TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY

On-Demand Host-Container Filesystem Sharing

Bassam Yassin and Guy Barshatski
Supervisor: Noam Shalev

NSSL Labs
Winter 2016

Implementing and adding a new feature to Docker open source project which allows a host to add extra volume to a running container

Contents

List Of Figures

1 Abstract

Volume of a Docker container is pre-set once the container got started for the very first time, hence, it is unlikely to add to its volume while running. Means, in order to add an extra volume to a container, you need to terminate, delete it and then re-create it with the new volume you wished you had in the first place.

The problem with this approach, as you have already guessed, is losing the data and state of the container. In other words, all the work that was previously made with the container prior deleting it should be conducted again with the re-created container.

Our project provide a new way to look at a running container, in terms of the file system and the namespaces so it would be possible to attach an external volume with our running container.

In this paper, we are going to show the process of attaching a directory to another filesystem located in a different namespace. Also, we will present the new API command that we added to Docker which does the above.

2 Introduction

The whole objective of this project is adding a new command to the API of Docker as we are going to profoundly explain.

Docker is the name of the company which created, sponsored and maintained the Docker software, which is basically an open source project.

One question which emerged on Docker's community forum was if it is actually possible to attach a volume to a container after it was initiated and currently running without the urge to restart it, and thus losing all the progress made so far.

Needles to say that such a question was catchy and worth investigating, since it conjures the latest incidence where former CIA employee and contractor for the US government Edward Snowden, had managed to copy classified documents from the United States National Security Agency (NSA).

As a computer professional, he exploited a major security breach in the filesystem of the operating system where one could look outside of the scoping allowed.

That being said, our mentor for the project, Noam Shalev, had motivated us to dig deeper to see if it was possible to counter attack such a loose end. Means to change the boundaries of the preset scope that one is allowed to see in the filesystem without impacting any other branches in the system's tree of files.

In this project we were faced with a lot of technical challenges and theoretical ones where Noam, our mentor, was there to direct us to the correct orientation when getting to a dead end. Thus, we had learned more about the jailing mechanisms of UNIX like machines, which Docker harnesses well.

Our main reference and starting point in this project is the blog post written by Jérôme Petazzoni, one of the developers of Docker platform, as he excellently defined the key points we had to look at in order to achieve our goal.

3 Background Theory

3.1 Device

Devices are files that provide simple interfaces to peripheral hardware devices, such as printers and serial ports. But they can also be used to access specific resources on those devices, such as disk partitions.

Device nodes[1] correspond to resources that an operating system's kernel has already allocated. Unix identifies those resources by a *major number* and a *minor number*, both stored as part of the structure of a node.

The assignment of these numbers (minor and major numbers) occurs uniquely in different operating systems and on different computer platforms.

Generally, the major number identifies the device driver and the minor number identifies a particular device that the driver controls.

There are two general kinds of device files in Unix-like operating systems:

1. character special files.
2. block special files.

The difference between them lies in how data written to them and read from them is processed by the operating system and hardware.

3.1.1 Character devices

Character devices are byte-oriented.

They provide unbuffered, synchronous direct access to the hardware device.

For example: a mouse, keyboard.

3.1.2 Block devices

Unlike character devices, block devices will always allow the programmer to read or write a block of any size (including single characters, bytes) and any alignment.

Most systems create both block and character devices to represent hardware like hard disks.

In Linux, to get a character device for a disk one must use the "raw" driver, though one can get the same effect as opening a character device by opening the block device with the Linux-specific `O_DIRECT` flag.

[1] The Inodes of the files which represents the devices.

3.2 Unix-Like Filesystem

Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy.

In other words, in those systems, there is one root directory, and every file existing on the system is located under it somewhere.

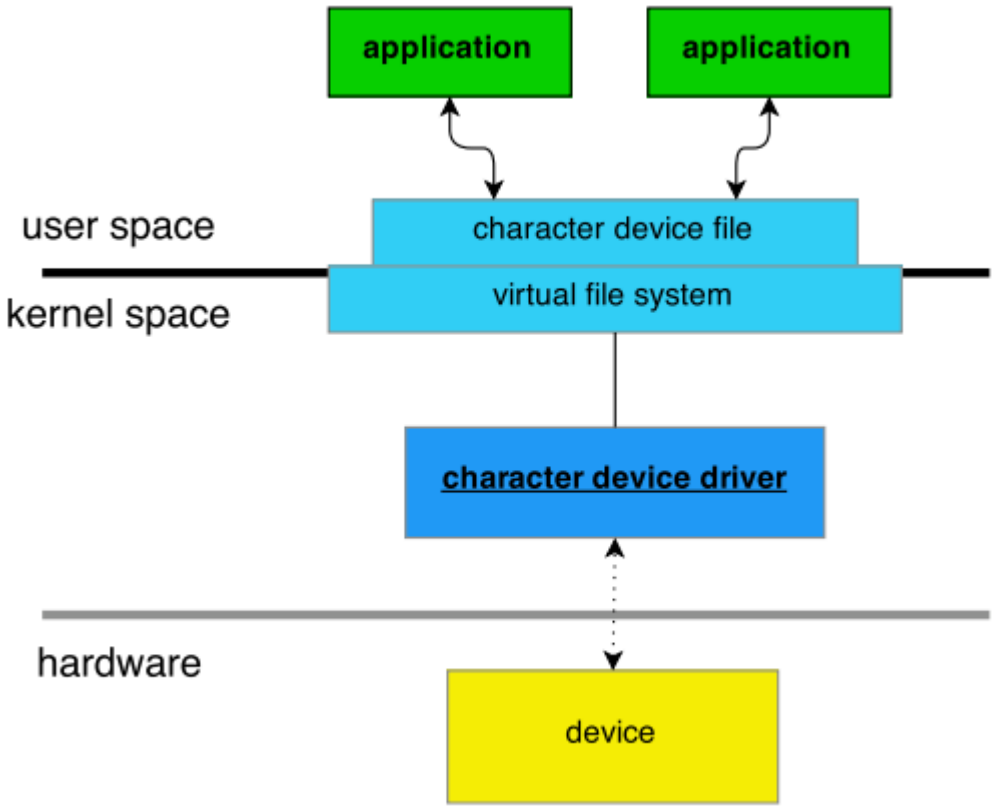


Figure 1 Filesytem in Unix-like operating systems.

To gain access to files on another device, the operating system must first be informed where in the directory tree those files should appear.

This process is called **mounting a file system**.

For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under a certain directory". The directory given to the operating system is called the **mounting point**.

Generally, only the administrator may authorize the mounting of file systems.

