

- [About InfoQ](#)
- [Our Audience](#)
- [Contribute](#)
- [About C4Media](#)

- Exclusive updates on:

- 
- 
- 
- 
- 

Facilitating the spread of knowledge and innovation in professional software development



[Login](#)



- [En](#)
- [中文](#)
- [日本](#)
- [Fr](#)
- [Br](#)

1,324,148 Aug unique visitors

- [Development](#)
 - [Java](#)
 - [Clojure](#)
 - [Scala](#)
 - [.Net](#)
 - [C#](#)
 - [Mobile](#)
 - [Android](#)
 - [iOS](#)
 - [IoT](#)
 - [HTML5](#)
 - [JavaScript](#)
 - [Functional Programming](#)
 - [Web API](#)

Featured in Development

[Java 9, OSGi and the Future of Modularity](#)



[The flagship feature of Java 9 will be the new Java Platform Module System \(JPMS\). Given the maturity of OSGi there were technical, political and commercial reasons why another Java module system will soon exist. In this article we compare the two from a technical perspective and see how JPMS and OSGi can work together.](#)

[All in Development](#)

- [Architecture & Design](#)
 - [Architecture](#)
 - [Enterprise Architecture](#)
 - [Scalability/Performance](#)
 - [Design](#)
 - [Case Studies](#)
 - [Microservices](#)
 - [Patterns](#)
 - [Security](#)

Featured in Architecture & Design

[Java 9, OSGi and the Future of Modularity](#)



[The flagship feature of Java 9 will be the new Java Platform Module System \(JPMS\). Given the maturity of OSGi there were technical, political and commercial reasons why another Java module system will soon exist. In this article we compare the two from a technical perspective and see how JPMS and OSGi can work together.](#)

[All in Architecture & Design](#)

- [Data Science](#)
 - [Big Data](#)
 - [Machine Learning](#)
 - [NoSQL](#)
 - [Database](#)
 - [Data Analytics](#)
 - [Streaming](#)

Featured in Data Science

[Wall St. Derivative Risk Solutions Using Geode](#)



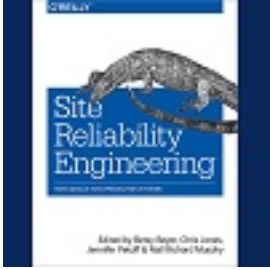
[Andre Langevin and Mike Stolz discuss how Geode forms the core of many Wall Street derivative risk solutions. By externalizing risk from trading systems, Geode-based solutions provide cross-product risk management at speeds suitable for automated hedging, while simultaneously eliminating the back office costs associated with traditional trading system based solutions.](#)

[All in Data Science](#)

- [Culture & Methods](#)
 - [Agile](#)
 - [Leadership](#)
 - [Team Collaboration](#)
 - [Testing](#)
 - [Project Management](#)
 - [UX](#)
 - [Scrum](#)
 - [Lean/Kanban](#)
 - [Personal Growth](#)

Featured in Culture & Methods

[Book Review: Site Reliability Engineering - How Google Runs Production Systems](#)



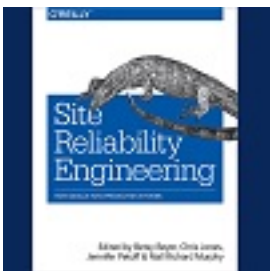
["Site Reliability Engineering - How Google Runs Production Systems" is an open window into Google's experience and expertise on running some of the largest IT systems in the world. The book describes the principles that underpin the Site Reliability Engineering discipline. It also details the key practices that allow Google to grow at breakneck speed without sacrificing performance or reliability.](#)

[All in Culture & Methods](#)

- [DevOps](#)
 - [Infrastructure](#)
 - [Continuous Delivery](#)
 - [Automation](#)
 - [Containers](#)
 - [Cloud](#)

Featured in DevOps

[Book Review: Site Reliability Engineering - How Google Runs Production Systems](#)



["Site Reliability Engineering - How Google Runs Production Systems" is an open window into Google's experience and expertise on running some of the largest IT systems in the world. The book describes the principles that underpin the Site Reliability Engineering discipline. It also details the key practices that allow Google to grow at breakneck speed without sacrificing performance or reliability.](#)

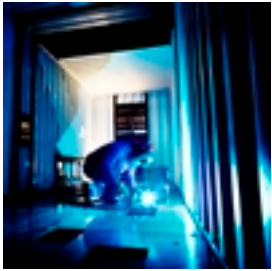
[All in DevOps](#)

- [Streaming](#)
- [Machine Learning](#)
- [Reactive](#)
- [Microservices](#)
- [Containers](#)
- [APM](#)

[All topics](#)

You are here: [InfoQ Homepage](#) [Articles](#) Build Your Own Container Using Less than 100 Lines of Go [The InfoQ Podcast](#)

Build Your Own Container Using Less than 100 Lines of Go



Posted by [Julian Friedman](#) on Apr 22, 2016 / [1 Discuss](#)

- Share



- |



- Y



- f



-



- ["Read later"](#)

- ["My Reading List"](#)

The open source release of Docker in March 2013 triggered a major shift in the way in which the software development industry is aspiring to package and deploy modern applications. The creation of many competing, complimentary and supporting container technologies has followed in the wake of Docker, and this has lead to much hype, and some disillusion, around this space. This article series aims to cut through some of this confusion, and explains how containers are actually being used within the enterprise.

This articles series begins with a look into the core technology behind containers and how this is currently being used by developers, and then examines core challenges with deploying containers in the enterprise, such as integrating containerisation into continuous integration and continuous delivery pipelines, and enhancing monitoring to support a changing workload and potential transience. The series concludes with a look to the future of containerisation, and discusses the role unikernels are currently playing within leading-edge organisations.

