Docker: die Linux-Basics unter der Container-Haube (https://entwickler.de/online/besuch-im-docker-maschinenraum-126456.html)

ONLINE (HTTPS://ENTWICKLER.DE/ONLINE)

seit 2 Jahren online

(https://ent... things-...

Inside IoT: Android Things ♥ Intel, IoT ♥ Blockchain & autonomes Auto von Amazon (https://entwickler.de/online/... things-intel-iot-blockchain-autonomes-auto-amazon-579768672.html)

IOT (HTTPS://ENTWICKLER.DE/ONLINE/IOT)

seit 48 Minuten online

(https://ent... features-...

ECMAScript 2017 – das sind die neuen Featur... (https://entwickler.de/online/javascript/ecma... features-579768668.html)

JAVASCRIPT (HTTPS://ENTWICKLER.DE/ONLINE/JA...

Besuch im Docker-...aschinenraum

Docker: d...e Linux-Basics unter der Containe...-Haube

jürgen Brunk, Mat...as Albert und Nils Magnus 2 Jahren online Kommentare

Der Hauptverdi...nst von Docker liegt in seiner standardisierten Hülle: Einheitliche Bes...hreibungen und Abmessungen vereinfachen das Verladen, Teiler...und Ausführen von Anwendungen. Unter der Motorhaube gre...ft Docker auf Bestehendes zurück: Die Basisfunktion...einer Sandbox sowie Namespaces, Cgroups und Chroot sind sch...n länger Bestandteil des Linux-Kerns. Wer also die richtigen Komm...ndos bemüht, kann nachvollziehen, wie es im Maschinenraum...von Docker aussieht. Das Wissen darüber nützt nicht zuletzt, um sich...einen Eindruck von der Wirkungsweise und Sicherheit des C...ntainers zu verschaffen.

...hutterstock.com/donvictorio

Viele Vorzüge von Docker schätzen Anwender erst dann richtig, wenn sie selbst Hand angeleg: und versucht haben, seine Funktionen nachzubilden. Dieser Artikel zeichnet mit Linux-Bordmitteln nach, was Docker leistet und wie ein Container funktioniert. Dabei geht es nicht darum, ein besseres Framework, sondern vielmehr Verständnis zu schaffen.

MEHR ZUM THEMA

Docker Network Magic – der elegante Umgang mit Netzwerken

(https://entwickler.de/online/development/docker-netzwerk-container-microservices-126443.html)

Die einfachste Form der Container-Virtualisierung gibt es schon bemerkenswert lange: Das erstmals 1982 eingesetzte Unix-Kommando *chroot* verändert das Root-Verzeichnis eines Prozesses und seiner Kinder (Kasten: „Chroot-Umgebungen"). Es spricht sich „change root" aus und verweist auf den gleichnamigen Systemaufruf im Kernel von Unix- und Linux-Systemen. Damit realisiert es eine sehr einfache Sandbox und verhindert, dass Programme auf Dateien außerhalb der neuen Wurzel zugreifen. Seit diesen ersten Schritten gab es bis heute eine ganze Reihe von Meilensteinen, die zur heutigen Container-Technik geführt haben (Abb. 1).

(https://entwickler.de/wp-content/uploads/2015/03/magnus_maschinenraum_1.png)

Abb. 1: Die Geschichte der Entwicklungen, die zur heutigen Container-Technik mit Docker geführt hat, ist lang: erste Ansätze zur Dateisystemisolation datieren zurück bis 1979 (zum Vergrößern klicken)

Chroot-Umgebungen

Den Systemaufruf *chroot()* gibt es schon seit 1979. Unix-Pionier Bill Joy brachte ihn 1982 in BSD-Unix ein – aus ganz ähnlichen Gründen, wie er heute noch eingesetzt wird: Er wollte nämlich die Installation und das Build-System testen, ohne sein eigentliches Betriebssystem zu gefährden. Heute kennt fast jedes unixoide Betriebssystem das oft auch als „Jail" bezeichnete Prinzip. Der Systemaufruf benötigt besondere Privilegien, die mit Root-Rechten einhergehen. Die Kernelfunktion selbst wechselt nicht in das neue Wurzelverzeichnis hinein. Das ist auch mit Grund dafür, dass die Funktion nicht als Sicherheitsfunktion herhalten kann. Der folgende Code demonstriert, von einem User mit Root-Rechten ausgeführt, wie sich aus einer solchen Sandbox entkommen lässt: Er startet nämlich eine neue Shell, die Zugriff auf die äußere Verzeichnisstruktur hat:

```
#include <unistd.h>
#define DIR "xxx"
int main() {
  int i;
  mkdir(DIR, 0755);
  chroot(DIR);
  for (i = 0; i < 1024; i++) chdir("..");
  chroot(".");
  execl("/bin/sh", "-i", NULL);
}
```

Der auf der Kommandozeile genutzte gleichnamige Befehl *chroot* umschifft diese Probleme aber und sorgt dafür, dass der Aufrufer in die Sandbox wechselt. Sollte der Aufruf mit dem dürren Fehler *No such file or directory (ENOENT)* scheitern, so liegt dies meist an fehlenden dynamischen Bibliotheken. Welche das sind, zeigt ldd für das zu startende Executable an. Für erste Experimente zum Herantasten bieten sich statisch gelinkte Binarys an, etwa die als */bin/static-sh* abgelegte Busybox.

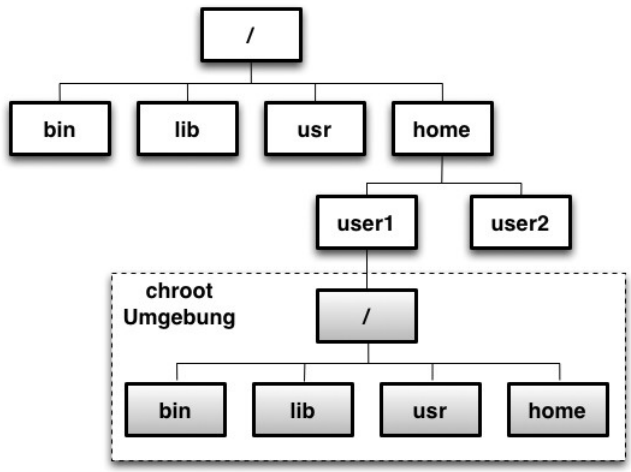
Die auch Jails genannten Umgebungen nutzen Entwickler beispielsweise, um Build Chains zu erstellen, in denen sie in einem abgetrennten Bereich neue Software übersetzen, installieren und testen. Administratoren hingegen sperren gerne bedrohte Dienste wie Bind, Apache oder einen FTP-Server in solchen Containern ein. Auf diese Weise halten sie Einbrecher in einem abgeschotteten Bereich gefangen, sollten diese Sicherheitslücken ausnutzen.