In many situations, file systems other than the root need to be available as soon as the operating system has booted. All Unix-like systems therefore provide a facility for mounting file systems at boot time.

3.3 Namespaces and Cgroups

3.3.1 Namespaces

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.

One use of namespaces is to implement containers.

Linux provides the following namespaces:

Namespace	**Isolates**
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
User	User and group IDs
UTS	Hostname and NIS domain name

3.3.1.1 Process Namespace

Historically, the Linux kernel has maintained a single process tree. The tree contains a reference to every process currently running in a parent-child hierarchy.

With the introduction of Linux namespaces, it became possible to have multiple “nested” process trees.

Each process tree can have an entirely isolated set of processes. This can ensure that processes belonging to one process tree cannot inspect or kill - in fact cannot even know of the existence of - processes in other sibling or parent process trees.

Every time a computer with Linux boots up, it starts with just one process, with process identifier (PID) 1. This process is the root of the

process tree, and it initiates the rest of the system[2] by performing the appropriate maintenance work and starting the correct daemons/services.

The PID namespace allows one to spin off a new tree, with its own PID 1 process. The process that does this remains in the parent namespace, in the original tree, but makes the child the root of its own process tree.

With PID namespace isolation, processes in the child namespace have no way of knowing of the parent process's existence. However, processes in the parent namespace have a complete view of processes in the child namespace, as if they were any other process in the parent namespace.

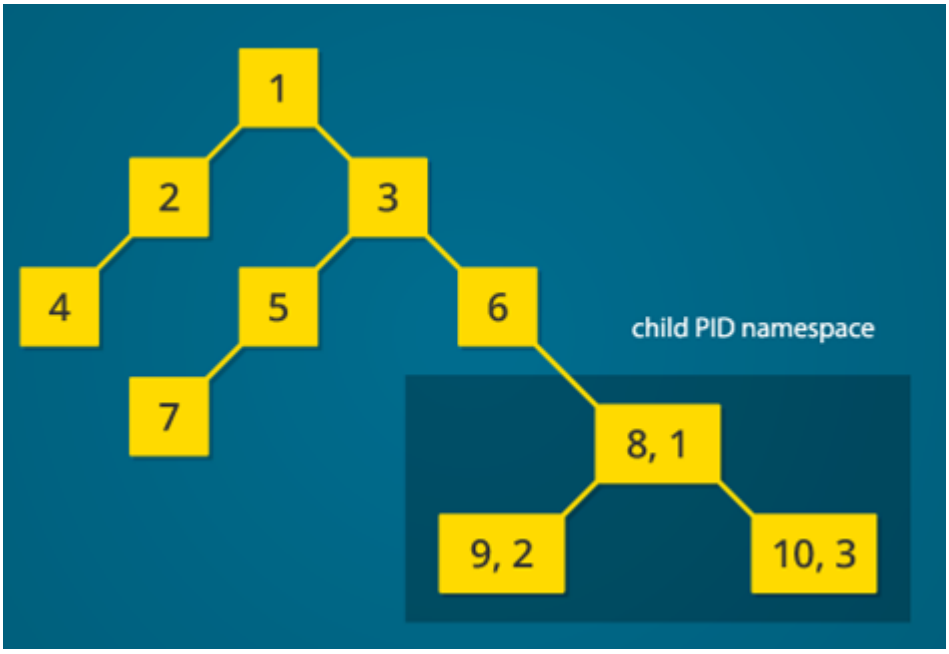


Figure 2 - parent PID Namespace

3.3.1.2 Mount Namespace

Linux maintains a data structure for all the mount-points of the system. It includes information like what disk partitions are mounted, where they are mounted, and whether they are read only, etc.

With Linux namespaces, one can have this data structure cloned, so that processes under different namespaces can change the mount-points without having impact on each other.

[2] Including the reset of the processes

Creating a separate mount namespace allows each of these isolated processes to have a completely different view of the entire system's mount-point structure from the original one. This allows you to have a different root for each isolated process, as well as other mount-points that are specific to those processes.

Example:

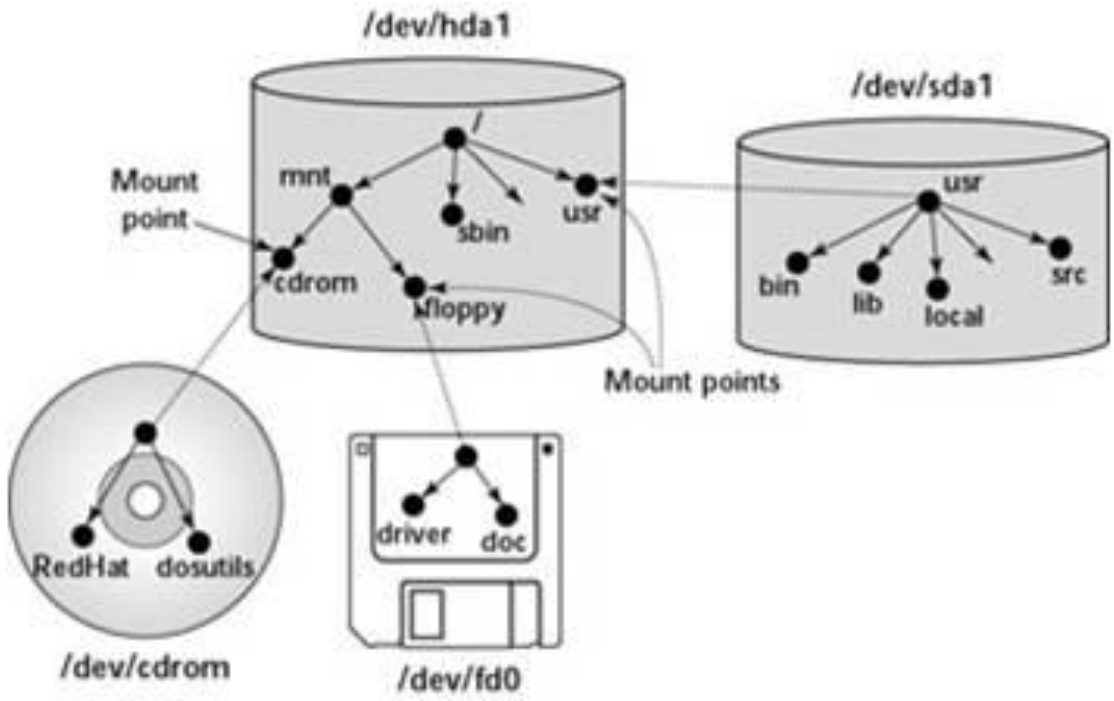


Figure 3 - Mounting Devices on the Red Hat Linux File System

The figure above, shows each device with a name that begins with `/dev`.