This InfoQ article is part of the series "[Containers in the Real World - Stepping Off the Hype Curve](#)". You

can [subscribe](#) to receive notifications via RSS.

Related Vendor Content

[Modern Java EE Design Patterns \(By O'Reilly\) - Download Now](#)

[2016 DevOps Salary Report](#)

[Cloud Portability - Download the FREE InfoQ eMag](#)

[New O'Reilly book: Microservices for Java Developers. A hands-on introduction to frameworks and containers.](#)

[GitHub Integration with Upsource: Why and How](#)

Related Sponsor

[Download Upsource, the Polyglot Code Review Tool](#)



The problem with analogies is that they tend to turn your brain off when you hear them. Some may say that software architecture is "just like" building architecture. No, it's not, and the fact that the analogy sounds good has arguably resulted in quite a lot of harm. In a related fashion, software containerisation is often pitched as providing the ability to move software around “just like” shipping containers move goods around. Not quite. Or at least, it is, but the analogy loses a lot of the detail.

Shipping containers and software containers do share a lot in common. Shipping containers - with their standard shape and size - enable powerful economies of scale and standardisation. And software containers promise many of the same benefits. But, this is a surface-level analogy - a goal rather than a fact.

To really understand what a container is in the world of software, we need to understand what goes into making one. And that's what this article is explains. In the process we'll talk about containers vs containerisation, linux containers (including namespaces, cgroups and layered filesystems), then we'll walk through some code to build a simple container from scratch, and finally talk about what this all really means.

What is a Container, really?

I'd like to play a game. In your head, right now, tell me what a “container” is. Done? Ok. Let me see if I can guess what you might've said:

You might have said one or more of:

- A way to share resources
- Process Isolation
- Kind of like lightweight virtualisation
- Packaging a root filesystem and metadata together
- Kind of like a chroot jail
- Something something shipping container something
- Whatever docker does

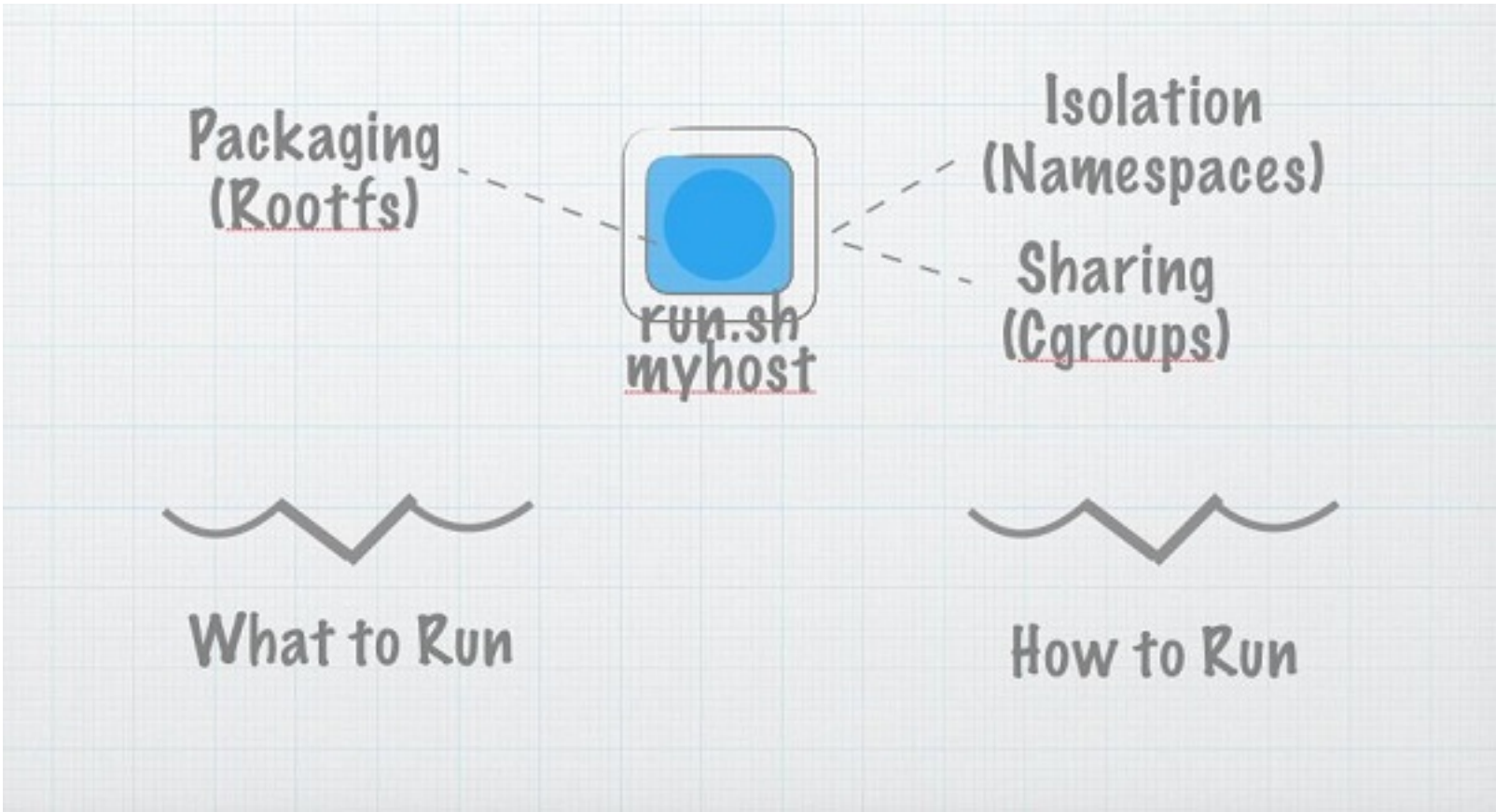
That is quite a lot of things for one word to mean! The word “container” has started to be used for a lot of (sometimes overlapping) concepts. It is used for the analogy of containerisation, and for the technologies used to implement it. If we consider these separately, we get a clearer picture. So, let’s talk about the why of containers, and then the how. (Then we’ll come back to why, again).

