gupta

NGINX

Containers without chaos

Flawless application delivery with NGINX Plus


Advanced load balancing and automated routing


On-the-fly reconfiguration for scalable service discovery


Application-aware health checks and container monitoring


Content caching for better availability and performance


Access controls and rate limiting to secure your applications

Learn more at:
nginx.com/microservices

Docker for Java Developers

Package, Deploy, and Scale with Ease

Arun Gupta

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Docker for Java Developers
by Arun Gupta

Copyright © 2016 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Melanie Yarbrough
Copyeditor: Christina Edwards
Proofreader: Colleen Toporek
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2016: First Edition

Revision History for the First Edition
2016-06-08: First Release
2016-09-16: Second Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Docker for Java Developers*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-97442-1

[LSI]

Table of Contents

Foreword v

Preface vii

1. Introduction to Docker 1
Docker Concepts 3
Docker Images and Containers 4
Installing Docker 5
Docker Tools 7
Kubernetes 16
Other Platforms 20

2. Docker and Your First Application 23
Dockerfile 23
Build Your First Docker Image 24
Run Your First Docker Container 27
Push Image to Docker Hub 28
Multicontainer and Multihost Applications 30
Deploying Using Docker Compose and Docker Swarm 31
Deploying Using Kubernetes 40

3. Docker and Java Tooling 45
NetBeans 45
Eclipse 48
IntelliJ IDEA 51
Maven 55

4. Application Deployment. 59
Load Balancing 59
Caching 64
Health Monitoring 70

5. Conclusion. 75
Getting Started with Java and Docker 76

Foreword

Docker is a seeming overnight sensation in application development and delivery. Only a few years ago it was a small open source project like many others. Now Docker is firmly established as a fundamental technology for companies that are moving applications to the cloud, building microservices, adopting continuous integration and delivery, or even simply making traditional-style apps more secure, resilient, and robust.

Not that long ago, Java was an overnight sensation of its own. Java helped bring object-oriented programming to the mainstream, while combining high performance and broad portability of code. Java is now the most popular and widely used programming language.

NGINX is extremely popular in the Docker world and, to a growing degree, the Java world as well. So uniting Docker, Java, and NGINX makes great sense.

Luckily, it's Arun Gupta who has stepped up to bring these technologies together. Arun was a driving force behind the development and early popularity of Java, first at Sun, then at Oracle. He's continued to work at the cutting edge of technology, helping to evangelize both Docker and Kubernetes.

In this ebook, Arun provides a complete introduction and user's guide to Docker for Java developers. Arun explains why Docker is so important, then shows how Java developers can easily develop and deploy their first Java application using popular, Java-friendly tools––including NGINX.

As Arun explains, NGINX and NGINX Plus serve as reverse proxies for Docker-based applications. Java clients depend on NGINX Plus to manage functions critical for app success, such as caching, load balancing, and health monitoring.

NGINX continues to be one of the most frequently downloaded applications on Docker Hub, with more than 10 million pulls to date. We're sure that guided by this excellent ebook, increasing numbers of Java developers will also discover how NGINX and NGINX Plus can make their apps better than ever. We hope you enjoy this ebook and that it helps you succeed as you deploy containers in production.

— Floyd Smith,
Technical Marketing Writer,
NGINX, Inc.

Preface

The Java programming language was created over 20 years ago. It continues to be the most popular and widely used programming language after all these years. The design patterns and antipatterns of Java deployment are well known. The usual steps to deploy a Java application involve using a script that downloads and installs an operating system package such as JDK on a machine—whether physical or virtual. Operating system threads and memory need to be configured, the network needs to be set up, the correct database identified, and several other such requirements need to be configured for the application to work. These applications are typically deployed on a virtual machine (VM). Starting up these VMs is an expensive operation and can take quite a few minutes in most cases. The number of VMs that can run on a host is also limited because the entire operating system needs to be started, and thus there are stringent requirements on the CPU and memory of the host.

Containers provide several benefits over traditional VM-based deployments. Faster startup and deployments, security and network sandboxing, higher density, and portability across different environments are some of the commonly known advantages. They also improve portability across machines and reduce the impedance mismatch between dev, test, and prod environments.

There are efforts like the Open Container Initiative (OCI) that aim to create an industry standard around container formats and runtime. Docker is the first container implementation based on OCI specifications, and is unarguably the most popular container format. Docker nicely complements the Java programming model by allowing you to package your application, including libraries, dependen-

cies, and configuration, as a single artifact. The unit of deployment becomes a Docker image as opposed to a *.war* or *.jar* file. Different components of an application such as an application server, database, or web server can be started as separate containers. All of these containers can then be connected to each other using orchestration frameworks. The entire setup can then be deployed in a variety of operating systems and run as containers.

This book is targeted toward developers who are interested in learning the basic concepts of Docker and commonly used orchestration frameworks around them. The first chapter introduces the basic concepts and terminology of Docker. The second chapter explains, using code samples, how to build and run your first Docker container using Java. The third chapter explains how support for Docker is available in popular developer toolchains. The fourth chapter is a quick summary. The examples in this book use the Java programming language, but the concepts are applicable for anybody interested in getting started with Docker.

Acknowledgments

I would like to express gratitude to the people who made writing this book a fun experience. First and foremost, many thanks to O'Reilly for providing an opportunity to write this book. The team provided excellent support throughout the editing, reviewing, proofreading, and publishing processes. At O'Reilly, Brian Foster believed in the idea and helped launch the project. Nan Barber was thorough and timely with her editing, which made the book fluent and consistent. Thanks also to the rest of the O'Reilly team, some of whom we may not have interacted with directly, but who helped in many other ways. Many thanks to Kunal Pariani to help me understand the simplicity and power of NGINX. Daniel Bryant (@danielbryantuk) and Roland Huß (@ro14nd) did an excellent technical review of the book. This ensured that the book stayed true to its purpose and explained the concepts in the simplest possible ways. A vast amount of information in this book is the result of delivering the Docker for Java Developers workshop all around the world. A huge thanks goes to all the attendees of these workshops whose questions helped clarify my thoughts. Last, but not least, I seek forgiveness from all those who have helped us over the past few months and whose names we have failed to mention.

CHAPTER 1
Introduction to Docker

This chapter introduces the basic concepts and terminology of Docker. You'll also learn about different scheduler frameworks.

The main benefit of the Java programming language is Write Once Run Anywhere, or WORA, as shown in Figure 1-1. This allows Java source code to be compiled to byte code and run on any operating system where a Java virtual machine is available.

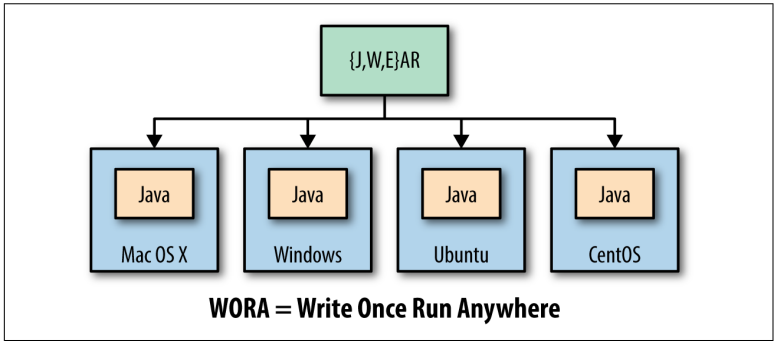


Figure 1-1. Write Once Run Anywhere using Java

Java provides a common API, runtime, and tooling that works across multiple hosts.

Your Java application typically requires an infrastructure such as a specific version of an operating system, an application server, JDK, and a database server. It may need binding to specific ports and requires a certain amount of memory. It may need to tune the configuration files and include multiple other dependencies. The appli-

cation, including its dependencies, and infrastructure together may be referred to as the *application operating system*.

Typically, building, deploying, and running an application requires a script that will download, install, and configure these dependencies. Docker simplifies this process by allowing you to create an *image* that contains your application and infrastructure together, managed as one component. These images are then used to create Docker *containers* that run on the container virtualization platform, which is provided by Docker.

Docker simplifies software delivery by making it easy to build, ship, and run distributed applications. It provides a common runtime API, image format, and toolset for building, shipping, and running containers on Linux. At the time of writing, there is no native support for Docker on Windows and OS X.

Similar to WORA in Java, Docker provides Package Once Deploy Anywhere, or PODA, as shown in Figure 1-2. This allows a Docker image to be created once and deployed on a variety of operating systems where Docker virtualization is available.

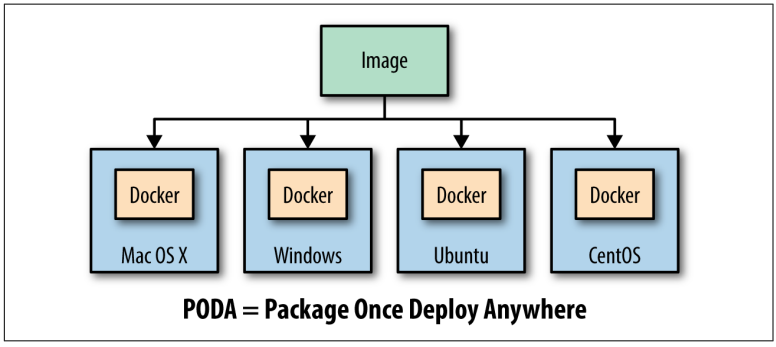


Figure 1-2. Package Once Deploy Anywhere using Docker



PODA is not the same as WORA. A container created using Unix cannot run on Windows and vice versa as the base operating system specified in the Docker image relies on the underlying kernel. However, you can always run a Linux virtual machine (VM) on Windows or a Windows VM on Linux and run your containers that way.

Docker Concepts

Docker simplifies software delivery of distributed applications in three ways:

Build
: Provides tools you can use to create containerized applications. Developers package the application, its dependencies and infrastructure, as read-only templates. These are called the *Docker image*.

Ship
: Allows you to share these applications in a secure and collaborative manner. Docker images are stored, shared, and managed in a *Docker registry*.

  Docker Hub is a publicly available registry. This is the default registry for all images.

Run
: The ability to deploy, manage, and scale these applications. A *Docker container* is a runtime representation of an image. Containers can be run, started, scaled, stopped, moved, and deleted.

A typical developer workflow involves running Docker Engine on a host machine as shown in Figure 1-3. It does the heavy lifting of building images, and runs, distributes, and scales Docker containers. The client is a Docker binary that accepts commands from the user and communicates back and forth with Docker Engine.

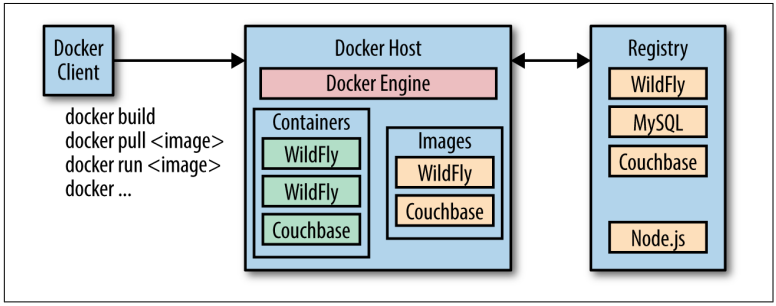


Figure 1-3. Docker architecture

These steps are now explained in detail:

Docker host
: A machine, either physical or virtual, is identified to run Docker Engine.

Configure Docker client
: The Docker client binary is downloaded on a machine and configured to talk to Docker Engine. For development purposes, the client and Docker Engine typically are located on the same machine. Docker Engine could be on a different host in the network as well.

Client downloads or builds an image
: The client can pull a prebuilt image from the preconfigured registry using the `pull` command, create a new image using the `build` command, or run a container using the `run` command.

Docker host downloads the image from the registry
: Docker Engine checks to see if the image already exists on the host. If not, then it downloads the image from the registry. Multiple images can be downloaded from the registry and installed on the host. Each image would represent a different software component. For example, WildFly and Couchbase are downloaded in this case.

Client runs the container
: The new container can be created using the `run` command, which runs the container using the image definition. Multiple containers, either of the same image or different images, run on the Docker host.

Docker Images and Containers

Docker images are read-only templates from which Docker containers are launched. Each image consists of a series of layers. Docker makes use of a *union filesystem* to combine these layers into a single image. Union filesystems allow files and directories of separate filesystems, known as branches, to be transparently overlaid, forming a single coherent filesystem.

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 VM, 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.

Docker images are built on Docker Engine, distributed using the registry, and run as containers.

Multiple versions of an image may be stored in the registry using the format *image-name:tag*. *image-name* is the name of the image and *tag* is a version assigned to the image by the user. By default, the tag value is `latest` and typically refers to the latest release of the image. For example, `jboss/wildfly:latest` is the image name for the WildFly's latest release of the application server. A previous version of the WildFly Docker container can be started with the image `jboss/wildfly:9.0.0.Final`.

Once the image is downloaded from the registry, multiple instances of the container can be started easily using the `run` command.

Installing Docker

Docker can be installed using a native application or using Oracle VirtualBox virtual machine.

Docker for Mac and Windows



Docker for Mac and Windows is the easiest way to get Docker up and running in development. They provide an integrated native application for building, assembling, and shipping applications from these operating systems. These tools are deeply integrated with the native virtualization technologies built into each operating system: Hypervisor framework on Mac OS and Hyper-V on Windows. In

addition, they are deeply integrated with the native networking systems and filesystems. Each tool provides an auto-update capability as well.

Docker for Mac and Windows are available from two channels—stable and beta. The stable channel is aligned with each major release of Docker Engine. The beta channel offers the latest cutting-edge features and comes with an experimental version of Docker Engine. Choose the stable channel if you want a reliable platform to work with. Choose the beta channel if you want to experiment with features that are being worked upon on as they become available, and can weather some instability and bugs.

The tools can be downloaded from the Docker website. Installing this tool requires Mac OS Yosemite 10.10.3 or above or Windows 10 Professional or Enterprise 64-bit. Make sure you read the complete set of requirements for either for Docker for Mac or Docker for Windows.

Instructions for installing Docker on Linux are available at the Docker website.

Docker Toolbox

If your operating system does not meet the requirements of Docker for Mac and Windows, then Docker Toolbox is the fastest way to get up and running with Docker in development. It provides different tools required to get started with Docker.

The complete set of installation instructions is available from the Docker website as well.