For example, `/dev/cdrom` is the CD-ROM drive and `/dev/fd0` is the floppy drive.

These physical devices are mounted at specific mount points on the Red Hat Linux file system.

For example, the CD-ROM drive, `/dev/cdrom`, is mounted on `/mnt/cdrom` in the file system.

After mounting the CD-ROM in this way, the RedHat directory on the CD-ROM appears as `/mnt/cdrom/RedHat` in the Red Hat Linux file system.

3.3.2 Cgroups

Cgroups (control groups) is a Linux kernel feature that limits and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

By using cgroups, system administrators gain fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources.

Cgroups are organized hierarchically, like processes, and child cgroups inherit some of the attributes of their parents.

3.4 NSENTER

It is a small tool allowing to enter into namespaces, especially Docker containers namespaces'.
Technically, it can enter existing namespaces, or spawn a process into a new set of container namespace's.

4 Docker

4.1 What is it?

Docker is a software which makes it possible for the user to wrap and package together a certain application with all of its dependencies and libraries for the purpose of sharing and developing.

The Docker containers wraps the application into a complete filesystem, means that along the code and dependencies, it also includes the system tools which the application is made to runs on. This is a very important concept, since now we have a full compatibility with almost every Linux[3] distribution.

For instance, if we created an application which runs on a Linux machine operating a Debian image, we can wrap that application in a Docker container, and run it on another Linux machine, however, that second machine might be operating on a Ubuntu image. The reason why the application successfully runs on the Ubuntu machine, although it was designed on a Debian machine, is due to the fact that the container simulates an environment of the image which the application compatible with.

4.2 Architecture overview

In this chapter, we will overview the whole architecture and logic-flow of Docker.

Docker is made up from various layers and levels, starting from the user typing the command in the terminal, until it is completely executed.

For efficiency and ease of use, the whole interaction between the user and Docker is maintained via the Docker daemon, a persistent process which is responsible for taking commands from the user and apply changes in the containers accordingly.

The main reason such daemon is vital, due to the fact that there are many components that modify the containers, for example, one component is needed for building the container, another for putting code in the

[3] In this project we worked with Docker on a Linux machine. Although Docker is compatible with Mac and Windows platforms, The feature we talked about is relevant to Linux in that context.

container, as well as a mechanism that monitors when the container exceeds its disk space limits, and many more.

Since all those mechanisms and components want to gain access to the container, we ended up with one huge resource locking problem and concurrency issues.

The solution was the daemon, the only process which is allowed to change in the containers, thus every component that needs to touch any container, must get to the daemon to do the job. That approach was neat and important, since it enables a modular development.

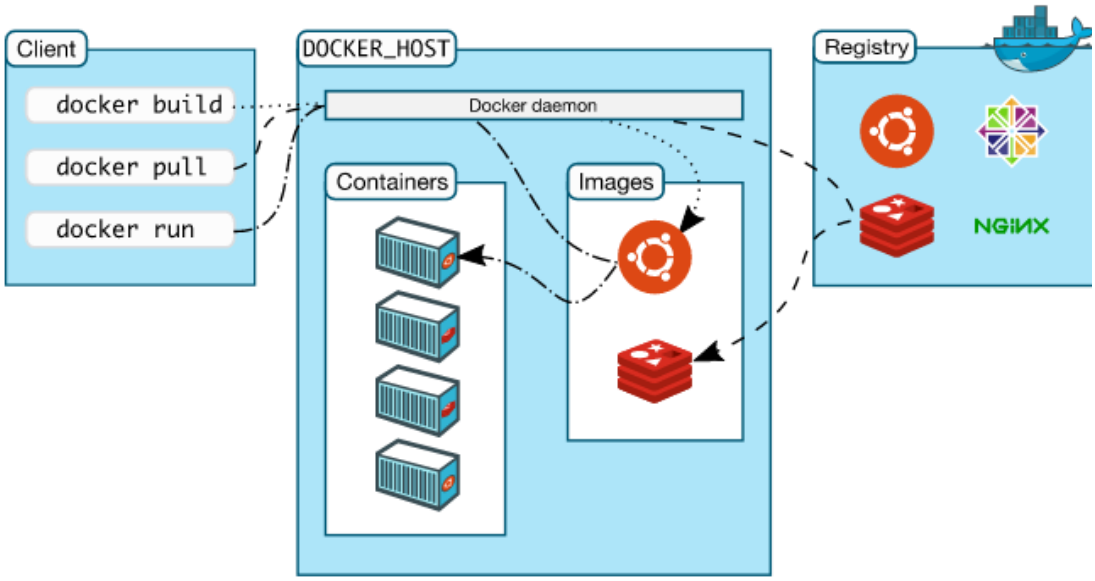


Figure 4: Docker architecture

At Figure above, we can see how the daemon reads the commands from the user and takes care of them.

Initially, when the daemon receives a command about a container, it checks whether it is actually exists in the disk. In case that such a container does not exist, then it must be build[4].

Docker harnesses the **client-server** technology in order to make its use more user friendly; as discussed above, all the interaction is made through the daemon alone.

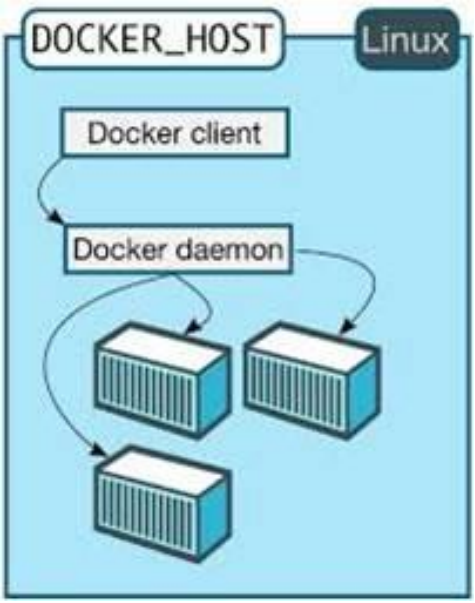


Figure 5: Docker client-server

In Figure 2, we see the overview on how the user interacts with the *Docker-client* whereas the *Docker-client* talks with the daemon.

[4] the command 'docker run' actually means that the user want to create a new container or run an existing container.

4.3 Docker and Virtual Machine

When Comparing between a Docker and VM[5] setup, there are three major elements to take into consideration as key players. Those are:

1. Size: in terms of memory and disk space recruited.
2. Startup (booting): how fast it takes to launch a Docker container and a VM.
3. Integration: how easy is it to integrate a container with a host machine.