In the beginning

In the beginning, there was a program. Let's call the program `run.sh`, and what we’d do is we’d copy it to a remote server, and we would run it. However, running arbitrary code on remote computers is insecure and hard to manage and scale. So we invented virtual private servers and user permissions. And things were good.

But little `run.sh` had dependencies. It needed certain libraries to exist on the host. And it never worked quite the same remotely and locally. (Stop me if you’ve heard this tune). So we invented AMIs (Amazon Machine Images) and VMDKs (VMware images) and Vagrantfiles and so on, and things were good.

Well, they were kind-of good. The bundles were big and it was hard to ship them around effectively because they weren’t very standardised. And so, we invented caching.



And things were good.

Caching is what makes Docker images so much more effective than vmdks or vagrantfiles. It lets us ship the deltas over some common base images rather than moving whole images around. It means we can afford to ship the entire environment from one place to another. It’s why when you ``docker run whatever`` it starts close to immediately even though whatever described the entirety of an operating system image. We’ll talk in more detail about how this works in (section N).

And, really, that’s what containers are about. They’re about bundling up dependencies so we can ship code around in a repeatable, secure way. But that’s a high level goal, not a definition. So let’s talk about the reality.

Building a Container

So (for real this time!) what is a container? It would be nice if creating a container was as simple as just a `create_container` system call. It’s not. But honestly, it’s close.

To talk about containers at the low level, we have to talk about three things. These things are namespaces, cgroups and layered filesystems. There are other things, but these three make up the majority of the magic.

Namespaces

Namespaces provide the isolation needed to run multiple containers on one machine while giving each what appears like it's own environment. There are - at the time of writing - six namespaces. Each can be independently requested and amounts to giving a process (and its children) a view of a subset of the resources of the machine.

The namespaces are:

- **PID:** The pid namespace gives a process and its children their own view of a subset of the processes in the system. Think of it as a mapping table. When a process in a pid namespace asks the kernel for a list of processes, the kernel looks in the mapping table. If the process exists in the table the mapped ID is used instead of the real ID. If it doesn't exist in the mapping table, the kernel pretends it doesn't exist at all. The pid namespace makes the first process created within it pid 1 (by mapping whatever its host ID is to 1), giving the appearance of an isolated process tree in the container.
- **MNT:** In a way, this one is the most important. The mount namespace gives the process's contained within it their own mount table. This means they can mount and unmount directories without affecting other namespaces (including the host namespace). More importantly, in combination with the `pivot_root` syscall - as we'll see - it allows a process to have its own filesystem. This is how we can have a process think it's running on ubuntu, or busybox, or alpine — by swapping out the filesystem the container sees.
- **NET:** The network namespace gives the processes that use it their own network stack. In general only the main network namespace (the one that the processes that start when you start your computer use) will actually have any real physical network cards attached. But we can create virtual ethernet pairs — linked ethernet cards where one end can be placed in one network namespace and one in another creating a virtual link between the network namespaces. Kind of like having multiple ip stacks talking to each other on one host. With a bit of routing magic this allows each container to talk to the real world while isolating each to its own network stack.
- **UTS:** The UTS namespace gives its processes their own view of the system's hostname and domain name. After entering a UTS namespace, setting the hostname or the domain name will not affect other processes.
- **IPC:** The IPC Namespace isolates various inter-process communication mechanisms such as message queues. See the [Namespace](#) docs for more details.
- **USER:** The user namespace was the most recently added, and is the likely the most powerful from a security perspective. The user namespace maps the uids a process sees to a different set of uids (and gids) on the host. This is extremely useful. Using a user namespace we can map the container's root user ID (i.e. 0) to an arbitrary (and unprivileged) uid on the host. This means we can let a container think it has root access - we can even actually give it root-like permissions on container-specific resources - without actually giving it any privileges in the root namespace. The container is free to run processes as uid 0 - which normally would be synonymous with having root permissions - but the kernel is actually mapping that uid under the covers to an unprivileged real uid. Most container systems don't map any uid in the container to uid 0 in the calling namespace: in other words there simply isn't a uid in the container that has real root permissions.

Most container technologies place a user's process into all of the above namespaces and initialise the namespaces to provide a standard environments. This amounts to, for example, creating an initial internet card in the isolated network namespace of the container with connectivity to a real network on the host.

CGroups

Cgroups could honestly be a whole article of their own (and I reserve the right to write one!). I'm going to

address them fairly briefly here because there's not a lot you can't find directly in the documentation once you understand the concepts.

Fundamentally cgroups collect a set of process or task ids together and apply limits to them. Where namespaces isolate a process, cgroups enforce fair (or unfair - it's up to you, go crazy) resource sharing between processes.

Cgroups are exposed by the kernel as a special file system you can mount. You add a process or thread to a cgroup by simply adding process ids to a tasks file, and then read and configure various values by essentially editing files in that directory.

Layered Filesystems

Namespaces and CGroups are the isolation and resource sharing sides of containerisation. They're the big metal sides and the security guard at the dock. Layered Filesystems are how we can efficiently move whole machine images around: they're why the ship floats instead of sinks.