Umfangreiche Zutatenliste

Um einen Jail anzulegen, muss der Administrator dem eingesperrten Prozess die von ihm benötigten Bibliotheken, Werkzeuge und Frameworks bereitstellen. Wer sich viel Arbeit sparen möchte, erledigt dies mittels *debootstrap* und erzeugt sich – quasi das Äquivalent zu einem Basis-Image bei Docker – eine minimale, aber voll funktionstüchtige Debian-Installation. Das Kommando muss mit Root-Rechten ausgeführt werden und erwartet als Argumente die Debian-Version und ein Verzeichnis, das später zum neuen Wurzelverzeichnis wird: *$ sudo debootstrap wheezy ./wheezy_chroot*. Dieser Vorgang lädt innerhalb einiger Minuten rund 250 Megabyte an Softwarepaketen von externen Mirror-Servern herunter und installiert sie im Verzeichnis *wheezy_chroot* (Abb. 2). Für andere Linux-Distributionen gibt

es ähnliche Werkzeuge und Skripte, etwa *febootstrap* für Fedora oder *pacstrap* für Arch Linux. Unter Ubuntu steht ebenfalls *debootstrap* als Paket bei den Bordmitteln zur Verfügung.

Docker nimmt Anwendern diese Fleißaufgabe ab, nennt das Ergebnis Docker-Image und bietet sie sogar in einem eingebauten Repository an. Für die meisten Aufgaben reichen die vielen dort angebotenen Basis-Images bereits aus. Nur wer darüber hinausgehende Anforderungen umsetzen möchte, sollte einmal einen Blick auf die Vielzahl angebotener Skripte werfen (https://github.com/rossbachp/dockerbox/tree/master/docker-images/base-ubuntu), die solche Miniumgebungen zusammenstellen.



(https://entwickler.de/wp-content/uploads/2015/01/magnus_maschinenraum_2.jpg)

Abb. 2: Mittels „debootstrap" entsteht im Unterverzeichnis „wheezy_chroot" eine neue Instanz einer Debian-Installation; anschließend wechselt „chroot" in diese neue Sandbox hinein

Mit dem Kommando *chroot* betritt der Administrator seine neue Umgebung. Dazu übergibt er dem Kommando zwei Argumente: zuerst das neue Wurzelverzeichnis *wheezy_chroot* und dann den Pfad zum Programm, welches als erster Prozess laufen soll. Im Beispiel ist das einfach eine Shell: *$ sudo chroot ./wheezy_chroot /bin/bash*.

Wer sich in seiner Chroot-Umgebung umsieht, bemerkt auf den ersten Blick nicht viel. Die Dateien und Verzeichnisse, die *ls* anzeigt, ähneln denen in jedem Unix-Wurzelverzeichnis. Auch ein Wechsel in die Unterverzeichnisse mit *cd* gelingt wie gewohnt. Die Dateien des Hostsystems bleiben jedoch verborgen.

Zu viel Freiheit

Experimentierfreudige verifizieren mit *ping* die Erreichbarkeit eines bekannten Hosts über das Netzwerk. Weiterhin lässt sich sogar der Hostname des Containers mittels *hostname* ändern. Hier zeigen sich jedoch die Grenzen des Jails: Die Namensänderung modifiziert nämlich gleichzeitig den Hostnamen der Gastgebermaschine. Schaut sich der Systemverwalter die aktuellen Prozesse an, meldet ps ihm einen Fehler:

```
wheezy_chroot# ps ax
Error, do this: mount -t proc proc /proc
```

Eine Kontrolle mit *mount* ohne Argumente zeigt, dass im Container bislang keine virtuellen Dateisysteme in der Chroot-Umgebung gemountet sind. Dies lässt sich leicht nachholen und erneut kontrollieren:

```
wheezy_chroot# mount -t proc proc /proc
wheezy_chroot# mount
proc on /proc type proc (rw)
```

Eine erneute Auflistung der aktuellen Prozesse mittels *ps ax* zeigt nun alle Prozesse des Hosts an – nicht nur den bislang einzigen im Container. Auch lassen sich alle Prozesse beenden – ebenfalls diejenigen, die nicht aus dem Jail heraus gestartet wurden. Es wäre sogar möglich, den Host neu zu booten. Auch wenn die Chroot-Umgebung ursprünglich nicht als Sicherheitsfeature entworfen wurde, implementiert sie mit ein paar Abstrichen eine Sandbox auf der Ebene des Dateisystems. Sie isoliert aber weder Prozesse, noch Benutzer oder die Netzwerkressourcen voneinander. Weil das Unix-API teilweise über vierzig Jahre in die Vergangenheit zurückreicht, erscheint es bei der Menge von vorhandenem Code nahezu aussichtslos, die vielen System- und Bibliothekaufrufe allesamt anzupassen. Genau das benötigt aber eine Isolation von Betriebsmitteln, wenn sie generisch funktionieren soll.

Eigene Bezeichner für Ressourcen durch Namensräume

Um dieses Problem zu lösen, haben die Kernelentwickler die Namespaces implementiert (Kasten: „Namespaces“). Möchte ein Prozess künftig für sich und seine Kinder private Prozess-IDs (PIDs) verwalten, so erzeugt er neuen Namensraum vom Typ *CLONE_NEWPID* und tritt in ihn ein. Danach erscheint es dem Prozess und seinen Abkömmlingen so, als ob die anderen PIDs gar nicht vorhanden wären.

Namespaces
Eine Instanz eines Namespace definiert eine neue Umgebung, die bestimmte Betriebsressourcen wie Prozessuser, das Dateisystem oder das Netzwerk auf eine sehr leichtgewichtige Weise virtualisiert [2]. Sie abstrahieren die jeweils globale Systemressource so, dass sie für einen Prozess innerhalb des jeweiligen Namespace wie eine eigenständige isolierte Instanz aussieht. Leichtgewichtig bedeutet dabei, dass dazu kein Hypervisor nötig ist, sondern die Prozesse schlicht die anderen Instanzen einer Ressource nicht sehen, diese aber weiter im gleichen Kernel weiterlaufen.
Im Jahr 2002 hat Kernelentwickler Al Viro den ersten Namespace *CLONE_NEWNS* in die Kernelversion 2.4.19 eingebracht. Er ist heute meist unter dem Namen „Mount-Namespace“ bekannt [3]. Seither haben die Kernelentwickler noch fünf weitere Systemressourcen abstrahiert. Seitdem Anfang 2013 im Kernel 3.8 auch der User-Namespace *(CLONE_NEWUSER)* fertig implementiert ist, gibt es Abstraktionen für das Dateisystem *(CLONE_NEWNS)*, den Hostnamen *(CLONE_NEWUTS)*, die Interprozesskommunikation *(CLONE_NEWIPC)*, das Netzwerk *(CLONE_NEWNET)* und die Prozess-IDs *(CLONE_NEWPID)*. Diese Abstraktionsschicht ist eine wichtige Voraussetzung, auf die Docker in seinem Kern aufsetzt. Auf diese Weise greift der Container nämlich auf bekannte Mechanismen wie Netzinterfaces oder Prozess-IDs zurück, ohne dass Anwender die darin laufende Software modifizieren müssen.
24 Namespaces sind aktuell theoretisch möglich, für 21 gibt es schon Namen in der Datei *include/uapi/linux/sched.h* des Kernels. Jedoch befinden sich außer den sechs aufgeführten viele noch in einem experimentellen Status (Tabelle 1). Die Einführung der Namespaces erforderte zwangsläufig eine Anpassung vieler interner

Kernelmechanismen, damit dieser alle Schnittstellen vollständig abstrahiert. Gerade bei der Vielzahl an Dateisystemen ist das noch nicht immer der Fall. Damit Entwickler mit Namespaces arbeiten können, nutzen sie drei Funktionen. Der Systemaufruf *clone()* erledigt die Hauptarbeit, wenn ein Programm einen neuen Prozess anlegt. Erhält er eine der genannten Bitmasken als Parameter flag, legt dies für den Nachwuchs eine neue Instanz des Namensraums an. Die Funktion *unshare()* legt unabhängig von neuen Prozessen einen neuen Namensraum an und *setns()* verschiebt einen bestehenden Prozess in einen bestehenden Namensraum.

