▶ Courses (/my-pluralsight-courses) Books (/books)

(https://twitter.com/EltonStoneman) (http://www.pluralsight.com/author/elton-stoneman) (https://github.com/sixeyed) in (http://uk.linkedin.com/in/eltonstoneman) (http://blog.sixeyed.com/rss)

© ELTON STONEMAN (/)

Consultant and trainer.
Microsoft MVP (https://mvp.microsoft.com/en-us/PublicProfile/4028368?fullName=Elton%20Stoneman).
Pluralsight Author (//pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fauthors%2Felton-stoneman).

Docker on Windows (https://amzn.to/2HWLarD)

My Pluralsight Courses (/my-pluralsight-courses)

Using Declarative Jenkins Pipelines (https://pluralsight.pxf.io/eLgGX)

May 2020 | ★★★★★

Using and Managing Jenkins Plugins (https://pluralsight.pxf.io/yNAPv)

April 2020 | ★★★★★

Site Reliability Engineering (SRE): The Big Picture (https://pluralsight.pxf.io/31WqX)

March 2020 | ★★★★⯪

Managing Apps on Kubernetes with Istio (https://pluralsight.pxf.io/Rrr3a)

February 2020 | ★★★★★

Serverless Programming with Fn Project (https://pluralsight.pxf.io/xqBnA)

May 2019 | ★★★★★

Handling Data and Stateful Applications in Docker (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fhandling-data-stateful-applications-docker)

January 2019 | ★★★★½

Monitoring Containerized Application Health with Docker (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fmonitoring-containerized-app-health-docker)

August 2018 | ★★★★★

Managing Load Balancing and Scale in Docker Swarm Mode Clusters (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fmanaging-load-balancing-scale-docker-swarm-clusters)

March 2018 | ★★★★★

Modernizing .NET Apps with Docker (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fmodernizing-dotnet-framework-apps-docker)

December 2017 | ★★★★½

Get Started with Docker Datacenter (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fgetting-started-docker-datacenter)

January 2017 | ★★★★½

Hadoop for .NET Developers (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fhadoop-for-dotnet-developers)

August 2016 | ★★★★½

Managing Azure IaaS with PowerShell (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fmanaging-azureipas-with-powershell)

April 2016 | ★★★★★

HDInsight Deep Dive: Storm, HBase, and Hive (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fhdinsight-deep-dive-storm-hbase-hive)

November 2015 | ★★★★⯪

Real World Big Data in Azure (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Freal-world-big-data-microsoft-azure)

June 2015 | ★★★★⯪

Getting Started with Ubuntu (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fubuntu-getting-started)

February 2015 | ★★★★⯪

Five Essential Tools for Building REST APIs (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Ffive-essential-tools-building-rest-api)

December 2014 | ★★★★⯪

Executable Specifications: End-to-End Acceptance Testing With SpecFlow (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fexecutable-specifications-specflow)

August 2014 | ★★★★⯪

IDisposable Best Practices for C# Developers (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fidisposable-best-practices-csharp-developers)

June 2014 | ★★★★⯪

Message Queue Fundamentals in .NET (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fmessage-queue-fundamentals-dotnet)

April 2014 | ★★★★⯪

C# Extension Methods (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fcsharp-extension-methods)

December 2013 | ★★★★⯪

Implementing the Reactive Manifesto with Azure and AWS (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fimplementing-reactive-manifesto-azure-aws)

October 2013 | ★★★★⯪

Nginx and PHP Fundamentals (https://pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fnginx-php-fundamentals)

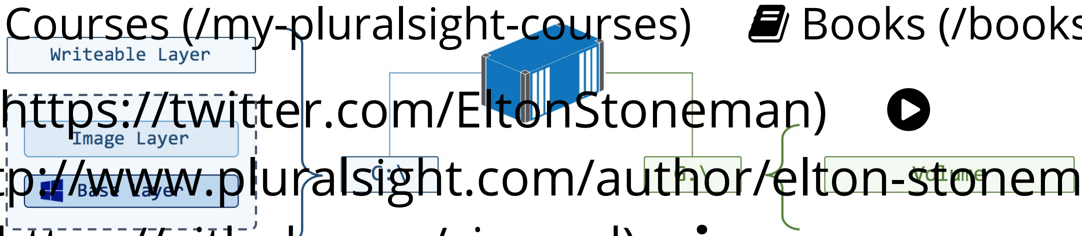
July 2013 | ★★★★⯪

Caching in the .NET Stack: Inside-Out (//pluralsight.pxf.io/c/1197078/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fdotnet-caching-inside-out)