At a basic level, layered filesystems amount to optimising the call to create a copy of the root filesystem for each container. There are numerous ways of doing this. [Btrfs](#) uses copy on write at the filesystem layer. [Aufs](#) uses “union mounts”. Since there are so many ways to achieve this step, this article will just use something horribly simple: we'll really do a copy. It's slow, but it works.

Building the Container

Step One: Setting up the skeleton

Let's just get the rough skeleton in place first. Assuming you have the latest version of the [golang programming language SDK](#) installed, then open an editor, and copy in the following listing.

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    switch os.Args[1] {
    case "run":
        parent()
    case "child":
        child()
    default:
        panic("wat should I do")
    }
}
```

```

func parent() {
    cmd := exec.Command("/proc/self/exe", append([]string{"child"}, os.Args[2:]...))
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        fmt.Println("ERROR", err)
        os.Exit(1)
    }
}

func child() {
    cmd := exec.Command(os.Args[2], os.Args[3:]...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        fmt.Println("ERROR", err)
        os.Exit(1)
    }
}

func must(err error) {
    if err != nil {
        panic(err)
    }
}

```

So what does this do? Well we start in main.go and we read the first argument. If it's 'run' then we run the parent() method, if it's child() we run the child method. The parent method runs `/proc/self/exe` which is a special file containing an in-memory image of the current executable. In other words, we re-run ourselves, but passing child as the first argument.

What is this craziness? Well, right now, not much. It just lets us execute another program that executes a user-requested program (supplied in `os.Args[2:]`). With this simple scaffolding, though, we can create a container.

Step Two: Adding namespaces

To add some namespaces to our program, we just need to add a single line. On line. On the second line of the parent() method, just add this line to tell go to pass some extra flags when it runs the child process.

```
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_NEWNS
}
```

If you run your program now, your program will be running inside the UTS, PID and MNT namespaces!

Step Three: The Root Filesystem

Currently your process is in an isolated set of namespaces (feel free to experiment with adding the other namespaces to your Cloneflags above at this point). But the filesystem looks the same as the host. This is because you're in a mount namespace, but the initial mounts are inherited from the creating namespace.

Let's change that. We need the following four simple lines to swap into a root filesystem. Place them right at the start of the `child()` function.

```
must(syscall.Mount("rootfs", "rootfs", "", syscall.MS_BIND, ""))
must(os.MkdirAll("rootfs/oldrootfs", 0700))
must(syscall.PivotRoot("rootfs", "rootfs/oldrootfs"))
must(os.Chdir("/"))
```

The final two lines are the important bit, they tell the OS to move the current directory at `^` to `rootfs/oldrootfs`, and to swap the new rootfs directory to `^`. After the `pivotroot` call is complete, the `/` directory in the container will refer to the rootfs. (The bind mount call is needed to satisfy some requirements of the `pivotroot` command — the OS requires that `pivotroot` be used to swap two filesystems that are not part of the same tree, which bind mounting the rootfs to itself achieves. Yes, it's pretty silly).

Step Four: Initialising the world of the container

At this point you have a process running in a set of isolated namespaces, with a root filesystem of your choosing. We've skipped setting up cgroups, although this is pretty simple, and we've skipped the root filesystem management that lets you efficiently download and cache the root filesystem images we `pivotroot`-ed into.

We've also skipped the container setup. What you have here is a fresh container in isolated namespaces. We have set the mount namespace by pivoting to the rootfs, but the other namespaces have their default contents. In a real container we'd need to configure the 'world' for the container before running the user process. So, for example, we'd set up networking, swap to the correct uid before running the process, set up any other limits we want (such as dropping capabilities and setting rlimits) and so on. This might well nudge us over 100 lines.

Step Five: Putting it Together

So here it is, a super super simple container, in (way) less than 100 lines of go. Obviously this is intentionally simple. If you use it in production, you are crazy and, more importantly, on your own. But I think seeing something simple and hacky gives a really useful picture of what's going on. So let's look through Listing A.

```
package main
```

```
import (
```

```

    "fmt"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    switch os.Args[1] {
    case "run":
        parent()
    case "child":
        child()
    default:
        panic("wat should I do")
    }
}

func parent() {
    cmd := exec.Command("/proc/self/exe", append([]string{"child"}, os.Args[2:]...))
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_NEWNS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        fmt.Println("ERROR", err)
        os.Exit(1)
    }
}

func child() {
    must(syscall.Mount("rootfs", "rootfs", "", syscall.MS_BIND, ""))
    must(os.MkdirAll("rootfs/oldrootfs", 0700))
    must(syscall.PivotRoot("rootfs", "rootfs/oldrootfs"))
    must(os.Chdir("/"))

    cmd := exec.Command(os.Args[2], os.Args[3:]...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout

```

```
cmd.Stderr = os.Stderr
```

```
if err := cmd.Run(); err != nil {  
    fmt.Println("ERROR", err)  
    os.Exit(1)  
}
```

```
func must(err error) {  
    if err != nil {  
        panic(err)  
    }  
}
```