Namespace	Name im Kernel	Erklärung
Mount	CLONE_NEWNS	Erzeugt ein neues Filesystem-Layout oder setzt bestimmte Mount-Optionen wie etwa ein Read-only Flag
UTS	CLONE_NEWUTS	Erlaubt das Ändern des Hostnamens
IPC	CLONE_NEWIPC	Isoliert die Interprocess Communications (IPC) zwischen Namespaces
PID	CLONE_NEWPID	Isoliert die Prozess-IDs (PIDs), die Liste der Prozesse und ihrer Details; Prozesse im Parent Namespace sehen jedoch immer alle Prozesse im Child Namespace
Network	CLONE_NEWNET	Isoliert die Netzwerkcontroller (phys. oder virtuell), Netfilter-Regeln und Routingtabellen; Network-Namespaces lassen sich durch virtuelle Interfaces mit dem Namen vethX miteinander verbinden
User	CLONE_NEWUSER	Isoliert die User-IDs (UIDs) zwischen Namespaces

Tabelle 1: Die sechs am häufigsten eingesetzten Namespaces; der Linux-Kernel kennt noch eine Handvoll weitere, die sich aber noch in verschiedenen Entwicklungsstadien befinden

(https://entwickler.de/wp-content/uploads/2015/01/tabelle12_lg2)

Auf der Kommandozeile lassen sich Namespaces mit dem Kommando *unshare* aus dem Paket *util-linux* instanziieren. Im gleichen Schritt startet der Befehl einen neuen Prozess in dieser so erzeugten Instanz. Sein Name rührt daher, dass eine Ressource wie beispielsweise die gemounteten Dateisysteme, die bislang allen Prozessen bekannt war, nun privat für den neuen Prozess zur Verfügung steht. Um alle Namespaces zu nutzen, ist ein Kernel ab Version 3.8 nötig, den die Linux-Entwickler Anfang 2013 veröffentlicht haben. Mit den Optionen *-m, -u, -i, -n, -p* und *-U* zeigt der Administrator an, für welche Namespaces er eine neue Instanz anlegen möchte. Mit *-f* legt er einen Prozess an, andernfalls führt der Kernel den aktuellen Prozess in den neuen Namespace-Instanzen mit dem angegebenen Programm fort. Einige weitere Optionen legen spezifische Details für einzelne Namespaces fest. Alternativ versetzen Anwender einen neuen Prozess mit *nsenter* in eine bereits laufende Namespace-Instanz. Er erinnert damit vage an das Kommando *docker attach*, das allerdings noch eine Reihe von weiteren Aufgaben wie Signalverarbeitung und das Verbinden von Dateideskriptoren erledigt.

Zalando setzen Docker bereits produktiv ein. Das Entwickler Magazin Spezial „Docker“ informiert kompetent über diese revolutionäre Technologie, von der viele meinen, dass sie eine neue Ära in der IT einläuten wird. Wir erklären technische Hintergründe, demonstrieren Best Practices und zeigen, wie Docker effektiv eingesetzt werden kann. Das Sonderheft vereint umfangreiches Wissen über die wichtigsten Aspekte von Docker, spannende Ideen für eigene Docker-Projekte und wertvolle Impulse für ihre strategische Planung.

Anlegen und Abfragen von Instanzen

Die Instanzen selbst haben keinen sprechenden Namen. Stattdessen repräsentiert sie der Linux-Kernel durch Zahlen, genau genommen eine Inode-Nummer des Proc-Dateisystems. Auf diese Weise lässt sich einiges über sie herausfinden (http://unix.stackexchange.com/a/113561). Da viele aktuelle Distributionen heute auf Systemd aufbauen, gibt es dort bereits instanziierte Namespaces, weil das neue Framework davon schon Gebrauch macht. Um die aktuellen Instanzen zu erheben, findet der Systemverwalter sie als Symlinks im Pfad */proc/<PID>/ns/<NAMESPACE>* auf die Inodes, die sie repräsentieren. Die Zuordnungen lassen sich mit *echo* und Shell Globbing, per *ls* oder schöner mit *readlink* auslesen. Ein erster Test ermittelt die IDs der Namespaces, in denen der aktuelle Prozess lebt:

```
# readlink /proc/self/ns/*
ipc:[4026531839]
mnt:[4026531840]
net:[4026532868]
pid:[4026531836]
uts:[4026531838]
```

Wer sich die ID *net:[4026532868]* des Netz-Namespace merkt, kann von diesem mit der Option -n eine neue Instanz mit unshare anlegen und das Ergebnis vergleichen:

```
# unshare -n -f /bin/bash
# readlink /proc/self/ns/net
net:[4026532803]
```

Der abgetrennte Netz-Namespace *net:[4026532803]* ist dabei entstanden. Als sichtbarer Effekt haben sich die Netzwerkinterfaces verändert. Es gibt nun nur noch ein Loopback-Device:

```
# ip link show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAU
LT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Weitere Schnittstellen, die der Systemverwalter jetzt anlegt und konfiguriert, sind aus Sicht des Netzwerks vom Rest des Systems isoliert.

Virtualisierter Hostname

Der Einsatz von Namespaces lässt sich auch kombinieren. Erhält das letzte Beispiel zusätzlich die Option *-u*, instanziiert der Linux-Kernel auch noch den UTS-Namespace, der für den Hostnamen zuständig ist:

```
# hostname
hans
# unshare -nu -f /bin/bash
# readlink /proc/self/ns/{net,uts}
net:[4026532807]
uts:[4026532809]
# hostname haenschen
# hostname
haenschen
# exit
# hostname
hans
```

Erneut sind Instanzen entstanden, in denen sowohl Interfaces als auch der Hostname eigene Werte erhalten können. Änderte der Anwender innerhalb des neuen Namespace den Hostnamen, so beträfe dies – anders als im Beispiel mit *chroot* – nicht den Namespace der Eltern. Namespaces separieren Prozesse vom Rest des Systems und verfügen über eigene abgetrennte Bereiche für beispielsweise PIDs, UIDs oder Netzinterfaces.

Gutes verbinden: Chroot und Namespaces im Zusammenspiel

Die beiden Verfahren Chroot und Namespaces lassen sich gut miteinander kombinieren. Das *unshare*-Kommando ist dafür bereits vorbereitet:

```
# unshare -muinp -f --mount-proc=./wheezy_chroot/proc \
  chroot ./wheezy_chroot /bin/bash
```

Der Aufruf von *unshare -muinp* legt fünf neue Namespaces auf einmal an. Sollte sich das Kommando über eine unbekannte Option beschweren, aktualisiert der Systemverwalter seinen Kernel und die zugehörigen Tools, da hier viel aktuelle Entwicklung stattfindet. Im Jahr 2014 erschienene Distributionen sind in der Regel mit dem Kernel 3.8 ausgestattet. Die Option *-f* forkt das angegebene Kommando *chroot,* anstatt es direkt auszuführen. Weil gerade die Isolation von PIDs knifflig ist, erfordert eine neue Instanz des Namensraums *NEWPID* auch einen neuen Prozess, den der Kernel nach dem Aufruf von *fork()* und damit letztlich via *clone()* erzeugt. Beim Aufruf von *clone()* erhält der Kernel im Parameter *flags* Hinweise darüber, welche Namensräume er instanziieren soll. Nur eine neue Instanz anzulegen, dem bestehenden Prozess per *unshare()* zuzuweisen und dann den aktuellen Prozess mittels *execl()* zu überschreiben, klappt für die PIDs nicht.

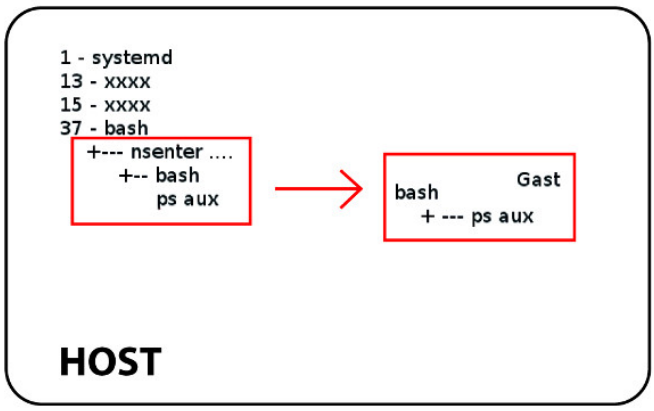
Bevor die Befehlsgruppe tatsächlich den *chroot*-Befehl ausführt, mountet sie durch das Argument *–mount-proc=./wheezy_chroot/proc* noch einmal ein frisches Proc-Dateisystem an die richtige Stelle innerhalb des Debian-Dateibaums. Das ist nötig, weil durch die geklonte *mount*-Instanz nur eine Kopie des elterlichen Prozessbaums entsteht. Geschähe das nicht, könnte der neue Prozess über das elterliche Proc-Filesystem des Hosts auch dessen Prozesse einsehen. Zusammengefasst erzeugt die Befehlszeile den Namespace, mountet das Proc-Dateisystem, führt das *chroot*-Kommando aus und startet damit eine neue Shell. Durch Kombinieren einer Chroot-Umgebung mit Namespaces erhalten wir ein Sandboxing auf Dateisystemebene sowie separierte PIDs, UIDs und Mountpoints. Damit steht ein erster Container, der allerdings noch kein Netzwerk besitzt.

Alternative: Systemd bringt eigene Tools mit

Für viel Diskussion hat in vergangenen Monaten das Systemd-Framework gesorgt. Seine streitbaren Entwickler haben es als Ersatz für den klassischen Startprozess des Linux-Betriebssystems erdacht, der ursprünglich aus einer Reihe von iterativ ablaufenden Shell-Skripten bestand. Heute ist Systemd Bestandteil vieler aktueller Distributionen und verkürzt dabei durch Parallelabarbeitung und eine Reihe weiterer Tricks die Boot-Zeit von Linux. Weil die Entwickler ein Werkzeug benötigten, mit dem sie den Startablauf einfach simulieren konnten, haben sie ihrem Framework das Kommando *systemd-nspawn* beigelegt.

Es lässt sich ebenfalls dazu verwenden, einen schlanken, auf Namespaces aufbauenden Container zu erzeugen. Es virtualisiert ähnlich wie Chroot das Dateisystem, aber zusätzlich noch den Prozessbaum, das IPC-Subsystem sowie Host- und Domainnamen. Die Kernelinterfaces */sys, /proc/sys* oder */sys/fs/selinux* stellt der Befehl nur lesend zur Verfügung. Bestehende Netzwerkinterfaces und die Systemuhr zu verändern, verbietet er ebenso wie das Erzeugen von Device-Nodes. Es verhindert, den Host zu rebooten. Darüber hinaus mountet das Kommando systemd-nspawn Dateisysteme wie */dev, /run* im Container privat – sie sind daher von außerhalb nicht sichtbar.

Analog zum letzten Beispiel errichtet das Werkzeug mit einem einzelnen Aufruf einen Container. Wenn es mit der Option *-D* das Verzeichnis mit dem per debootstrap erzeugten Dateisystem erhält, startet es dort eine Shell: # *systemd-nspawn -M mycontainer –private-network -D ./wheezy_chroot*. Die Option *-M* vergibt einen Hostnamen, *–private-network* deaktiviert effektiv alle Netzinterfaces, solange der Anwender mit weiteren Optionen keine explizit anfordert. Eine Kontrolle der Namespaces innerhalb und außerhalb des Containers mit *ls /proc/self/ns/** bestätigt, dass der Kernel für den Container neue Namespaces erzeugt hat.



(https://entwickler.de/wp-content/uploads/2015/01/magnus_maschinenraum_3.jpg)

Abb. 3: Wenn „systemd-nspawn" mit der Option „–macvlan" Interfaces in einem Gast anlegt, erlaubt das Host dem Gast einen direkten Zugriff auf das externe Netz, das ist zwar einfach eingerichtet, erlaubt aber nur wenig Kontrolle

Um den Container ans Netz anzubinden, erzeugt der Anwender mit der Option *–network-macvlan* ein gleichnamiges Interface für den Gast. Dabei fügt der Linux-Kernel einer Schnittstelle des Hosts eine weitere MAC-Adresse hinzu. Im Container ist die physikalische NIC daraufhin unter gleichem Namen, aber mit dem Präfix *mv-* vorhanden (Abb. 3). Das Beispiel in Listing 1 verwendet das Interface *eth0* des Hosts und lässt diesem im Container per DHCP eine Adresse zuweisen.

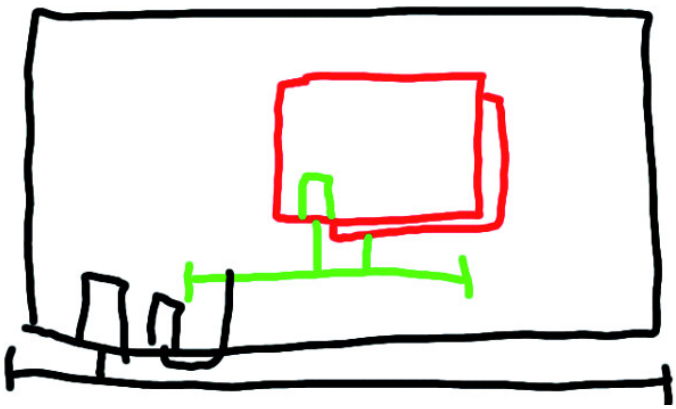
Listing 1

