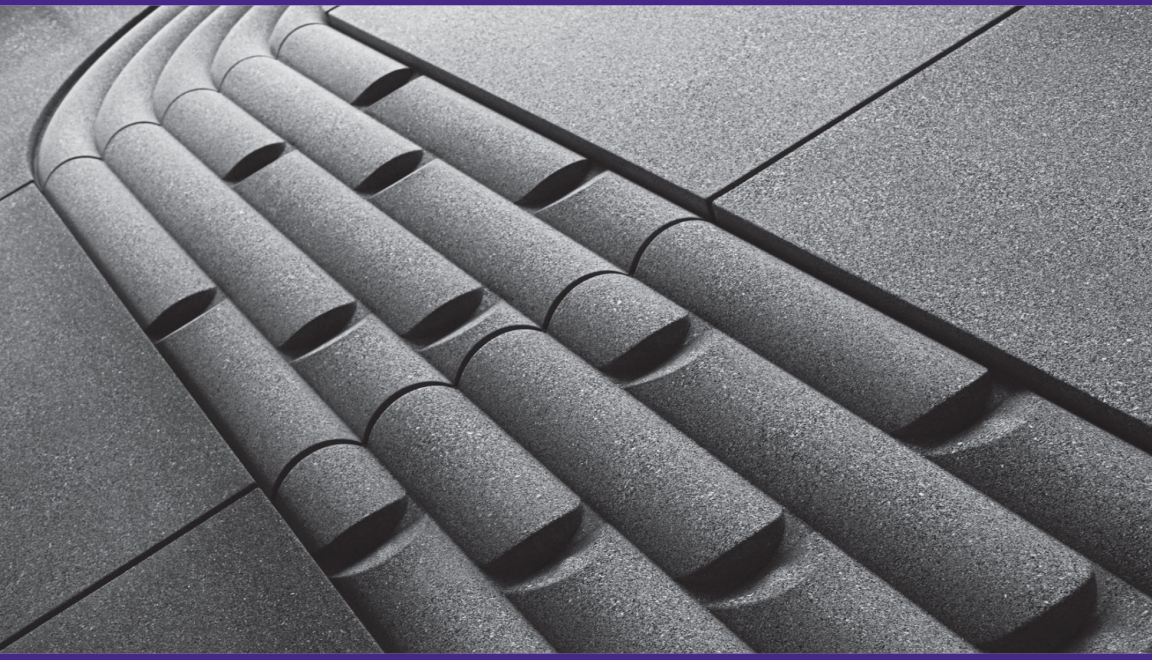
O'REILLY®

Compliments of NGINX

Containerizing Continuous Delivery in Java

Docker Integration for Build Pipelines and Application Architecture



Daniel Bryant

Containerizing Continuous Delivery in Java

Docker Integration for Build Pipelines and Application Architecture

Daniel Bryant

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

Containerizing Continuous Delivery in Java
by Daniel Bryant

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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian Foster
Production Editor: Nicholas Adams
Copyeditor: Gillian McGarvey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2017: First Edition

Revision History for the First Edition
2017-01-13: First Release

978-1-491-97960-0

[LSI]

Table of Contents

Foreword

Despite being around for more than two decades, Java is one of the most widely used programming languages today. Though it's often associated with legacy, monolithic J2EE apps, when combined with Docker and continuous delivery (CD) practices, tried-and-true Java becomes a valuable tool in a modern microservices-based application development environment.

In this report, author Daniel Bryant covers everything you need to know as you containerize your Java apps, fit them into CD pipelines, and monitor them in production. Bryant provides detailed step-by-step instructions with screenshots to help make the complex process as smooth as possible. By the end of the report, you will be well on your way to self-service deployments of containerized Java apps.

One of the biggest challenges with containers in production—Java or otherwise—is interconnecting them. With monolithic apps, everything resides on the same server, but microservices and containerizing apps introduces a new component you have to design and maintain: the network they use to communicate. And with CD, the containers are continuously being updated and changing location, further complicating things.

To help you solve these networking and service discovery problems, we've created a series of three reference architectures for microservices, powered by NGINX Plus. We've made it easy to connect, secure, and scale your distributed application by simply adding a few lines to a Dockerfile to embed NGINX Plus in your containers. Each container can then independently discover, route, and load balance to other services without the need for any middleware. This greatly speeds up communications between services and enables fast

SSL/TLS connections. To learn more about our microservice reference architectures, please visit *nginx.com/mra*.

We hope that you enjoy this report and that it helps you use Java in exciting new ways.

— Faisal Memon,
Product Marketer, NGINX Inc.

Introduction

> "It is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself."
>
> —C. Megginson, interpreting Charles Darwin

Java is a programming language with an impressive history. Not many other languages can claim 20-plus years of active use, and the fact that the platform is still evolving to adapt to modern hardware and developer requirements will only add to this longevity. Many things have changed during the past 20 years in the IT industry, and the latest trend of packaging and deploying applications in containers is causing a seismic shift in the way software is developed. Since the open source release of Docker in March 2013, developers have rapidly become captivated with the promises that accompany this technology. The combination of a well-established language like Java and an emerging technology such as Docker may appear at first glance like a strange mix. But, it is not. Packaging applications that were developed using the production-ready, battle-hardened Java language within flexible container technology can bring the best of both worlds to any project.

As it is with the introduction of any new paradigm or technology, some things must change. The process of containerizing Java applications is no exception, especially when an organization or development team aspires to practice continuous delivery. The emergence of practices like continuous integration and continuous delivery is currently transforming the software development industry, where short technical feedback cycles, improved functional/nonfunctional validation, and rapid deployment are becoming business enablers. If

we combine this with public cloud technology and programmable infrastructure, we have a potent recipe for deploying applications that can scale cost-effectively and on demand. Because not everything can be covered in a book of this size, the primary aim is to provide the reader with a practical guide to continuously delivering Java applications deployed within Docker containers. Let's get started!

Acknowledgments

I would like to express gratitude to the people who made writing this book not only possible, but also a fun experience! First and foremost, I would like to thank Brian Foster for providing the initial opportunity, and also extend this thanks to the entire O'Reilly team for providing excellent support throughout the writing, editing, reviewing, and publishing process. Thanks also to Arun Gupta for his very useful technical review, and to Nic Jackson (and the entire notonthehighstreet.com technical team), Tareq Abedrabbo, and the OpenCredo and SpectoLabs teams for support and guidance throughout the writing process. Finally, I would like to thank the many people of the IT community that reach out to me at conferences, meetups, and via my writing—your continued sharing of knowledge, feedback, and questions are the reason I enjoy what I do so much!

CHAPTER 1
Continuous Delivery Basics

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
>
> —First principle of the Agile Manifesto

Continuous delivery (CD) is fundamentally a set of practices and disciplines in which software delivery teams produce valuable and robust software in short cycles. Care is taken to ensure that functionality is added in small increments and that the software can be reliably released at any time. This maximizes the opportunity for rapid feedback and learning. In 2010, Jez Humble and Dave Farley published their seminal book *Continuous Delivery* (Addison-Wesley), which collated their experiences working on software delivery projects around the world, and this publication is still the go-to reference for CD. The book contains a very valuable collection of techniques, methodologies, and advice from the perspective of both technology and organizations. One of the core recommended technical practices of CD is the creation of a build pipeline, through which any candidate change to the software being delivered is built, integrated, tested, and validated before being determining that it is ready for deployment to a production environment.

Continuous Delivery with a Java Build Pipeline

Figure 1-1 demonstrates a typical continuous delivery build pipeline for a Java-based monolithic application. The first step of the process of CD is *continuous integration* (CI). Code that is created on a developer's laptop is continually committed (integrated) into a

shared version control repository, and automatically begins its journey through the entire pipeline.

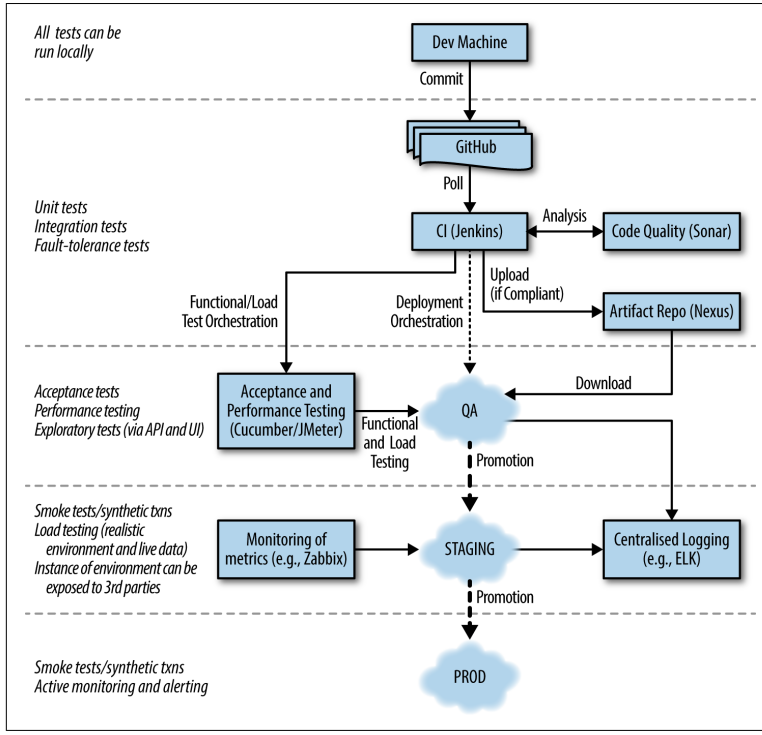


Figure 1-1. A typical Java application continuous delivery pipeline

The primary goal of the build pipeline is to prove that changes are production-ready. A code of configuration modification can fail at any stage of the pipeline, and this change will accordingly be rejected and not marked as ready for deployment to production.

Feature Branching and CI

It's worth noting that many people argue that code must be continually integrated (at least daily) into a trunk/master/development branch if one is truly practicing CI. An associated antipattern for this practice is long-lived feature branches that are not continually merged with the main codebase, and which often result in "merge hell" when they are integrated. Steve Smith has contributed several informative articles on this subject.

Initially, the software application to which a code change is being applied is built and tested in isolation, and some form of code quality analysis may (should) also be applied, perhaps using a tool like Sonar Qube. Code that successfully passes the initial unit and component tests and the code-quality metrics moves to the right in the pipeline, and is exercised within a larger integrated context. Ultimately, code that has been fully validated emerges from the pipeline and is marked as ready for deployment into production. Some organizations automatically deploy applications that have successfully navigated the build pipeline and passed all quality checks—this is known as continuous delivery.

Maximize Feedback: Automate All the Things

The build pipeline must provide rapid feedback for the development team in order to be useful within their daily work cycles. Accordingly, automation is used extensively (with the goal of 100% automation), as this provides a reliable, repeatable, and error-free approach to processes. The following items should be automated:

- Software compilation and code-quality static analysis
- Functional testing, including unit, component, integration, and end-to-end
- Provisioning of all environments, including the integration of logging, monitoring, and alerting hooks
- Deployment of software artifacts to all environments, including production
- Data store migrations
- System testing, including nonfunctional requirements like fault tolerance, performance, and security
- Tracking and auditing of change history

The Impact of Containers on Continuous Delivery

Java developers will look at Figure 1-1 and recognize that in a typical Java build pipeline, two things are consistent regardless of the application framework (like Java EE or Spring) being used. The first is that Java applications are often packaged as Java Archives (JAR) or web application archive (WAR) deployment artifacts, and the second is that these artifacts are typically stored in an artifact reposi-

tory such as Nexus, Apache Achiva, or Artificatory. This changes when developing Java applications that will ultimately be deployed within container technology like Docker.