```
}
```

So, what does it mean?

Here's where I'm going to be a bit controversial. To me, a container is a fantastic way to ship things around and run code cheaply with a good deal of isolation, but that isn't the end of the conversation. Containers are a technology, not a user experience.

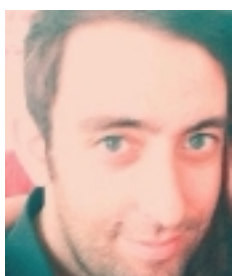
As a user I don't want to push around containers into production any more than a shopper using amazon.com wants to actually phone the docks to organise shipment of their goods. Containers are a fantastic technology to build on top of, but we shouldn't be distracted by an ability to move machine images around from the need to build really great developer experiences.

Platforms as a Service (PaaS) built on top of containers, such as [Cloud Foundry](#), start with a user experience based on *code* rather than containers. For most developers, what they want to do is to push their code and have it run. Behind the scenes, Cloud Foundry - and most other PaaSes - take that code and create a containerised image which is scaled and managed. In the case of Cloud Foundry this uses a buildpack, but you can skip this step and push a Docker image created from a Dockerfile too.

With a PaaS, all of the advantages of containers are still present - consistent environments, efficient resource management etc - but by controlling the user experience a PaaS can both offer a simpler user experience for a developer and perform a few extra tricks like patching the root file system when there are security vulnerabilities. What's more, platforms provide things such as databases and message queues as *services* you can bind to your apps, removing the need to think of everything as containers.

So, we have examined what containers are. Now, what shall we do with them?

About the Author



Julian Friedman is an IBMer working as the engineering lead on Garden, Cloud Foundry's container technology. Before Cloud Foundry Julian worked on a large number of emerging technology projects, from performance work on IBM Watson - the jeopardy playing computer - to some of the earliest iterations of IBM Cloud technologies. He also recently completed a doctorate in the area of Map/Reduce, so intends now, if possible, to spend the rest of his life never thinking about Map/Reduce again. He tweets at @doctor_julz.

The open source release of Docker in March 2013 triggered a major shift in the way in which the software development industry is aspiring to package and deploy modern applications. The creation of many competing, complimentary and supporting container technologies has followed in the wake of Docker, and this has lead to much hype, and some disillusion, around this space. This article series aims to cut through some of this confusion, and explains how containers are actually being used within the enterprise.

This articles series begins with a look into the core technology behind containers and how this is currently being used by developers, and then examines core challenges with deploying containers in the enterprise, such as integrating containerisation into continuous integration and continuous delivery pipelines, and enhancing monitoring to support a changing workload and potential transience. The series concludes with a look to the future of containerisation, and discusses the role unikernels are currently playing within leading-edge organisations.

This InfoQ article is part of the series "[Containers in the Real World - Stepping Off the Hype Curve](#)". You can [subscribe](#) to receive notifications via RSS.

- Personas
- [DevOps](#)
- [Development](#)
- Topics
- [Containers](#)
- [Docker](#)
- [Cloud Computing](#)
- [Containers in the Real World - Stepping Off the Hype Curve](#)

Related Editorial

[Controlling Hybrid Cloud Complexity with Containers: CoreOS, rkt, and Image Standards](#)

[Article Series: Containers in the Real World - Stepping Off the Hype Curve](#)

[Continuous Deployment with Containers](#)