```
# systemd-nspawn -M mycontainer -D ./wheezy_chroot --network
-macvlan eth0
root@mycontainer:~# dhclient mv-eth0
root@mycontainer:~# ifconfig mv-eth0 | grep -A2 mv-eth0
mv-eth0   Link encap:Ethernet  HWaddr 0e:17:83:02:1f:01
          inet addr:10.3.3.77  Bcast:10.3.3.255  Mask:255.25
5.255.0
          inet6 addr: fe80::c17:83ff:fe02:1f01/64 Scope:Link
root@mycontainer:~# ping -nc1 www.google.de
PING www.google.de (74.125.232.24) 56(84) bytes of data.
64 bytes from 74.125.232.24: icmp_req=1 ttl=54 time=8.12 ms
```

Virtuelles Subnetz trennt Host und Gast

Das Framework bietet noch einen alternativen Weg an, um dem Gast einen Netzanschluss zu verschaffen. Die so genannten Veth-Devices erzeugen ein virtuelles Ethernet, an die der Admin sowohl Host als auch die Container anschließt. Dazu erzeugt er im Host eine virtuelle Ethernet-Bridge *mybr* aus dem neuen Subnetz 192.168.200.0/24, um dort seine Container zu bündeln. Auf Hostseite hat *systemd-nspawn* das Interface in dieses Netz bereits nach dem Systemd-Container benannt und ihm das Präfix ve- vorangestellt. Im Beispiel heißt es also *ve-mycontainer*. Innerhalb des Containers erhält das Interface den Namen *host0*. Schließlich legt er eine Masquerading-Regel für den Netztraffic des Containers an (Abb. 4):

```
# brctl create mybr
# ip link set mybr up
# ip addr add 192.168.200.1/24 dev mybr
# iptables -t nat -A POSTROUTING -s 192.168.200.0/24 -j MASQ
UERADE
```



(https://entwickler.de/wp-content/uploads/2015/01/magnus_maschinenraum_4.jpg)

Abb. 4: Einen besseren Schutz erhält „systemd-nspawn" durch ein virtuelles Subnetz, an dem alle Container hängen und in dem der Host als Gateway agiert. Systemverwalter richten dort dann per SNAT ein Masquerading ein und können dann noch zusätzliche Firewallregeln implementieren

Nach dem Start des Containers aktiviert der Systemverwalter auf dem Host das dadurch erzeugte Device *ve-mycontainer* mit *ip link ... up* und verbindet es mittels *brctl* mit der Host-Bridge:

```
# systemd-nspawn -M mycontainer -D ./wheezy_chroot/ --networ
k-veth
# ip link set ve-mycontainer up
# brctl addif mybr ve-mycontainer
```

Im Container konfiguriert der Admin anschließend sein Interface *host0* auf die IP-Adresse 192.168.200.2 aus dem Range der *mybr*-Bridge, sowie die IP des Hosts in diesem Subnetz als Default-Gateway. Um die Adressen der Container zu verbergen, richtet er noch per SNAT ein Masquerading auf dem Host ein, sodass die Container über das Hostsystem auf das Internet zugreifen können, aber von dort aus nicht automatisch exponiert sind:

```
root@mycontainer:~# ifconfig host0 192.168.200.2/24 up
root@mycontainer:~# route add default gw 192.168.200.1
root@mycontainer:~# ping -nc1 www.google.de
PING www.google.de (74.125.232.87) 56(84) bytes of data.
64 bytes from 74.125.232.87: icmp_req=1 ttl=53 time=8.20 ms
```

Beim Stoppen des Containers löscht der Kernel übrigens das Device *veth* und entfernt es aus der Bridge des Hosts. Systemverwalter sollten also daran denken, die entsprechenden Bridge-Einträge dort beim Neustart des Gasts wieder zu setzen. Die beiden Verfahren unterscheiden sich darin, dass Macvlan zwar einfach eingerichtet ist, aber dafür nur wenig Steuerungsmöglichkeiten bietet. Der mit Veth-Interfaces beschriebene Weg ist im Grundsatz mit dem vergleichbar, den Docker auch normalerweise geht: Alle Container sind in einem virtuellen Subnetz zusammengefasst und nutzen den Host als Übergabepunkt zum Internet.

Auf praktische Weise isoliert *systemd-nspawn* Ressourcen mittels Chroot sowie Namespaces in einem Werkzeug und konfiguriert gleich noch Netzinterfaces und viele weitere nützliche Einstellungen. Damit erhalten Anwender eine Sandbox auf der Ebene des Dateisystems und einfach konfigurierbare Netzanbindung. Doch trotz des Komforts, den das Tool bietet, warnen seine Entwickler davor, in den Containern unbeaufsichtigt privilegierten Code ablaufen zu lassen. Sie möchten nicht garantieren, dass es für diesen nicht doch Schlupflöcher in das Hostsystem hinein gibt. Sie sehen *systemd-nspawn* daher als Test- und Debugging-Umgebung, die dabei unterstützt, Anwendungen in Container zu verpacken.

Gerechte Aufteilung von CPU und Speicher mittels Control Groups

Auch wenn ein Gast von den Daten und Subsystemen des Hosts und seinen Geschwistern isoliert ist, so kann er diese dennoch ärgern, indem er selbst ungezügelt Ressourcen konsumiert. Ein amoklaufender Prozess könnte beispielsweise einen Großteil der CPU beanspruchen oder sehr viel Hauptspeicher belegen, der dann anderen Containern nicht mehr zur Verfügung steht. Ein sehr generischer Ansatz, um den Umgang mit solchen Ressourcen zu koordinieren sind die Control Groups, deren Hintergründe der Kasten „Control Groups“ erläutert. Um auf eigenen Systemen Cgroups einzusetzen, interessiert Administratoren zunächst, welche ihr Kernel überhaupt anbietet und welche Parameter sich dort einstellen lassen. Mit diesem Wissen richten sie dann individuelle Gruppen ein und konfigurieren diese. Erst dann lassen sich neue Prozesse mit den gewünschten Restriktionen anlegen.

Praktisch lassen sich Control Groups auf mehrere Weise erzeugen und verwalten: Auf der Kommandozeile erledigen das *cgcreate* und *cgexec.* Das erste Kommando legt eine neue Control Group an, das zweite erzeugt einen neuen Prozess, für den eine bestehende Control Group gelten soll. Beide Tools sind Teil des Debian-Pakets *libcgroup*, das auch den Rules Engine Daemon mitbringt. Der sorgt dafür, die Einstellungen als Dateien im Verzeichnis */etc/cgrules.conf* auch über einen Reboot hinweg zu persistieren.

Manche Software wie Docker, LXC oder Systemd greift direkt auf das Subsystem zu. Dazu nutzt sie das virtuelle /sys–Dateisystem, das sich auch manuell abfragen lässt. Mit *ls /sys/fs/cgroup/* findet der Administrator heraus, welche Cgroup-Typen sein Kernel aktuell unterstützt. Eine gute Beschreibung der prinzipiellen Möglichkeiten und Konfigurationen von Cgroups finden Sie hier (http://www.admin-magazin.de/Das-Heft/2011/06/Cgroups-zur-Ressourcenkontrolle-in-Linux).

Dynamische Control Groups legen individuelle Grenzen fest

Seine Container vermag Docker mit unterschiedlichen Execution-Backends zu realisieren. Baute es ursprünglich auf LXC auf, so nutzt es in der aktuellen Fassung per Default die vom Projekt selbst entwickelten Libcontainer. Weitere Backends sind ebenfalls möglich, darunter sind libvirt und systemd-nspawn. Die per Execution Driver angesteuerten Subsysteme sind für den Grad der Isolierung und den Grad der möglichen Begrenzungen verantwortlich. Der Default, die Libcontainer, implementiert zurzeit nur das Einschränken von CPU- und Speicherressourcen. Um dieses Verhalten nachzustellen, erzeugt der Anwender eine neue Ressourcengruppe *mycontainer.* Mit ein paar Shell-Tricks (Listing 2) prüft er für die CPU, welche Einstellmöglichkeiten sie bietet und wie die Gruppe vorbelegt ist.

Listing 2