Docker Isn't the Only Container Technology

It is worth noting that although this book focuses on the combination of Docker and Java, Docker is not the only container technology available. Indeed, many people (including the author at times) conflate the words "container" and "Docker," but this is not correct. Other container technologies include CoreOS's rkt, Canonical's LXD, and Microsoft's Windows Server Containers. Much of the high-level CD advice and patterns presented within this book will apply to all of these container technologies.

Additional information for teams developing on a Windows platform can be found in the O'Reilly mini-book *Continuous Delivery with Windows and .NET* by Chris O'Dell and Matthew Skelton. Although this is .NET-focused, there are many useful chapters on Windows-related CD topics.

Saying that a JAR file is similar to a Docker image is not a fair comparison. Indeed, it is somewhat analogous to comparing an engine and a car—and following this analogy, Java developers are much more familiar with the skills required for building engines. Due to the scope of this book, an explanation of the basics of Docker is not included, but the reader new to containerization and Docker is strongly encouraged to read Arun Gupta's *Docker for Java Developers* to understand the fundamentals, and Adrian Mouat's *Using Docker* for a deeper-dive into the technology.

Because Docker images are run as containers within a host Linux operating system, they typically include some form of operating system (OS), from lightweight ones like Alpine Linux or Debian Jessie, to a fully-functional Linux distribution like Ubuntu, CentOS, or RHEL. The exception of running on OS within a Docker container is the "scratch" base image that can be used when deploying statically compiled binaries (e.g., when developing applications) in Golang. Regardless of the OS chosen, many languages, including Java, also require the packaging of a platform runtime that must be considered (e.g., when deploying Java applications, a JVM must also be installed and running within the container).

The packaging of a Java application within a container that includes a full OS often brings more flexibility, allowing, for example, the installation of OS utilities for debugging and diagnostics. However, with great power comes great responsibility (and increased risk). Packaging an OS within our build artifacts increases a developer's area of responsibility, such as the security attack surface. Introducing Docker into the technology stack also means that additional build artifacts now have to be managed within version control, such as the Dockerfile, which specifies how a container image should be built.

One of the core tenets of CD is testing in a production-like environment as soon as is practical, and container technology can make this easier in comparison with more traditional deployment fabrics like bare metal, OpenStack, or even cloud platforms. Docker-based orchestration frameworks that are used as a deployment fabric within a production environment, such as Docker Swarm, Kubernetes (nanokube), and Mesos (minimesos), can typically be spun up on a developer's laptop or an inexpensive local or cloud instance.

The increase in complexity of packaging and running Java applications within containers can largely be mitigated with an improved build pipeline—which we will be looking at within this book—and also an increased collaboration between developers and operators. This is one of the founding principles with DevOps movement.

CHAPTER 2

Continuous Delivery of Java Applications with Docker

> "Integrating containers into a continuous-delivery pipeline is far from easy. Along with the benefits Docker brings, there are also challenges both technological and process-related."
>
> —Viktor Farcic, author of DevOps 2.0

This chapter examines the impact of introducing Docker into a Java application build pipeline. In addition to looking at the theoretical issues, practical examples will also be provided, with the goal being to enable a Java developer already familiar with the basic concepts of Docker to start creating an appropriate pipeline. For Java developers looking for an introductory explanation of Docker, Arun Gupta's *Docker for Java Developers* is an excellent primer.

Adding Docker to the Build Pipeline

Introducing Docker into a typical Java application build pipeline will primarily impact four locations within the pipeline. The figure below shows an extended example of the earlier Java application build pipeline with the locations of change highlighted (Figure 2-1).

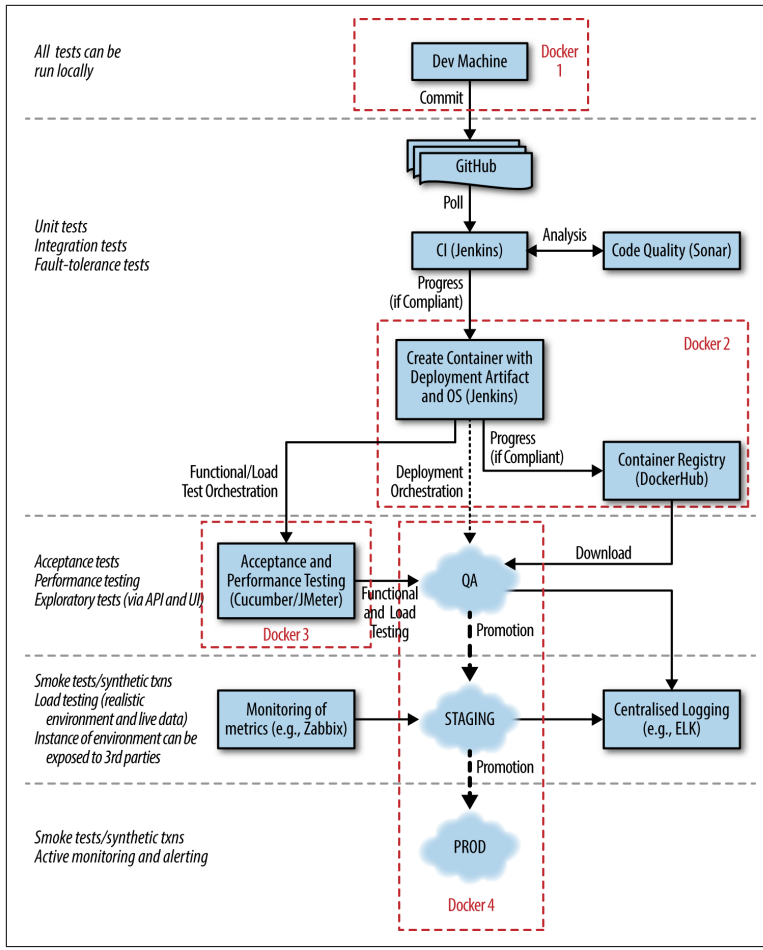


Figure 2-1. A Dockerized Java application continuous delivery pipeline

Location 1, the developer's machine, will now include packaging and running Java application within a Docker container. Typically, testing will occur here in two phases: the first ensures that the Java build artifact (JAR, WAR, etc.) is thoroughly tested, and the second asserts that the container image wrapping the artifact has been assembled correctly and functions as expected.

The CI-driven packaging of the application within a container is highlighted in Location 2. Notice now that instead of pushing a Java build artifact (JAR, WAR, etc.) to an artifact repository, a Docker image is being pushed to a centralized container registry, such as

Docker Hub. This Docker image now becomes the single source of truth, as opposed to the WAR previously, and this is the artifact promoted through the pipeline. It is worth noting that some organizations may want to store code artifacts in addition to Docker images, and accordingly JFrog's Artifactory can store both.

Location 3 pinpoints where tooling for defining and running multi-container Docker application (such as Docker Compose) could be used to drive acceptance and performance testing. Other existing options for orchestration and execution of build artifacts can also be used, including Maven/Gradle scripts, configuration management tooling (e.g., Ansible, Chef, Puppet), and deployment tooling like Capistrano.

Location 4 highlights that all application environments from QA to production must now change in order to support the execution of Docker containers, with the goal of testing an application within a production-like (containerized) environment as early in the pipeline as possible. Typically, the creation of a container deployment environment would be implemented by the use of a container orchestration framework like Docker Swarm, Kubernetes, or Apache Mesos, but discussing this is outside of the scope of this book.

The easiest way to understand these changes is to explore them with an example Java application and associated build pipeline. This is the focus of the next section.

Introducing the Docker Java Shopping Application

Throughout the remainder of this chapter, we will be working with an example project that is a simplified representation of an e-commerce application.

Obtaining the Example Project Code and Pipeline

For the examples in this chapter, we will use a simple e-commerce-style Java application named Docker Java Shopping, which is available via GitHub. The source repository contains three applications (with Docker build templates), a Docker Compose file for orchestrating the deployment of these applications, and a fully functional Jenkins build pipeline Vagrant virtual machine (VM) configuration. If you would like to follow along with the examples in this chapter,

please clone the repository locally (↵ indicates where a code line has been broken to fit the page):

```
git clone ↵
https://github.com/danielbryantuk/oreilly-docker-java-shopping/
```

The GitHub repository root project directory structure can be seen in Figure 2-2.

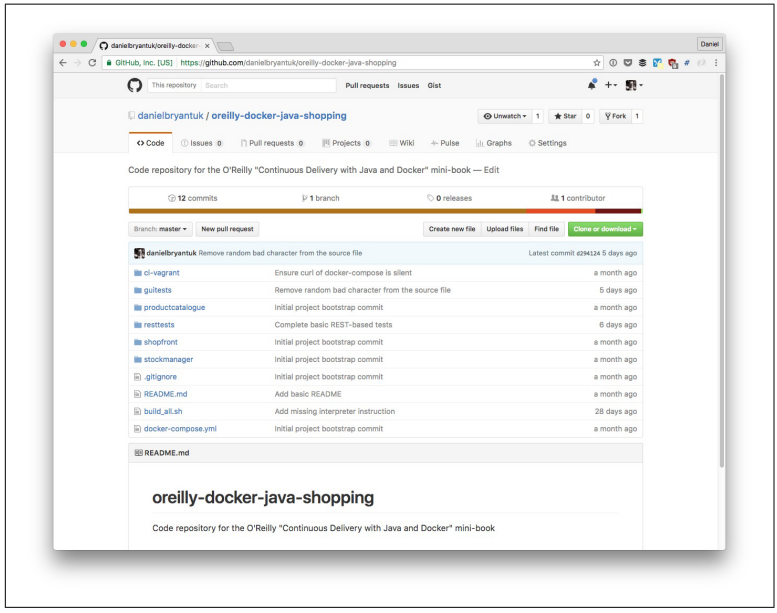


Figure 2-2. The Docker Java Shopping project on GitHub

Docker Java Shopping Application Architecture

The application follows the microservices-style architecture and consists of three applications, or services. Don't be too concerned with the use of the microservice pattern at the moment, as this will be discussed in Chapter 3, and we will deal with each of the three services as applications in their own right. The shopfront service has a UI that can be seen in Figure 2-3.