Here is a list of the tools included in the Docker Toolbox:

1. Docker Engine or the `docker` binary
2. Docker Machine or the `docker-machine` binary
3. Docker Compose or the `docker-compose` binary
4. Kitematic, the desktop GUI for Docker
5. A preconfigured shell for invoking Docker commands
6. Oracle VirtualBox
7. Boot2docker ISO

Docker Engine, Docker Machine, and Docker Compose are explained in detail in the following sections. Kitematic is a simple application for managing Docker containers on Mac, Linux, and Windows. Oracle VirtualBox is a free and open source hypervisor for x86 systems. This is used by Docker Machine to create a VirtualBox VM and to create, use, and manage a Docker host inside it. A default Docker Machine is created as part of the Docker Toolbox installation. The preconfigured shell is just a terminal where the environment is configured to the default Docker Machine. Boot2Docker ISO is a lightweight Linux distribution based on Tiny Core Linux. It is used by VirtualBox to provision the VM.

Let's look at some tools in detail now.

Docker Tools

Docker Engine

Docker Engine is the central piece of Docker. It is a lightweight runtime that builds and runs your Docker containers. The runtime consists of a daemon that communicates with the Docker client and executes commands to build, ship, and run containers.

Docker Engine uses Linux kernel features like cgroups, kernel namespaces, and a union-capable filesystem. These features allow the containers to share a kernel and run in isolation with their own process ID space, filesystem structure, and network interfaces.

Docker Engine is supported on Linux, Windows, and OS X.

On Linux, it can typically be installed using the native package manager. For example, `yum install docker-engine` will install Docker Engine on CentOS.

On Windows and Mac, it is available as a native application as explained earlier. Alternatively it can be installed using Docker Machine. This is explained in the section "Docker Machine" on page 10.

Swarm mode

An application typically consists of multiple containers. Running all containers on a single Docker host makes that host a single point of failure (SPOF). This is undesirable in any system because the entire system will stop working, and thus your application will not be accessible.

Docker 1.12 introduces a new swarm mode that allows you to natively manage a cluster of Docker Engines called a swarm. This mode allows to run a multicontainer application on multiple hosts. It allows you to create and access a pool of Docker hosts using the full suite of Docker tools. Because swarm mode serves the standard Docker API, any tool that already communicates with a Docker daemon can use a swarm to transparently scale to multiple hosts. This means an application that consists of multiple containers can now be seamlessly deployed to multiple hosts.



Prior to Docker 1.12, multiple Docker Engines can be configured in a cluster using Docker Swarm. This is explained in "Docker Swarm" on page 13.

All engines participating in a cluster are running in swarm mode, see Figure 1-4.

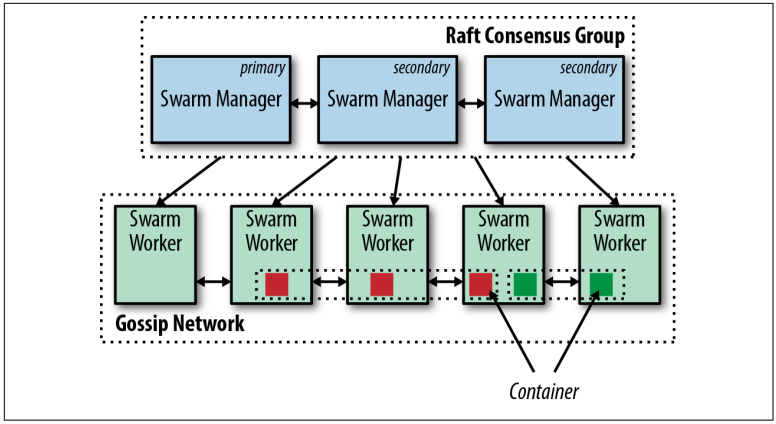


Figure 1-4. Swarm mode architecture

Let's learn about the key components of swarm mode, as shown in Figure 1-4, and how they avoid SPOF:

Node
: A node is an instance of the Docker Engine participating in the swarm. There are *manager* and *worker* nodes.

  An application is submitted to a manager node using a `service` definition. A service typically consist of multple tasks. The manager node dispatches the tasks to worker nodes. By default manager nodes are also worker nodes, but can be configured to be manager-only nodes.

  Multiple managers may be added to the swarm for high availability. Manager nodes elect a single leader to conduct orchestration tasks using the Raft consensus protocol. Worker nodes talk to each other using the Gossip protocol.

  Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm.

Service
: A service is the definition of the tasks to execute on the worker nodes. A task is an image such as `nginx` for an HTTP server or `couchbase` for a database server. Each service in the swarm has a unique name.

The service definition also includes options such as the port where the service will be accessible, number of replicas of the task, and CPU and memory limits. The number of tasks within a service can be dynamically scaled up or down. Each service definition is the *desired* state of the service. The manager ensures that the desired and the *actual* state are reconciled. So, if the service definition asks for three replicas of the task and only two replicas are running then the manager will start another replica of the task.

The manager load balances the running containers. By default, the containers are spread across all manager and worker nodes.

Each service can be published on a port. The manager uses an *ingress load balancer* to make the service accessible outside the swarm. Similarly, the manager uses an internal load balancer to distribute requests between different task replicas of the service.

There are multiple kinds of filters, such as *constraints* and *affinity*, that can be assigned to nodes. A combination of different filters allow for creating your own scheduling algorithm.

An application created using Docker Compose can be targeted to a cluster of Docker Engines running in swarm mode. This allows multiple containers in the application to be distributed across multiple hosts, thus avoiding SPOF. This is explained in "Docker Compose" on page 12.

Multiple containers talk to each other using an *overlay* network. This type of network is created by Docker and supports multihost networking natively out-of-the-box. It allows containers to talk across hosts.

Docker Machine

Docker Machine allows you to create Docker hosts on your computer, on cloud providers, and inside your own data center. It creates servers, installs Docker on them, and then configures the Docker client to talk to them. The `docker-machine` CLI comes with Docker Toolbox and allows you to create and manage machines.

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

- Start, stop, restart container

- Upgrade Docker
- Configure the Docker client to talk to a host

Commonly used commands for Docker Machine are listed in Table 1-1.

Table 1-1. Common commands for Docker Machine

Command	Purpose
`create`	Create a machine
`ls`	List machines
`env`	Display the commands to set up the environment for the Docker client
`stop`	Stop a machine
`rm`	Remove a machine
`ip`	Get the IP address of the machine

The complete set of commands for the `docker-machine` binary can be found using the command `docker-machine --help`.

Docker Machine uses a driver to provision the Docker host on a local network or on a cloud. By default, at the time of writing, a Docker Machine created on a local machine uses `boot2docker` as the operating system. A Docker Machine created on a remote cloud provider uses `Ubuntu LTS` as the operating system.

Installing Docker Toolbox creates a Docker Machine called `default`.

A Docker Machine can be easily created on a local machine as shown here:

```
docker-machine create -d virtualbox my-machine
```

The machine created using this command uses the VirtualBox driver and `my-machine` as the machine's name.

The Docker client can be configured to give commands to the Docker host running on this machine as shown in Example 1-1.

Example 1-1. Configure Docker client for Docker Machine

```
eval $(docker-machine env my-machine)
```

Any commands from the `docker` CLI will now run on this Docker Machine.

Docker Compose

Docker Compose is a tool that allows you to define and run applications with one or more Docker containers. Typically, an application would consist of multiple containers such as one for the web server, another for the application server, and another for the database. With Compose, a multicontainer application can be easily defined in a single file. All the containers required for the application can be then started and managed with a single command.

With Docker Compose, there is no need to write scripts or use any additional tools to start your containers. All the containers are defined in a configuration file using *services*. The `docker-compose` script is used to start, stop, restart, and scale the application and all the services in that application, as well as all the containers within each service.

Commonly used commands for Docker Compose are listed in Table 1-2.

Table 1-2. Common commands for Docker Compose

Command	Purpose
`up`	Create and start containers
`restart`	Restart services
`build`	Build or rebuild services
`scale`	Set number of containers for a service
`stop`	Stop services
`kill`	Kill containers
`logs`	View output from containers
`ps`	List containers

The complete set of commands for the `docker-compose` binary can be found using the command `docker-compose --help`.

The Docker Compose file is typically called *docker-compose.yml*. If you decide to use a different filename, it can be specified using the `-f` option to `docker-compose` script.

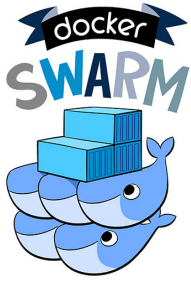
All the services in the Docker Compose file can be started as shown here:

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

This starts all the containers in the service in a background, or detached mode.

Docker Swarm

Before Docker 1.12, multihost Docker was achieved using Docker Swarm.



Docker Swarm had to be explicitly installed in Docker Engine using a separate image. This is in contrast to swarm mode, which is already available in Docker Engine as another feature. Once Docker Swarm is configured, it allows you to access a pool of Docker hosts using the full suite of Docker tools. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts. This means an application that consists of multiple containers can now be seamlessly deployed to multiple hosts.

Figure 1-5 shows the main concepts of Docker Swarm.

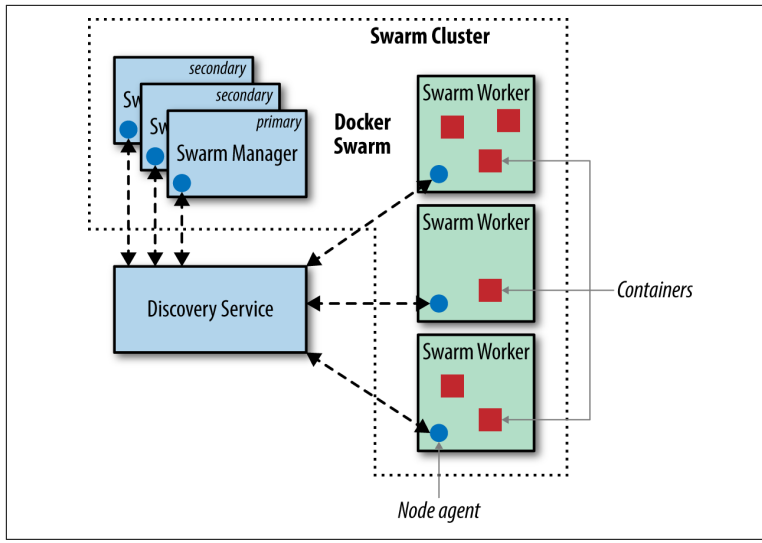


Figure 1-5. Docker Swarm Architecture

Swarm Manager
: Docker Swarm has a manager that is a predefined Docker Host in the cluster and manages the resources in the cluster. It orchestrates and schedules containers in the entire cluster.

  The Swarm manager can be configured with a *primary* instance and multiple *replica* instances for high availability.

Discovery Service
: The Swarm manager talks to a hosted discovery service. This service maintains a list of IPs in the Swarm cluster. Docker Hub hosts a discovery service that can be used during development. In production, this is replaced by other services such as `etcd`, `consul`, or `zookeeper`. You can even use a static file. This is particularly useful if there is no Internet access or you are running in a closed network.

Swarm Worker
: The containers are deployed on nodes that are additional Docker Hosts. Each node must be accessible by the manager. Each node runs a Docker Swarm agent that registers the referenced Docker daemon, monitors it, and updates the discovery services with the node's status.

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

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. This means that a multicontainer application can now be easily deployed on multiple hosts configured through a Docker Swarm cluster.

Docker Machine and Docker Compose are integrated with Docker Swarm. A Docker Machine can participate in the Docker Swarm cluster using `--swarm`, `--swarm-master`, `--swarm-strategy`, `--swarm-host`, and other similar options. This allows you to easily create a Docker Swarm sandbox on your local machine using VirtualBox.

An application created using Docker Compose can be targeted to a Docker Swarm cluster. This allows multiple containers in the application to be distributed across multiple hosts, thus avoiding SPOF.

Multiple containers talk to each other using an *overlay* network. This type of network is created by Docker and supports multihost networking natively out-of-the-box. It allows containers to talk across hosts.

If the containers are targeted to a single host then a *bridge* network is created and it only allows the containers on that host to talk to each other.

Differences Between Docker Swarm and Docker Engine Swarm Mode

There are some key differences between Docker Swarm and Docker Engine swarm mode:

Integrated
: Docker Swarm requires quite a bit of explicit configuration such as downloading and running the swarm container, setting up

the discovery service, and providing CLI options for Docker Machine to be part of Docker Swarm. Swarm mode is just another feature of Docker Engine, for example, like networking.

Secure
: Swarm mode is secure by default, out of the box. For example, each node uses mutually authenticated TLS providing authentication, authorization, and encryption to the communication of every node participating in the swarm. Certificate generation and rotation, reasonable defaults for Public Key Infrastructure (PKI), and support for an external Certificate Authority (CA) are some more features that makes it a compelling feature. All this had to be explicitly set up in Docker Swarm.

Resilient
: One or more managers are configured in a self-healing group using the Raft consensus protocol. The type of node, manager, or worker can be changed at any time. All managers and workers had to be explicitly registered to a discovery service in Docker Swarm.

Pluggability
: Swarm mode is built using SwarmKit. This is a toolkit for orchestrating distributed systems. By default, it comes with *Docker Container Executor*. But it can be easily swapped. This aligns with the *batteries included, but replaceable* approach of Docker. This is not possible using Docker Swarm.

At the time of this writing, both Docker Swarm and Docker Engine swarm mode are available in Docker 1.12. Docker Compose files can be easily deployed to a Docker Swarm cluster. However, support for deploying Docker Compose files into Docker Engine swarm mode is only in experimental release. This will likely be included in a future stable release.

Kubernetes

Kubernetes is an open source orchestration system for managing containerized applications. These can be deployed across multiple hosts. Kubernetes provides basic mechanisms for deployment, maintenance, and scaling of applications. An application's desired state, such as "3 instances of WildFly" or "2 instances of Couchbase,"

can be specified declaratively. And Kubernetes ensures that the state is maintained.

Kubernetes is a container-agnostic system and Docker is one of the container formats supported.

The main application concepts in Kubernetes are explained below:

Pod
: The smallest deployable units that can be created, scheduled, and managed. It's a logical collection of containers that belong to an application. An application would typically consist of multiple pods.

  Each resource in Kubernetes is defined using a configuration file. For example, a Couchbase pod can be defined as shown here:

```
apiVersion: v1
kind: Pod
# labels attached to this pod
metadata:
  name: couchbase-pod
  labels:
    name: couchbase-pod
spec:
  containers:
  - name: couchbase
    # Docker image that will run in this pod
    image: couchbase
    ports:
    - containerPort: 8091
```

  Each pod is assigned a unique IP address in the cluster. The `image` attribute defines the Docker image that will be included in this pod.