A virtual machine(VM) requires some sort of an emulation layer or a hypervisor complete with an operating system installation for each VM. Means, with each VM on your computer, there is a hypervisor, which is heavy since it emulates a different hardware from the one the host already uses, thus occupies more space on the disk comparing to containers, as we will see how later on.

On the other hand, a container uses features[6] in the Linux kernel in order to be able to provide an isolated virtual environment (i.e. disk, memory, networking, etc.), on the same operating system.
In other words, containers use the operating system in question and do not require extra installation of any other tool or utility other than the files required for your application, because it shares the ones of the host machine.

When considering the application in terms of what files on disk it requires, as well as considering what processes are required when instantiated using a container, it will be clear that containers need far less disk space and memory than what is required to run on a virtual machines with all of its own resource prerequisites, as discussed above.

[5] Virtual Machine
[6] Features like tools and utilities, including the namespace jailing mechanism which is already used by the linux host.

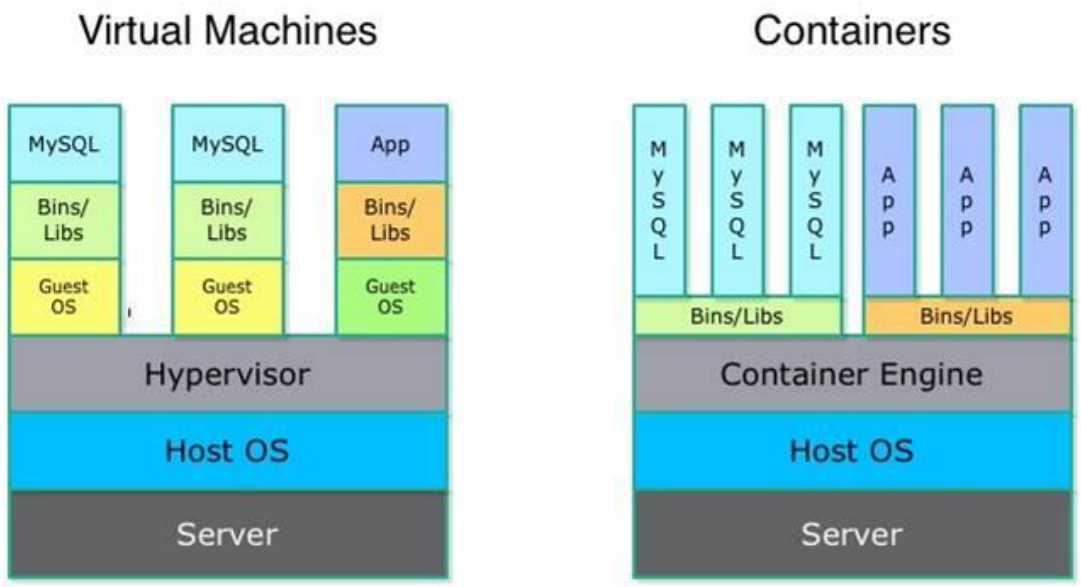


Figure 6: Difference between Docker and VMs

The comparison above will eventually conjures up the question of which of the two is better? The answer is neither.

It really depends on what the user is trying to accomplish, as each has its own purpose.

Sometimes, a virtual machine is needed to provide an environment that is satisfied with a full, complete operating system installation and other associated functionalities. However with other applications, where developers wish to ship the application, either to a testing group or a customer, as a single unit, then Docker will fit the bill.

Moreover, the isolation in VM is much stronger than Docker containers, which yield more immunity to security breaches, due to the hypervisor layer which isolates the hardware from the software layer, which obviously Docker containers lack.

4.4 Docker API

As already been discussed, the Docker daemon interacts with the user, via a certain set of commands.
Those commands (typed in a cmd terminal) are the API of Docker, that tell the daemon which service the user needs to get.

In this chapter, we are going to review the following topics:

1- Important API commands used in the project.
2- Investigating the procedure which occurs when typing the command in the terminal.

How to use the API?

Using the API-command starts by typing the key word docker followed by the name of the command, and finally the set of arguments needed.

Among the API commands which Docker provide are staring and deleting a certain container. Here is how it looks:

```
$ docker start my_container
$ docker rm    my_container
```

As been said earlier, using the commands starts with the keyword 'docker'.
In the first line we used the API command start which takes the name of the container, in this case its name is my_container, which we are interested in starting it.
The second line, demonstrates the use of rm, which deletes a container named my_container, means we now lost access to it – including all data stored in it.

Note how both commands performed completely different tasks, yet received the same argument: the name of the container!

5 Solution Outline

5.1 Project's main objective

In this project we are trying to add an *external volume* to a *running container*.

There are two key words in the above paragraph which sets the boundary of our workbench:

1. External volume.
2. Running container.

An external volume stands for a directory located in the host machine. Since containers are isolated from the host's resources (using namespaces) they have no access (yet!) for that directory which we are trying to attach.

Our challenge is attaching that directory while the container is running. The reason why this is no trivial is due to the fact that the container's volume is pre set to its launch, and due to the namespace isolation as well.

Means that the moment the container is created, its volume is already defined and cannot be tweaked. Therefore, in case of requesting a change, one ought to terminate (kill) the container and re-create it again with the wanted volumes.

The problem stems when all the previous content of the container is lost when it is killed.

5.2 Solution Flow Chart

In the Figure bellow, there are the whole layers of the solution.