June 2013 | ★★★★★

▶ Courses (/my-pluralsight-courses) Books (/books)

(https://twitter.com/EltonStoneman)
(http://www.pluralsight.com/author/elton-stoneman)
(https://github.com/sixeyed) in
(http://uk.linkedin.com/in/eltonstoneman)
(http://blog.sixeyed.com/rss)



Docker Volumes on Windows - Introducing the `G` Drive

JULY 16, 2017 | << DOCKER ON WINDOWS: THE BOOK (/DOCKER-ON-WINDOWS-THE-BOOK/) | >> WINDOWS WEEKLY DOCKERFILE #1 (/WINDOWS-WEEKLY-DOCKERFILE-1/)

> Update! From Windows 1809 onwards this is no longer an issue!
>
> See 6 Things You Can Do with Docker in Windows Server 2019 That You Couldn't Do in Windows Server 2016 (/what-you-can-do-with-docker-in-windows-server-2019-that-you-couldnt-do-in-windows-server-2016/)

You use Docker volumes (https://docs.docker.com/engine/tutorials/dockervolumes/) to store state outside of containers, so your data survives when you replace the container to update your app. Docker uses symbolic links (https://en.wikipedia.org/wiki/Symbolic_link) to give the volume a friendly path inside the container, like `C:\data`. Some application runtimes try to follow the friendly path to the real location - which is actually outside the container - and get themselves into trouble.

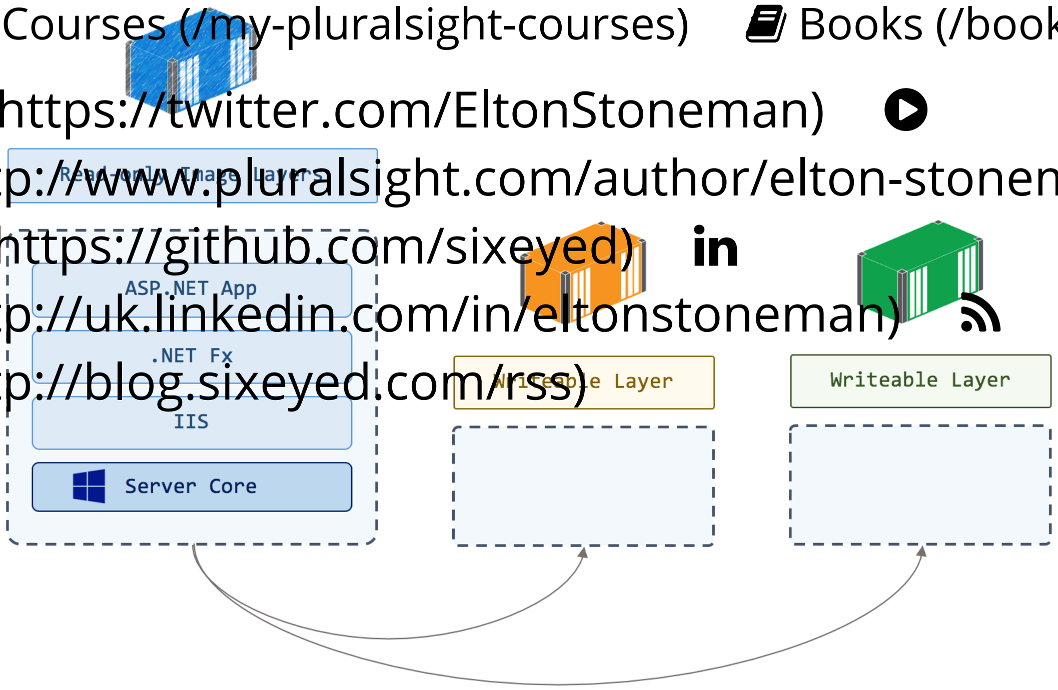
> This issue may not affect all application runtimes, but I have seen it with Windows Docker containers running Java, Node.JS, Go, PHP and .NET Framework apps. So it's pretty widespread.