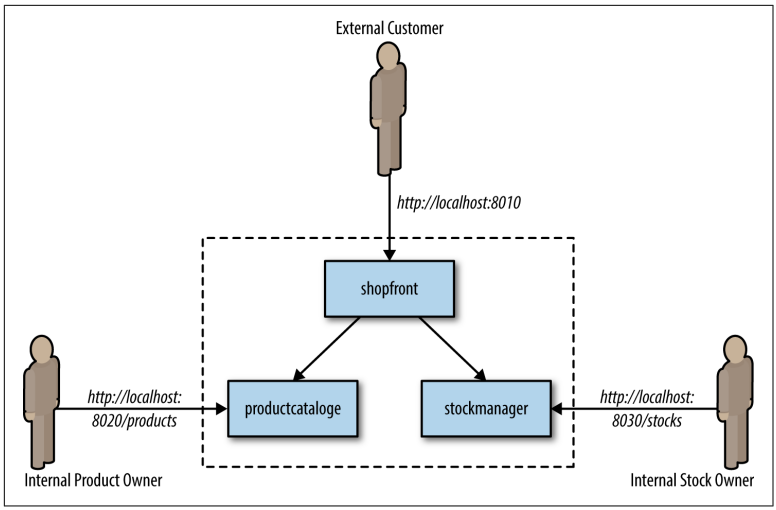


Figure 2-3. Docker Java Shopping application architecture

Figure 2-3 shows the architecture of this system, with the shopfront Spring Boot–based application acting as the main customer entry point. The shopfront application is mainly concerned with displaying and aggregating information from two other services that expose data via REST-like endpoints: the productcatalogue Dropwizard/Java EE–based application provides data on the products in the system; and the stockmanager Spring Boot–based application provides stock (SKU and quantity) data on the associated products.

In the default configuration (running via Docker Compose), the productcatalogue runs on port 8020, and the products can be viewed as JSON as shown in the following code:

```
$ curl localhost:8020/products | jq
[
  {
    "id": "1",
    "name": "Widget",
    "description": "Premium ACME Widgets",
    "price": 1.2
  },
  {
    "id": "2",
    "name": "Sprocket",
    "description": "Grade B sprockets",
    "price": 4.1
  },
  {
```

```
    "id": "3",
    "name": "Anvil",
    "description": "Large Anvils",
    "price": 45.5
  },
  {
    "id": "4",
    "name": "Cogs",
    "description": "Grade Y cogs",
    "price": 1.8
  },
  {
    "id": "5",
    "name": "Multitool",
    "description": "Multitools",
    "price": 154.1
  }
]
```

Also in the default configuration, the stockmanager runs on port 8030, and the product stock data can be viewed as JSON as shown in the following code:

```
$ curl localhost:8030/stocks | jq
[
  {
    "productId": "1",
    "sku": "12345678",
    "amountAvailable": 5
  },
  {
    "productId": "2",
    "sku": "34567890",
    "amountAvailable": 2
  },
  {
    "productId": "3",
    "sku": "54326745",
    "amountAvailable": 999
  },
  {
    "productId": "4",
    "sku": "93847614",
    "amountAvailable": 0
  },
  {
    "productId": "5",
    "sku": "11856388",
    "amountAvailable": 1
  }
]
```

Local Development Environment Configuration

If you plan to follow along with the examples in this book, the remainder of this chapter assumes that your local development system has the following software installed:

- Docker and Docker Compose
  - This can be installed for Mac and Windows via the Docker Toolbox
  - Developers using a Linux distribution can find instructions for installation on the Docker website
- Java 8
  - OpenJDK or Oracle JDK
- Maven
- Git
- Vagrant
- Oracle VM VirtualBox

For developers keen to explore the example Docker Java Shopping application, first check out the project repository and build the applications locally. All three of the applications must be built via Maven (there is a convenient `build_all.sh` script for Mac and Linux users), and then run via the `docker compose up --build` command, which should be executed in the root of the project. The shopfront application can be found at *http://localhost:8010/*.

Let's look at the project in more detail, and see how the Java applications have been deployed via Docker.

Building a Docker Image Locally

First we will look at the shopfront application located in the *shopfront* directory at the root of the project. This is a simple Spring Boot application that acts as an ecommerce "shop front." The application's build and dependencies are managed by Maven, and can be built with the `mvn clean install` command. Doing this triggers the compilation and unit (Maven Surefire) and integration (Maven Failsafe) testing, and creates a Fat JAR deployment artifact in the project's *target* directory. The contents of the target directory after a successful build can be seen in Example 2-1.

Example 2-1. Typical output

```
(master) oreilly-docker-java-shopping $ ls
README.md          docker-compose.yml resttests
build_all.sh       guitests           shopfront
ci-vagrant         productcatalogue   stockmanager
```

So far, this is nothing different from a typical Java project. However, there is also a *Dockerfile* file in the *shopfront* directory. The contents are as follows:

```
FROM openjdk:8-jre
ADD target/shopfront-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8010
ENTRYPOINT ["java", ↵
"-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Assuming basic Docker knowledge, we can see that this Dockerfile is built from the default OpenJDK Java image with the `8-jre` tag (ensuring that an OpenJDK 8 JRE is installed within the base image), the shopfront application Fat JAR is added to the image (and renamed to *app.jar*), and that the entrypoint is set to execute the application via the `java -jar <jar_file>` command. Due to the scope of this book, the Dockerfile syntax won't be discussed in further detail in this chapter, as Arun's *Docker for Java Developers* contains the majority of the commands that a Java developer will need.

Now that the Java application JAR has been built, we can use the following Docker command in the *shopfront* directory to build the associated Docker image:

```
docker build -t danielbryantuk/djshopfront .
```

This builds a Docker image using the Dockerfile in the current directory (the specified build context of `.`) and tags (`-t`) the image as `danielbryantuk/djshopfront`. We now have a Docker image that can be run as a container via the following command:

```
docker run -p 8010:8010 danielbryantuk/djshopfront
```

After the application has initialized (visible via the logs shown after executing the `docker run` command), visit *http://localhost:8010/health* in a browser to confirm the successful startup (it's worth noting that attempting to visit the shopfront UI will result in an error response, as the dependent productcatalogue and stockmanager

applications have not been started). The `docker run` command can be stopped by issuing a SIGINT via the key combination Ctrl-C.

You should now have a basic understanding of how to build, package, and run Java applications locally via Docker. Let's look at a few important questions that using Docker with Java imposes, before we move to creating a Docker-based continuous delivery pipeline.

Packaging Java Within Docker: Which OS, Which Java?

Packaging a Java application within a Docker image may be the first time many Java developers have been exposed to Linux (especially if you develop Java applications on MS Windows), and as the resultant Docker container will be running as is within a production environment, this warrants a brief discussion.

A Java developer who works in an organization that implements a similar build pipeline to that demonstrated in Figure 2-1 may not have been exposed to the runtime operational aspects of deployment; this developer may simply develop, test, and build a JAR or WAR file on their local machine, and push the resulting code to a version control system. However, when an application is running in production, it runs on a deeper operational stack: a JAR or WAR file is typically run within an application container (e.g., Tomcat, Websphere, etc.) that runs on a Java JDK or JRE (Oracle, OpenJDK, etc.), which in turn runs on an operating system (Ubuntu, CentOS, etc.) that runs on the underlying infrastructure (e.g., bare metal, VMs, cloud, etc.). When a developer packages an application within a Docker container, they are implicitly taking more responsibility further down this operational stack.

Accordingly, a developer looking to incorporate the packaging of Java applications with a Docker container will have to discuss several choices with their operations/sysadmin team (or take individual responsibility for the deployment). The first decision is what operating system (OS) to use as the base image. For the reasons of size, performance, and security, Docker developers generally prefer base images with the smaller minimal Linux distributions (e.g., Debian, Jessie, or Alpine) over traditional full-featured distributions (Ubuntu, RHEL). However, many organizations already have licences or policies that dictate that a certain distribution must be used. The same can often be said with the version of Java being used to run applications. Many organizations prefer to run the Oracle JDK

or JRE, but the default Java Docker images available from the Docker Hub public container repository only offer OpenJDK for licensing reasons.

Regardless of which OS and Java are chosen, it is essential that the usage is consistent throughout the pipeline; otherwise, unexpected issues may show up in production (e.g., if you are building and testing on Ubuntu running the OpenJDK but running in production on CentOS with Oracle's JRE, then your application hasn't been tested within a realistic production environment).

Packing Java Within Docker: JDK or JRE?

Although most Java development occurs alongside a full Java Development Kit (JDK), it is often beneficial to run an application using only a Java Runtime Environment (JRE). A JRE installation takes up less disk space, and due to the exclusion of many build and debugging tools has a smaller security attack surface. However, when developing and testing, it is often convenient to run your application on a JDK, as this includes additional tooling for easy debugging. It should be stated again, though, that whatever you build and test on must be identical to what is used in the production environment).

Pushing Images to a Repository and Testing

Traditional Java application deployment involved creating a JAR or WAR and pushing this to an artifact repository such as Nexus using a build pipeline tool like Jenkins. Jenkins is an open source automation server written in Java that enables automation of the nonhuman (build pipeline) parts of the software development process. Alternatives to Jenkins include Bamboo, Team City, and SaaS-based offerings like TravisCI and CircleCI.

Builds can be triggered in Jenkins by a variety of methods, such as code commit or a cron schedule, and resulting artifacts can be exercised via scripts and external tooling like SonarQube. As shown in Location 2 in Figure 2-1, with Docker-based Java deployment a Docker image artifact is now pushed to a container registry, such as Docker Hub or JFrog's Artifactory, rather than a traditional Java artifact repository.

In order to explore this next stage of the Java/Docker CD process, we need a Jenkins installation to illustrate the changes. Using Hashi-

Corp's Vagrant tool, the oreilly-docker-java-shopping project repository enables the construction of a Docker-enabled Jenkins server running on Ubuntu 16.04. To follow the instructions here, you must have locally installed the latest version of Vagrant and Oracle's VirtualBox virtual machine hypervisor.

Create a Jenkins Server via Vagrant

From the oreilly-docker-java-shopping project root, navigate into the *ci-vagrant* folder that contains a single Vagrant file. The Jenkins VM can be created by issuing the `vagrant up` command within this folder. The creation of the VM may take some time, especially on the first run, which downloads the Ubuntu 16.04 base image. While the VM is being created, the Vagrantfile can be explored to learn how the machine has been provisioned.

The simple `shell` provisioning option has been used to install the necessary dependencies on to a basic Ubuntu 16.04 base image. If you are familiar with Ubuntu operations, none of the commands should come as any surprise; if you aren't, don't be overly concerned as this book focuses on the developer perspective on CD. It is worth noting that for a production-grade deployment of a Jenkins build server, it would be typical to provision a machine using a configuration management tool like Puppet, Chef, or Ansible.

Once the Jenkins Vagrant box is provisioned, you can navigate to *http://localhost:8080*, supply the Jenkins key that is provided, in the terminal window at the end of the Vagrant provisioning process, create a Jenkins account, and install the recommended Jenkins plugins. One of the biggest technical strengths of Jenkins is that it is extremely extensible through the use of plugins, which can be installed to provide additional build customization and functionality.

Now navigate to the Manage Jenkins menu on the left of the Jenkins home page, and select Manage Plugins from the resulting main menu. Select the Available tab and search for docker. Approximately a quarter of the way down the page is a plugin named CloudBees Docker Build and Publish Plugin, which provides the ability to build projects with a Dockerfile and publish the resultant tagged image (repo) to the docker registry. Install this plugin without a restart and navigate back to the Jenkins home page.

Building a Docker Image for a Java Application

Now create a new job by selecting New Item from the menu on the left. Enter the item name of djshopfront and choose "Freestyle project" before pressing the OK button. On the item configuration page that is displayed, scroll down to the Source Code Management section and select the Git radiobox. Enter the oreilly-docker-java-shopping GitHub details in this section of the configuration, as shown in Figure 2-4.

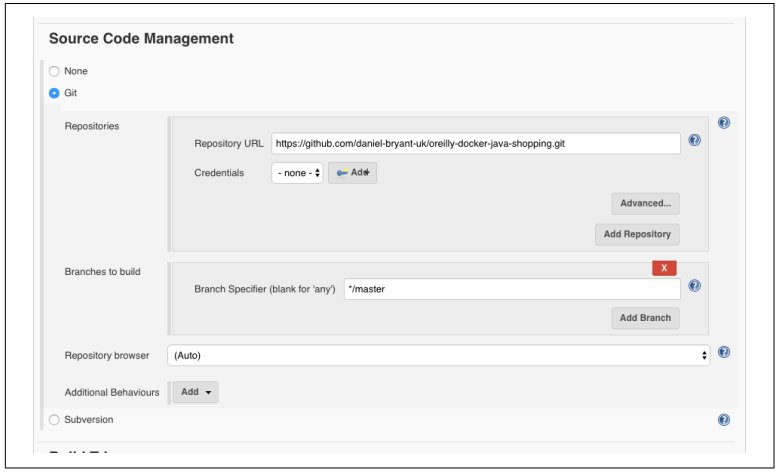


Figure 2-4. Configuring git in the djshopfront Jenkins build item

Now add an "Invoke top-level Maven targets" in the Build section of the configuration, shown in Figure 2-5. Enter the Goals as specified, and don't forget to specify the correct POM, as this Git project repository contains applications within subdirectories from the root.

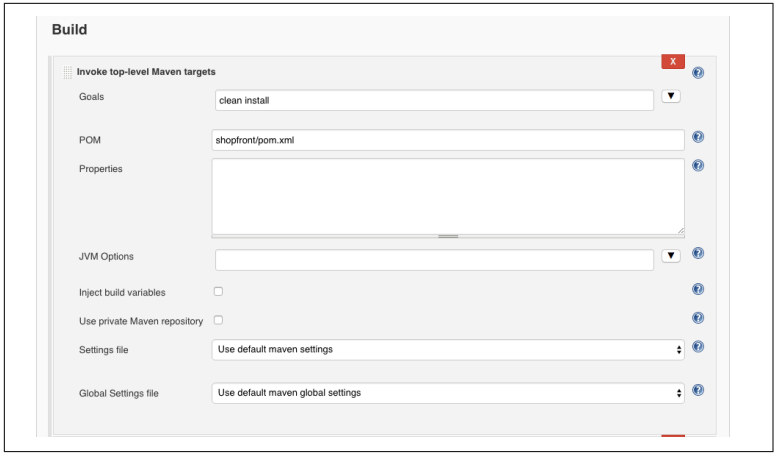


Figure 2-5. Configuring Maven in the djshopfront Jenkins build item

Next add a "Docker Build and Publish" build step, and enter the information as specified in Figure 2-6, but be sure to replace the tag in Repository Name with your Docker Hub account details (e.g., janesmith/djshopfront). Docker Hub is free to use for the public hosting of images, but other commercial container registries are available. If you wish to follow along with the example, then you will need to sign up for a free account and add your credentials to the "Registry credentials" in the Docker Build and Publish section of each build job.

Typically when building and pushing Docker images locally the `docker build -t <image_name:tag> .` and `docker push <image_name:tag>` commands are used, but the CloudBees Jenkins Docker Build and Publish plugin allows this to be managed via the Jenkins job configuration under Build.

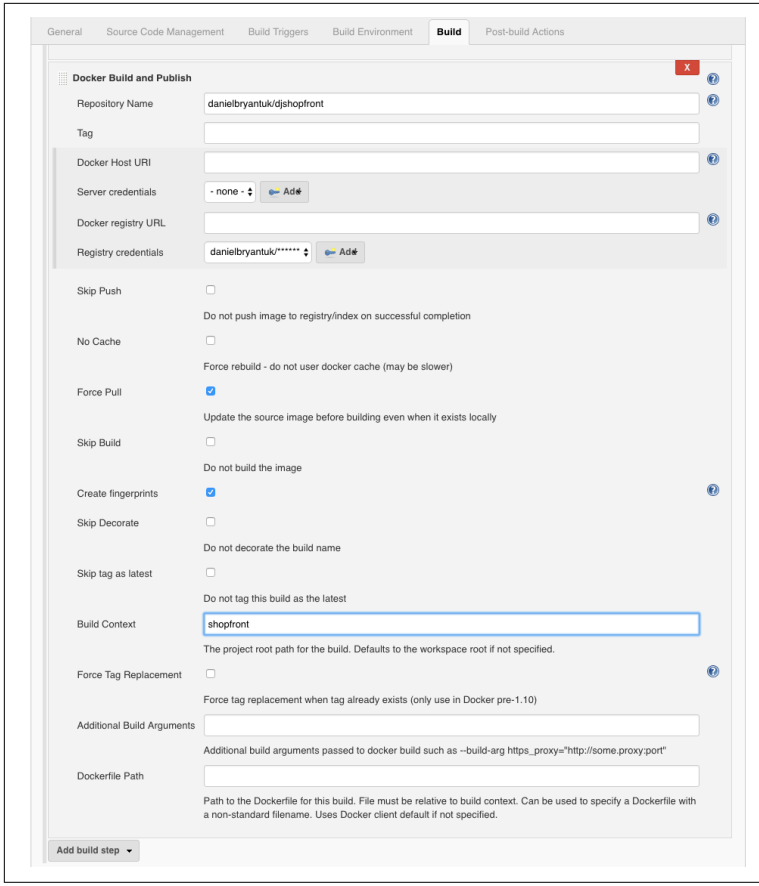


Figure 2-6. Configuring the CloudBees Docker Build and Publish data in the Jenkins build item

Don't Use the "latest" Docker Image Tag

Any application artifact deployed via a build pipeline must have a version associated with it; otherwise, it is impossible to know when the artifact was built and what functionality (and compatibility with other applications) it offers. In the Java world, we are used to using GAV Maven coordinates—groupId, artifactIs, and version. Docker allows images to be versioned with a tag, which can be an arbitrary string; we will discuss the impact this has on versioning in more depth later. However, it is worth mentioning that Docker offers a `latest` tag, but this is unfortunately one of the most misunderstood tags, and simply indicates the last build/tag that ran without a spe-

cific tag/version specified. Don't use the `latest` tag in your build pipeline, and instead pass tag version information as a parameter to the build item. As Adrian Mouat has suggested, if you don't version your Docker images properly, then you are heading toward a bad place.

The djshopfront job item can now be saved and run. By clicking on the Console Output menu item for the job, the build initialization in Figure 2-7 should initially be displayed:

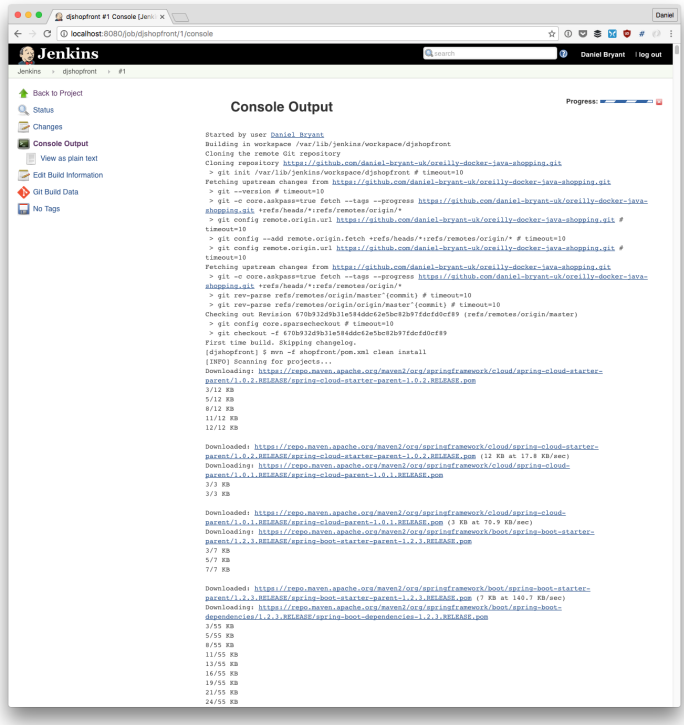


Figure 2-7. Jenkins build log from the start of the djshopfront build item

Upon completion of the job, the resultant Docker image should have been pushed to the Docker Hub (if there are any errors displayed here, please check that you have entered your Docker Hub registry credentials correctly in the job configuration):

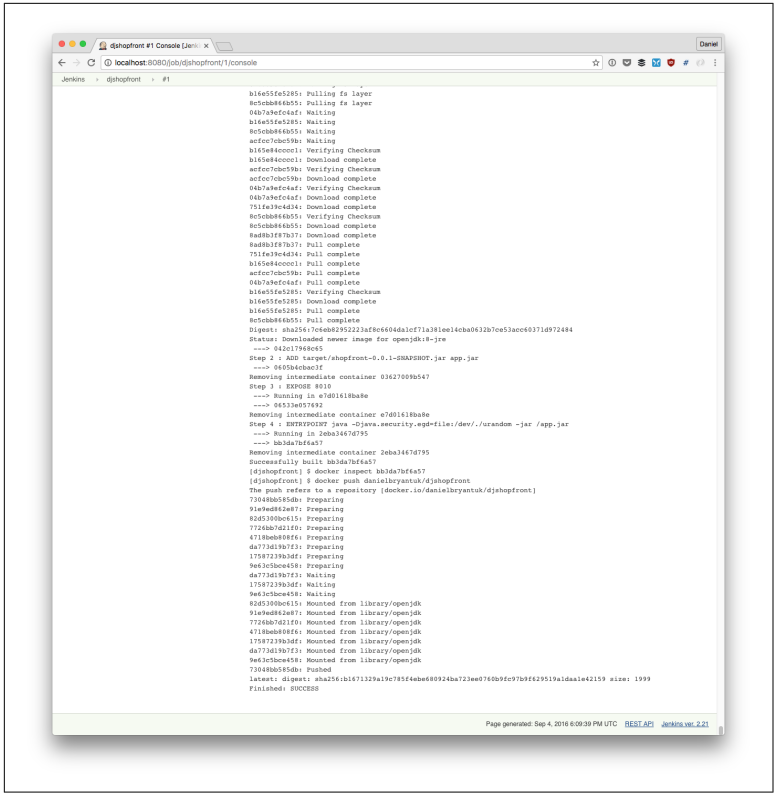


Figure 2-8. Jenkins build logs from a successful build of the djshopfront item

Once this job is run successfully, the resultant Docker image that contains the Java application JAR will be available for download by any client that has access to the registry (or, as in this case, the registry is publicly accessible) via `docker pull danielbryantuk/djshopfront`. This Docker image can now be run in any of the additional build pipeline test steps instead of the previous process of running the Fat JAR or deploying the WAR to an application container.

Now repeat the above process for the productcatalogue and stockmanager applications under the project root. Once complete, your Jenkins home page should look similar to Figure 2-9.

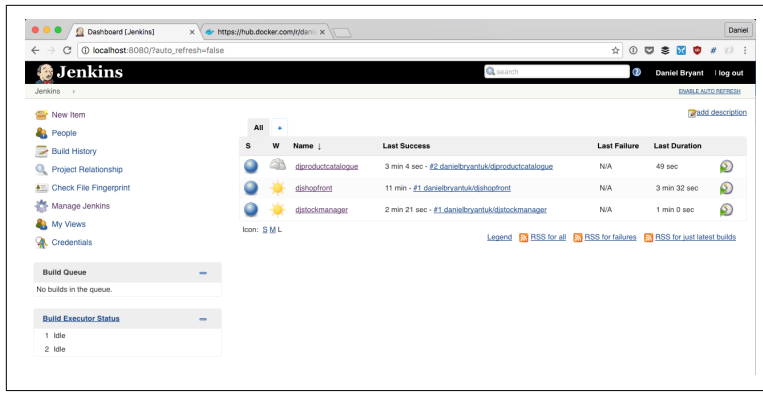


Figure 2-9. The Jenkins UI showing all three service build items

If all three Java application images have been successfully pushed to Docker Hub, you should be able to log in to the Docker Hub UI with your credentials and view the image details, as shown in Figure 2-10.

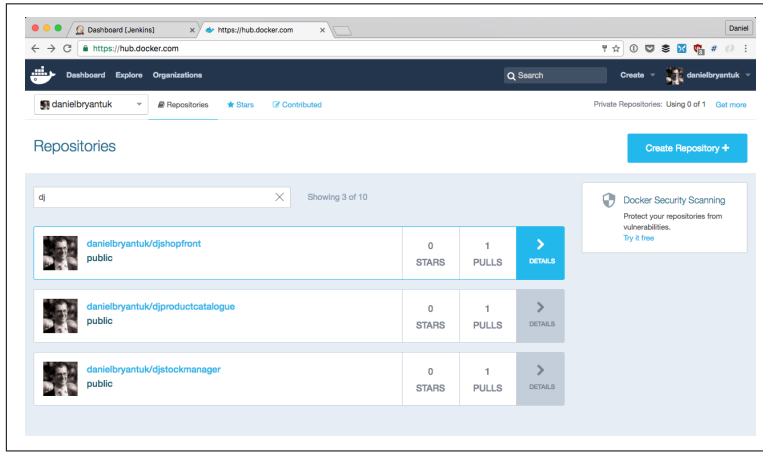


Figure 2-10. DockerHub UI showing the successful push of the three service Docker images

Potential Improvements to the Build Pipeline

The previous section of this book provided an introduction to building Java applications in Docker using Jenkins. It is worth noting that many of the techniques used with traditional Java Jenkins build pipelines can still be applied here:

- The Java artifact build process (and any unit, integration, and code quality tests) can be conducted separately from a job that packages the resulting JAR or WAR within a Docker image. This can save time and compute resources if a Docker image is not required upon every build.
- Java artifacts can still be stored in a Java artifact repository, such as Nexus, and pulled from this location into a Docker image within an additional build job.
- Docker labels can be used to indicate artifact versions and metadata in much the same fashion as the Maven groupId, artifactId, and version (GAV) format.
- Jenkins jobs can be parameterized, and users or upstream jobs can pass parameters into jobs to indicate the (label) version or other configuration options.
- The Jenkins Promoted Build plugin can be used to "gate" build stages, and indicate when an artifact is ready to be promoted from one stage of the pipeline to the next.

The Importance of Metadata

It is important to be able to understand, track, and manage artifacts that are produced within the build pipeline. Typical Java application artifacts are versioned using GAV Maven coordinates, which follow principles from semantic versioning (semver); for example, 1.12.01 (major.minor.patch). Additional metadata, such as the VCS commit SHA tag, QA control/audit information, and date/time of build is often included in manifest files. Docker has not yet standardized an approach to packaging this type of metadata, but several prominent members of the container community have proposed a label schema specification based on the use of Docker/container labels. You can learn more at *label-schema.org*.

Running Docker as a Deployment Fabric

The changes at Location 3 in Figure 2-1 indicate that any environment that previously ran Java applications as Fat JARs or deployed WARs must now be modified to instead run Docker containers.

Component Testing with Docker

Creation of component test environments and the triggering of associated tests (Location 2 in Figure 2-1) can be achieved via the Jenkins Pipeline job type and the CloudBees Docker Pipeline plugin, which can be installed in a similar manner to the previously mentioned CloudBees plugin. After this has been installed, create a new item named djshopfront-component-test of type Pipeline and click O.K.

The Power of Jenkins' Pipelines (Formerly Workflow)

As stated on the Jenkins website, while standard Jenkins freestyle jobs support simple continuous integration by allowing you to define sequential tasks in an application lifecycle, they do not create a record of execution that persists through any planned or unplanned restarts, enable one script to address all the steps in a complex workflow, or offer the other advantages of pipelines.

Pipeline functionality helps Jenkins to support continuous delivery (CD). The Pipeline plugin was built with requirements for a flexible, extensible, and script-based CD workflow capability in mind.

A key feature of pipelines is the ability to define stages within a build, which both help to clearly separate the configuration of disparate parts of a build process and also separate the reporting of the outcomes.

Scroll the resulting item configuration page to the Pipeline section, and enter the following:

```
node {
    stage ('Successful startup check') {
        docker.image('danielbryantuk/djshopfront')
        .withRun('-p 8010:8010') {
            timeout(time: 30, unit: 'SECONDS') {
                waitUntil {
                    def r = sh script: ↵
                    'curl http://localhost:8010/health | grep ↵
                    "UP"', returnStatus: true
                    return (r == 0);
                }
            }
        }
    }
}
```

This pipeline command uses the CloudBees `docker.image` library to run the `danielbryantuk/djshopfront` image, sets a timeout for 30 seconds (to allow a max limit to wait for the Spring Boot application initialization), and then waits until the application health endpoint returns `UP` from a `curl`.

The Value of Health Checks

The provisioning of health checks are extremely important when deploying applications into production, as an applications that is in trouble or cannot initialize correctly most likely indicates that it should not be sent traffic (for example, by adding the health check verification to a load balancer that is managing traffic). Pausing for an arbitrary time for tests within a build pipeline should also be avoided, as this can introduce nondeterministic results depending on the load of the build server and timing of the build, and accordingly it is best to set a timeout and continually probe the health check endpoint until a good response is received. Many container orchestration frameworks offer health checks "out of the box" and Docker has introduced native support for a `healthcheck` command in a Dockerfile in version 1.12.

The `curl/grep` test used above can be replaced with any command or execution of any application that returns an exit code to indicate success or failure (e.g., a test Docker container or JUnit-powered test suite obtained from a Git repo).

Figure 2-11 shows the example pipeline code within the Jenkins configuration page. Save this item and run the associated job via the Build Now option in the left menu.

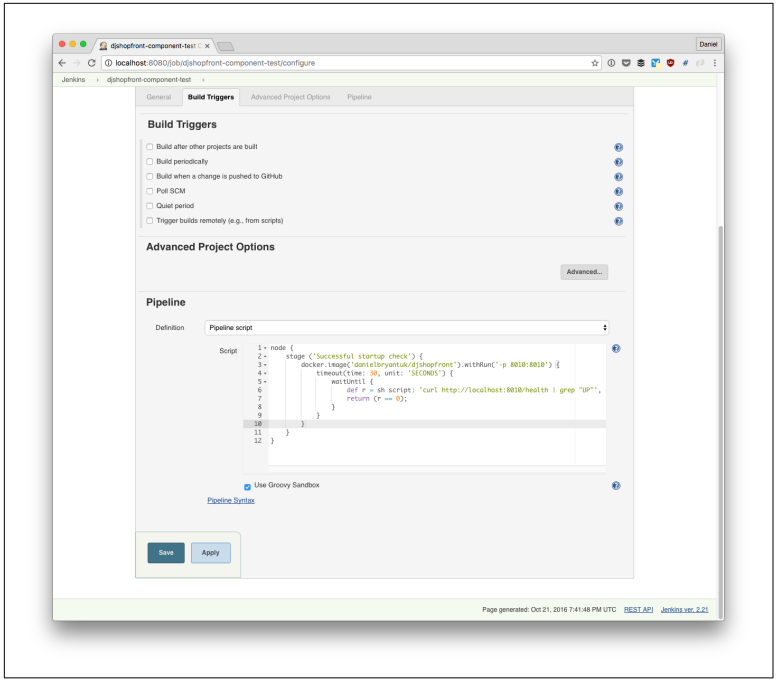


Figure 2-11. Jenkins pipeline build configuration for djshopfront-component-test

Figure 2-12 shows the output from a typical execution of this test.

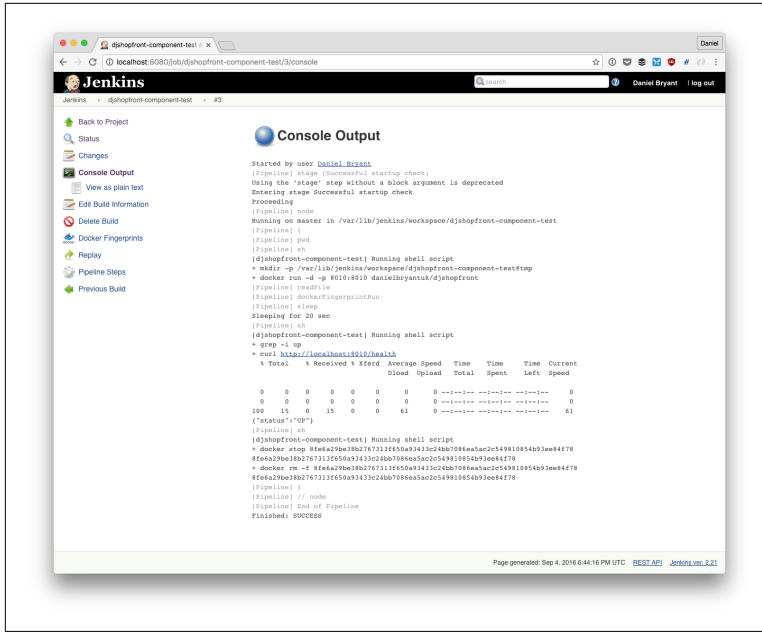


Figure 2-12. Jenkins build logs from a successful run of djshopfront-component-test

Using Docker Compose for E2E Testing

The acceptance and performance testing stage of the pipeline, Location 4 in Figure 2-1, can be further modified with the introduction of Docker by utilizing Docker Compose, a tool for defining and running multicontainer Docker applications. Using the same approach above with the Jenkins pipeline job we can write jobs using Docker Compose, as follows:

```
node {
    stage ('build') {
        git url: 'https://github.com/ ↵
        danielbryantuk/oreilly-docker-java-shopping.git'
        // conduct other build tasks
    }

    stage ('end-to-end tests') {
        timeout(time: 60, unit: 'SECONDS') {
            try {
                sh 'docker-compose up -d'

                waitUntil { // application is up
                    def r = sh script: ↵
```

```
                    'curl http://localhost:8010/health | ↵
                    grep "UP"', returnStatus: true
                    return (r == 0);
                }

                // conduct main test here
                sh 'curl http://localhost:8010 | ↵
                grep "Docker Java"'