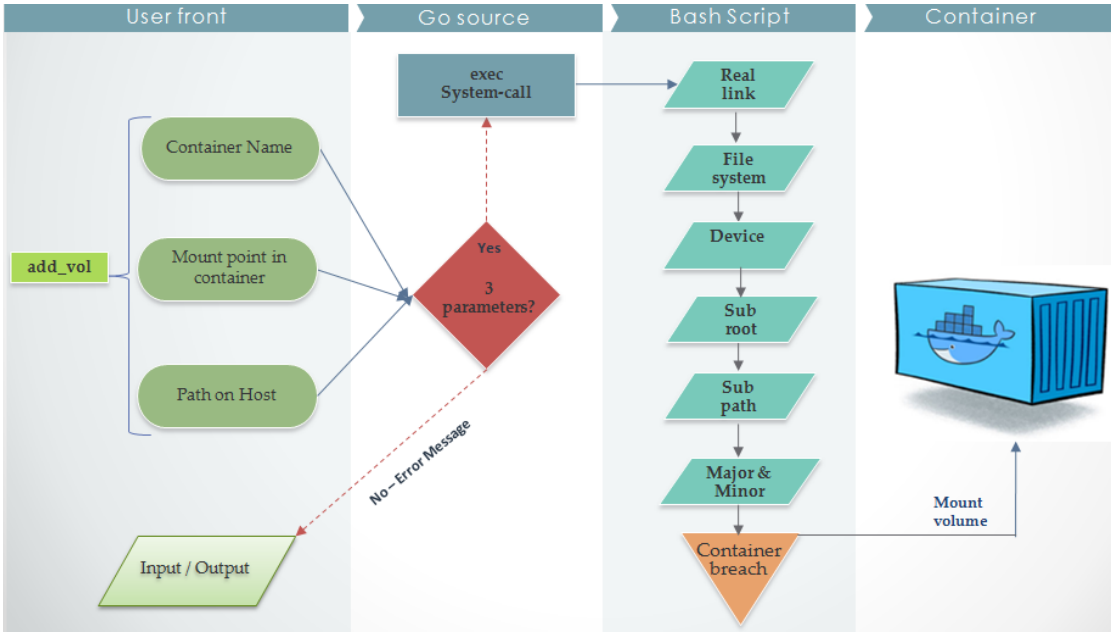


Figure 10: flow diagram of the whole solution

The Figure above explains the solution outline, as it consists of three main layers:

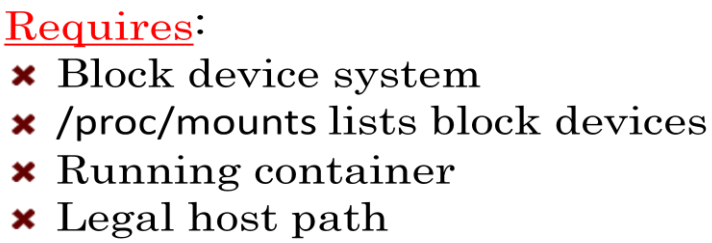
1. User front
2. Go script
3. Bash script

The user front layer is basically when the user provides the command with the three arguments, as we already have seen in chapter 5.5.

The next layer, Go script, checks whether those arguments are valid, where the validity of the arguments is based on their quantity(tree arguments) and their order and whether the container name actually stands for a running container.

In case there was an error, then the solution halts (dos not proceed to layer 3) as well as printing an error message to the user[7] .

Finally, if all the argument are checked positive, then we reach the script. As you can see in Figure, there are seven levels in the script, where the final level (container breach) actually happens in the container namespace.



5.3 Go-lang In Docker

The programming language in which Docker got finally implemented is called *Go*. It was developed at Google Inc in 2007, as it proved practical for system programming due to the various supported system calls and safety features. Moreover, it is a statically typed language (like Python and Bash) with a concurrent programming feature to it.

Another important feature is its compatibility with Bash commands, means, it is possible to run Bash build-in commands in Go! For example:

[7] Error messages are printed in the Linux terminal. Note that each API command could have its unique error message.

```
package main

import (
 "fmt"
 "os/exec"
)

func main() {
  cmd := "echo 'hello world!' "
  out, err := exec.Command("sh","-c",cmd).Output()
  if err != nil {
    fmt.Printf("%s", err)
  }
  fmt.Printf("%s", out)
}
```

Running this simple program from the terminal is done by:

```
$  go run  helloworld.go
```

That feature by itself granted us a huge leap in achieving our solution, which we'll discuss later in the book.

5.4 Adding new command

The following steps show how to add your own custom command to the API of Docker.

First, you must have a GitHub account with the source code of Docker been forked to it.

1- Add a `'xyz.go'` file in the directory `'/api/client'` along with all the API commands. The name of the file(in this example I used `xyz`) must be identical to the name of your command.
2- In that `'xyz.go'` file you should include whatever your command does in a function named `'Cmd[YourCommandName]'`
3- Commit and push the changes to your forked github branch.
4- Make the binary file

Example:

Say you want to add the command '**talk**', so you do the following:

1) Add a new source file named ***'talk.go'***

2) <contents of the file>

3) Update the project on GitHub:

```
a.    $ git add talk.go
b.    $ git commit -s -m "Making a dry run test."
c.    $ git push --set-upstream origin dry-run-test
d.    $ <add your github account name + password>
```

4) Now change directory to docker-master directory and 'make' the project

5) Change directory to /bundles/1.10.0-dev/binary

6) Execute: ./docker-1-10-dev talk

7) DONE!

A very important thing to pay attention to when adding a new API command is the names of the Go script file, it must be the same as your command's name.

Inside the Go script, the function which defines the command – equivalent to `main` in other high level languages, has a naming convention of its own, as follows:

```
func (<return type>) Cmd<Your_command_name>(<args>) error {
     ...
}
```

Basically, what you need to take from the above declaration is `Cmd<Your_command_name>.`
For example, if your command is called `talk`,
then you want to call the function `CmdTalk.` **Pay attention to the uppercase letter T!**

Next, that new command must be 'registered' with all the API commands, so that it will appear when typing `--help` at the terminal. This is achieved by the following code:

```
cmd := Cli.Subcmd("<your_command_name>",
     []string{"CONTAINER"},
     Cli.DockerCommands["<your_command_name>"].Description, true)
```

For sanity check, you need to verify that the number of arguments been passed to the command equals what expected to be received!

In order to see the new command appear in Docker, you must push and commit all the changes to the repository(which been forked earlier), and only then 'compiling' Docker again, by typing `make` in the main directory.

5.5 add_vol.go

So far we understood some key features in the Go-language including system calls and the procedure to be taken in order to integrate a new API to Docker.

The new command which is responsible for adding the new volume, uses two bash scripts for manipulating the file system of the of the container and gathering information from the host's filesystem[8] as we will explore in details in a later chapter.

As already been stated, our new API command in bash is going to read the inputs from the user and call to the bash scripts (execute them).

Bellow is the code snippet for our Go script, called `add_vol.go`. This script sits with the rest of the API commands' scripts in the project tree under `api/client`, as in the Figure bellow.

[8] The filesystem of the host as well as that of the container sit in the same disk of the same machine. Although the terminology here induces the reader to think of them as two different places, they are physically sitting in the same physical disk, in spite of the jailing, which is achieved due to namespaces.