You can avoid that issue by using a mapped drive (say `G:\`) inside the container. Your app writes to the `G` drive and the runtime happily lets the Windows filesystem take care of actually finding the location, which happens to be a symlink to a directory on the Docker host.

Filesystems in Docker Containers

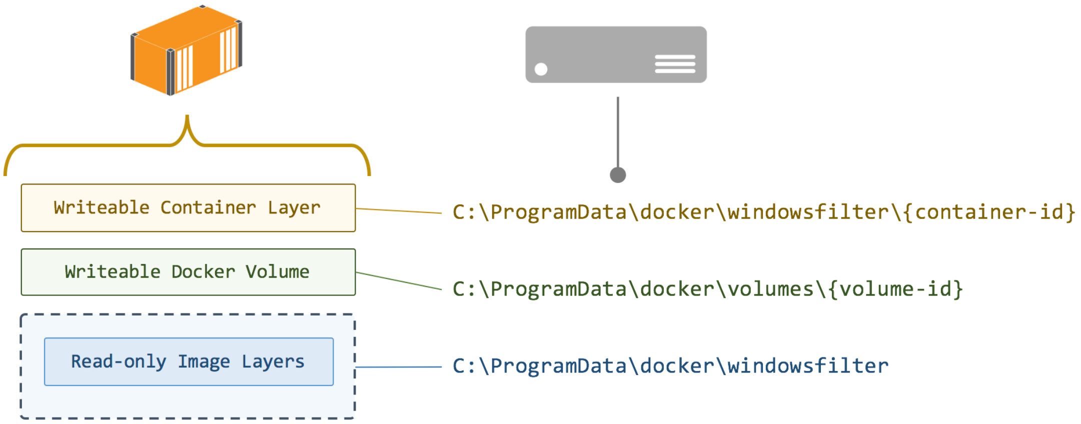
An application running in a container sees a complete filesystem, and the process can read and write any files it has access to. In a Windows Docker container the filesystem consists of a single `C` drive, and you'll see all the usual file paths in there - like `C:\Program Files` and `C:\inetpub`. In reality the `C` drive is composed of many parts, which Docker assembles into a virtual filesystem.

It's important to understand this. It's the basis for how images are shared (https://docs.docker.com/engine/userguide/storagedriver/imagesandcont between multiple containers, and it's the reason why data stored in a container is lost when the container is removed. The virtual filesystem the container sees is built up of many image layers which are read-only and shared, and a final writeable layer which is unique to the container:



When processes inside the container modify files from read-only layers, they're actually copied into the writeable layer. That layer stores the modified version and hides the original. The underlying file in the read-only layer is unchanged, so images don't get modified when containers make changes.

Removing a container removes its writeable layer and all the data in it, so that's not the place to store data if you run a stateful application in a container. You can store state in a volume, which is a separate storage location that one or more containers can access, and has a separate lifecycle to the container:


Writeable Container Layer
Writeable Docker Volume
Read-only Image Layers
C:\ProgramData\docker\windowsfilter\{container-id}
C:\ProgramData\docker\volumes\{volume-id}
C:\ProgramData\docker\windowsfilter

Storing State in Docker Volumes

Using volumes is how you store data in a Dockerized application, so it survives beyond the life of a container. You run your database container with a volume for the data files. When you replace your container from a new image (to deploy a Windows update or a schema change), you use the same volume, and the new container has all the data from the original container.

> The SQL Server Docker lab (https://github.com/docker/labs/tree/master/windows/sql-server) on GitHub walks you through an example of this.

You define volumes in the Dockerfile, specifying the destination path where the volume is presented to the container. Here's a simple example which stores IIS logs in a volume:

```
#escape=`
FROM microsoft/iis
VOLUME C:\inetpub\logs
```

You can build an image from that Dockerfile and run it in a container. When you run `docker container inspect` you will see that there is a mount point listed for the volume:

```
"Mounts": [
            {
                "Type": "volume",
                "Name": "cfc1ab55dbf6e925a1705673ff9f202d0ee2157dcd199c021118
13b05ddddf22",
                "Source": "C:\\ProgramData\\docker\\volumes\\cfc1ab55dbf6e925
a1705673ff9f202d0ee2157dcd199c02111813b05ddddf22\\_data",
                "Destination": "C:\\inetpub\\logs",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ]
```

The source location of the mount shows the physical path on the Docker host where the files for the volume are written - in `C:\ProgramData\docker\volumes`. When IIS writes logs from the container in `C:\Inetpub\logs`, they're actually written to the directory in `C:\ProgramData\docker\volumes` on the host.