            } finally {
                sh 'docker-compose stop'
            }
        }
    }

    stage ('deploy') {
        // deploy the containers/application here
    }
}
```

The resulting job output can be seen in the Figure 2-13.

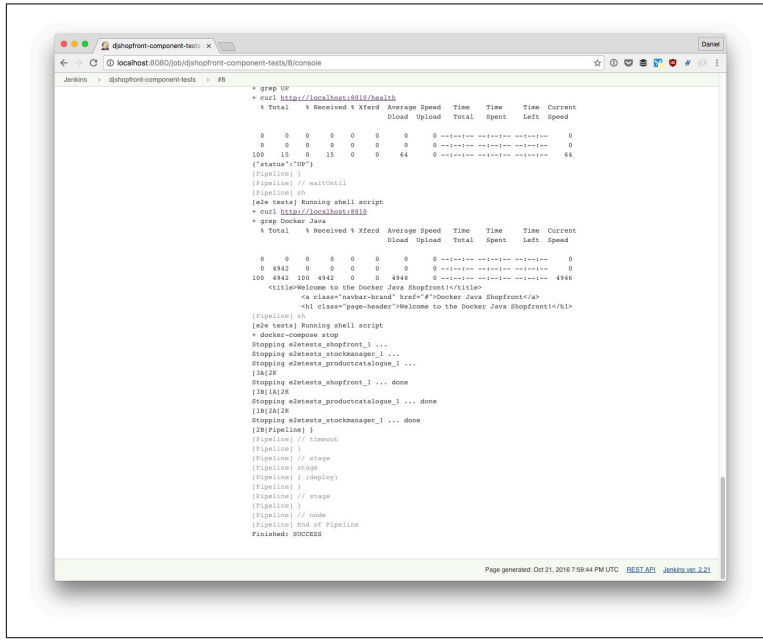


Figure 2-13. Jenkins build logs for a successful run of end-to-end tests

Navigating back to the dj-end-to-end-tests build page shows the multiple pipeline stages, and whether each stage has succeeded or

not. Figure 2-14 shows an example of this with several builds succeeding.

Adrian Mouat talks more about using Docker Compose in this fashion in his *Using Docker* book.

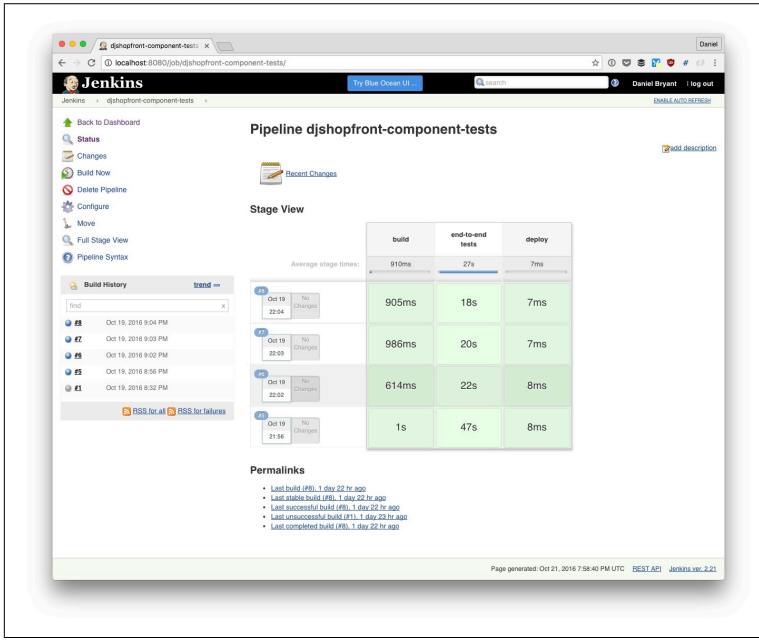


Figure 2-14. An example dj-end-to-end-tests build item page showing pipeline stages

Running Docker in Production

Discussing the running of Docker in production is out of scope in a book of this size, but we strongly recommend researching technologies such as Docker Swarm, Kubernetes, and Mesos, as these platforms make for excellent Docker-centric deployment fabrics that can be deployed on top of bare metal, private VMs, or the public cloud.

It's a Brave New (Containerized) World

There is no denying the flexibility afforded by deploying Java applications in Docker containers. However, deploying applications in containers adds several new constraints, and there are also a series of best practices emerging that can affect CD. Skills must be learned and new techniques mastered across architecture, testing, and operations. Let's look at this over the next three chapters.

CHAPTER 3

The Impact of Docker on Java Application Architecture

> "If you can't build a [well-structured] monolith, what makes you think microservices are the answer?"
>
> —Simon Brown, Coding the Architecture

Continuously delivering Java applications within containers require several long-standing architectural patterns and paradigms be challenged. The potential for automation and immutability that containers provide (along with new properties imposed like resource-restriction and transience) means that new architecture and configuration approaches are required. This chapter looks at examples of these changes, such as the Twelve-Factor Application, the decomposition of monolithic applications into smaller API-driven composable components (microservices), and the need to develop *mechanical sympathy*—an appreciation and respect for the underlying deployment fabric.

Cloud-Native Twelve-Factor Applications

In early 2012, Platform-as-a-Service (PaaS) pioneer Heroku developed the Twelve-Factor App, a series of rules and guidance for helping developers build cloud-ready PaaS applications that:

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project

- Have a clean contract with the underlying operating system, offering maximum portability between execution environments
- Are suitable for deployment on modern cloud platforms, minimizing the need for servers and systems administration
- Minimize divergence between development and production, enabling continuous deployment for maximum agility
- Can scale up without significant changes to tooling, architecture, or development practices

These guidelines were written before container technology became mainstream, but are all equally (if not more) relevant to Docker-based applications. Let's looks briefly at each of the factors now, and see how they map to continuously deploying Java applications within containers:

1. Codebase: One codebase tracked in revision control, many deploys

Each Java application (or service) should be tracked in a single, shared code repository, and Docker configuration files, such as Dockerfiles, should be stored alongside the application code.

2. Dependencies: Explicitly declare and isolate dependencies

Dependencies are commonly managed within Java applications using Maven or Gradle, and OS-level dependencies should be clearly specified in the associated Dockerfile.

3. Config: Store config in the environment

The Twelve-Factor App guidelines suggest that configuration data should be injected into an application via environment variables, although in practice many Java developers prefer to use configuration files, and there can be security issues with exposing secrets via environment variables. Storing nonsensitive configuration data in a remote service like Spring Cloud Config (backed by Git or Consul) and secrets in a service like HashiCorp's Vault can be a good compromise. It is definitely not recommended to include secrets in a Dockerfile or Docker image.

4. Backing services: Treat backing services as attached resources (typically consumed over the network)

Java developers are accustomed to treating data stores and middleware in this fashion, and in-memory substitutes (e.g., HSQLDB, Apache Qpid, and Stubbed Cassandra) or service virtualization (e.g.,

Hoverfly) can be used for in-process component testing within the build pipeline.

5. Build, release, run: Strictly separate build and run stages

For a compiled language such as Java, this guideline comes as no surprise (and with little choice of implementation!). It is worth mentioning that the flexibility provided by Docker means that separate containers can be used to build, test, and run the application, each configured as appropriate. For example, a container can be created for build and test with a full OS, JDK, and diagnostic tools; and a container can be built for running an application in production with only a minimal OS and JRE. However, some may see this as an antipattern, as there should only be one "source of truth" artifact that is created, tested, and deployed within the pipeline, and using multiple Docker images can lead to an impedance mismatch and configuration drift between images.

6. Processes: Execute the app as one or more stateless processes

Building and running a Java application as a series of microservices can be made easier by using Docker (the concept of microservice is explained later in this chapter).

7. Port binding: Export services via port binding

Java developers are used to exposing application services via ports (e.g., running an application on Jetty or Apache Tomcat). Docker compliments this by allowing the declarative specification of exposed ports within an application's Dockerfile, and Docker Compose also enables the same functionality when orchestrating multiple containers.

8. Concurrency: Scale out via the process model

Traditional Java applications typically take the opposite approach to scaling, as the JVM runs as a giant "uberprocess" that is often vertically scaled by adding more heap memory, or horizontally scaled by cloning and load-balancing across multiple running instances. However, the combination of decomposing Java applications into microservices and running these components within Docker containers can enable this approach to scalability. Regardless of the approach taken to implement scalability, this should be tested within the build pipeline.

9. Disposability: Maximize robustness with fast startup and graceful shutdown

This can require a mindset shift with developers who are used to creating a traditional long-running Java application, where much of the expense of application configuration and initialization was front-loaded in the JVM/application startup process. Modern, container-ready applications should utilize more just-in-time (JIT) configuration, and ensure that best efforts are taken to clean up resource and state during shutdown. The JDK exposes hooks for JRE/application startup and shutdown, and Docker (and the OS) can be instructed to call these, such as by issuing SIGTERM (versus SIGKILL) to running processes within a container.

10. Dev/prod parity: Keep development, staging, and production as similar as possible

The use of Docker in combination with orchestration technologies like Docker Swarm, Kubernetes, and Mesos can make this easier in comparison with traditional bare metal deployments where the underlying hardware and OS configuration is often significantly different than developer or test machines. As an application artifact moves through the build pipeline, it should be exposed to more and more realistic environments (e.g., unit testing can run in-memory [or within a Docker container] on a build box). However, end-to-end testing should be conducted in a production-like environment (e.g., if you are running Docker Swarm in production, you should be testing on Docker Swarm).

11. Logs: Treat logs as event streams

Java has had a long and sometimes arduous relationship with logging frameworks, but modern frameworks like Logback and Log4j 2 can be configured to stream to standard output (and hence viewed when running in a container by using the `docker logs` command) or streamed to disk (which can be mounted from the host running the container).

12. Admin processes: Run admin/management tasks as one-off processes

The ability to create simple Java applications that can be run within a Docker container allows administrative tasks to be run as one-off processes. However, these processes must be tested within (or as part of) the build pipeline.

This section of the chapter has attempted to summarize the benefits of using the principles from the Twelve-Factor App. For developers looking to develop a deeper understanding, Kevin Hoffman's *Beyond the Twelve-Factor App* (O'Reilly) is a recommended read.

The Move to Microservices

One of the early use cases for Docker containers was the process isolation guarantees—the fact that multiple application instances could be deployed onto the same hardware without the need for a virtual machine hypervisor. Arguments emerged that suggested running only a single process per container was best practice, and that container runtimes should be minimal and the artifacts deployed into them should not be monolithic applications. This proved a difficult (if not impossible) set of requirements for existing Java applications that were already running in production. These arguments, in combination with other trends in the software development industry, led to the increase in popularity of a new architectural style we now refer to as *microservices*.

This book won't cover the concept of microservices in depth; instead, an introduction to the topic can be found in Christian Posta's *Microservices for Java Developers* (O'Reilly), and a more thorough treatment can be found in Sam Newman's *Building Microservices* (O'Reilly) and Amundsen et al's *Microservice Architecture* (O'Reilly). However, it is worth mentioning that many developers are embracing this architectural style in combination with adopting containerization technologies like Docker.

Don't Conflate Microservices with Container Technology