Labels
: A label is a key/value pair that is attached to objects, such as pods. Multiple labels can be attached to a resource. Labels can be used to organize and to select subsets of objects. Labels are designed to specify identifying attributes of the object that are meaningful and relevant to the users, but do not directly imply semantics to the core system.

  In the previous example, `metadata.labels` define the labels attached to the pod.

Replication controller
: A replication controller ensures that a specified number of pod replicas are running on worker nodes at all times. It allows both up- and downscaling of the number of replicas. Pods inside a replication controller are re-created when the worker node reboots or otherwise fails.

  A replication controller that creates two instances of a Couchbase pod can be defined as shown here:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: couchbase-controller
spec:
  # Two replicas of the pod to be created
  replicas: 2
  # Identifies the label key and value on the pod
  # that this replication controller is responsible
  # for managing
  selector:
    app: couchbase-rc-pod
  # "Cookie cutter" used for creating new pods when
  # necessary
  template:
    metadata:
      labels:
        # Label key and value on the pod
        # These must match the selector above
        app: couchbase-rc-pod
    spec:
      containers:
      - name: couchbase
        image: couchbase
        ports:
        - containerPort: 8091
```

Service
: Each pod is assigned a unique IP address. If the pod is inside a replication controller, then it is re-created but may be given a different IP address. This makes it difficult for an application server such as WildFly to access a database such as Couchbase using its IP address.

  A service defines a logical set of pods and a policy by which to access them. The IP address assigned to a service does not change over time, and thus can be relied upon by other pods.

Typically the pods belonging to a service are defined by a label `selector`.

For example, a Couchbase service might be defined as shown here:

```
apiVersion: v1
kind: Service
metadata:
  name: couchbase-service
  labels:
    app: couchbase-service-pod
spec:
  ports:
    - port: 8091
  # Label keys and values of the pod started elsewhere
  selector:
    app: couchbase-rc-pod
```

Note that the labels used in `selector` must match the metadata used for the pods created by the replication controller.

So far, we have learned the application concepts of Kubernetes. Let's look at some of the system concepts of Kubernetes, as shown in Figure 1-6.

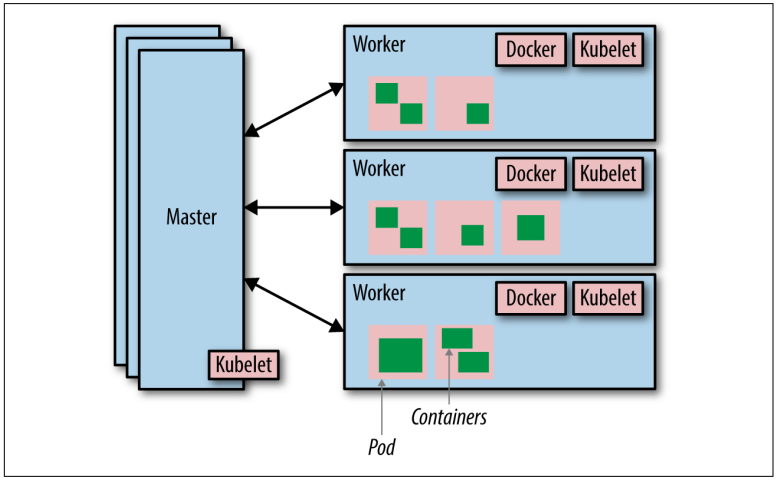


Figure 1-6. Kubernetes architecture

Let's break down the pieces of the Kubernetes architecture:

Cluster
: A Kubernetes cluster is a set of physical or virtual machines and other infrastructure resources that are used to run your applications. The machines that manage the cluster are called *masters* and the machines that run the containers are called *workers*.

Node
: A node is a physical or virtual machine. It has the necessary services to run application containers.

  A master node is the central control point that provides a unified view of the cluster. Multiple masters can be set up to create a high-availability cluster.

  A worker node runs tasks as delegated by the master; it can run one or more pods.

Kubelet
: This is a service running on each node that allows you to run containers and be managed by the master. This service reads container manifests as YAML or JSON files that describe the application. A typical way to provide this manifest is using the configuration file.

A Kubernetes cluster can be started easily on a local machine for development purposes. It can also be started on hosted solutions, turn-key cloud solutions, or custom solutions.

Kubernetes can be easily started on Google Cloud using the following command:

```
curl -sS https://get.k8s.io | bash
```

The same command can be used to start Kubernetes on Amazon Web Services, Azure, and other cloud providers; the only difference is that the environment variable `KUBERNETES_PROVIDER` needs to be set to `aws`.

The Kubernetes Getting Started Guides provide more details on setup.

Other Platforms

Docker Swarm allows multiple containers to run on multiple hosts. Kubernetes provides an alternative to running multicontainer appli-

cations on multiple hosts. This section lists some other platforms that allow you to run multiple containers on multiple hosts.

Apache Mesos

Apache Mesos provides high-level building blocks by abstracting CPU, memory, storage, and other resources from machines (physical or virtual). Multiple applications that use these blocks to provide resource isolation and sharing across distributed applications can run on Mesos.

Marathon is one such framework that provides container orchestration. Docker containers can be easily managed in Marathon. Kubernetes can also be started as a framework on Mesos.

Amazon EC2 Container Service

Amazon ECS is a container management service that makes it easy to run, stop, and manage Docker containers on a cluster of Amazon EC2 instances. Amazon ECS integrates well with rest of the AWS infrastructure and eliminates the need to operate your own cluster or configuration management systems.

Amazon ECS lets you launch and stop container-enabled applications with simple API calls, allows you to get the state of your cluster from a centralized service, and gives you access to many familiar Amazon EC2 features such as security groups, elastic load balancing, and EBS volumes.

Docker containers run on AMIs hosted on EC2. This eliminates the need to operate your own cluster management systems or worry about scaling your management infrastructure.

More details about ECS are available from Amazon's ECS site.

Rancher Labs

Rancher Labs develops software that makes it easy to deploy and manage containers in production. They have two main offerings—Rancher and RancherOS.

Rancher is a container management platform that natively supports and manages your Docker Swarm and Kubernetes clusters. Rancher takes a Linux host, either a physical machine or virtual machine, and makes its CPU, memory, local disk storage, and network connectiv-

ity available on the platform. Users can now choose between Kubernetes and Swarm when they deploy environments. Rancher automatically stands up the cluster, enforces access control policies, and provides a complete UI for managing the cluster.

RancherOS is a barebones operating system built for running containers. Everything else is pulled dynamically through Docker.

Joyent Triton

Triton by Joyent virtualizes the entire data center as a single, elastic Docker host. The Triton data center is built using Solaris Zones but offers an endpoint that serves the Docker remote API. This allows Docker CLI and other tools that can talk to this API to run containers easily.

Triton can be used as a service from Joyent or installed on-premise from Joyent. You can also download the open source version and run it yourself.

Red Hat OpenShift

OpenShift is Red Hat's open source PaaS platform. OpenShift 3 uses Docker and Kubernetes for container orchestration. It provides a holistic and simplistic experience of provisioning, building, and deploying your applications in a self-service fashion.

It provides automated workflows, such as source-to-image (S2I), that takes the source code from version control systems and converts them into ready-to-run, Docker-formatted images. It also integrates with continuous integration and delivery tools, making it an ideal solution for any development team.

CHAPTER 2

Docker and Your First Application

This chapter explains how to build and run your first Docker container using Java.

You'll learn the syntax needed to create Docker images using Dockerfiles and run them as containers. Sharing these images using Docker Hub is explained. Deploying a sample Java EE application using prebuilt Docker images is then covered. This application will consist of an application server and a database container on a single host. The application will be deployed using Docker Compose and Docker Swarm. The same application will also be deployed using Kubernetes.

Dockerfile

Docker builds images by reading instructions from a text document, usually called a *Dockerfile*. This file contains all the commands a user can usually call on the command line to assemble an image. The `docker build` command uses this file and executes all the instructions in this file to create an image.

The `build` command is also passed a context that is used during image creation. This context can be a path on your local filesystem or a URL to a Git repository. The context is processed recursively, which means any subdirectories on the local filesystem path and any submodules of the repository are included.

It's recommended to start with an empty directory in order to keep the build process simple. Any directories or files that need to be included in the image can be added to the context.

A file named *.dockerignore* may be included in the root directory of the context. This file has a newline-separated list of patterns for the files and directories to be excluded from the context.

Docker CLI will send the context to Docker Engine to build the image.

Take a look at the complete list of commands that can be specified in the Dockerfile. The common commands are listed in Table 2-1.

Table 2-1. Common commands for Dockerfiles

Command	Purpose	Example
`FROM`	First noncomment instruction in the Dockerfile	`FROM ubuntu`
`COPY`	Copies multiple source files from the context to the filesystem of the container at the specified path	`COPY .bash_profile / home`
`ENV`	Sets the environment variable	`ENV HOSTNAME=test`
`RUN`	Executes a command	`RUN apt-get update`
`CMD`	Default for an executing container	`CMD ["/bin/echo", "hello world"]`
`EXPOSE`	Informs the network ports that the container will listen on	`EXPOSE 8093`

Any line starting with `#` in the Dockerfile is treated as a comment and not processed.

Build Your First Docker Image

Any valid Dockerfile must have `FROM` as the first noncomment instruction. The argument to `FROM` defines the base image upon which subsequent instructions in the Dockerfile are executed, such as add packages or install JDK. This base image could be for an operating system such as `ubuntu` for the Ubuntu operating system, or `centos` for the CentOS operating system. There are base images available for different operating systems at the Docker website. Additional packages and software can then be installed on these images.

Other images can use this new image in the `FROM` command. It allows you to create a chain where multipurpose base images are used and additional software is installed—for example, the Dockerfile for WildFly. The complete chain for this image is shown here:

```
jboss/wildfly -> jboss/base-jdk:7 -> jboss/base-jdk ->
jboss/base -> centos
```

Often the base image for your application will be a base image that already has some software included in it—for example, the base image for Java. So your application's Dockerfile will have an instruction as shown here:

```
FROM openjdk
```

Each image has a tag associated with it that defines multiple versions of the image. For example, `openjdk:8` is the JDK that has OpenJDK 8 already included in it. Similarly, the `openjdk:9-jre` image has the OpenJDK 9 Java runtime environment (JRE) included in it.

The Dockerfile can also contain a `CMD` instruction. `CMD` provides defaults for executing the container. If multiple `CMD` instructions are listed then only the last `CMD` will take effect. This ensures that the Docker container can run one command, and only one.

Our First Dockerfile

Let's create our first Dockerfile:

1. Create a new directory.

   This directory will contain the Dockerfile and any other artifacts that need to be included in the image.

2. In this directory, create a new text file and name it *Dockerfile*. In this file, enter the following code:

   ```
   FROM openjdk

   CMD ["java", "-version"]
   ```

   Here's a breakdown of the image definition:

   - This Dockerfile uses `openjdk` as the base image. This is a prebuilt image on Docker Hub and can generally be used as the base image for all images that need the Java runtime.
   - This `openjdk` image is built on the Debian Jessie release and uses OpenJDK. By default, at the time of this writing, the

OpenJDK 8 release is used. For example, the OpenJDK 8 JRE may be specified using `8-jre` as the image tag.

- The `CMD` instruction defines the command that needs to run. The command in this case is simply printing the version of the Java interpreter.

  Any other dependencies or libraries, such as JAR files, can be included in this image using the `COPY` instruction. Then a Java command using that JAR file can be invoked by setting the appropriate classpath.

Build Your First Docker Image

Build the image as shown in Example 2-1.

Example 2-1. Build your first Docker image

```
docker build -t hello-java .
Sending build context to Docker daemon 2.048 kB ❶
Step 1 : FROM openjdk ❷
latest: Pulling from library/openjdk
8ad8b3f87b37: Pull complete
751fe39c4d34: Pull complete
ae3b77eefc06: Pull complete
8b357fc28db9: Pull complete
1a614fcb4b1b: Pull complete
1fcd29499236: Pull complete
1df99ed2f401: Pull complete
c4b6cf75aef4: Pull complete
Digest: sha256:581a4afcbbedd8fdf194d597cb5106c1f91463024fb3a49 \
a2d9f025165eb675f
Status: Downloaded newer image for openjdk:latest
 ---> ea40c858f006
Step 2 : CMD java -version ❸
 ---> Running in ea2937cdc268
 ---> 07fdc375a91f
Removing intermediate container ea2937cdc268
Successfully built 07fdc375a91f
```

The output shows:

❶ The `docker build` command builds the image. `-t` provides a name for the image. `hello-java` is the name of the image. `.` is the context for the command. This context is used as the base

directory for copying any files to the image. No files are copied in this case.

❷ The `openjdk` image is downloaded from Docker Hub. It also downloads the complete chain of base images.

❸ The `CMD` instruction adds a new layer to the image.

List the Docker Image

List the images available using the `docker images` command as shown in Example 2-2.

Example 2-2. List of Docker images

```
REPOSITORY    TAG       IMAGE ID        CREATED          SIZE
hello-java    latest    07fdc375a91f    3 minutes ago    643.1 MB
openjdk       latest    ea40c858f006    9 days ago       643.1 MB
```

Our image `hello-java` and the base image `openjdk` are both shown in this list.

Each image can optionally be tagged using the `name:tag` format. This allows multiple versions of the image to be created. By default, an image is assigned the `latest` tag. The total size of the image is shown in the last column.

Run Your First Docker Container

You can run a Docker container by using the `docker run` command and specifying the image name. Let's run our image as shown here:

```
docker run hello-java
```

This shows the following output:

```
openjdk version "1.8.0_102"
OpenJDK Runtime Environment (build 1.8.0_102-8u102-b14.1-1~bpo8
                             +1-b14)
OpenJDK 64-Bit Server VM (build 25.102-b14, mixed mode)
```

The `docker run` command has multiple options that can be specified to customize the container. Multiple options can be combined together. Some of the common options are listed in Table 2-2.

Table 2-2. Common options for the docker run command

Option	Purpose
`-i`	Keep STDIN open even if not attached
`-t`	Allocate a pseudo-TTY
`-d`	Run container in background and print container ID
`--name`	Assign a name to the container
`--rm`	Automatically remove the container when it exits
`-e`	Set environment variable
`-P`	Publish all exposed ports to random ports on the host
`-p`	Publish a container's port(s) to the specified host port
`-m`	Limit the memory

The following command runs the container in the background, gives it a name, and automatically removes it when it exits:

```
docker run --name hello -d --rm hello-java
```

Push Image to Docker Hub

Docker Hub is a software-as-a-service (SaaS) registry service. You can search, manage, push, and pull images to this registry. The images can be manually pushed to the registry using the `docker push` command. Alternatively, they can be built when changes are pushed to a GitHub or Bitbucket repository. User and team collaboration can be facilitated by creating public and private registries.

Table 2-3 lists some of the common Docker CLI commands used to interact with Docker Hub.

Table 2-3. Common Docker Hub commands

Command	Purpose
`login`	Register or log in to a Docker registry
`search`	Search Docker Hub for images
`pull`	Pull an image or a repository from a registry
`push`	Push an image or a repository to a registry
`logout`	Log out from a Docker registry
`tag`	Tag an image into a repository

The `search` and `pull` commands can be invoked without having an account on Docker Hub. However, the `push` command requires you to have an account on Docker Hub, which can be created using the `docker login` command or at the Docker Hub website.

The list of Docker images already downloaded on your Docker host is shown in Example 2-2.

The image `hello-java` does not have a Docker Hub username associated with it, which is needed to push the image to Docker Hub.

The `IMAGE ID` column in the following code block shows a unique identifier assigned to this image. A Docker Hub username can be associated with this image using `docker tag` and the image ID as shown here:

```
docker tag aae0104e1169 arungupta/hello-java:latest
```

The Docker Hub username here is `arungupta`. Make sure to change this to your Docker Hub username. By default, the `latest` tag is assigned to the image. In this case, an explicit `latest` tag is assigned.

Listing the images using the `docker images` command shows the updated output, as shown in Example 2-3.

Example 2-3. List of Docker images with tags

```
REPOSITORY           TAG     IMAGE ID      CREATED         SIZE
arungupta/hello-java latest  07fdc375a91f  4 minutes ago   643.1 MB
hello-java           latest  07fdc375a91f  4 minutes ago   643.1 MB
openjdk              latest  ea40c858f006  9 days ago      643.1 MB
```

The newly tagged image now shows up correctly.

Now push this image to Docker Hub:

```
docker push arungupta/hello-java:latest
```

It shows the output as:

```
The push refers to a repository [docker.io/arungupta/hello-java]
f9297671100e: Pushed
692485f0cd71: Pushed
e7f30faf37db: Pushed
6eeff1cea9fa: Pushed
adaee1fb0269: Pushed
751f5d9ad6db: Pushed
17587239b3df: Pushed
9e63c5bce458: Pushed
```

```
latest: digest: sha256:7dbb6a9951065ca22874f0ac0fd6240fa05680d \
621bf1dcd6a1793e15c09dc70 size: 2000
```

Anyone can download this image using the `docker pull` command. Alternatively, a container using this image can be started using the `docker run` command. This command downloads the image as well, if it does not already exist on your Docker Host.

Multicontainer and Multihost Applications

A typical application consists of multiple components, such as an application server, a web server, and a database server. A Docker container does one thing only, and does it well. Each component from the application thus would be a Docker container. And your application would consist of multiple Docker containers.

These containers need to communicate with each other. For example, the application deployed on the application server will need to know the host address and port on which the database server is running. This can be achieved by using a framework that allows Docker containers to be scheduled. These containers need to be distributed over multiple hosts to avoid SPOF.

The following sections explain how to deploy a simple Java EE application on an application server and access a NoSQL database. WildFly is used as the application server and Couchbase is used as the NoSQL database. The Java EE application will use Java API for RESTful services (JAX-RS) to publish REST endpoints. You'll learn how the application can be deployed using the following orchestration frameworks:

- Docker Compose and Docker Swarm
- Kubernetes

The support for deploying multicontainer applications using Docker Compose on Docker Engine with swarm mode is experimental in Docker 1.12 at the time of this writing, and thus will not be covered here.

Deploying Using Docker Compose and Docker Swarm

To deploy an application we need to define the application using a Compose file, set up the Docker Swarm cluster, and then run our application on the Docker Swarm cluster. Let's look at these steps in more detail.

Defining the Application Using the Compose File

"Docker Compose" on page 12 explained how an application consisting of multiple containers can be defined using a Compose file. The Compose file definition for our Java EE application is defined here:

```
version: '2'
services:
  db: ❶
    container_name: "db" ❻
    image: arungupta/oreilly-couchbase ❷ ❹
    ports: ❸
      - 8091:8091
      - 8092:8092
      - 8093:8093
      - 11210:11210
  web: ❶
    image: arungupta/oreilly-wildfly ❷ ❺
    depends_on: ❼
      - db
    environment: ❽
      - COUCHBASE_URI=db
    ports: ❸
      - 8080:8080
```

This file is available on GitHub. This Compose file has:

❶ Two services named `db` and `web`.

❷ An image name for each service defined using the `image` attribute.

❸ Port forwarding from host to container defined using the `ports` attribute.

❹ The `arungupta/oreilly-couchbase` image, which starts the Couchbase server and configures it using the Couchbase REST API. The image is built using this Dockerfile.

❺ The `arungupta/oreilly-wildfly` image, which starts the WildFly application server and deploys the WAR file built from GitHub. The image is built using this Dockerfile.

This Java EE application uses JAX-RS to publish REST endpoints over JSON documents stored in the Couchbase server.

❻ The Couchbase container is given the name `db` using the `container_name` attribute.

❼ `depends_on` requires the `web` service to start before the `db` service. This will create a dependency between the containers but the application-level health check is still up to the user. For example, the Couchbase database may not be completely ready before the application is deployed in WildFly. In this case, the `web` service needs to be restarted as `docker-compose restart web`.

❽ The `COUCHBASE_URI` environment variable identifies the database container to be used by this `web` container. This environment variable is then used in the application to connect to the Couchbase server using the Couchbase Java SDK API.

Set Up the Docker Swarm Cluster

This application can be targeted to a single host. But as mentioned earlier, this will be a SPOF. Let's set up a Docker Swarm cluster to avoid that.

Setting up a Docker Swarm cluster using Docker Machine and running an application on it typically involves the following steps:

Define discovery service
: All nodes in a Docker Swarm cluster need to register with a discovery service. By default, Docker Hub has a hosted service that returns a token. This token can be used to register a Docker Machine as part of the cluster.

Optionally, a new Docker Machine may be created. This machine can run a discovery service, e.g., using Consul. This machine does not need to be part of the Docker Swarm cluster.

Create Swarm manager and worker nodes
: Create at least one Docker Swarm manager node and one Docker Swarm worker node. Make sure to use `--swarm-discovery` to point to the discovery service. All machines need to use `--swarm` and other related options to ensure they are part of the cluster. The Swarm manager node is identified by the `--swarm-master` option.

Configure Docker CLI
: The Docker CLI is configured to talk to the Docker Swarm cluster using the code shown here:

```
eval $(docker-machine env --swarm <SWARM-MASTER-NAME>)
```

`<SWARM-MASTER-NAME>` is the name of the Docker Machine, that was identified as the Swarm master. Use `docker info` to check the number of nodes in the cluster.

Run application
: Use the Docker Compose file to start the multicontainer application on the Docker Swarm cluster. A sample script to create a three-node Docker Swarm cluster is in Example 2-5.

Example 2-5. Three-node Swarm cluster

```
echo "Creating Docker Machine for Consul ..."
docker-machine \
   create \
   -d virtualbox \
   consul-machine

echo "Starting Consul ..."
docker $(docker-machine config consul-machine) run -d \
         --restart=always \
         -p "8500:8500" \
         -h "consul" \
         progrium/consul -server -bootstrap

echo "Creating Docker Swarm master ..."
docker-machine \
  create \
  -d virtualbox \
  --swarm \
```

```
  --swarm-master \
  --swarm-discovery="consul://$(docker-machine ip \
  consul-machine):8500" \
  --engine-opt="cluster-store=consul://$(docker-machine ip \
  consul-machine):8500" \
  --engine-opt="cluster-advertise=eth1:2376" \
  swarm-master

echo "Creating Docker Swarm worker node 1 ..."
docker-machine \
  create \
  -d virtualbox \
  --swarm \
  --swarm-discovery="consul://$(docker-machine ip \
  consul-machine):8500" \
  --engine-opt="cluster-store=consul://$(docker-machine ip \
  consul-machine):8500" \
  --engine-opt="cluster-advertise=eth1:2376" \
  swarm-node-01

echo "Creating Docker Swarm worker node 2 ..."
docker-machine \
  create \
  -d virtualbox \
  --virtualbox-disk-size "5000" \
  --swarm \
  --swarm-discovery="consul://$(docker-machine ip \
  consul-machine):8500" \
  --engine-opt="cluster-store=consul://$(docker-machine ip \
  consul-machine):8500" \
  --engine-opt="cluster-advertise=eth1:2376" \
  swarm-node-02

echo "Configure to use Docker Swarm cluster ..."
eval "$(docker-machine env --swarm swarm-master)"
```

This script is available on GitHub.

Run Application on Docker Swarm Cluster

After you've configured the Docker CLI for the cluster of Docker Engines, start the services using the Compose file from Example 2-4 and using the command `docker-compose up -d`.

Check the services using `docker-compose ps`. Alternatively, the containers can be seen using `docker ps`. This shows the worker node on which WildFly and Couchbase containers are running. The relevant output is shown in Example 2-6.

Example 2-6. WildFly and Couchbase containers on Docker Swarm nodes

```
IMAGE                                 NAMES
arungupta/oreilly-wildfly:latest      swarm-node-02/hellojavaee_web_1
arungupta/oreilly-couchbase:latest    swarm-node-01/db
```

The output shows that WildFly is running on `swarm-node-02` and Couchbase is running on `swarm-node-01`.

Check the logs using `docker-compose logs`. You may have to restart the `web` service as previously explained.

The application expects a JSON document that represents a book entity as input, which can be added as shown in Example 2-7.

Example 2-7. Add new entity using REST

```
curl -v \
-H "Content-Type: application/json" \
-X POST -d '{
  "isbn": "978-1-4919-1889-0",
  "name": "Minecraft Modding with Forge",
  "cost": 29.99
}' \
http://$(docker-machine ip swarm-node-02):8080/books/resources/book
```

Make sure to change the node to where WildFly is running, which is `swarm-node-02` in this case.

This will display the output in Example 2-8.

Example 2-8. Result from creating a new entity using REST

```
{"name":"Minecraft Modding with Forge","cost":29.99,"id":"1",
"isbn":"978-1-4919-1889-0"}
```

The complete list of entities can be queried, as shown in Example 2-9.

Example 2-9. Query entities using REST

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

If the Docker Compose application is started on a Docker Swarm cluster then a default, application-specific overlay network is cre-

ated. A user-defined overlay network can be created as shown in the following code and assigned to the application using the `networks` attribute in the Docker Compose file:

```
docker network create --driver overlay my-network
```

More details about using Docker Compose with Docker Swarm are available on the Docker website.

Swarm Mode

So far, we've seen how to deploy a multicontainer application to a multihost cluster using Docker Swarm. An alternative way to deploy multicontainer applications on multiple hosts is to use a cluster of Docker Engines in swarm mode. As of Docker 1.12, this requires the experimental version of the Docker client that can be downloaded from the beta channel on the Getting Started with Docker for Mac page.

Setting up a multinode Docker cluster using swarm mode and running a multicontainer application on it typically involves the following steps:

1. Create Swarm manager and worker nodes.

   Create at least one Docker Swarm manager node using `docker swarm init`. By default, a manager node is also a worker node, but it can be configured to be manager-only. Additional manager and worker noders can be created and join the cluster using `docker swarm join`. Each node also need to pass a token that can be obtained from the master using `docker swarm join-token [ manager | worker]`.

   The following ports need to be open between the hosts for communication:

Protocol	Port	Purpose
TCP	2377	Cluster management
TCP and UDP	7946	Among nodes
TCP and UDP	4789	Overlay network traffic

2. Configure the Docker CLI.

The Docker CLI is configured to talk to the manager Docker Engine using the code shown here:

```
eval $(docker-machine env <DOCKER-ENGINE-MANAGER-NAME>)
```

`<DOCKER-ENGINE-MANAGER-NAME>` is the name of the Docker Machine that was identified as the manager. Use `docker info` to check the number of nodes in the cluster.

3. Run the application.

   Deploying a multicontainer application requires the creation of a Distributed Application Bundle (DAB) from a Compose definition. A Dockerfile is used to create an image, and an image is used to create containers. Similarly, a Docker Compose file is used to create a DAB, and *stacks* can be created from the bundle.

   As of Docker 1.12 and Compose 1.8, DAB and Stacks are experimental features. They'll become available in a stable release in a future version.

Deploying a stack on a cluster of Docker Engines in swarm mode creates multiple *services*. A service is a replicated, distributed, and load-balanced container image running in the cluster.

A sample script to create a six-node cluster of Docker Engines in swarm mode using Docker Machine is in Example 2-10.

Example 2-10. Six-node cluster of Docker Engines in swarm mode

```
#!/bin/bash

# swarm mode using Docker Machine

managers=3
workers=3

# create manager machines
echo "======> Creating $managers manager machines ...";
for node in $(seq 1 $managers);
do
  echo "======> Creating manager$node machine ...";
  docker-machine create -d virtualbox manager$node;
done

# create worker machines
echo "======> Creating $workers worker machines ...";
for node in $(seq 1 $workers);
do
```

```
  echo "======> Creating worker$node machine ...";
  docker-machine create -d virtualbox worker$node;
done

# list all machines
docker-machine ls

# initialize swarm mode and create a manager
echo "======> Initializing first swarm manager ..."
docker-machine ssh manager1 "docker swarm init --listen-addr \
$(docker-machine ip manager1) --advertise-addr \
$(docker-machine ip manager1)"

# get manager and worker tokens
export manager_token=`docker-machine ssh manager1 \
"docker swarm join-token manager -q"`
export worker_token=`docker-machine ssh manager1 \
"docker swarm join-token worker -q"`

echo "manager_token: $manager_token"
echo "worker_token: $worker_token"

# other masters join swarm
for node in $(seq 2 $managers);
do
  echo "======> manager$node joining swarm as manager ..."
  docker-machine ssh manager$node \
    "docker swarm join \
    --token $manager_token \
    --listen-addr $(docker-machine ip manager$node) \
    --advertise-addr $(docker-machine ip manager$node) \
    $(docker-machine ip manager1)"
done