["Using Docker" Book Review and Q&A with Author Adrian Mouat](#)

[The Challenge of Monitoring Containers at Scale](#)

[Controlling Hybrid Cloud Complexity with Containers: CoreOS, rkt, and Image Standards](#)

[Article Series: Containers in the Real World - Stepping Off the Hype Curve](#)

[Continuous Deployment with Containers](#)

["Using Docker" Book Review and Q&A with Author Adrian Mouat](#)

[The Challenge of Monitoring Containers at Scale](#)

Hello stranger!

You need to [Register an InfoQ account](#) or [Login](#) or login to post comments. But there's so much more behind being registered.

Get the most out of the InfoQ experience.

Tell us what you think

Please enter a subject

Message

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

Post Message

Community comments [Watch Thread](#)
[OSX by Paul Fortin Posted Apr 25, 2016 12:43](#)

OSX Apr 25, 2016 12:43 by "Paul Fortin"

Any idea how to rootfs and SysProcAttr for OSX? or is it even possible?

- [Reply](#)
- [Back to top](#)

[Close](#)

by

on

- View
- [Reply](#)
- [Back to top](#)

[Close](#)

Subject Your Reply

[Quote original message](#)

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

Post Message Cancel

[Close](#)

Subject Your Reply






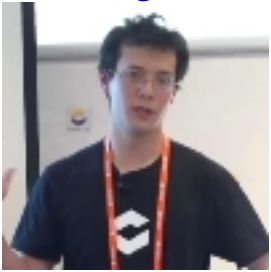

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

☐ Cancel

[Close](#)

RELATED CONTENT

- [HyperGrid Announces Platform for Application Migration to Containers](#) Sep 22, 2016
- [Memory Issues with Linux Control Groups Might Affect Containerized Applications](#) Aug 31, 2016
- [Amir Chaudhry on Unikernels, Docker](#) Aug 07, 2016 
- [InfoQ eMag: Exploring Container Technology in the Real World](#) Jul 15, 2016 
- [Access and Secret Management in Cloud Services](#) Aug 20, 2016 
- [Is Mesos DC/OS a Better Way to Run Docker on AWS?](#) Jul 24, 2016 
- [Peter Bourgon Discusses Coding in Idiomatic Go, Building Microservices with Go-kit, and Weave](#) Jul 13, 2016 
- [Pair Programming in the Cloud with Eclipse Che, Eclipse Flux, Orion, Eclipse IDE and Docker](#) Jul 13, 2016 
- [Controlling Hybrid Cloud Complexity with Containers: CoreOS, rkt, and Image Standards](#) May 01, 2016 
- [Article Series: Containers in the Real World - Stepping Off the Hype Curve](#) Apr 17, 2016



• [rs](#) Apr 14, 2016

• [Download the FOREVER FREE and fully-featured version of TeamCity](#)

Do you practice a traditional approach to Continuous Integration, or prefer Feature Branches with Git or Mercurial? Either way TeamCity has got you covered with a wide range of developer-oriented features to take your team's performance to the next level.

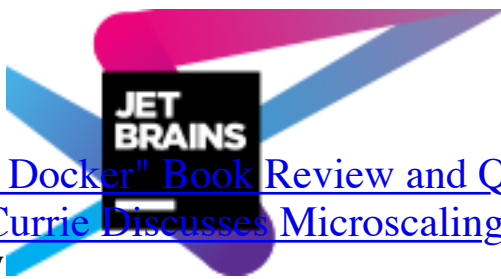
 **TeamCity**
Powerful Continuous Integration
Out of the Box



• [Powerful Continuous Integration Out of the Box - Try the Fully-Featured Version of TeamCity](#)

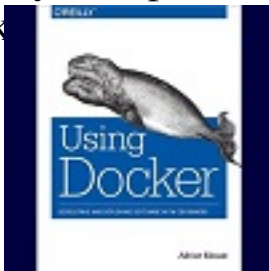
RELATED CONTENT

Initially a Continuous Integration server, TeamCity has encompassed all the features you expect from a mature Continuous Deployment platform. Today it is the best choice you can make



• ["Using Docker" Book Review and Q&A with Author Adrian Mouat](#) Apr 02, 2016

• [Anne Currie Discusses Microscaling, Following in the Footsteps of Google, and the Future of](#)



[Containers](#) Jun 14, 2016



- [Scala, ECS, and Docker: Delayed Execution @Coursera](#) Jun 06, 2016



- [Is HyperContainer the Answer for Cloud Native Applications?](#) Mar 31, 2016



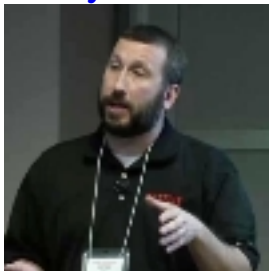
- [Build, Ship and Run Unikernels](#) Jun 28, 2016



- [Lessons Learned from Scheduling One Million Containers with HashiCorp Nomad](#) Apr 08, 2016



- [Netflix Keystone - How We Built a 700B/day Stream Processing Cloud Platform in a Year](#) May 20,

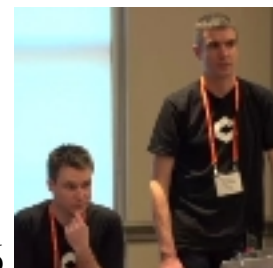


2016

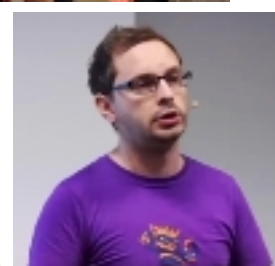
- [The State of Docker and Vagrant Tooling in Eclipse](#) May 05, 2016



- [Code in the Cloud with Eclipse Che and Docker](#) Apr 28, 2016