```
# cgcreate -g cpu,memory:mycontainer
# cd /sys/fs/cgroup/cpu/mycontainer; grep . *
cgroup.clone_children:0
cpuacct.stat:user 0
cpuacct.stat:system 0
cpuacct.usage:0
cpuacct.usage_percpu:0 0 0 0 0 0 0 0
cpu.cfs_period_us:100000
cpu.cfs_quota_us:-1
cpu.rt_period_us:1000000
cpu.rt_runtime_us:0
cpu.shares:1024
cpu.stat:nr_periods 0
cpu.stat:nr_throttled 0
cpu.stat:throttled_time 0
notify_on_release:0
```

Begrenzungen gelten kumulativ für alle Prozesse, die einer Cgroup zugeordnet sind. Um dem erzeugten Container etwa den Speicherverbrauch auf 128 Megabyte zu begrenzen und rechnerisch nur ein Achtel der CPU zuzuweisen, genügen die Befehle

```
# echo $((128*2**20)) &gt /sys/fs/cgroup/memory/mycontainer/
memory.limit_in_bytes
# echo 128 &gt /sys/fs/cgroup/cpu/mycontainer/cpu.shares
# cgexec -g cpu,memory:mycontainer /bin/bash
```

Das erste Kommando erlaubt maximal 128 x 2^{20} Byte, also 128 Megabyte an Hauptspeicher. Die Cgroups teilen die CPU in jeweils 1 024 so genannte Shares auf, sodass 128 einem Achtel entspricht. Zuletzt startet cgexec unter diesen Bedingungen eine neue Shell. Wer nun beispielsweise mit *a=$(dd*

if=/dev/zero bs=1M count=128 | base64) der Shell-Variable einen deutlich mehr als 128 Megabyte großen String zuweist, erntet eine Out of Memory Exception. Im Systemlog erscheinen daraufhin zwei Kernelmeldungen:

```
kernel: Memory cgroup out of memory: Kill process 31208 (bas
h) score 985 or sacrifice child
kernel: Killed process 31208 (bash) total-vm:146424kB, anon-
rss:129572kB, file-rss:3324kB
```

Alle Zutaten zusammen kombiniert

Um das Beispiel der Namespaces aufzugreifen, aber die Container mittels Cgroups gegen übergroßen Ressourcenhunger eines Gasts abzusichern, kombiniert der Admin die Kommandozeile um den Aufruf von *cgexec*:

```
# cgexec -g cpu,memory:mycontainer unshare -muinp -f \
  --mount-proc=./wheezy_chroot/proc chroot ./wheezy_chroot /
bin/bash
```

Innerhalb des Containers gelten fortan die Begrenzungen der Cgroup mycontainer für CPU und den Speicher:

```
wheezy_chroot# grep mycontainer /proc/self/cgroup
4:memory:/mycontainer
3:cpu,cpuacct:/mycontainer
```

Um eine moderne Sandbox aufzusetzten, die Anwendungen auf leichtgewichtige Weise voneinander isoliert, bringt der Linux-Kernel von Haus aus eine Menge mit: Das schon lange verfügbare Chroot teilt das Dateisystem in Unterbereiche auf. Die zwei relativ jungen Subsysteme der Namespaces und Control Groups haben schon einen wesentlich weiter gefassten Blick auf die Isolation von Ressourcen und ihrer Rationierung. Sie sind beide in der Lage, sehr feine Zuordnung zu treffen, die sich oft nur in Nuancen unterscheiden.

Linux hat zur Container-Verwaltung einiges an Bord

Von Hand ausgeführt sind eine Menge Details zu beachten, um den eingesperrten Prozessen nicht doch einen Weg ins Hostsystem zu ermöglichen. Die Docker-Entwickler nehmen Anwendern hier eine Menge Arbeit ab, aber auch sie sind nicht davor gefeit, einmal etwas zu übersehen (Kasten: „Sicherheitsfunktionen in Docker“). Da das Linux-API sehr umfassend ist und seine Maintainer an den genannten Subsystemen teilweise noch aktiv entwickeln, erfordert der Einsatz trotz Container weiterhin ein wachsames Auge. Auch Docker ist kein automatischer Schutz vor böswilligem Code. Das gilt auch dann, wenn er zwar im Container, aber mit privilegierten Root-Rechten läuft. Darum sollten Anwendungen auch in Docker-Umgebungen möglichst frühzeitig ihre privilegierten Startrechte aufgeben.

Control Groups
Ein Kernstück von Docker sind Cgroups, eine Kurzform von „Control Groups“. Sie verantworten die Ressourcenvergabe im Linux-Kernel und sind dort seit Version 2.6.24 fester Bestandteil. Die ursprüngliche Entwicklung begann 2006 durch Google. Mit Cgroups lässt sich der Speicher von Prozessen oder Prozessgruppen limitieren sowie die Zuteilung von CPU-Rechenzeit und Disk-I/O priorisieren. Bei einer

Container-Virtualisierung laufen alle Prozesse auf dem gleichen Kernel ab. Das schont viele Ressourcen wie etwa den Hauptspeicher. Trotzdem möchten Anwender ihre Prozesse gerne untereinander isolieren, damit ein einzelner Amok laufender Prozess nicht alle anderen Container beeinträchtigt.

Dazu sind im Wesentlichen zwei Maßnahmen notwendig: Die Prozesse eines Containers sollen zum einen für die Prozesse der anderen Container verborgen bleiben. Dafür sorgen die Namespaces mit ihren Prozessgruppen (Kasten: „Namespaces“). Zum anderen gilt es, die zur Verfügung stehenden Ressourcen auf die Container so aufzuteilen, dass ein besonders gieriger Prozess nicht eine Ressource komplett für sich beanspruchen kann. Dies ist die Aufgabe von Cgroups. Die Entwickler haben vier Methoden identifiziert, mit denen das geschehen kann: So lassen sich Grenzen definieren, die Ressourcen nicht überschreiten dürfen (Limiting), beispielsweise eine Menge von Speicher. Anwender können manchen Ressourcen Vorrang gegenüber anderen einräumen (Priorisierung), etwa häufigere Bearbeitung durch die CPU. Cgroups können auch messen, wie viele Betriebsmittel konsumiert wurden (Accounting). Die letzte Methode steuert ganze Gruppen von Prozessen, hält sie an oder lässt sie weiterlaufen (Control).

Ressourcen unter Beobachtung

Je nach Linux-Distribution kann die Anzahl der vorhandenen Cgroups variieren. Typisch in aktuellen Distributionen sind die in Tabelle 2 verzeichneten Gruppen. Weil die Linux-Entwickler dieses Subsystem zurzeit sehr aktiv erweitern, gibt es beispielsweise auch noch die Cgroup *net_prio*, um Netzwerk-I/O zu priorisieren, die Gruppe *systemd* oder auch noch weitere.

Cgroup	Beschreibung der kontrollierten Ressource
blkio	Limitiert I/O von Block Devices oder physikalischen Laufwerken
cpu	Verwendet den CPU-Scheduler, um CPU-Zeit zu limitieren
cpuacct	Erzeugt Reports über verwendete CPU-Ressourcen
cpuset	Bindet Prozesse an bestimmte CPUs oder CPU-Cores
devices	Kontrolliert den Zugriff auf Gerätoeinträge, beispielsweise in /dev
freezer	Kann Prozesse einfrieren und wieder aufwecken
memory	Begrenzt den Speicher von Prozessen und erzeugt Reports über verwendete Speicherressourcen
net_cls	Versieht Netzwerkpakete mit einem so genannten Class Identifier, die der Linux Traffic Controller tc verarbeitet

(https://entwickler.de/wp-content/uploads/2015/01/tabelle2.jpg)

Tabelle 2: Über Cgroups lassen sich verschiedene Ressourcen des Linux-Kernels kontrollieren, begrenzen und ihre Nutzung auslesen

Historischer Vorfahr

Die Idee, Schranken für Prozesse einzuführen, ist fast so alt wie das Unix-Betriebssystem selbst. Einfache Beispiele sind der Nice-Level für Prozesse oder die Ulimits: Jeder Prozess besitzt einen Nice-Level, der dem Scheduler einen Hinweis für die Auswahl des nächsten aktiven Threads gibt. Per Default erben alle den Wert 0 von ihren Eltern, aber ein höherer Wert bedeutet eine höhere „Zuvorkommenheit“ für andere Prozesse – und damit eine niedrigere eigene Priorität. Der Aufruf *nice -n 10 /tmp/batchjob.sh* startet das Skript mit einer um zehn Punkte niedrigeren Priorität als die aufrufende Shell. Der Wert 20 ist meist das Maximum an möglicher Zurückhaltung, Prozesse mit einer Niceness von -19 haben die höchste Priorität.

Andere Ressourcen schränkt das Kommando *ulimit* ein. Mit *ulimit -t 100* begrenzen Anwender etwa die erlaubte Rechenzeit auf 100 Sekunden. Die Option -s begrenzt die Größe des Stack und *-c* den maximalen Umfang von Coredumps. Die Cgroups erlauben jedoch erheblich feinere Einstellmöglichkeiten, nicht nur für einzelne Prozesse, sondern frei definierbare Gruppen.

Sicherheitsfunktionen in Docker

Die wichtigsten Sicherheitsfunktionen von Docker bestehen darin, zu gewährleisten, dass ein Prozess nicht aus seinem Container ausbricht oder unbotmäßig viele Ressourcen nutzt. Ist diese Anforderung nicht erfüllt, könnten Prozesse auf Daten des Hosts und damit auch auf Geschwister-Container zugreifen. In einer Hostingumgebung, die explizit dazu geschaffen wurde, Dienste zu separieren, wäre das die Höchststrafe für Betreiber und Anwender. Der Ansatz der Container-Virtualisierung hat den Vorteil, dass ein einziger laufender Kernel pro Host ausreicht. Das spart eine Menge Ressourcen, gerade wenn sehr viele Container im Einsatz sind. Gleichzeitig ist dieses Paradigma auch die größte Herausforderung, weil der Kernel eine Vielzahl von Schnittstellen besitzt, die der Betreiber alle einzeln vor einem womöglich böswilligen Gast schützen muss. Die aufgeführten Subsysteme wie Chroot, Namespaces und Cgroups sind wichtige Antworten der Kernelentwickler auf diese Fragen, kontrollieren sie doch den Zugang zu bestimmten Ressourcen des Hosts, seien es Dateisysteme, Prozesse oder die Nutzung von Ressourcen.