# show members of swarm
docker-machine ssh manager1 "docker node ls"

# workers join swarm
for node in $(seq 1 $workers);
do
  echo "======> worker$node joining swarm as worker ..."
  docker-machine ssh worker$node \
  "docker swarm join \
  --token $worker_token \
  --listen-addr $(docker-machine ip worker$node) \
  --advertise-addr $(docker-machine ip worker$node) \
  $(docker-machine ip manager1):2377"
done

# show members of swarm
docker-machine ssh manager1 "docker node ls"
```

This script is available at GitHub.

The `docker-compose` CLI has a command `bundle` to generate a DAB from *docker-compose.yml*. We'll use the *docker-compose.yml* in the *hello-javaee* directory of this GitHub repo to create our DAB:

```
docker-compose bundle
WARNING: Unsupported key 'depends_on' in services.web - ignoring
Wrote bundle to hellojavaee.dab
```

The output shows that *hellojavaee.dab* is the generated DAB file. This bundle can be deployed as a stack using the `docker deploy` command. Note, this `docker deploy` command is only available in the experimental release at this time. This may be included in a stable release in the future:

```
docker deploy hellojavaee
Loading bundle from hellojavaee.dab
Creating network hellojavaee_default
Creating service hellojavaee_db
Creating service hellojavaee_web
```

The output shows that a network and two services are created. A new overlay network that spans across the cluster with the name `hellojavaee_default` is created. One service for the WildFly application server with the name `hellojavaee_db` is created. Another service for the Couchbase server with the name `hellojavaee_web` is created. The network can be verified using `docker network ls`. The list of services can be seen using the `docker service ls` command:

```
ID            NAME             REPLICAS  IMAGE
32y6vbp33klp  hellojavaee_db   1/1       arungupta/oreilly- \
couchbase@sha256:f150fcb9fca5392075c96f1baffc7f893858ba763f \
3c05cf0908ef2613cbf34c
6bqb2fiq3crs  hellojavaee_web  1/1       arungupta/oreilly- \
wildfly@sha256:d567ade7bb82ba8f15a85df0c6d692d85c15ec5a78d8 \
826dfba92756babcb914
```

More details about the `hellojavaee_web` service can be obtained using `docker service inspect hellojavaee_web`.

Each service runs Docker containers as *tasks* on the worker nodes. Each task represents a Docker container, and is the atomic scheduling unit of a swarm. A service can be scaled using the `docker service scale` command. This instructs the manager node to start additional tasks distributed between different worker nodes. Multiple replicas of the tasks are also load-balanced.

The swarm assigns a random port to each service. The exact port assigned to our service can be found using `docker service inspect --format='{{.Endpoint.Ports}}' hellojavaee_web`. It shows the output:

```
[{ tcp 8080 30004}]
```

The output shows that container port 8080 is mapped to port 30004 on the host. The exact port number may be different in your case.

The service is then accessible at *http://$(docker-machine env manager1):30004*. A new entity can now be added as explained in Example 2-7, the result can be seen as explained in Example 2-8, and finally the list of entities can be seen as shown in Example 2-9. Make sure to replace the service address with the new one obtained here.

Deploying Using Kubernetes

To deploy an application using Kubernetes, we need to define the application using a Kubernetes configuration as explained in "Kubernetes" on page 16 and then run our application on the Kubernetes cluster. Let's look at these steps in more detail.

Defining an Application Using a Kubernetes Configuration File

"Kubernetes" on page 16 explained how an application can be defined using a configuration file. This section will use the Java EE application from the previous section and deploy it using Kubernetes.

The Kubernetes configuration for the application is defined in Example 2-11.

Example 2-11. Kubernetes configuration for the application

```
apiVersion: v1
kind: Service ❶
metadata:
  name: couchbase-service
  labels:
    name: couchbase-service-pod
    context: oreilly-docker-book
spec:
  ports:
    - name: admin
```

```
      port: 8091
    - name: api
      port: 8092
    - name: query
      port: 8093
    - name: bucket
      port: 11210
  # label keys and values that must match in order to
  # receive traffic for this service
  selector: ❷
    name: couchbase-rc-pod
    context: oreilly-docker-book
----
apiVersion: v1
kind: ReplicationController ❶
metadata:
  name: couchbase-rc
  labels:
    name: couchbase-rc
    context: oreilly-docker-book
spec:
  replicas: 1
  template:
    metadata:
      name: couchbase-rc-pod
      labels: ❷
        name: couchbase-rc-pod
        context: oreilly-docker-book
    spec:
      containers:
      - name: couchbase-rc-pod
        image: arungupta/oreilly-couchbase:latest ❸ ❹
        ports:
        - containerPort: 8091
        - containerPort: 8092
        - containerPort: 8093
        - containerPort: 11210
----
apiVersion: v1
kind: ReplicationController ❶
metadata:
  name: wildfly-rc
  labels:
    name: wildfly-rc
    context: oreilly-docker-book
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: wildfly
```

```
    context: oreilly-docker-book
spec:
  containers:
  - name: wildfly-rc-pod
    image: arungupta/oreilly-wildfly:latest ❸ ❺
    ports:
    - containerPort: 8080
```

This configuration file has:

❶ One service and two replication controllers. The service is for Couchbase and the replication controllers are for WildFly and Couchbase. A service is required for Couchbase only as this provides a reliable IP address on which the application deployed in WildFly can rely.

❷ The service, which uses the metadata `name: couchbase-rc-pod` and `context: oreilly-docker-book` in the `selector` attribute. This matches the metadata for the pod specified in the `spec.template.metadata.labels` attribute of the Couchbase replication controller.

❸ The image specified using the `image` attribute in the `spec.template.spec.containers.image` attribute.

❹ The `arungupta/oreilly-couchbase:latest` image, which starts the Couchbase server and configures it using the Couchbase REST API. The image is built using this Dockerfile.

❺ The `arungupta/oreilly-wildfly:latest` image, which starts the WildFly application server and deploys the WAR file built from the Java EE NoSQL sample. The image is built using this Dockerfile.

This Java EE application deployed in WildFly uses JAX-RS to publish REST endpoints over the JSON documents stored in the Couchbase server.

Kubernetes provides two primary means of service discovery.

The Java EE application uses Kubernetes Service-style environment variables to talk to the database. This environment variable is then used in the application to connect to the Couchbase server using the Couchbase Java SDK API. Using environment variables for service discovery also means that any service a pod wants to access must be

created before the pod itself, or the environment variables will not be populated.

An alternative, and strongly recommended, way to discover a service is by using DNS server. There is no ordering restriction between a service and a pod when using the DNS server. The DNS server watches the Kubernetes API for new services and creates a set of DNS records for each.

The Couchbase database may not be completely ready before the application is deployed in WildFly. In this case, the WildFly pod needs to be removed, and another one will automatically be created. This ensures the database has started before the application is deployed.

Run the Application on the Kubernetes Cluster

This application can be started as shown here:

```
kubectl create -f app.yml
```

The list of created replication controllers can be seen using the command `kubectl get rc`, which gives the output shown here:

```
NAME           DESIRED   CURRENT   AGE
couchbase-rc   1         1         6m
wildfly-rc     1         1         6m
```

The list of services can be seen using the command `kubectl get svc`, which gives this output:

```
NAME                CLUSTER-IP    EXTERNAL-IP   PORT(S)     AGE
couchbase-service   10.0.54.170   <none>       8091/TCP,... 6m
kubernetes          10.0.0.1      <none>        443/TCP     2h
```

The list of ports is truncated here, but the command will show a complete output. Similarly, the list of pods can be seen using the command `kubectl get po`.

Replication controllers, pods, and services are accessible within the Kubernetes cluster. The WildFly replication controller needs to be exposed as an external service outside the Kubernetes cluster as shown in the following code. This allows us to perform the CRUD operation on the application deployed in WildFly:

```
kubectl expose rc wildfly-rc --target-port=8080 --port=8080
--type=LoadBalancer
```

The complete service description can now be seen using the command `kubectl describe svc wildfly-rc`, which produces this output:

```
Name:     wildfly-rc
Namespace:    default
Labels:     context=oreilly-docker-book,name=wildfly-rc
Selector:   context=oreilly-docker-book,name=wildfly
Type:     LoadBalancer
IP:     10.0.242.62
LoadBalancer Ingress: aca5ae155f86011e5aa71025a2ab0be1-658723113
.us-west-2.elb.amazonaws.com
Port:     <unset> 8080/TCP
NodePort:   <unset> 30591/TCP
Endpoints:    10.244.2.3:8080
Session Affinity: None
Events:
  FirstSeen LastSeen  Count From      SubobjectPath Type...
  --------- --------  ----- ----      ------------- ----...
  1m    1m    1 {service-controller }     Normal    Crea...
  1m    1m    1 {service-controller }     Normal    Crea...
```

The output is truncated, but executing the command will show the complete output. Wait about three minutes for the load balancer to settle. The value of attribute `LoadBalancer Ingress` shows the externally accessible address of the service. The application is now fully deployed and ready to be accessed.

The application expects a JSON document as input that defines a book entity, which can be added as shown in Example 2-7. Make sure to change the IP address obtained from the previous step. Executing this command will display the exact output shown in Example 2-8.

The complete list of entities can be queried as shown in Example 2-9. Make sure to change the IP address obtained in the previous step. Executing the command will display the output shown in Example 2-9.

For more information, check out the Kubernetes documentation.

CHAPTER 3

Docker and Java Tooling

A typical toolchain of Java developers consists of an integrated development environment (IDE) such as NetBeans, Eclipse, or IntelliJ IDEA. These IDEs assist in easily writing and debugging the Java application. They also provide features like syntax highlighting and code completion, making the developer more effective.

These applications are packaged as an archive using a build system such as Maven, Gradle, or Ant. All required dependencies of the project are specified in these scripts. Typically one command can create an archive that can then be used for deployment.

This chapter explains how Docker images and containers can be easily managed in NetBeans, Eclipse, and IntelliJ IDEA. It will then show how a Docker image can be created and a container started using a Maven plug-in.

All the IDEs can configure a Docker host, pull an image, run a container, and build an image. These features are provided so that they integrate with the rest of the workflow within the IDE.

NetBeans

NetBeans is adding integrated support for Docker in their next release. At the time of writing, this feature is currently available in the nightly build, which can be downloaded from the NetBeans website.

A new Docker node is added in the Services tab as shown in Figure 3-1.

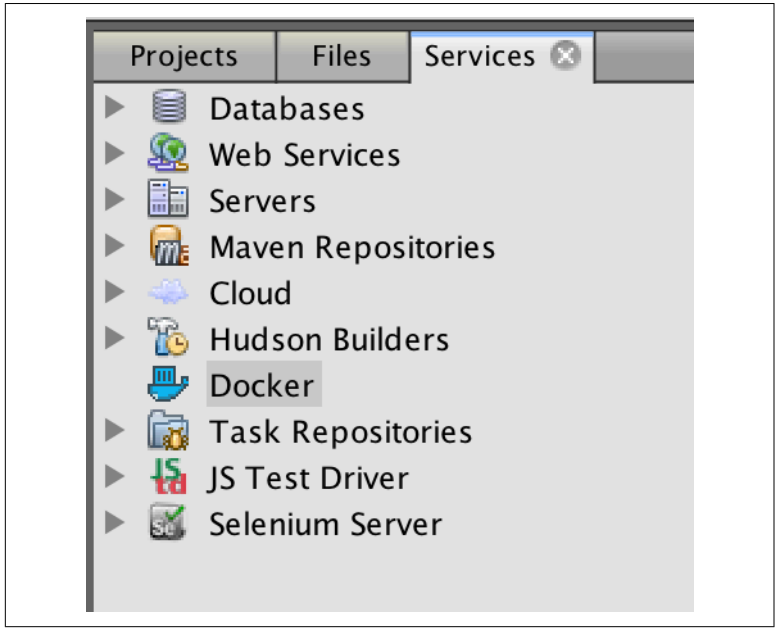


Figure 3-1. Docker node in NetBeans

Configure a Docker Machine

A new Docker host can be configured by specifying the remote API URL of the Docker host as shown in Figure 3-2.

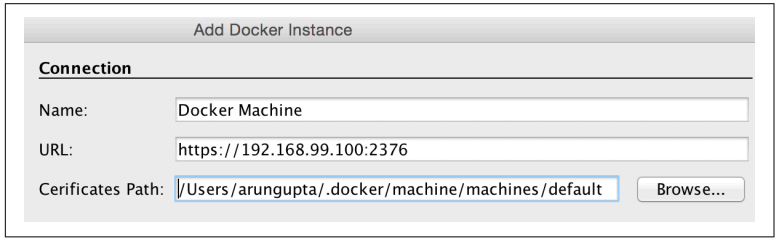


Figure 3-2. Configuring a Docker host in NetBeans

Pull an Image

A new image can be downloaded by right-clicking the newly created connection and selecting "Pull…" as shown in Figure 3-3.

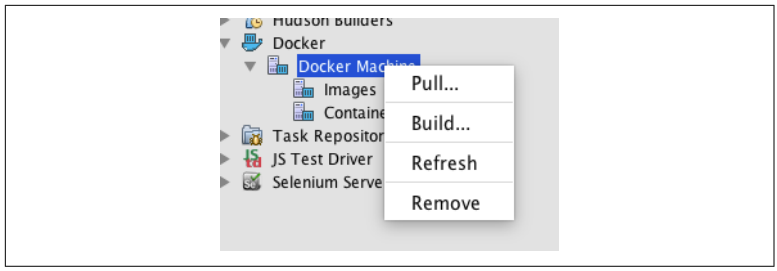


Figure 3-3. Pulling or building an image using NetBeans

The exact image name can then be specified, and this image is downloaded from Docker Hub or the specified registry.

The logs from executing the command are displayed in the Output window.

Build an Image

A new image can be created by selecting "Build…" as shown in Figure 3-3 and specifying the directory name that contains the Dockerfile. This directory serves as the build context for creating the image.

The Dockerfile editor provides syntax coloring to highlight instructions from comments.

This image can then be pushed to a registry (e.g., Docker Hub).

Run a Container

Once an image is downloaded, a container can be started by right-clicking on the image and selecting "Run…" as shown in Figure 3-4.

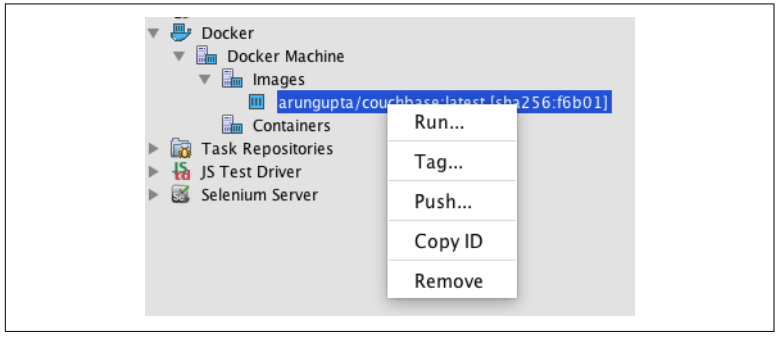


Figure 3-4. Running a Docker container in NetBeans

A new tag can be attached to this image by using the "Tag…" menu.

More details about NetBeans and Docker are available on the NetBeans website.

Eclipse

Docker support in Eclipse is available by installing a plug-in.

Eclipse introduces a new perspective as shown in Figure 3-5. It contains Docker Explorer, which allows you to manage images and containers.

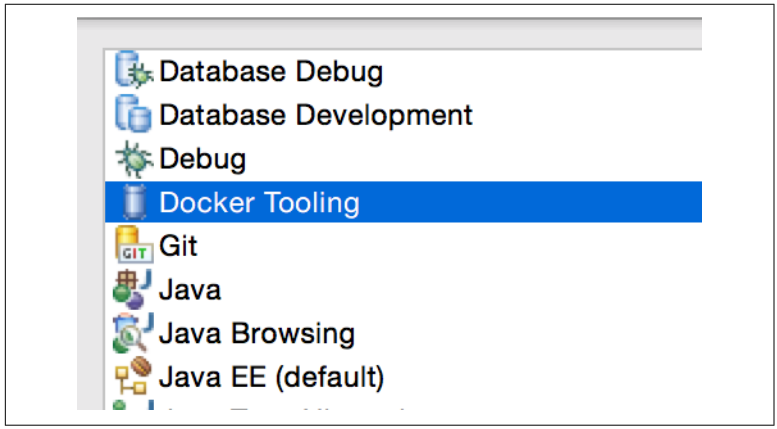


Figure 3-5. Docker perspective in Eclipse

This perspective offers the following views:

Docker Explorer
: A tree view showing Docker connections plus their images and containers.

Docker Images
: A table view of all downloaded images for a selected connection.

Docker Containers
: A table view of all containers for a selected connection (filter enabled).

Console
: Display logs of `stdout/stderr` output from containers; in some instances, this allows input to `stdin`.

Properties
: Detailed information about connections, images, and containers.

Configure Docker Machine

A new Docker host can be configured by specifying either the Unix socket or the remote API URL of the Docker host as shown in Figure 3-6.

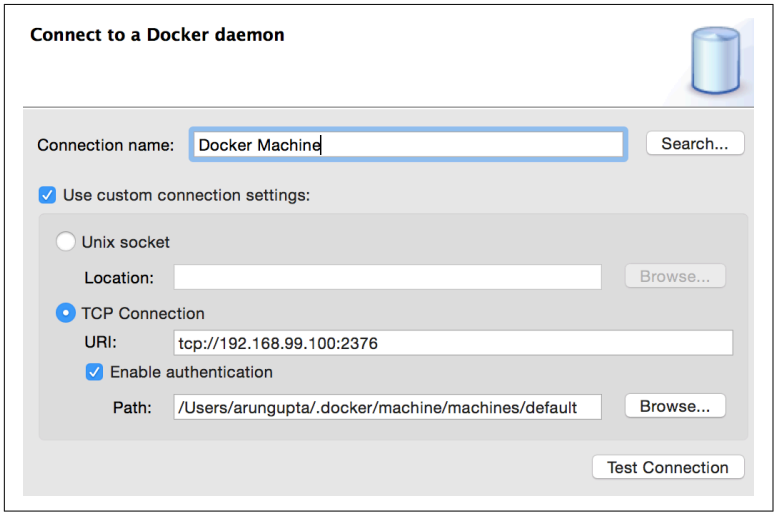


Figure 3-6. Configuring a Docker host in Eclipse

Click the Test Connection button to ensure the credentials are valid.

Pull an Image

A new image can be downloaded by right-clicking on the newly created connection and selecting "Pull…" as shown in Figure 3-7.

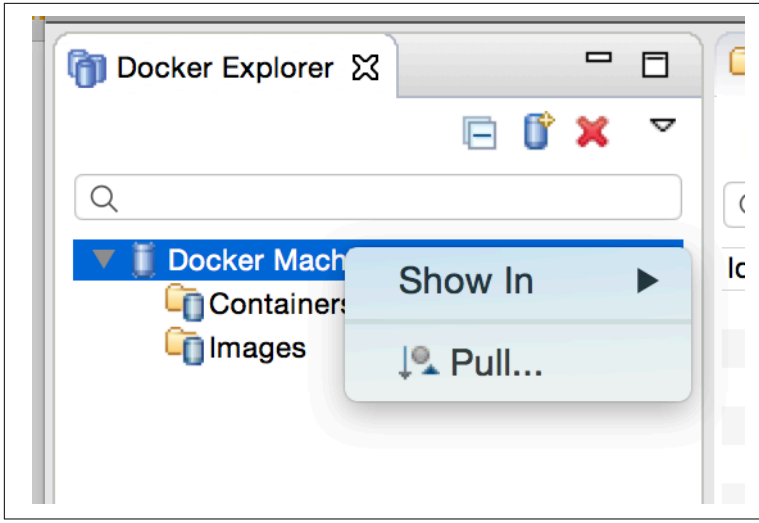


Figure 3-7. Pulling an image using Eclipse

The exact image name can then be specified. This image is downloaded from Docker Hub or the specified registry.

Build an Image

A new image can be created by clicking the Build Image wizard as shown in Figure 3-8.

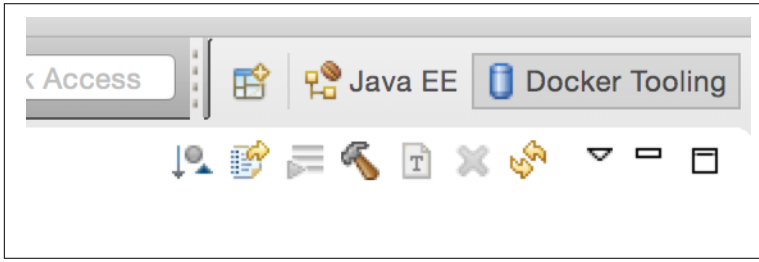


Figure 3-8. Building an image in Eclipse

You need to specify the image name. A directory where the Dockerfile exists or will be generated also needs to be specified. This directory serves as the build context for creating the image.

This image can then be pushed to a registry (e.g., Docker Hub).

Run a Container

Once an image is downloaded, a container can be started by right-clicking on the image and selecting "Run…" as shown in Figure 3-9.

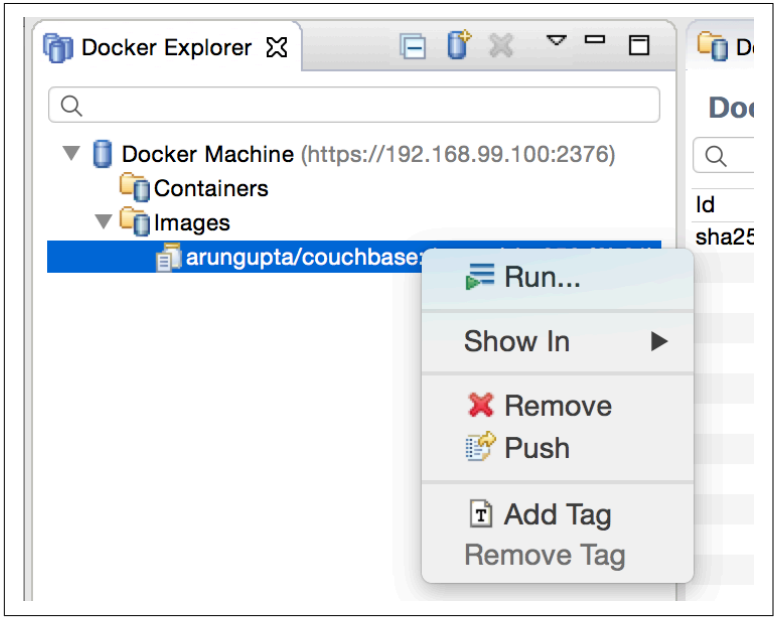


Figure 3-9. Running a Docker container in Eclipse

A new tag can be attached to this image by using the "Add Tag" menu.

Learn more about using Docker and Eclipse from the user guide.

IntelliJ IDEA

Docker support in IntelliJ IDEA is available by installing a plug-in from the plug-in repository. Install the plug-in as shown in Figure 3-10.

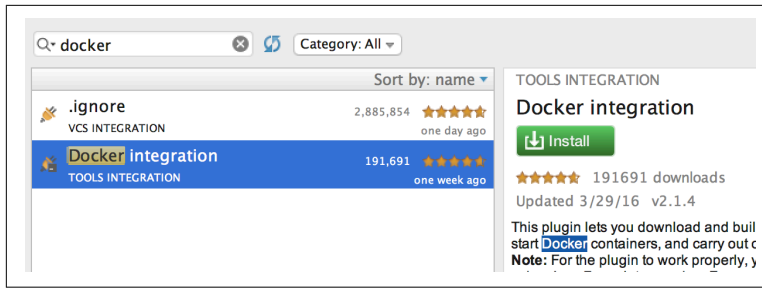


Figure 3-10. Installing the Docker plug-in in IntelliJ

Docker support needs to be configured per project.

Configure Docker Machine

A new Docker host can be configured by specifying the remote API URL of the Docker host (see Figure 3-11):

1. Select "Import credentials from Docker Machine" to show the list of currently running Docker Machines. This also allows you to configure them with a single click.
2. Navigate to View→Tool Windows→Docker to bring up the Docker Tooling window.
3. Connect to the selected remote server.

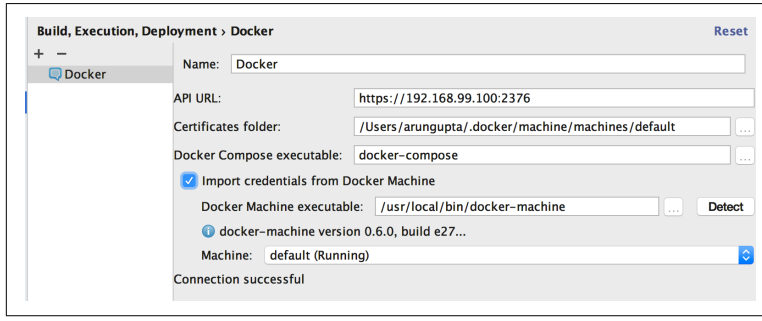


Figure 3-11. Configuring a Docker host in IntelliJ

Pull an Image

A new image can be downloaded by right-clicking on the newly created connection and selecting "Pull image" as shown in Figure 3-12.

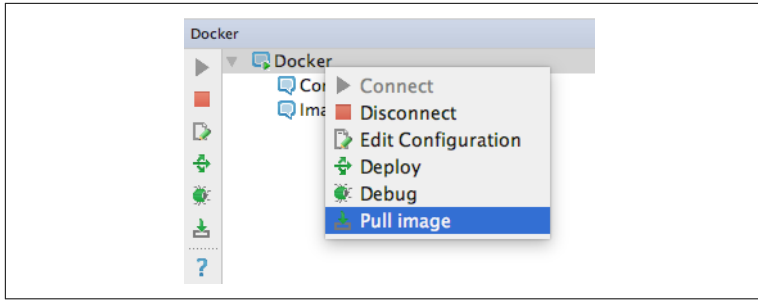


Figure 3-12. Pulling an image using IntelliJ

The exact image name can then be specified. By default, Docker Hub is the registry. A different registry can be specified by clicking "New…" as shown in Figure 3-13.

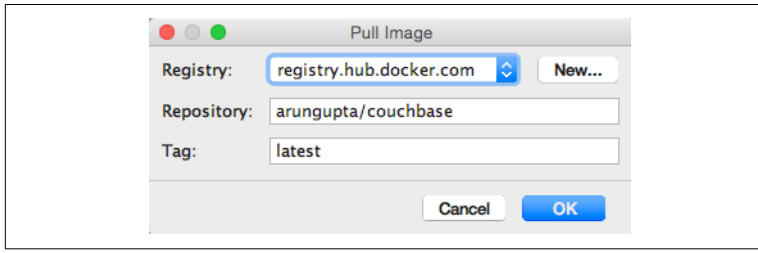


Figure 3-13. Registry name in IntelliJ

This image is downloaded from Docker Hub or the specified registry.

Build an Image

Creating a new image requires us to create a new directory, typically called *docker-dir*. This directory will contain the Dockerfile, the application archive such as the *.war* file, and a file that has settings for running the container. This directory serves as the build context for creating the image.

The application needs to be configured so the archive is generated in this directory. A new Docker deployment configuration can be created so this archive is built before the image as shown in Figure 3-14.

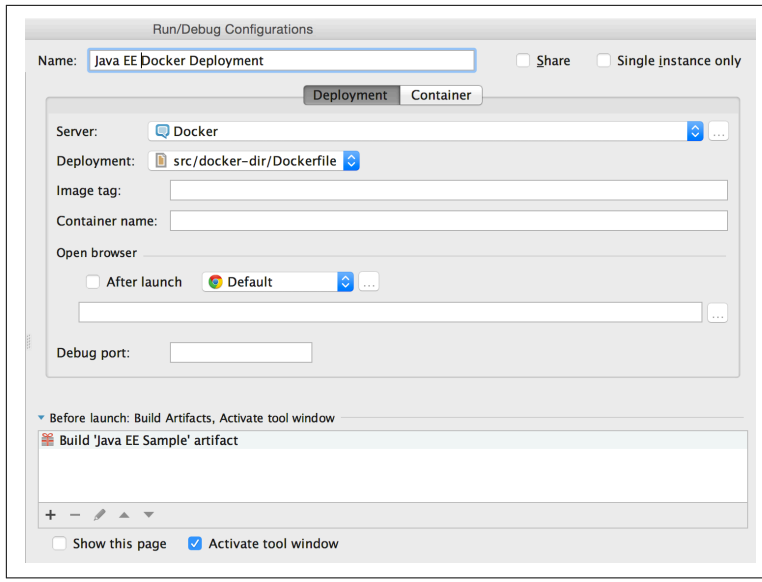


Figure 3-14. Building an image in IntelliJ

You need to specify the image name. A directory where the Dockerfile exists or will be generated also needs to be specified.

This image can then be pushed to a registry (e.g., Docker Hub).

Run a Container

Once an image is downloaded, a container can be started by right-clicking on the image and selecting "Create container" as shown in Figure 3-15.

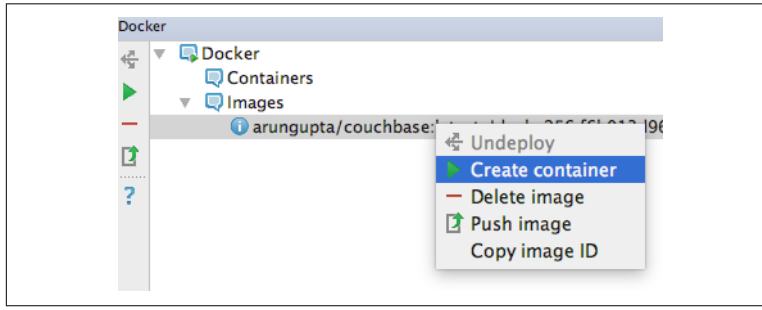


Figure 3-15. Running a Docker container in IntelliJ

More details about the container are available in the output console.

More details about Docker and IntelliJ are available from the IntelliJ IDEA Docker help page.

Maven

Maven is a build tool for Java applications that makes the build process easy by providing a uniform build system. Maven projects are built using a project object model (POM) defined in a file in the main directory of the project. This file, typically called *pom.xml*, contains a set of plug-ins that builds the project.

Each Maven project has a well-defined location for source, tests, resources, and other related files of the project. The process for building and distributing a project is clearly defined. There are predefined *phases* that map to the lifecycle of a project. For example, the `compile` phase will compile the code and the `package` phase will take the compiled code and package it into a distributable format, such as a JAR.

Each plug-in in *pom.xml* offers multiple *goals* and the plug-in associates a goal with one or more phases. For example, `maven-compiler-plugin` is the plug-in that compiles your source. One of the goals offered by this plug-in is `compiler:compile`, and it's tied to the `compile` phase. So the developer can invoke the `mvn compile` command, which invokes the associated goal from the plug-in. Read more about the Maven lifecycle at the Apache website.

There are a few Maven plug-ins that provide goals to manage Docker images and containers:

- fabric8io/docker-maven-plugin
- spotify/docker-maven-plugin
- wouterd/docker-maven-plugin
- alexec/docker-maven-plugin

All plug-ins offer goals that allow Docker lifecycle commands to be tied to a Maven phase. For example, the standard Maven `package` phase packages the compiled code into a JAR or WAR archive. Associating a goal from the plug-in to this phase can take the created archive and package it as a Docker image. Similarly, the standard Maven `install` phase installs the archive in the local repository. Attaching a plug-in's goal to this phase can run the container as well.

The plug-ins listed here offer similar functionality. In this book, we use the fabric8io/docker-maven-plugin plug-in. It is recommended to look at the latest state of plug-ins and pick the one that meets your requirements.

Maven allows *profiles* to be specified using the `<profile>` element in *pom.xml*. These profiles allow you to modify certain elements of the POM. The profiles can be triggered in a variety of ways. It's often convenient to create a separate profile that includes one of the Docker Maven plug-ins and its configuration elements. This allows you to keep the usual build process and still trigger the Docker image and container lifecycle elements by invoking the profile.

Each profile is given a unique name. One of the common ways to invoke the profile is by using `-P<profile-name>` with the `mvn` CLI.

Example 3-1 shows a Maven profile that uses the `fabric8io/docker-maven-plugin` plug-in to create Docker images and run container.

Example 3-1. Docker Maven plug-in profile