- [Real World Experience Report on Running Docker](#) Apr 22, 2016



SPONSORED CONTENT

- [Real World Experience Report on Running Docker](#) Apr 22, 2016
- [John Willis on IT Burnout and Mismatch Between Organizational and Personal Values](#) Apr 17, 2016

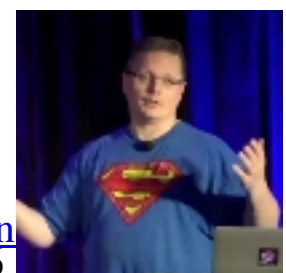


[Exploring the Nuxeo REST API](#)



[Java for Cloud Natives: The Lesson](#)

- [Docker, Vagrant and Kubernetes Walk into an Eclipse'd Bar](#) Apr 06, 2016

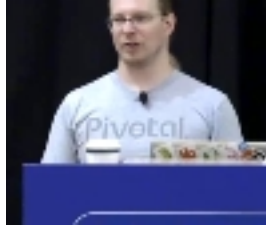


- [Cyber-dojo: Executing Your Code for Fun and Not-for Profit!, Part 2](#) Mar 29, 2016



- [Containers, FTW!](#) Sep 14, 2016

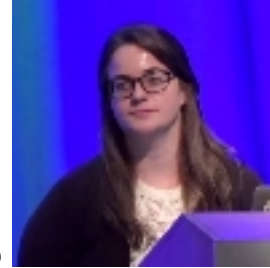




- [The Twelve-Factor Container](#) Sep 07, 2016



- [Reaching Production Faster with Containers in Testing](#) Sep 04, 2016



- [Getting Towards Real Sandbox Containers](#) Aug 29, 2016



- [Help Developers Do What They Love - SpringOne Keynote](#) Aug 29, 2016



- [Scaling Container Architectures with OSS & Mesos](#) Aug 07, 2016



- [Scheduling a Fuller House: Container Mgmt @Netflix](#) Jul 30, 2016

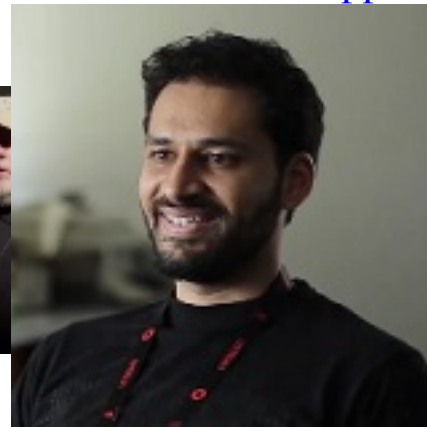
RELATED CONTENT



- [HyperGrid Announces Platform for Application Migration to Containers](#) Sep 22, 2016
- [Memory Issues with Linux Control Groups Might Affect Containerized Applications](#) Aug 31, 2016
- [Offense at Scale](#) Jul 24, 2016



- [DevOps'n the Operating System](#) Jul 24, 2016
- [Amir Chaudhry on Unikernels, Docker](#) Aug 07, 2016



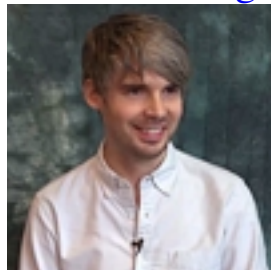
- [InfoQ eMag: Exploring Container Technology in the Real World](#) Jul 15, 2016



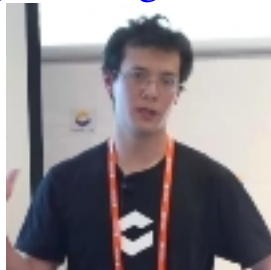
- [Access and Secret Management in Cloud Services](#) Aug 20, 2016



- [Is Mesos DC/OS a Better Way to Run Docker on AWS?](#) Jul 24, 2016
- [Peter Bourgon Discusses Coding in Idiomatic Go, Building Microservices with Go-kit, and Weave](#)



- [Net](#) Jun 30, 2016
- [Pair Programming in the Cloud with Eclipse Che, Eclipse Flux, Orion, Eclipse IDE and Docker](#) Jul



- 13, 2016
- [Controlling Hybrid Cloud Complexity with Containers: CoreOS, rkt, and Image Standards](#) May 01,



- 2016
- [Article Series: Containers in the Real World - Stepping Off the Hype Curve](#) Apr 17, 2016

SPONSORED CONTENT



- [rs](#) Apr 14, 2016

- [Download the FOREVER FREE and fully-featured version of TeamCity](#)

Do you practice a traditional approach to Continuous Integration, or prefer Feature Branches with Git or Mercurial? Either way TeamCity has got you covered with a wide range of developer-oriented features to take your team's performance to the next level.

TC TeamCity
Powerful Continuous Integration
Out of the Box

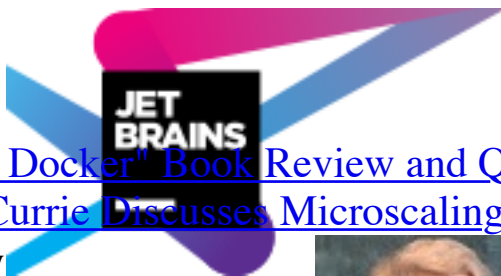


[Powerful Continuous Integration Out of the Box - Try the](#)

[Fully-Featured Version of TeamCity](#)

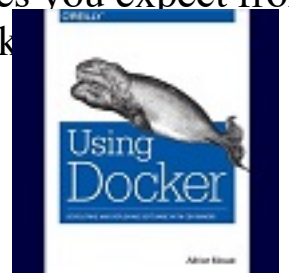
RELATED CONTENT

Initially a Continuous Integration server, TeamCity has encompassed all the features you expect from a mature Continuous Deployment platform. Today it is the best choice you can make



- ["Using Docker" Book Review and Q&A with Author Adrian Mouat](#) Apr 02, 2016

- [Anne Currie Discusses Microscaling, Following in the Footsteps of Google, and the Future of](#)



Sponsored by

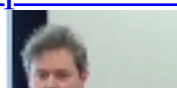


[Containers](#) Jun 14, 2016

- [Scala, ECS, and Docker: Delayed Execution @Coursera](#) Jun 06, 2016



- [Is HyperContainer the Answer for Cloud Native Applications?](#) Mar 31, 2016





- [Build, Ship and Run Unikernels](#) Jun 28, 2016
- [Lessons Learned from Scheduling One Million Containers with HashiCorp Nomad](#) Apr 08, 2016



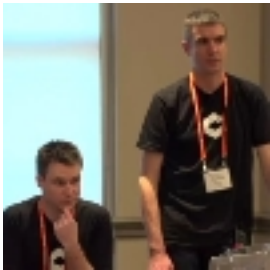
- [Netflix Keystone - How We Built a 700B/day Stream Processing Cloud Platform in a Year](#) May 20,



2016

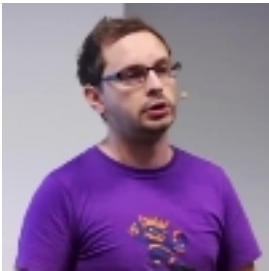


- [The State of Docker and Vagrant Tooling in Eclipse](#) May 05, 2016



- [Code in the Cloud with Eclipse Che and Docker](#) Apr 28, 2016

SPONSORED CONTENT