> The destination path for a volume must be a new folder, or an existing empty folder. Docker on Windows is different from linux in that respect, you can't use a destination folder which already contains data from the image, and you can't use a single file as a destination.

Docker surfaces the destination directory for the volume as a symbolic link (**symlink**) inside the container, and that's where the trouble begins.

Symlink Directories

Symbolic links have been a part of the Windows filesystem for a long time, but they're nowhere near as popluar as they are in Linux. A symlink is just like an alias, which abstracts the physical location of a file or directory. Like all abstractions, it lets you work at a higher level and ignore the implementation details.

In Linux it's common to install software to a folder which contains the version name - like `/opt/hbase-1.2.3` and then create a sylmink to that directory, with a name that removes the version number - `/opt/hbase`. In all your scripts and shortcuts you use the symlink. When you upgrade the software, you change the symlink to point to the new version and you don't need to change anything else. You can also leave the old version in place and rollback by changing the symlink.

You can do the same in Windows, but it's much less common. The symlink mechanism is how Docker volumes work in Windows. If you `docker container exec` into a running container and look at the volume directory, you'll it listed as a symlink directory (`SYMLINKD`) with a strange path:

```
C:\>dir C:\inetpub
 Volume in drive C has no label.
 Volume Serial Number is 90D3-C0CE

 Directory of C:\inetpub

06/30/2017  10:08 AM    <DIR>          .
06/30/2017  10:08 AM    <DIR>          ..
01/18/2017  07:43 PM    <DIR>          custerr
01/18/2017  07:43 PM    <DIR>          history
01/18/2017  07:43 PM    <SYMLINKD>     logs [\\?\ContainerMappedDirectories\8
305589A-2E5D...]
06/30/2017  10:08 AM    <DIR>          temp
01/18/2017  07:43 PM    <DIR>          wwwroot
```

The `logs` directory is actually a symlink directory, and it points to the path `\\?\ContainerMappedDirectories\8305589A-2E5D...`. The Windows filesystem understands that symlink, so if apps write directly to the `logs` folder, Windows writes to the symlink directory, which is actually the Docker volume on the host.

The trouble really begins when you configure your app to use a volume, and the application runtime tries to follow the symlink. Runtimes like Go, Java, PHP, NodeJS (https://github.com/moby/moby/issues/27537) and even .NET will do this - they resolve the symlink to get the real directory and try to write to the real path. When the "real" path starts with `\\?\ContainerMappedDirectories\`, the runtime can't work with it and the write fails. It might raise an exception, or it might just silently fail to write data. Neither of which is much good for your stateful app.

DOS Devices to the Rescue

The solution - as always - is to introduce another layer of abstraction, so the app runtime doesn't directly use the symlink directory. In the Dockerfile you can create a drive mapping to the volume directory, and configure the app to write to the drive. The runtime just sees a drive as the target and doesn't try to do anything special - it writes the data, and Windows takes care of putting it in the right place.

I use the `G` drive in my Dockerfiles, just to distance it from the `C` drive. Ordinarily you use the subst (https://technet.microsoft.com/en-us/library/bb491006.aspx) utility to create a mapped drive, but that doesn't create a map which persists between sessions. Instead you need to write a registry entry in your Dockerfile to permanently set up the mapped drive:

```
VOLUME C:\data

RUN Set-ItemProperty -Path 'HKLM:\SYSTEM\CurrentControlSet\Control\Session Ma
nager\DOS Devices' -Name 'G:' -Value "\??\C:\data" -Type String;
```

This creates a fake `G` drive which maps to the volume directory `C:\data`. Then you can configure your app to write to the `G` drive and it won't realise the target is a symlink, so it won't try to resolve the path and it will write correctly.

> I use this technique in these Jenkins (https://github.com/sixeyed/docker-on-windows/blob/master/ch10/ch10-jenkins/Dockerfile) and Bonobo (https://github.com/sixeyed/docker-on-windows/blob/master/ch10/ch10-bonobo/Dockerfile) Dockerfiles, where I also set up the `G` drive as the target in the app configuration.