```
<profiles>
  <profile>
    <id>docker</id> ❶
    <build>
      <plugins>
        <plugin>
          <groupId>io.fabric8</groupId>
          <artifactId>docker-maven-plugin</artifactId>
          <version>0.14.2</version>
          <configuration> ❷
            <images>
              <image>
                <name>hello-maven</name> ❸
                <build> ❹
                  <from>java</from> ❺
                  <assembly> ❻
                    <descriptorRef>artifact</descriptorRef>
                  </assembly>
                  <cmd>
                  java -cp maven/${project.name}-${project.
                  version}.jar com.oreilly.example.docker.
                  maven.App
                  </cmd> ❼
                </build>
                <run> ❽
                  <wait>
```

```
                <log>Hello</log>
              </wait>
            </run>
          </image>
        </images>
      </configuration>
    </plugin>
  </plugins>
 </build>
</profile>
</profiles>
```

This profile has the following elements:

❶ The profile's name, given by the `<id>` element, appropriately named `docker` in our case.

❷ The plug-in configuration section defined by the `<configuration>` element. The plug-in configuration has a *general* part and a list of *image-specific* configurations, one for each image. With Docker Machine, the general configuration is not required if the environment is configured as explained in Example 1-1.

This particular plug-in has an image-specific configuration for one image only. The image-specific configuration has three parts, described in items 3, 4, and 5.

❸ A general part containing the image's name and alias.

❹ An optional `<build>` configuration defining how the images are built.

❺ `<from>`, which defines the base image for the generated image.

❻ The `<assembly>` element is used to define what needs to be bundled in the image using the standard assembly descriptor format. Using `artifact` ensures that the generated artifact is packaged in the image.

❼ `<cmd>`, which specifies the command that will run when the container is started.

❽ An optional `<run>` configuration defining how the containers should be created and started.

The plug-in used here offers the goals listed in Table 3-1.

Table 3-1. Docker Maven plug-in goals

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

Building this project using `mvn package docker:build -Pdocker` will now compile the source and create the JAR file, create a Dockerfile using the specification from the `<build>` section, pull all the layers needed to build the Docker image, and build the Docker image. The generated image looks like this:

```
REPOSITORY    TAG     IMAGE ID       CREATED          SIZE
hello-maven   latest  3432332a5c80   32 seconds ago   642.9 MB
```

The command `mvn install docker:start -Pdocker` will not only install the generated JAR file in the local Maven repository but will also run the container, which generates this output:

```
[INFO] --- docker-maven-plugin:0.14.2:start (docker:start) @...
[INFO] DOCKER> [hello-maven] : Start container 6fd55878c304
[INFO] DOCKER> [hello-maven] : Waited on log out 'Hello' 505 ms
[INFO]
[INFO] --- docker-maven-plugin:0.14.2:logs (docker:start) @...
6fd558> Hello World!
```

The complete source code for this sample is available on GitHub.

CHAPTER 4

Application Deployment

Application availability, low latency, and health check are some of the most important concerns for successful application delivery. This chapter explains how to load-balance multiple Java application servers running as Docker containers. You'll learn how to enable health monitoring on your applications and provide low latency to the client by caching responses.

You'll learn how NGINX provides round-robin, least-connected, and IP-hash load-balancing algorithms. Weighted load balancing will be discussed to ensure the number of requests are appropriately assigned to each server. Caching of responses from a Java application server to provide low latency will be discussed in detail. Finally, application health monitoring using NGINX directives will be explained.

Load Balancing

Load balancing across multiple Java application servers is a commonly used technique to optimally utilize resources, maximize throughput, and reduce latency. This also helps improve performance, scalability, and reliability of web applications by reducing a single point of failure.

A load balancer (LB) receives a request from the client, such as a browser, and dispatches it to one of the multiple instances of the application server specified in the configuration. If the application server receiving the request is not accessible, the LB can reroute the

request to the next available server. The client does not know about this routing but also does not see the error message because the LB has done its job well.

Dispatching the request to the server can be done using different criteria. For example, LB may choose the next server in a round-robin fashion. Alternatively, it may choose the server based upon a number of active connections.

Some commonly used options for software load balancing are NGINX, Apache HTTP and mod_proxy_balancer, Amazon's Elastic Load Balancer, and HAProxy. This section will explain how to set up NGINX as a load balancer.

All the source code used in this chapter is available on GitHub.

1. The *lb* directory has a simple Compose file (Example 4-1).

   Example 4-1. Compose file for load-balancing WildFly with NGINX

   ```
   version: "2"
   services:
     web:
       image: jboss/wildfly
     lb:
       image: nginx
       ports:
         - 80:80
         - 8080:8080
   ```

   This file has two services: the `web` name for the WildFly container and the `lb` name for the NGINX container. The NGINX service exposes port 80, which is the usual port where the web server is accessible. It also exposes port 8080, which is used as a frontend for WildFly.

2. Start the services with `docker-compose up -d`. This will start one instance of a WildFly container and one instance of an NGINX container.

3. Scale the WildFly service to three replicas with `docker-compose scale web=3`.

4. Build the Java application and package it as a WAR file using the command `mvn -f webapp/pom.xml package`. This command will generate the *webapp/target/webapp.war* file. The web appli-

cation has a simple page that displays the host that is serving the request. In our case, this is going to be the container ID.

5. Copy the WAR file in all three WildFly containers:

```
docker cp webapp/target/webapp.war \
  lb_web_1:/opt/jboss/wildfly/standalone/deployments/ \
  webapp.war
docker cp webapp/target/webapp.war \
  lb_web_2:/opt/jboss/wildfly/standalone/deployments/ \
  webapp.war
docker cp webapp/target/webapp.war \
  lb_web_3:/opt/jboss/wildfly/standalone/deployments/ \
  webapp.war
```

   The `docker cp` command is used to copy the WAR file to each container identified by their name.

6. The NGINX configuration is specified in a configuration file. A sample NGINX configuration file is available at *conf/nginx.conf* in the GitHub repo. It contains the fragment in Example 4-2.

Example 4-2. NGINX configuration file for load balancing

```
upstream wildfly { ❶
    server IP1:8080; ❷ ❸
    server IP2:8080; ❷
    server IP3:8080; ❷
}

server { ❹
    listen 8080; ❺

    location / { ❻
     proxy_pass http://wildfly; ❼
    }
}
```

❶ NGINX functionality is extended by using *modules*. An `upstream` module defines a group of servers where the incoming requests will be load-balanced. By default, the servers are chosen using a round-robin system.

❷ Three servers are defined to equal the number of containers started earlier. The IP address is only specified as a regular expression here and will be updated later.

❸ Each server is listening at port 8080.

❹ Each module is controlled by *directives*, which are divided into simple directives and block directives. `server` is a block directive that defines a virtual server.

❺ A simple directive `listen` is used to indicate that the virtual server will listen on port 8080. By default, the virtual server listens on `0.0.0.0:80`.

❻ `location` is a block directive inside a `server` directive. It defines how NGINX should process the request URI for the parent server. In our case, all requests are processed by this server as the specified location is `/`. The NGINX Beginner's Guide contains more details about NGINX terminology. A more elaborate and comprehensive process on how NGINX processes a request, including selecting the `server` and `location` directive, is defined on the NGINX website.

❼ The `proxy_pass` directive proxies the incoming request to the specified server. In our case, the request will be proxied to a named group of servers.

7. Replace the IP address regular expression in *nginx.conf* with the exact IP address of the containers started earlier:

```
sed -i -e "s/IP1/`docker inspect \
  --format='{{range .NetworkSettings.Networks}} \
  {{.IPAddress}}{{end}}' lb_web_1`/g" conf/nginx.conf
sed -i -e "s/IP2/`docker inspect \
  --format='{{range .NetworkSettings.Networks}} \
  {{.IPAddress}}{{end}}' lb_web_2`/g" conf/nginx.conf
sed -i -e "s/IP3/`docker inspect \
  --format='{{range .NetworkSettings.Networks}}
  {{.IPAddress}}{{end}}' lb_web_3`/g" conf/nginx.conf
```

   This replaces the regular expressions `IP1`, `IP2`, and `IP3` with the IP address of the respective container. The IP address is retrieved using the `docker inspect` command.

8. Copy the NGINX configuration file to the container: `docker cp conf/nginx.conf lb_lb_1:/etc/nginx/nginx.conf`.

9. Reload the NGINX Docker container configuration using `docker exec -it lb_lb_1 nginx -s reload`.

Now, all our setup is done and it's time to see it live!

Invoke the command `curl http://localhost:8080/webapp/index.jsp` and see how a different `Hostname` is printed. This shows the round-robin algorithm of the NGINX load balancer in action.

The log from the NGINX container can be seen using the command `docker logs lb_lb_1`.

As mentioned earlier, the default-load balancing algorithm is round-robin. The following two algorithms are alternative options.

least-connected
: The request is assigned to the server with the least number of active connections. This is useful in situations where some of the requests take longer to complete. As a result, a busy application server will not be overloaded and the requests will be more evenly distributed.

  This can be configured in the `upstream` module:

```
upstream wildfly {
    least_conn;

    server IP1:8080;
    server IP2:8080;
    server IP3:8080;
}
```

ip-hash
: A request is assigned to the server based on applying a hash function to the client's IP address. This is useful if there is a need to tie a client to a particular application server. This allows for `sticky` sessions.

  This can be configured in the `upstream` module:

```
upstream wildfly {
    ip-hash;

    server IP1:8080;
    server IP2:8080;
    server IP3:8080;
}
```

Weighted Load Balancing

Another technique to influence load balancing using NGINX is by assigning server weights. Let's say a specific Docker container is started with more memory and CPU shares; it can be assigned more weight using the `weight` parameter.

Example 4-3 shows how a different weight can be assigned to one of the servers in the `upstream` block.

Example 4-3.