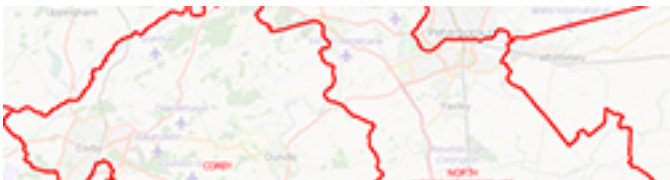


- [g Docker](#) Apr 22, 2016
- [Between Organizational and Personal Values](#) Apr 17, 2016

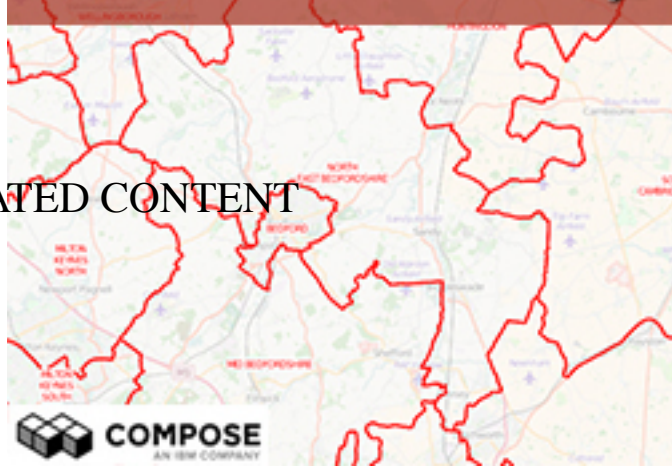


- APPDYNAMICS

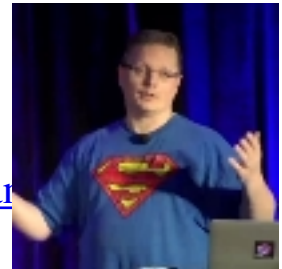
[Top 10 Java Performance Problems](#)



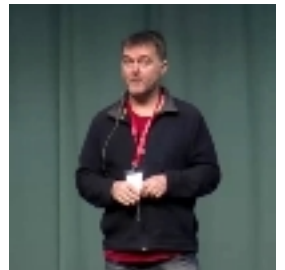
Getting started with Elasticsearch and Node.js



[Getting started with Elasticsearch and Node.js](#)



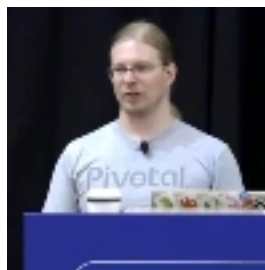
- [Getting started with Elasticsearch and Node.js](#) Apr 06, 2016



- [Cyber-dojo: Executing Your Code for Fun and Not-for Profit!, Part 2](#) Mar 29, 2016



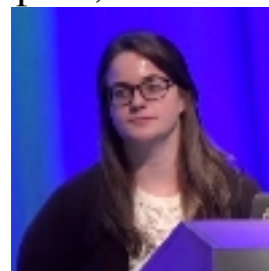
- [Containers, FTW!](#) Sep 14, 2016



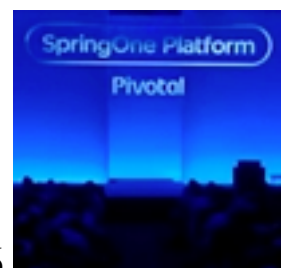
- [The Twelve-Factor Container](#) Sep 07, 2016



- [Reaching Production Faster with Containers in Testing](#) Sep 04, 2016



- [Getting Towards Real Sandbox Containers](#) Aug 29, 2016



- [Help Developers Do What They Love - SpringOne Keynote](#) Aug 29, 2016

- [Scaling Container Architectures with OSS & Mesos](#) Aug 07, 2016

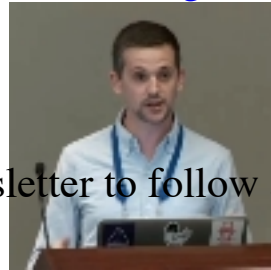


- [Scheduling a Fuller House: Container Mgmt @Netflix](#) Jul 30, 2016



InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ



[Offensive at Scale](#) Jul 24, 2016

Click to view
how it looks like



Your email here

Subscribe

- [DevOps'n the Operating System](#) Jul 24, 2016



Development

[JavaOne 2016 Keynotes Cover the Future of Java Near and Far](#)

[Java 9, OSGi and the Future of Modularity](#)

[Spring Releases Versions 2.3.1 and 2.4.0 Web Services](#)

Architecture & Design

[Java 9, OSGi and the Future of Modularity](#)

[Spring Releases Versions 2.3.1 and 2.4.0 Web Services](#)

[Agile Architecture](#)

Culture & Methods

[Scaling Scrum to Build a New-Technology Printer](#)

[Book Review: Site Reliability Engineering - How Google Runs Production Systems](#)

[Refactoring and Code Smells – A Journey Toward Cleaner Code](#)

Data Science

[Wall St. Derivative Risk Solutions Using Geode](#)

[DeepMind Unveils WaveNet - A Deep Neural Network for Speech and Audio Synthesis](#)

[The InfoQ Podcast: Cathy O'Neil on Pernicious Machine Learning Algorithms and How to Audit Them](#)

DevOps

[HyperGrid Announces Platform for Application Migration to Containers](#)

[JavaOne 2016 Keynotes Cover the Future of Java Near and Far](#)

[Book Review: Site Reliability Engineering - How Google Runs Production Systems](#)

- [Home](#)
- [All topics](#)
- [QCon Conferences](#)
- [About InfoQ](#)
- [Our Audience](#)
- [Contribute](#)
- [About C4Media](#)
- [Create account](#)
- [Login](#)

- **QCons Worldwide**
- [Shanghai](#)
[Oct 20-22, 2016](#)
- [San Francisco](#)
[Nov 7-11, 2016](#)
- [Tokyo 2016](#)
- [London](#)
[Mar 6-10, 2017](#)
- [Beijing](#)
[Apr 16-18, 2017](#)
- [São Paulo](#)
[Apr 24-26, 2017](#)
- [New York](#)
[Jun 26-30, 2017](#)

InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ

Click to view
an example



Your email here

Subscribe

- [Your personalized RSS](#)
- [For daily content and announcements](#)
- [For major community updates](#)
- [For weekly community updates](#)

Personalize Your Main Interests

- ☒ Development
- ☒ Architecture & Design
- ☒ Data Science
- ☒ Culture & Methods
- ☒ DevOps

This affects what content you see on the homepage & your RSS feed. Click preferences to access more fine-grained personalization.

General Feedback

Bugs

Advertising

Editorial

Marketing

feedback@infoq.combugs@infoq.comsales@infoq.comeditors@infoq.commarketing@infoq.com

InfoQ.com and all content copyright © 2006-2016 C4Media Inc. InfoQ.com hosted at [Contegix](#), the best ISP we've ever worked with. [Privacy policy](#)



We notice you’re using an ad blocker

We understand why you use ad blockers. However to keep InfoQ free we need your support. InfoQ will not provide your data to third parties without individual opt-in consent. We only work with advertisers relevant to our readers. Please consider whitelisting us.