It is tempting to conflate microservices with container technology (or Docker specifically), but the microservice pattern is an architectural style, and containers are a packaging/deployment technology. Although microservices and Docker are complementary, implementing one does not necessarily require the other.

A core concept of microservices revolves around creating services that follow the single-responsibility principle and have one reason to

change. Building Java-based microservices impacts the implementation of CD in several ways:

- Multiple build pipelines (or branches within a single pipeline) must be created and managed
- Deployment of multiple services to an environment now have to be orchestrated, managed, and tracked
- Component testing may now have to mock, stub, or virtualize dependent services
- End-to-end testing must now orchestrate multiple services (and associated state) before and after executing tests
- Process must be implemented to manage service version control (e.g., the enforcement of only allowing the deployment of compatible interdependent services)
- Monitoring, metrics, and application performance management (APM) tooling must be adapted to handle multiple services

Decomposing an existing monolithic application, or creating a new application that provides functionality through a composite of microservices is a non-trivial task. Techniques such as context mapping, from domain-driven design, can help developers (working alongside stakeholders and the QA team) understand how application/business functionality should be composed as a series of *bounded contexts* or focused services. It is also critical to understand how to design service application programming interfaces (APIs).

API-Driven Applications

Once service boundaries have been determined, the development team (and stakeholders) can define service functionality and the associated interfaces—the application programming interfaces (APIs). Mike Amundsen's O'Reilly video series "Designing APIs for the Web" is a good place to start if you want to learn more about API design. Many teams attempt to define a service API up front, but in reality the design process will be iterative. A useful technique to enable this iterative approach is the behavior-driven development (BDD) technique named "The Three Amigos," where any requirement should be defined with at least one developer, one QA specialist, and one project stakeholder present.

The typical outputs from this stage of the service design process include: a series of BDD-style acceptance tests that asserts compo-

nent (single microservice) level requirements, such as Cucumber Gherkin syntax acceptance test scripts; and an API specification, such as a Swagger or RAML file, which the test scripts will operate against. It is also recommended that each service has basic (happy path) performance test scripts created (for example, using Gatling or JMeter) and also security tests (for example, using bdd-security). These service-level component tests can then be run continuously within the build pipeline, and will validate local microservice functional and nonfunctional requirements. Additional internal resource API endpoints can be added to each service, which can be used to manipulate internal state for test purposes or expose metrics.

The benefits to the CD process of exposing application or service functionality via an API include:

- Easier automation of test fixture setup and teardown via internal resource endpoints (and this limits or removes the need to manipulate state via file system or data store access).
- Easier automation of specification tests (e.g., REST-assured). Triggering functionality through a fragile UI is no longer required for every test.
- API contracts can be validated automatically, potentially using techniques like consumer contracts and consumer-driven contracts (e.g., Pact-JVM).
- Dependent services that expose functionality through an API can be efficiently mocked (e.g., WireMock), stubbed (e.g., stubby4j), or virtualized (e.g., Hoverfly).
- Easier access to metrics and monitoring data via internal resource endpoints (e.g., Codahale Metrics or Spring Boot Actuator).

Containers and Mechanical Sympathy

Martin Thompson and Dave Farley have talked about the concept of *mechanical sympathy* in software development for several years. They were inspired by the Formula One racing driver Jackie Stewart's famous quote "You don't have to be an engineer to be a racing driver, but you do have to have mechanical sympathy", meaning that understanding how a car works will make you a better driver; and it has been argued that this is analogous to programmers understanding how computer hardware works. You don't necessarily need a

degree in computer science or to be a hardware engineer, but you do need to understand how hardware works and take that into consideration when you design software. The days of architects sitting in ivory towers and drawing UML diagrams is over. Architects and developers must continue to develop practical and operational experience from working with the new technologies.

Using container technologies like Docker can fundamentally change the way your software interacts with the hardware it is running on, and it is beneficial to be aware of these changes:

- Container technology can limit access to system resources, due to developer/operator specification, or to resource contention.
  - In particular, watch out for the restriction of memory available to a JVM, and remember that Java application memory requirements are not simply equal to heap size. In reality, Java applications' memory requirements include the sum of Xmx heap size, PermGen/Metaspace, native memory thread requirements, and JVM overhead.
  - Another source of potential issues is that containers typically share a single source of entropy (*/dev/random*) on the host machine, and this can be quickly exhausted. This manifests itself with Java applications unexpectedly stalling/blocking during cryptographic operations such as token generation on the initialization of security functionality. It is often beneficial to use the JVM option `-Djava.security.egd=file:/dev/urandom`, but be aware that this can have some security implications.
- Container technology can (incidentally) expose incorrect resource availability to the JVM (e.g., the number of processor cores typically exposed to a JVM application is based on the underlying host hardware properties, not the restrictions applied to a running container)
- When running containerized deployment fabric, it is often the case that additional layers of abstraction are applied over the operating system (e.g., orchestration framework, container technology itself, and an additional OS).
- Container orchestration and scheduling frameworks often stop, start, and move containers (and applications) much more often compared to traditional deployment platforms.

- The hardware fabric upon which containerized applications are run is typically more ephemeral in nature (e.g., cloud computing).
- Containerized applications can expose new security attack vectors that must be understood and mitigated.

These changes to the properties of the deployment fabric should not be a surprise to developers, as the use of many new technologies introduce some form of change (e.g., upgrading the JVM version on which an application is running, deploying Java applications within an application container, and running Java applications in the cloud). The vast majority of these potential issues can be mitigated by augmenting the testing processes within the CD build pipeline.

CHAPTER 4

The Importance of Continuous Testing

> *"If you don't like testing your product, most likely your customers won't like testing it either."*
>
> —(Anonymous)

Testing is a core part of any continuous delivery build pipeline, and the introduction of Docker into the system technology stack has no impact in some areas of testing and a great impact in others. This chapter attempts to highlight these two cases, and makes recommendations for modifying an existing Java build pipeline to support the testing of containerized Java applications.

Functional Testing with Containers

The good news is that running Java applications within containers does not fundamentally affect functional testing. The bad news is that many systems being developed do not have an adequate level of functional testing regardless of the deployment technology.

The approach to unit and integration testing with Java applications that will ultimately be executed within containers remains unchanged. However, any component or system-level end-to-end tests should be run against applications running within containers in an environment as close to production as possible. In reality, placing an application inside a container can often make it easier to

"spin up" and test, and the major area for concern is avoiding port and data clashes when testing concurrently.

Although it is not yet generally recommended to run production data stores from within containers, running RDBMS and other middleware within containers during the build pipeline test phase can allow easier scenario creation by the loading of "pre-canned" data within containers, "data containers," or by cloning and specifying data directories to be mounted into a container.

Finally, regardless of whether or not an application is being deployed within a container, it is strongly recommended to use executable specifications and automated acceptance tools to define, capture, and run functional tests. Favorites in the Java development ecosystem include Serenity BDD, Cucumber, JBehave, REST-assured, and Selenium.

Nonfunctional Testing: Performance

Running Java applications within containers makes it much easier to configure hardware and operating system resource limits (for example, using cgroups). Because of this, it is imperative that during the later stages of the build pipeline, performing nonfunctional testing of the runtime environment is as production-like as possible. This includes executing the container with production-like options (e.g., with Docker CPU cores configured via the `--cpuset` command-line option, CPU shares via `-c`, and memory limits set via `-m`), running the container within the same orchestration framework used in production, and executing the orchestration/container runtime on comparable hardware infrastructure (perhaps scaled down).

Performance and load testing can be implemented via Jenkins by using a tool like Docker Compose to orchestrate deployment of an application (or a series of services) and tools like Gatling or JMeter to run the tests. In order to isolate individual services under test, the technique of service virtualization can be used to control the performance characteristics of interdependent services. More information on this technique in the context of testing microservices can be found in my "Proposed Recipe for Designing, Building, and Testing Microservices" blog post.