```
upstream wildfly {
    server IP1:8080 weight=4;
    server IP2:8080;
    server IP3:8080;
}
```

Now, for every one request to the server with IP address `IP2` and `IP3`, four requests will be served by the server with IP address `IP1`.

Read more details about load balancing using NGINX on the NGINX website.

Caching

A web application typically consists of some static content such as HTML pages, JavaScript, stylesheets, and images. It may also consist of dynamic pages that are updated after querying a database or making a REST invocation to another endpoint. The static pages don't change often. Even the dynamic pages can have an `expiry` associated with them. This means that the same page can be served again (i.e., not expired), if it is requested within that duration. And if that page is requested outside the duration (i.e., expired), then a new page is generated and served by the server and cached as well.

Adding a caching tier between the client and server will allow the content to be temporarily stored and served directly to the client, without a full round trip from the client to the server. This helps reduce the latency of the web application, as the caching tier is sitting closer to the client than the server. It also reduces the amount of network traffic for pages that are frequently accessed. Overall, it has the perceived effect of improved performance for your application.

Multiple types of cache can sit between the client and server: the client's browser cache, content delivery networks (CDNs), and a reverse proxy sitting in front of the application servers. This section will explain how to set up caching using a reverse proxy with NGINX.

Configure Caching Using NGINX

Caching can be easily configured in NGINX using two directives in the configuration file: `proxy_cache_path` and `proxy_cache`. The `proxy_cache_path` directive sets the path and configuration of the cache, and the `proxy_cache` directive activates it.

The following code fragment, similar to Example 4-2, load-balances a single WildFly server. A real-world use case would have multiple WildFly application servers specified here. It uses `proxy_cache_path` and `proxy_cache` directives to configure a cache for the set of servers:

```
proxy_cache_path /tmp/cache levels=1:2 keys_zone=oreilly_cache:
10m max_size=1G active=15m; ❶

upstream wildfly {
    server IP1:8080;
}

server {
    listen 8080;

    location / {
        proxy_pass http://wildfly;
        proxy_cache oreilly_cache; ❷
        add_header X-Cache-Status $upstream_cache_status; ❸
    }
}
```

❶ `proxy_cache_path` sets the path and the configuration of the cache. The first parameter, `/tmp/cache`, is the directory where all the files will be stored.

The `levels` parameter defines hierarchy levels under the directory. `1:2` creates a two-level directory hierarchy under the specified path. This is the recommended configuration as well, otherwise NGINX will put all files in the same directory, and this will slow down the file access.

Each request coming to NGINX is uniquely identified by a hash key. This key is generated based upon parameters such as protocol, host name, request URI, and HTTP method. `keys_zone` sets a shared memory zone for storing the hash keys. Each zone has a name and a size. In our case, the name is `oreilly_cache`. A 1MB zone can store about 8,000 keys. The specified size is 10MB and so it can store 80,000 keys. The key is stored in the memory and the response is stored in the filesystem path defined earlier. The response is cached during the first request. Subsequent requests for the same key are served by the NGINX cache instead of going to the server.

`max_size` defines the maximum size of the cache. By default, if no size is specified, then the cache will grow to consume all available disk space. When the specified size is exceeded, the least recently used data is removed.

Data in the cache is removed if it is not accessed for the time specified by `inactive`. By default, `inactive` is set to 10 minutes. In our case, if the data is not accessed for 15 minutes then it will be removed independent of its freshness.

❷ The `proxy_cache` directive activates the caching of all content that matches the URL of the parent location block. In our case, all requests served at the root context will be cached. Only certain directories may be specified in the location block and then only those directories will be cached.

❸ `add_header` is an NGINX directive that allows you to add a specified field to a response header. The header name is `X-Cache-Status`. The header value is obtained using the `$upstream_cache_status` embedded variable. The complete set of embedded variables is defined in the NGINX documentation.

More details about configuring the `proxy_cache_path` directive can be found in the NGINX documentation.

Caching in Action

The source code for this section is on GitHub. Let's see how we can achieve caching for Java applications with NGINX. Both components, the Java application and NGINX, are running as Docker containers.

1. The caching directory has a simple Compose file:

```
version: "2"
services:
  web:
    image: jboss/wildfly
  lb:
    image: arungupta/oreilly-nginx-caching
    ports:
      - 80:80
      - 8080:8080
```

   This file is similar to Example 4-1. The only difference is that it uses the `arungupta/oreilly-nginx-caching` image for the `lb` service. This image creates a */tmp/cache* directory in the NGINX container filesystem. It is created using the Dockerfile in the `caching` directory.

2. Start the services using `docker-compose up -d`. This will start one instance of a WildFly container and one instance of our custom NGINX container.

3. Build the Java application and package it as a WAR file using the command `mvn -f webapp/pom.xml package`. This command will generate the *webapp/target/webapp.war* file. The web application has one JSP page, one HTML page, and a few static images.

4. Copy the WAR file in the WildFly container:

```
docker cp webapp/target/webapp.war \
  caching_web_1:/opt/jboss/wildfly/standalone/ \
  deployments/webapp.war
```

5. Update the NGINX configuration file using the IP regular expression:

```
sed -i -e "s/IP/`docker inspect \
  --format='{{range .NetworkSettings.Networks}} \
  {{.IPAddress}}{{end}}' caching_web_1`/g" \
  conf/nginx.conf
```

6. Copy the NGINX configuration file to the container: `docker cp conf/nginx.conf caching_lb_1:/etc/nginx/nginx.conf`.

7. Reload the NGINX Docker container configuration with `docker exec -it caching_lb_1 nginx -s reload`.

Let's see caching in action. The `curl -I` command issues a HTTP HEAD request to a URI and so displays only the HTTP headers in the response message.

1. Use the command `curl -I http://localhost:8080/webapp/index.html` to get the response headers for *index.html* in the web application. The output of this command looks like this:

```
HTTP/1.1 200 OK
Server: nginx/1.11.1
Date: Tue, 23 Aug 2016 08:04:27 GMT
Content-Type: text/html
Content-Length: 148
Connection: keep-alive
Last-Modified: Mon, 22 Aug 2016 19:20:48 GMT
X-Powered-By: Undertow/1
X-Cache-Status: MISS
```

   All the response headers are standard HTTP headers. The first response header of interest is `Server: nginx/1.11.1`. It shows that the request was served with NGINX and also shows the version of the software. The second response header is `X-Cache-Status`. It is a custom header generated as specified in our NGINX configuration file. The value `MISS` indicates that the response was not found in the cache and was fetched from the server. The response might then have been cached.

2. Accessing the HTML document again using the `curl -I http://localhost:8080/webapp/index.html` command shows similar output. The only difference is that the value of `X-Cache-Status` is now `HIT`. This value indicates that the response contains valid, fresh content direct from the cache.

3. Use the command `curl -I http://localhost:8080/webapp/index.jsp` to get the response headers for *index.jsp* in the web application. The set of headers is very similar to the first request for *index.html*. The only difference is the `Set-Cookie` header, which is included because this is a JSP page.

```
HTTP/1.1 200 OK
Server: nginx/1.11.1
Date: Tue, 23 Aug 2016 08:04:34 GMT
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 200
Connection: keep-alive
X-Powered-By: Undertow/1
X-Powered-By: JSP/2.3
```

```
Set-Cookie: JSESSIONID=ajREvwSjyoW8c_BMDvJEPOpJ4BLVH3V_ \
5PLp0eVb.25ef9480ca59; path=/webapp
X-Cache-Status: MISS
```

4. Any subsequent requests for *index.jsp* continue to show the response header `X-Cache-Status: MISS`. This is because any response with the `Set-Cookie` header is not cached by NGINX.
5. Use the command `curl -I http://localhost:8080/webapp/images/couchbase.png` to access a static image. The response headers look like:

```
HTTP/1.1 200 OK
Server: nginx/1.11.1
Date: Tue, 23 Aug 2016 08:04:47 GMT
Content-Type: image/png
Content-Length: 19748
Connection: keep-alive
Last-Modified: Mon, 25 Jan 2016 10:55:32 GMT
X-Powered-By: Undertow/1
X-Cache-Status: MISS
```

   As expected, all the response headers, and their values, are very similar to the first request of *index.html*.
6. Any subsequent requests for the *couchbase.png* static image show the header as `X-Cache-Status: HIT`. This shows that the content is served from the cache.

Here are some other common directives that can configure NGINX caching behavior:

`proxy_cache_methods`
: By default, NGINX caches GET and HEAD requests. This can be used to change the methods that should be cached.

`proxy_cache_min_uses`
: By default, NGINX will cache all requests coming to the specified location. This directive will cache a response only after it has been requested a certain number of times.

`proxy_cache_key`
: This directive can be used to change which information should be used to generate the hash key.

`proxy_cache_valid`
: This directive sets the caching time for different HTTP response codes.

`proxy_ignore_headers`
: Disable processing of certain HTTP headers. For example, generally, Java Server Pages are generated dynamically and the HTTP response contains a `Set-Cookie` header. NGINX does not cache responses with a `Set-Cookie` header. In order for JSPs to be cached, the `proxy_ignore_headers Set-Cookie;` directive can be added in the appropriate `location` block.

Complete details about all NGINX primitives are in the NGINX documentation. More details about caching using NGINX are available in the NGINX caching guide.

Health Monitoring

Monitoring the health of a web application allows you to detect and diagnose application performance problems to maintain an expected level of service. The Java Development Kit provides tools like jconsole and VisualVM that can provide performance and resource consumption of applications running on the Java platform. A wide variety of open source and commercial offerings also exist in this space.

Often, these applications are frontended by a web server like NGINX or Apache HTTP. This section will explore what primitives are provided by NGINX for monitoring the health of your application.

Passive Health Monitoring

Application servers may stop responding or take longer than expected to respond. NGINX provides a built-in application/server-level health check. It can mark the server as unavailable and stop sending requests to it for a defined amount of time. This is defined using the following two parameters in the `server` directive:

`fail_timeout`
: The amount of time within which a specified number of connection attempts must fail for the server to be considered unavailable. This is also the amount of time that NGINX considers the server unavailable after marking it so. The default value for this is 10 seconds.

`max_fails`
: The number of failed attempts that must happen during the specified time for NGINX to consider the server unavailable. The default value for this is 1.

This can be defined in the configuration file as:

```
upstream wildfly {
    server IP1:8080 max_fails=3 fail_timeout=30s;
    server IP2:8080;
    server IP3:8080;
}
```

With this configuration file, if a connection attempt times out or fails at least three times in a 30-second period, NGINX marks the server as unavailable for 30 seconds.

Application Health Monitoring

NGINX Plus adds enterprise-ready features for load balancing, out-of-band application health checks, and advanced monitoring and management to open source NGINX. A comparison of open source NGINX and NGINX Plus is outlined in this feature matrix.

This section will explain how application-level health checks can be performed for Java applications servers running as Docker containers with NGINX Plus.

An NGINX Plus Docker image is not available at Docker, and needs to be explicitly built as explained on the NGINX blog. After the image is built, the updated Compose file to start a WildFly application server and NGINX Plus is:

```
version: "2"
services:
  web:
    image: jboss/wildfly
  lb:
    image: nginx
    ports:
      - 80:80
      - 8080:8080
```

In its simplest form, out-of-band HTTP health checks are performed for different application servers in the `upstream` block. An HTTP probe is sent every few seconds and the server is marked `healthy` or `failed` based upon the HTTP status code.

The updated *nginx.conf* configuration file is:

```
upstream wildfly {
    zone wildfly 64k; ❶

    server IP1:8080;
    server IP2:8080;
    server IP3:8080;
}

server {
    listen 8080;

    location / {
     proxy_pass http://wildfly;
     health_check; ❷
    }
}
```

❶ Defines a memory zone that is shared among worker processes and is used to store the configuration of the server group.

❷ Enable health checks. By default, a request for the `/` URI is made to each server in the upstream group every five seconds. Servers that respond with a well-formed 2xx or 3xx response are considered healthy; otherwise they are marked as failed.

The default behavior of a health check can be overridden by specifying parameters of the `health_check` directive:

```
health_check interval=15 fails=3 passes=3;
```

The `interval` parameter, with a default value of 1 second, sets the interval between two consecutive probes to 10 seconds. The `fails` parameter, with a default value of 1, defines that the server will be considered `unhealthy` after three consecutive failed health checks of a particular server. The `passes` parameter, with a default value of 1, defines that the server will be considered `healthy` after three consecutive passed health checks of a particular server.

Amplify

NGINX Amplify is a comprehensive tool for NGINX monitoring. It helps you proactively analyze and fix problems related to running and scaling NGINX-based web applications. It provides powerful features such as recommendations for performance and security

improvements, a customizable monitoring dashboard, and a configurable alert system.

NGINX Amplify consists of three components:

1. NGINX Amplify agent, which runs on the same host as NGINX
2. NGINX Amplify cloud receiver and backend
3. NGINX Amplify web interface with dashboard

CHAPTER 5
Conclusion

Containers are changing the way Java applications are deployed.

The predominant and classical way to deploy an application is to install a JDK, application server, web server, database, and other required components in your data center, whether on-premises or on the public cloud. The application archive (e.g., *.war*) is then deployed to the application server. This application will then talk to the database and be exposed using the web server. Typically each component of the application can be scaled independently, which allows for optimum resource utilization.

During installation, the exact version of the JAR, the number of threads in the operating system, the correct parameters for the Java runtime, and many other similar parameters must be configured. There are a variety of tools available to get these right, and they've worked well, but each tool comes with its own APIs and workflows.

Docker provides a common runtime API, image format, and toolset for building, shipping, and running containers on both Windows and Linux. This allows you to package your application as Docker containers in an easy and portable way, and to resolve dependencies between them. These applications can run in your on-premises data center or on public clouds. The applications can be packaged using your usual Java toolchain such as NetBeans and Maven. There are Docker images available for JDK, application servers, databases, web servers, and other components.

Big players like Google (Kubernetes, Container Engine), Microsoft (Azure), Red Hat (OpenShift), and Amazon (Container Service) have been continually evolving their container story. Companies that have been doing Linux containers for quite some time such as Mesosphere and Joyent are aligning with Docker. New frameworks and platforms such as Rancher that provide support for both Docker Swarm and Kubernetes are beginning to grow. The Open Container Initiative is working on creating open industry standards around container formats and runtimes.

Even though not a strict requirement, containers simplify microservices deployment. There are some common principles behind both: single responsibility principle, isolation, explicitly published interface, and ability to scale.

There is a lot of excitement and an equal amount of work happening. This is not just hype—Docker provides real and credible benefits of simplicity and portability.

By no means is the work done; this is only a start and the landscape is evolving rapidly. Docker doesn't just simplify packaging your Java application; once you buy into the vision, the concepts and tools can be applied to other languages and projects as well.

Getting Started with Java and Docker

Docker Labs is a self-paced hands-on introduction to packaging and deploying your Java applications using Docker.

You can take the blue pill and continue to deploy applications in the classical way. Or you can take the red pill and get ready to explore the world of containers!

About the Author

Arun Gupta is Vice President of Developer Advocacy at Couchbase. He built and led developer communities for over 10 years at Sun Microsystems, Oracle, and Red Hat, and has deep expertise in leading cross-functional teams to develop and execute strategy, planning, content, marketing campaigns, and programs. Arun is a founding member of the Java EE team, has authored more than 2,000 blog posts on technology, and has extensive speaking experience in more than 40 countries on a myriad of topics. He has been a JavaOne Rock Star for three years in a row, and also founded the US chapter of Devoxx4Kids and continues to promote technology education among children. An author of a best-selling book, an avid runner, a globe trotter, a Java Champion, a JUG leader, and a Docker Captain, he is easily accessible on Twitter at @arungupta.