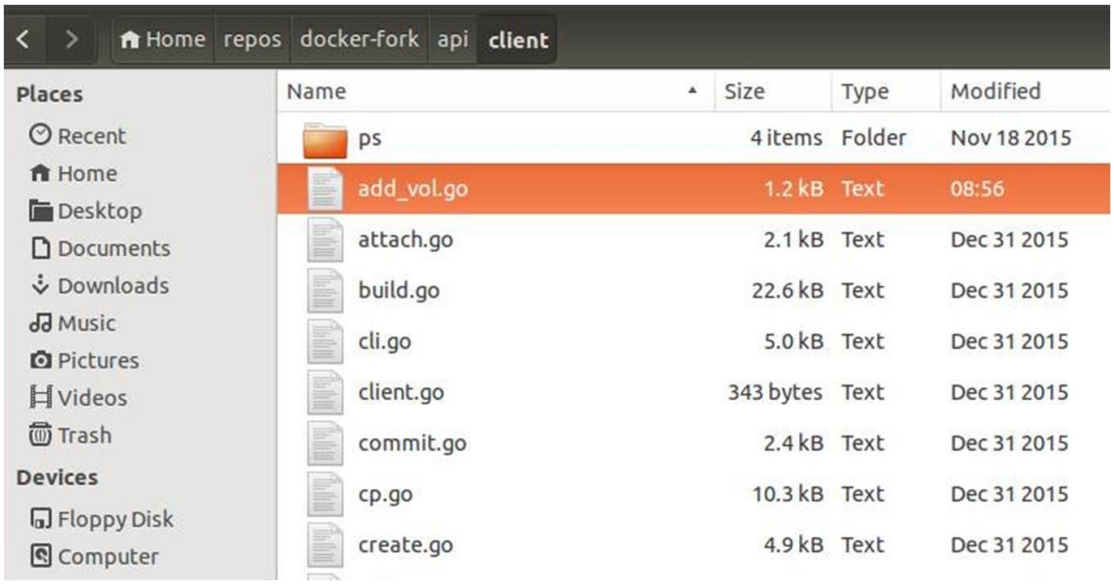


Figure 7: Go script location

Note how it starts with the line *`package client,`* indicating that it is among the scripts which the user (client) calls.

The script opens with registering the new command with all the other API commands of Docker in the same data structure.

This allow Docker to recognise it as a valid command and list it along all the API valid commands when typing `docker --help`.

```
cmd := Cli.Subcmd("add_vol", []string{"CONTAINER"},
     Cli.DockerCommands["add_vol"].Description, true)
```

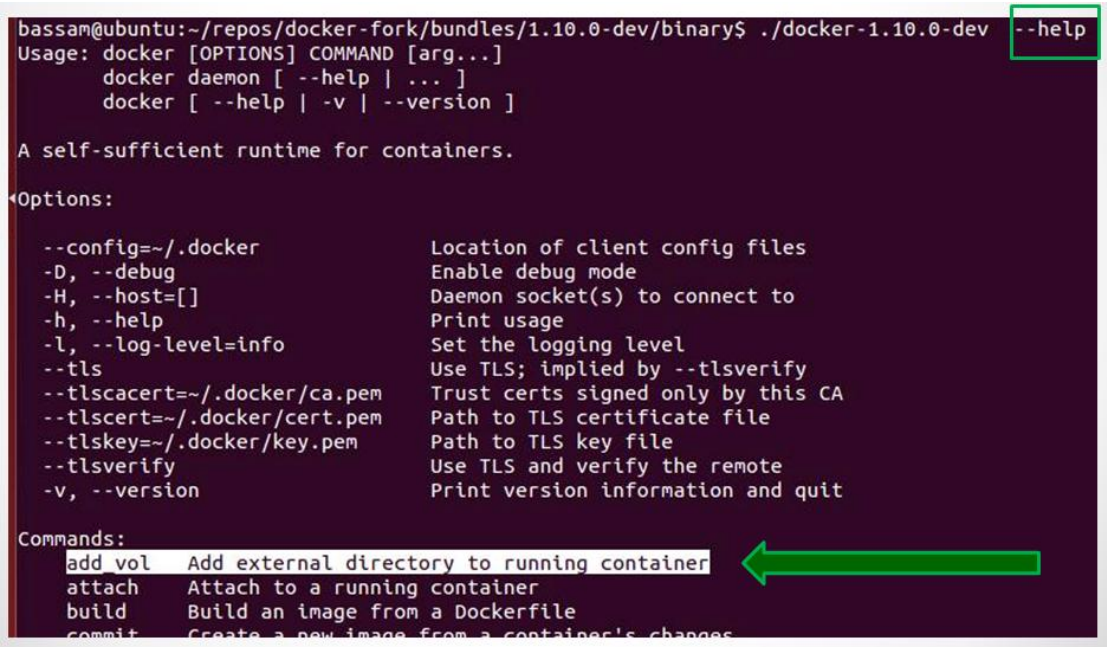


Figure 8: add_vol in --help list

As a part of the sanity check for the correct usage of the command, the script also checks whether the user gave it all the required arguments:

```
cmd.Require(flag.Exact, 3)

cmd.ParseFlags(args, true)
```

The core function called `runBashScript` which receives the path to the script[9] as well as the arguments which the script expects.
Those arguments which being moved to the script are taken from the user:

```
pathOnHost := os.Args[2]

containerName := os.Args[3]

pathOnContainer := os.Args[4]
```

[9] The script should be sitting in the project tree(repository), therefore, the path is in reference to the api/client directory. More on that script in later chapter.

The script is being executed using the sys-call of Go-lang as appears in the script at the following line:

```
out , err := exec.Command("/bin/sh", path,
                          pathOnHost, containerName,
                          pathOnContainer).Output()
```

```
package client

import (
	"fmt"
	"os"
	"os/exec"
	Cli "github.com/docker/docker/cli"
	flag "github.com/docker/docker/pkg/mflag"
)

// adds a file system to the runing container's file system.
// input(parameters): id of the runing container, root of the-
// sub-file-system which to be added.
func (cli *DockerCli) CmdAdd_vol (args ...string) error {

	cmd := Cli.Subcmd("add_vol", []string{"CONTAINER"},
		Cli.DockerCommands["add_vol"].Description, true)

	cmd.Require(flag.Exact, 3)
	cmd.ParseFlags(args, true)

	pathOnHost := os.Args[2]
	containerName := os.Args[3]
	pathOnContainer := os.Args[4]

	bashScriptPath := "../../../add_vol_c/script"

	fmt.Println("adding directory %s to container %s " ,
pathOnHost, containerName)

	runBashScript(bashScriptPath, pathOnHost, containerName,
pathOnContainer)
	return nil
}
//*************************************************************
// runs a bash script which found int path.
func runBashScript(path string,  pathOnHost string,
containerName string,   pathOnContainer string) {
	out , err := exec.Command("/bin/sh", path, pathOnHost,
containerName, pathOnContainer).Output()
	if err != nil {
		fmt.Println("Error: %s", err)
	}
	fmt.Printf("%s", out)
```

Basically, the new command's API would look to the user something like in the Figure bellow.