Nonfunctional Testing: Security Across the System

Security should be of paramount concern to any developer, regardless of whether deployment is occurring in containers. However, executing Java applications within a containerized runtime can add new security attack vectors, and these must be mitigated.

Host-Level Security

Any host running containers must be hardened at the operating system level. This includes:

- Ensuring the latest operating system version available is being used, and that the OS is fully patched (potentially with additional kernel hardening options like grsecurity)
- Ensuring the application attack surface exposed in minimized (e.g., correctly exposing ports, running applications behind a firewall with DMZ, and using certificate-based login)
- Using application-specific SELinux or AppArmour
- Using an application-specific seccomp whitelist if possible
- Enabling user namespaces

The Center for Internet Security (CIS) regularly publishes guidance for running containers in production. The CIS Docker 1.12.0 benchmark can be found on the CIS website. The Docker team has also created the Docker Bench for Security, which attempts to automate many of the checks and assertions that the CIS documentation recommends. The Docker Bench for Security tool can be downloaded and executed as a container on the target host, and a comprehensive report about the current state of the machine's security will be generated. Ideally, execution of the Docker Bench Security should be conducted on the initialization of any host that will be running containers, and also periodically against all servers to check for configuration drift or new issues. This process can be automated as part of an infrastructure build pipeline.

Container-Level Security

Java developers may not be used to dealing with OS-level security threats, but will be exposed to this when packaging applications

within containers. A Docker image running a Java application will typically include a Linux-based operating system such as Alpine, Ubuntu, or Debian Jessie, and may also have other tooling installed via a package manager. Ensuring the OS and all associated software is up-to-date and configured correctly is very challenging without automation. Fortunately, tooling like CoreOS's open source Clair project can be used to statically analyze Docker images for vulnerabilities. This tool can be integrated within a build pipeline, or if Docker images are being pushed to a commercial centralized image repository like Docker Hub or CoreOS Quay, then this will most likely be enabled by default (but do take care to ensure it is, and also that actionable results—such as vulnerability detection—are fed back into the build pipeline).

Application-Level Security

Any build pipeline should include automated security testing in addition to manual vulnerability analysis and penetration testing. Tooling such as OWASP's ZAP and Continuum's bdd-security can also be included in a standard build pipeline, and run at the (micro)service/application and system-level testing phases.

Additional Reading

Further information on security aspects of running containers can be found in Adrian Mouat's *Using Docker* (O'Reilly) or Aaron Grattafiori's white paper "Understanding and Hardening Linux Containers."

CHAPTER 5

Operations in the World of Containers

> "You were not in control. You had no visibility: maybe there was a car in front of you, maybe not."
>
> —Alain Prost, Formula One Driver

With continuous delivery, the build pipeline provides confidence that new or modified application features will work correctly, both at the functional and nonfunctional level. However, this is only part of the story for continuously delivering valuable software to users—as soon as an application is released to production it must be monitored for signs of malfunction, performance issues, and security vulnerabilities.

Host-Level Monitoring

Starting at the host level, it is essential that any machine running production containers (and applications) provide hardware- and OS-level metrics. All cloud vendors provide an API for obtaining metrics such as CPU usage, and disk and network I/O performance, and such an API is relatively easy to create and expose when running on bare metal (for example, using sar/sysstat). All Linux-based OSes provide excellent metrics, such as number of processes running, run queue averages, and swap space usage.

Regardless of where metrics originate, they should be collected and processed centrally, such as by using a tool like Prometheus, the

InfluxData TICK stack, or a SaaS-based offering like Datadog. Centralization not only provides a single place for developers and operators to manually monitor metrics, but also enables core alerts and automated warnings to be defined. Any unexpected or critical issues that require operator intervention should trigger a paging system like PagerDuty, and the on-call developer or operator must action the alert.

All core host and operating system–level logs should also be collected centrally, and this is typically undertaken by streaming logs to a tool like the ElasticSearch-Logstash-Kibana (ELK) stack (shown in Figure 5-1) or a commercial equivalent.

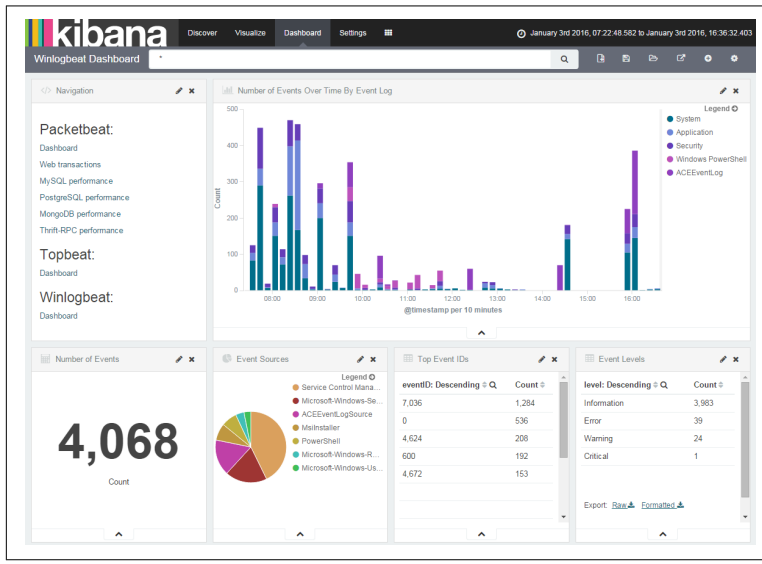


Figure 5-1. Kibana web UI dashboard, part of the ELK stack

Operators may also consider running a host intrusion detection system (HIDS) if security or governance is of paramount concern (over and above hardening systems), and open source offerings such as OSSEC or Bro Network Security Monitor.

Container-Level Metrics

Monitoring at the container level ranges from as simple as the output to `docker stats` or using the more detailed overview provided by cAdvisor, to the full power of Sysdig.

The built-in Docker `docker stats` command returns a live data stream of basic metrics for running containers, as shown in Figure 5-2.

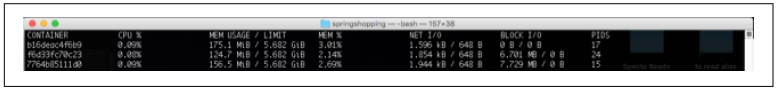


Figure 5-2. Command line output from docker stats command

More detailed information about a container's resource usage can also be found enabling and using the Docker Remote API, which is exposed as a REST-like API.

cAdvisor (Container Advisor) is Google's container monitoring daemon that collects, aggregates, processes, and exports information about running containers (see Figure 5-3). Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage, and network statistics. This data is exported by container and machine-wide.

Sysdig is a container-friendly, open source, system-level exploration tool that can capture system state and activity from a running Linux instance, then save, filter, and analyze this data. Sysdig is scriptable in Lua and includes a command-line interface and a powerful interactive UI, csysdig, that runs in a terminal.

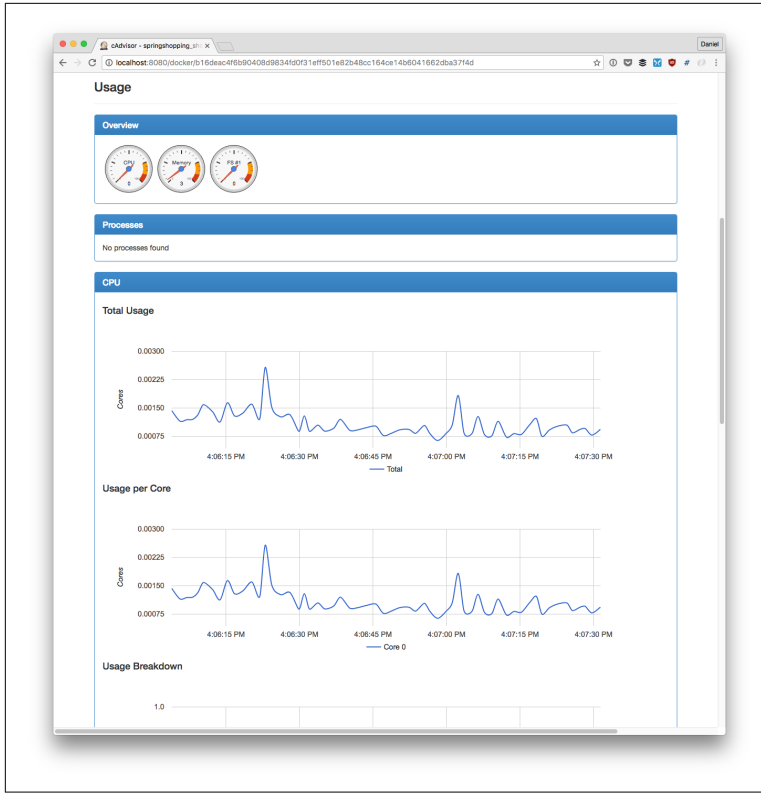


Figure 5-3. Web UI of cAdvisor showing local container metrics

Application-Level Metrics and Health Checks

Developers should expose important application metrics via API endpoints, like those provided by CodaHale's Metrics library and Spring Boot's Actuator. Metrics are typically exposed as gauges, counters, histograms, and timers, and can relay vital information to both operators and business stakeholders. Many Java-based metric libraries also support the creation of health-check endpoints that can be queried by monitoring systems and load balancers in order to determine if the application instance is healthy and ready to accept traffic (Figure 5-4).

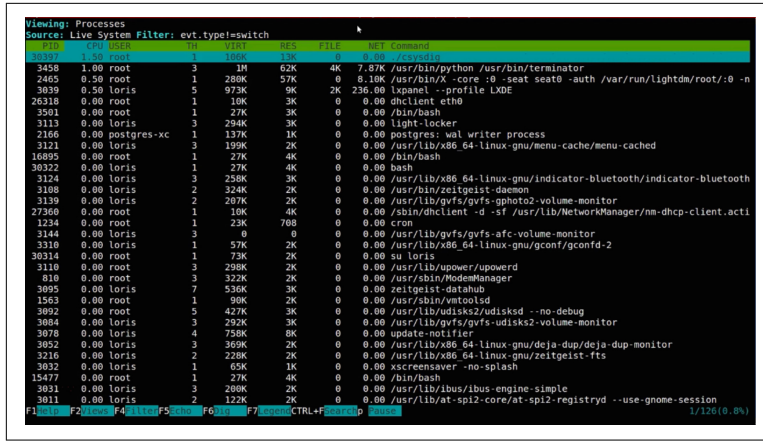


Figure 5-4. Sysdig's interactive UI running at the command line

Application performance management (APM) is also a useful tool for developers and operators to understand and debug a system. Historically, the commercial solutions have had much more functionality in comparison with open source tooling, but Naver's Pinpoint is now offering much of the expected core functionality (see Figure 5-5).

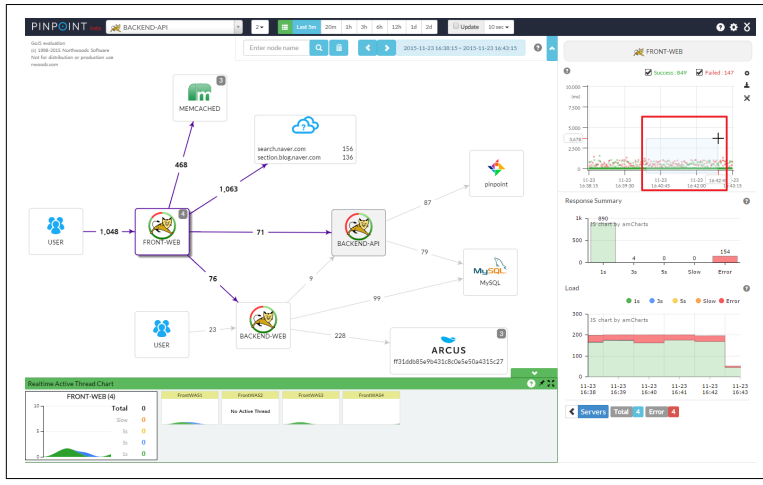


Figure 5-5. Pinpoint web UI dashboard

As many Java developers who are embracing container technology like Docker are also looking to develop microservice-based applications, distributed tracing solutions like OpenZipkin's Zipkin are very

valuable when exploring the request/response and data flows through a distributed system (see Figure 5-6).

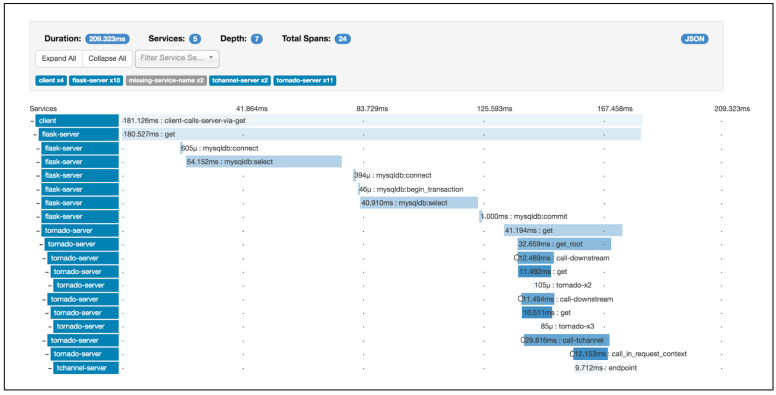


Figure 5-6. OpenZipkin web UI

CHAPTER 6
Conclusion and Next Steps

In a book of this size, it would be impossible to cover every detail about continuously delivering Java applications in Docker. Instead, we have attempted to highlight core areas for consideration, and also provide several practical examples for getting started. The value of implementing continuous delivery cannot be underestimated, and the increase in speed of feedback combined within the ability to reliably and repeatedly guarantee quality is a true game changer. The flexibility provided by packaging Java applications within Docker is also not to be underestimated, and what might at first glance appear to be a strange marriage between a 20-year-old programming language and an emerging packaging and runtime format can add a lot of flexibility to building, testing, and deploying applications.

The true benefits of continuously delivering Java applications within Docker containers truly starts to emerge when these approaches and technologies are combined with a holistic approach to agility—embracing agile and lean practices at the business level, designing evolutionary architectures using patterns like microservices, developing and testing software using techniques from extreme programming (XP), deploying software to programmatically defined agile infrastructure such as cloud and modern cluster managers, and also supporting the growth of a DevOps culture and shared responsibility mindset.

There is much to learn, but we hope this guide serves you well as an introduction!

About the Author

Daniel Bryant is chief scientist at OpenCredo and CTO at SpectoLabs. Acting as a technical leader and aspiring to be the archetypal "generalizing specialist," his current work includes enabling agility within organizations by introducing better requirement gathering, focusing on the relevance of architecture within agile development, implementing high-quality testing strategies, and facilitating continuous integration/delivery.

Daniel's current technical expertise focuses on DevOps tooling, cloud/container platforms, and microservice implementations. He is also a leader within the London Java Community (LJC), contributes to several open source projects, writes for well-known technical websites such as O'Reilly, InfoQ, and Voxxed, and regularly presents at international conferences such as QCon, JavaOne, and Devoxx.