How you configure the storage target depends on the app. Jenkins uses an environment variable, which is very easy. Bonobo uses `Web.config`, which means running some XML updates with PowerShell in the Dockerfile. This technique means you need to mentally map the fake `G` drive to a real Docker volume, but it works with all the apps I've tried, and it also works with volume mounts.

Mounting Volumes

Docker volumes on Windows are always created in the path of the graph driver (https://docs.docker.com/edge/engine/reference/commandline/dockerd/# storage-driver-option), which is where Docker stores all image layers, writeable container layers and volumes. By default the root of the graph driver in Windows is `C:\ProgramData\docker`, but you can mount a volume to a specific directory when you run a container.

I have a server with a single SSD for the `C` drive, which is where my Docker graph is stored. I get fast access to image layers at the cost of zero redundancy, but that's fine because I can always pull images again if the disk fails. For my application data, I want to use the `E` drive which is a RAID array of larger but slower spinning disks.

When I run my local Git server and Jenkins server in Docker containers I use a volume mount, pointing the Docker volume in the container to a location on my RAID array:

```
docker container run -v E:\bonobo:C:\data sixeyed/bonobo
```

> Actually I use a compose file for my services, but that's for a different post.

So now there are multiple mappings from the `G` drive the app uses to the Docker volume, and the underlying storage location:

```
G:\ -> C:\data -> \\?\ContainerMappedDirectories\xyz -> E:\bonobo
```

Book Plug

I cover volumes - and everything else to do with Docker on Windows - in my book Docker on Windows (https://amzn.to/2yxcQxN), which is out now.

If you're not into technical books, all the code samples are on GitHub: sixeyed/docker-on-windows (https://github.com/sixeyed/docker-on-windows) and every sample has a Docker image on the Hub: dockeronwindows (https://hub.docker.com/r/dockeronwindows/).

Use the G Drive For Now

I've hit this problem with lots of different app runtimes, so I've started to do this as the norm with stateful applications. It saves a lot of time to configure the `G` drive first, and ensure the app is writing state to the `G` drive, instead of chasing down issues later.

The root of the problem actually seems to be a change in the file descriptor for symlink directories in Windows Server 2016. Issues have been logged with some of the application runtimes to work correctly with the symlink (like in Go (https://github.com/golang/go/issues/17540) and in Java (bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8172711)), but until they're fixed the G drive solution is the most robust that I've found.

It would be nice if the image format supported this, so you could write `VOLUME G:` in the Dockerfile and hide all this away. But this is a Windows-specific issue and Docker is a platform that works in the same way across multiple operating systems. Drive letters don't mean anything in Linux so I suspect we'll need to use this workaround for a while.

DOCKER (/TAG/DOCKER/) WINDOWS (/TAG/WINDOWS/)

SHARE THIS ARTICLE ON

(HTTPS://WWW.FACEBOOK.COM/SHARER/SHARER.PHP?U=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/)

(HTTPS://TWITTER.COM/SHARE?TEXT=DOCKER%20VOLUMES%20ON%20WINDOWS%20-%20INTRODUCING%20THE%20%60G%60%20DRIVE&URL=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/)

(HTTPS://PLUS.GOOGLE.COM/SHARE?URL=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/)

(HTTP://WWW.DIGG.COM/SUBMIT?URL=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/)

(HTTP://REDDIT.COM/SUBMIT?URL=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/&TITLE=DOCKER VOLUMES ON WINDOWS - INTRODUCING THE `G` DRIVE)

(HTTP://WWW.LINKEDIN.COM/SHAREARTICLE?MINI=TRUE&URL=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/)

(HTTP://WWW.STUMBLEUPON.COM/SUBMIT?URL=HTTP://BLOG.SIXEYED.COM/DOCKER-VOLUMES-ON-WINDOWS-THE-CASE-OF-THE-G-DRIVE/&TITLE=DOCKER VOLUMES ON WINDOWS - INTRODUCING THE `G` DRIVE)

(/author/elton/)

Written by Elton Stoneman (/author/elton/)

LONDON https://pluralsight.pxf.io/YMBGB (https://pluralsight.pxf.io/YMBGB)

(https://twitter.com/EltonStoneman) (http://www.pluralsight.com/author/elton-stoneman) (https://github.com/sixeyed) (http://uk.linkedin.com/in/eltonstoneman) (http://blog.sixeyed.com/rss)