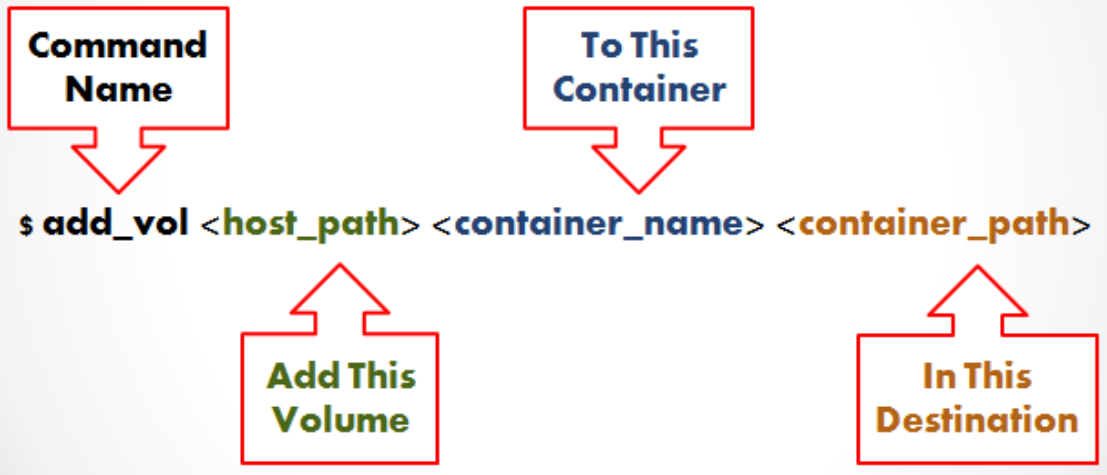


Figure 9: add_volume API

5.6 Bash Script

The bash script plays the major part in the solution, as it does all the heavy lifting of accessing the namespace of the container and attaching the directory in that other namespace.

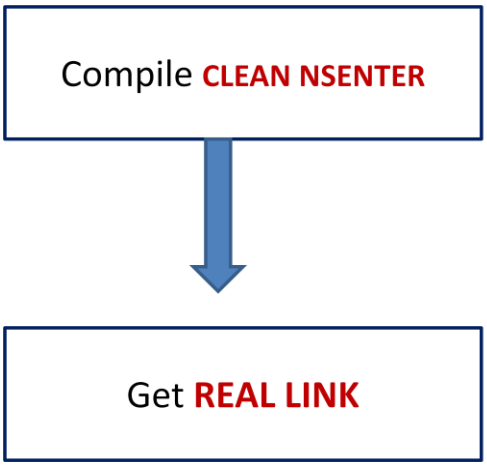
However, there are quit few procedures running behind this fairly complex layer, therefore, we have dedicated already a whole chapter explaining about the filesystem characteristics that the script harnesses, and how it's done exactly.

Meanwhile let us have a look at the script itself (next page).

```
!/bin/sh


set -e
CONTAINER=$2
#the directory in the host ubuntu to be attached.
HOSTPATH=$1
#the directory in the coontainer to add to it the volume
CONTPATH=$3
OLD_WORKING_DIR=`pwd`
NEW_WORKING_DIR=../../../add_vol_c
#------------------------------------------------------------------------
cd $NEW_WORKING_DIR
rm -f my_nsenter importenv
gcc -std=gnu99 -Wall -Werror nsenter.c -o my_nsenter
make LDFLAGS=-static CFLAGS=-Wall importenv
#------------------------------------------------------------------------
#get the real path(not the symlink) of the file system which we need to attach to
the container.
REALPATH=$(readlink --canonicalize $HOSTPATH)
FILESYS=$(df -P $REALPATH | tail -n 1 | awk '{print $6}')
#find on which device the volume(filesystem which we need to attach) sits
while read DEV MOUNT JUNK
do [ $MOUNT = $FILESYS ] && break
done </proc/mounts
[ $MOUNT = $FILESYS ] # Sanity check!
#get the real path for DEV:
DEV=$(readlink --canonicalize $DEV)

while read A B C SUBROOT MOUNT JUNK
do [ $MOUNT = $FILESYS ] && break
done < /proc/self/mountinfo
[ $MOUNT = $FILESYS ] # Moar sanity check!

SUBPATH=$(echo $REALPATH | sed s,^$FILESYS,,)
DEVDEC=$(printf "%d %d" $(stat --format "0x%t 0x%T" $DEV))
#------------------------------------------------------------------------
# since the script 'docker-enter' is sitting with the rest of the API source files
# in '/api/client' , we need to execute it using 'source' cmd (alias is .)
./container_breach $CONTAINER  sh -c \
         "[ -b $DEV ] || mknod --mode 0600 $DEV b $DEVDEC"
./container_breach $CONTAINER  mkdir /tmpmnt
./container_breach $CONTAINER  mount $DEV /tmpmnt
./container_breach $CONTAINER  mkdir -p $CONTPATH
./container_breach $CONTAINER  mount -o bind /tmpmnt/$SUBROOT/$SUBPATH $CONTPATH
./container_breach $CONTAINER  umount /tmpmnt
./container_breach $CONTAINER  rmdir /tmpmnt
cd $OLD_WORKING_DIR
```

5.6.1 Steps in the script

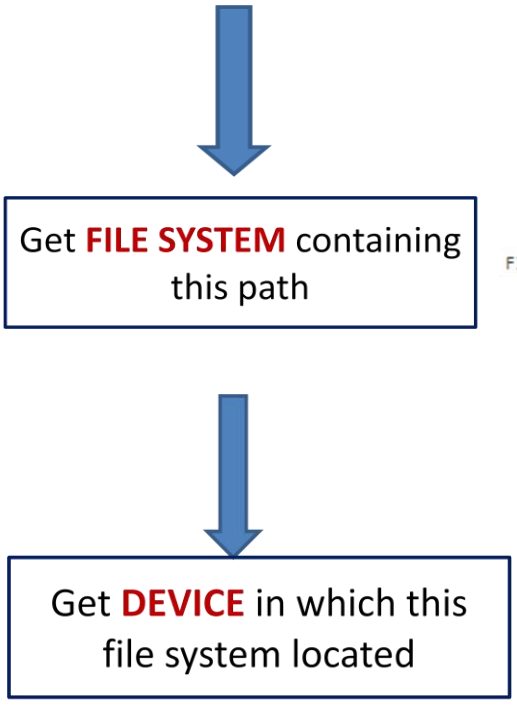
Our goal is to mount the host path directory ($1) into a running container ($2) in the destination path ($3).




```
# create nsenter executable
echo "Script: building nsenter from source: "
gcc -std=gnu99 -Wall -Werror nsenter.c -o my_nsenter
```

```
#get the real path(not the symlink) of the file which we need to attach to the container.
REALPATH=$(readlink --canonicalize $HOSTPATH)
```

Canonicalize the host path, just in case it is a symbolic link - or its path contains any symbolic link.