Docker setzt diese Mechanismen alle automatisch ein, sodass sich Anwender um die in diesem Artikel beschriebenen, händischen Maßnahmen nicht selbst kümmern müssen. Dazu kommt noch eine Reihe weiterer Vorkehrungen: Mittels Pivot-Mount, ein weiterer Systemaufruf, lässt sich das Root-Dateisystem innerhalb eines Containers durch einen komplett neuen Mount ersetzten. Das verhindert die im Kasten „Chroot-Umgebungen“ geschilderte Flucht aus der Chroot-Sandbox.

Namespaces setzen zwar neue Umgebungen auf, aber sie verhindern per se nicht, dass ein privilegierter Gastprozess einem seiner Prozesse wieder den ursprünglichen Namespace zuweist. Um das zu verhindern, entzieht Docker dem Gast per Capability das Recht, das zu tun. Dieses weitere Subsystem des Kernels erlaubt eine feingranularere Vergabe von Rechten, die sonst dem Superuser zustehen. Mittels Cgroups sorgt es ferner dafür, dass sich innerhalb des Containers bestimmte Device-Dateien nicht mehr anlegen und zugreifen lassen. Denn die sind der direkte Weg für den eingesperrten Prozess zu den Festplattenpartitionen des Hosts und sämtlicher angeschlossener Peripherie. Das gilt insbesondere auch für Netzinterfaces.

Wie schwierig das zu bewerkstelligen ist, zeigte ein erster Exploit, der im Sommer 2014 die Runde machte: Trotz eigenem Namespace war eine Kernelstruktur des Gastes noch zugreifbar. Die nutzte der Exploit als Eintrittsticket und zeigte sie dem nur wenig bekannten Systemaufruf *open_by_handle_at()* vor. Auf diese Weise konnte der Gast Dateien des Hosts auslesen (http://stealth.openwall.net/xSports/shocker.c). Das gelang nur unter einigen Voraussetzungen und ist in der aktuellen Version 1.2 auch längst behoben, weil Docker zwischenzeitlich alle Capabilities aufgibt und nur noch explizit die beibehält, die es wirklich benötigt. Trotzdem dokumentiert der Vorfall gut, dass Docker an vielen Stellen gleichzeitig für Sicherheit sorgen muss und nicht den Überblick verlieren darf, ob seine Entwickler nicht doch noch einen Zugangsvektor übersehen haben (https://opensource.com/business/14/9/security-for-docker). Bislang haben die Entwickler das glücklicherweise recht zeitnah erledigt und schnell Patches bereitgestellt.

Aufmacherbild: A modern ships telegraph isolated on white background – all settings from full astern to full speed ahead (http://www.shutterstock.com/de/pic-137175620/stock-photo-a-modern-ships-

telegraph-isolated-on-white-background-all-settings-from-full-astern-to-full-speed.html?src=Bbb_lMd9pimaLuvTQO3ZNw-1-73&ws=1) von Shutterstock / Urheberrecht: donvictorio

DOCKER (HTTPS://ENTWICKLER.DE/TAG/DOCKER)

LINUX (HTTPS://ENTWICKLER.DE/TAG/LINUX)

RELEVANTE BEITRÄGE

Web Server Survey im Dezember 2016 (https://entwickler.de/online/web-server-survey-dezember-2016-579759115.html)

Von **Patrica Stübig (https://entwickler.de/author/2q51r)** / 1 Monat online

XAML-Tipp: „XamlParseException“ debuggen (https://entwickler.de/online/windowsdeveloper/xaml-tipp-xamlparseexception-debuggen-579749135.html)

Von **Gregor Biswanger (https://entwickler.de/author/gregorbiswanger)** / 1 Monat online

XAML-Tipp: XAML-Designer bei Fehlern debuggen (https://entwickler.de/online/windowsdeveloper/xaml-tipp-xaml-designer-debuggen-579749129.html)

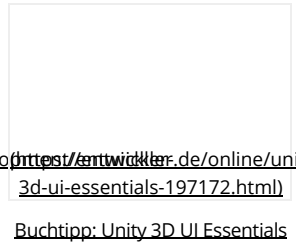
Von **Gregor Biswanger (https://entwickler.de/author/gregorbiswanger)** / 2 Monaten online

Neues vom RAD Studio 10.1 Berlin (https://entwickler.de/online/rad-studio-10-1-berlin-273629.html)

Von **Tom Wiesseckel (https://entwickler.de/author/divb)** / 5 Monaten online

entwickler.kiosk Contest– ihr bestimmt kostenlose Inhalte! (https://entwickler.de/online/development/entwickler-kiosk-contest-246012.html)

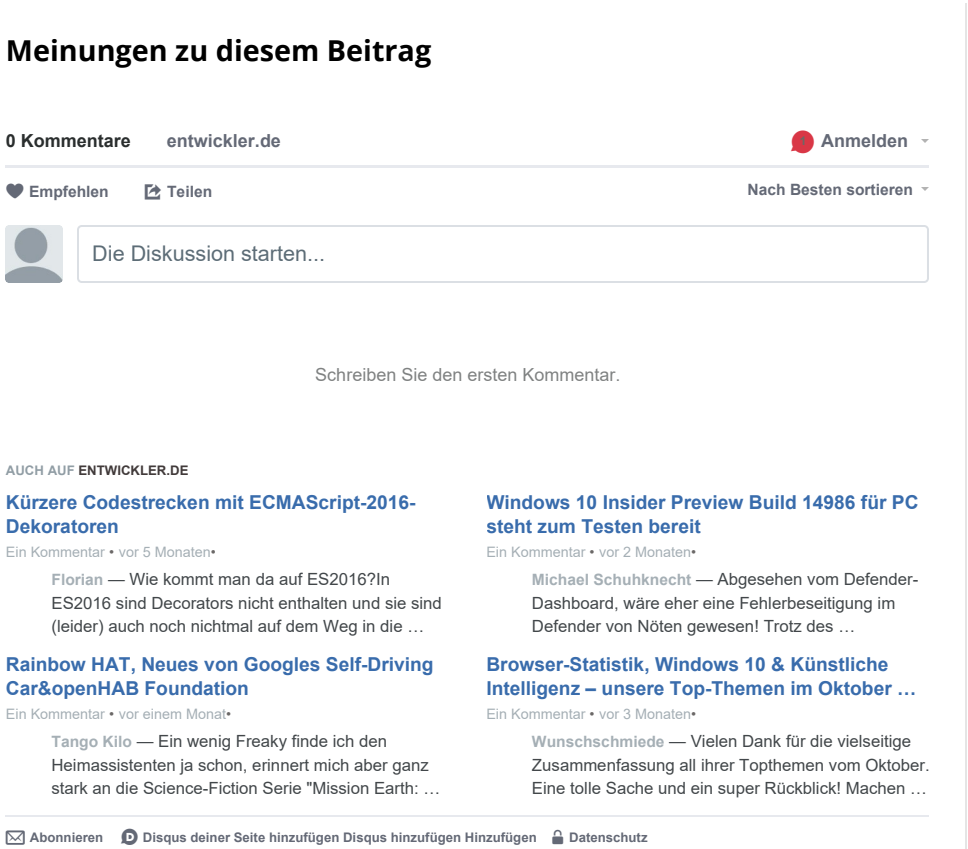
Von **Tom Wiesseckel (https://entwickler.de/author/divb)** / 9 Monaten online



Buchtipp: Unity 3D UI Essentials (https://entwickler.de/online/unity-3d-ui-essentials-197172.html)

Von **Tam Hanna (https://entwickler.de/author/tamhanna)** / 10 Monaten online

Meinungen zu diesem Beitrag



AUCH AUF **ENTWICKLER.DE**

Kürzere Codestrecken mit ECMAScript-2016-Dekoratoren
Ein Kommentar • vor 5 Monaten•
Florian — Wie kommt man da auf ES2016?In ES2016 sind Decorators nicht enthalten und sie sind (leider) auch noch nichtmal auf dem Weg in die …

Windows 10 Insider Preview Build 14986 für PC steht zum Testen bereit
Ein Kommentar • vor 2 Monaten•
Michael Schuhknecht — Abgesehen vom Defender-Dashboard, wäre eher eine Fehlerbeseitigung im Defender von Nöten gewesen! Trotz des …

Rainbow HAT, Neues von Googles Self-Driving Car&openHAB Foundation
Ein Kommentar • vor einem Monat•
Tango Kilo — Ein wenig Freaky finde ich den Heimassistenten ja schon, erinnert mich aber ganz stark an die Science-Fiction Serie "Mission Earth: …

Browser-Statistik, Windows 10 & Künstliche Intelligenz – unsere Top-Themen im Oktober …
Ein Kommentar • vor 3 Monaten•
Wunschschmiede — Vielen Dank für die vielseitige Zusammenfassung all ihrer Topthemen vom Oktober. Eine tolle Sache und ein super Rückblick! Machen …

Abonnieren Disqus deiner Seite hinzufügen Disqus hinzufügen Hinzufügen Datenschutz

ONLINE

JAXenter.de (http://jaxenter.de)
JAXenter.com (http://jaxenter.com)

MAGAZINE

PHP Magazin (http://entwickler.de/php-magazin)
Windows Developer (http://windowsdeveloper.de/shop)
Entwickler Magazin (http://entwickler.de/entwickler-magazin)
Mobile Technology (http://entwickler.de/mobile-technology)
SharePoint Kompendium (http://windowsdeveloper.de/shop)
Java Magazin (https://jaxenter.de/magazine/java-magazin)
Eclipse Magazin (https://jaxenter.de/magazine/eclipse-magazin)
Business Technology (https://jaxenter.de/magazine/business-technology)

KONFERENZEN

JAX (https://jax.de)
BASTA! (https://basta.net)
International PHP Conference (https://phpconference.com)
WebTech Conference (https://webtechcon.de)
webinale (https://webinale.de)
MobileTech Conference (https://mobiletechcon.de/)
Internet of Things Conference (https://iotcon.de/)
DevOpsCon (http://devopsconference.de/de/)

S&S MEDIA

Über S&S Media (http://sandsmedia.com/de/unternehmen)
Entwickler Akademie (http://entwickler-akademie.de/)

Software & Support Media GmbH (http://sandsmedia.com/) Impressum (/impressum)
Datenschutz (/datenschutz) AGB (/entwicklerde_agb) Lieferkonditionen (/lieferkonditionen)