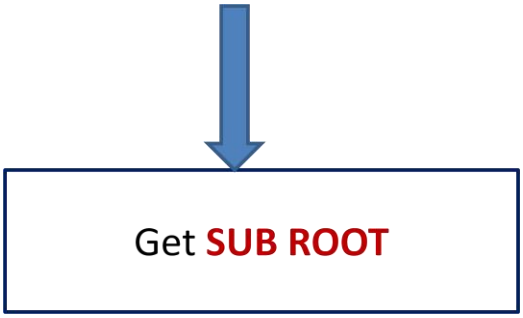

```
FILESYS=$(df -P $REALPATH | tail -n 1 | awk '{print $6}')
```

```
while read DEV MOUNT JUNK
do [ $MOUNT = $FILESYS ] && break
done </proc/mounts
```

```
bassam@ubuntu:~$ sudo cat /proc/mounts
sysfs /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
udev /dev devtmpfs rw,relatime,size=495892k,nr_inodes=123973,mode=755 0 0
devpts /dev/pts devpts rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000 0 0
tmpfs /run tmpfs rw,nosuid,noexec,relatime,size=101316k,mode=755 0 0
```

`/proc/mounts` gives information about current mounted disks

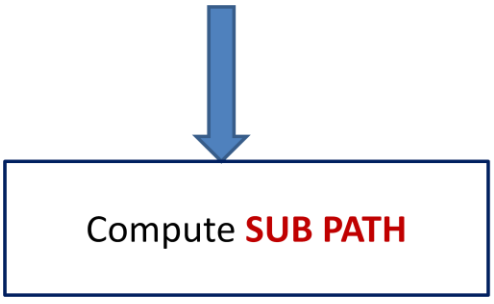
The 1st column specifies the **device** that is mounted.
The 2nd column reveals the **mount point**.
The 3rd column tells the **file-system type**.
The 4th column tells you if it is mounted **read-only (ro)** or **read-write (rw)**.
The 5th and 6th columns are **dummy values** designed to match the format used in `/etc/mtab`.




```
while read A B C SUBROOT MOUNT JUNK
do [ $MOUNT = $FILESYS ] && break
done < /proc/self/mountinfo
```

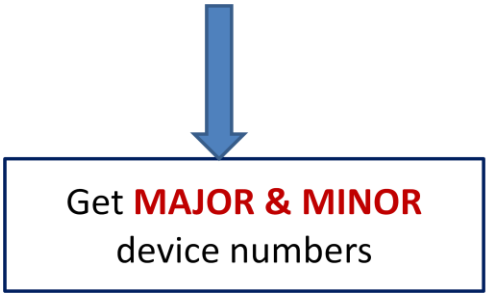
```
bassam@ubuntu:~$ sudo cat /proc/self/mountinfo
17 22 0:16 / /sys rw,nosuid,nodev,noexec,relatime - sysfs sysfs rw
18 22 0:4 / /proc rw,nosuid,nodev,noexec,relatime - proc proc rw
19 22 0:6 / /dev rw,relatime - devtmpfs udev rw,size=495892k,nr_inodes=123973,mode=755
20 19 0:13 / /dev/pts rw,nosuid,noexec,relatime - devpts devpts rw,gid=5,mode=620,ptmxmode=000
21 22 0:17 / /run rw,nosuid,noexec,relatime - tmpfs tmpfs rw,size=101316k,mode=755
```

Here we find the path of the mounted filesystem, within the global filesystem living in this device.



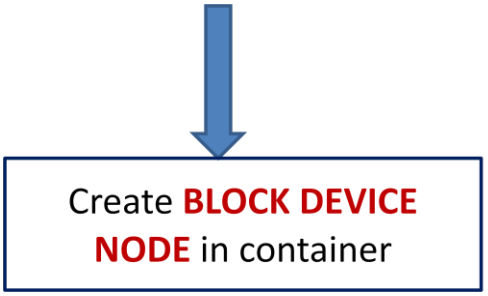

```
SUBPATH=$(echo $REALPATH | sed s,^$FILESYS,,)
```

Here we compute the remaining path - from the mounted filesystem to the volume we want to mount inside the container.



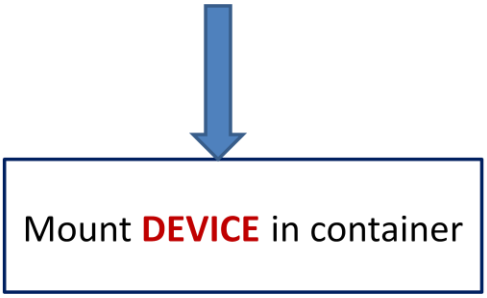

```
DEVDEC=$(printf "%d %d" $(stat --format "0x%t 0x%T" $DEV))
```

Here we calculate the major and minor device numbers for this block device using stat command. Those numbers are in hexadecimal, but we need them in decimal so we will convert them.



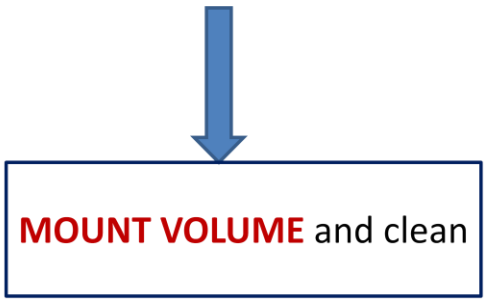

```
./container_breach $CONTAINER  sh -c \
        "[ -b $DEV ] || mknod --mode 0600 $DEV b $DEVDEC"
```

Here we use `container_breach` script that uses NSENTER in order to access containers' namespace from hosts' namespace!




```
./container_breach $CONTAINER  mkdir /tmpmnt
./container_breach $CONTAINER  mount $DEV /tmpmnt
```

We create a temporary mount point and mount the DEVICE.




```
./container_breach $CONTAINER  mkdir -p $CONTPATH
./container_breach $CONTAINER  mount -o bind /tmpmnt/$SUBROOT/$SUBPATH $CONTPATH
./container_breach $CONTAINER  umount /tmpmnt
./container_breach $CONTAINER  rmdir /tmpmnt
```

Here we create users' destination volume if does not exist and bind mount the wanted volume there. Afterwards we `unmount` the temporary mount point and remove it from the container.