Bernd Öggl
Michael Kofler

Docker

Das Praxisbuch für Entwickler und DevOps-Teams

2., aktualisierte und erweiterte Auflage

- Schritt für Schritt vom Setup bis zur Orchestrierung
- Continuous Delivery: Grundlagen, Konzepte und Beispiele
- Praxiswissen zu Projekt-Migration, Sicherheit, Kubernetes u. v. m.

Alle Beispielprojekte zum Download

Rheinwerk Computing

Vorwort

Zu Beginn der 2000er-Jahre stellte Virtualisierungssoftware den Alltag vieler Entwickler auf den Kopf: Plötzlich war es möglich, auf einem Rechner Linux *und* Windows auszuführen, Programme unkompliziert in verschiedenen Umgebungen bzw. Web-Apps in alten Versionen von Webbrowsern auszuprobieren, verschiedene Software-Stacks in virtuellen Maschinen parallel zu installieren und zu testen und vieles mehr.

Natürlich spielen virtuelle Maschinen für Entwickler weiter eine große Rolle; außerdem ist die Cloud in ihrer jetzigen Form ohne Virtualisierung gar nicht denkbar. Dennoch hat vor einigen Jahren ein Umbruch weg von virtuellen Maschinen hin zu Containern begonnen – und dieser Umbruch scheint sich mehr und mehr zu beschleunigen.

Container ermöglichen es, bestimmte Softwarekomponenten (Webserver, Programmiersprachen, Datenbanken) ohne den Overhead einer virtuellen Maschine auszuführen. Warum ein ganzes Betriebssystem (meist Linux) in eine virtuelle Maschine installieren, wenn es doch nur um *eine* ganz spezifische Funktion geht?

Selten trifft das Paradigma »Weniger ist mehr« so gut zu wie auf die Container-Technologie. Das *Weniger* drückt sich in unzähligen Vorteilen aus: Container sind viel schneller aufgesetzt als virtuelle Maschinen, lassen sich leichter auf verschiedenen Entwicklungssystemen replizieren, beanspruchen weniger Ressourcen und bieten wesentlich bessere Möglichkeiten zur Skalierung und Lastverteilung. Container sind insofern nicht nur ein Segen für Entwicklerteams, sondern bieten auch vollkommen neue Möglichkeiten im Deployment, also im produktiven Betrieb der entwickelten Lösung.

Docker

Docker ist *die* Container-Software schlechthin. Zwar gibt es mittlerweile durchaus interessante Alternativen, aber Docker hat den Container-Markt als solchen geschaffen. In einer Umfrage der Website Stack Overflow unter Tausenden von Entwicklern war Docker die Nummer 1 in der Kategorie *Most Wanted Platform*. Docker ist also das Werkzeug, das die meisten Entwickler neu erlernen wollen. Und bei den Entwicklern, die Docker bereits anwenden, erreichte Docker die Nummer 2 bei der Fragestellung *Most Loved Platform*. (Ein paar Prozent mehr Zustimmung hatte in diesem Fall Linux.)

https://insights.stackoverflow.com/survey/2019

Allerdings geriet die Firma *Docker Inc.* gerade zur Fertigstellung der zweiten Auflage dieses Buchs in finanzielle Schwierigkeiten. Im November 2019 kaufte der Cloud-Anbieter Mirantis die Enterprise-Sparte von Docker. Die so erzielten Einnahmen sichern die Weiterführung der verbleibenden Firma, die sich stärker als bisher um die Bedürfnisse von Entwicklern kümmern will. *Docker Community* und die Konfigurationshilfe *Docker Desktop* für macOS und Windows werden also weiterentwickelt und bleiben kostenlos verfügbar.

Müssen Sie sich Sorgen machen, dass Docker aufgrund wirtschaftlicher Probleme einfach verschwinden könnte? Glücklicherweise nicht! Nahezu alle Komponenten von Docker sind als Open-Source-Software verfügbar. Einige Schnittstellen von Docker wurden zudem durch die *Open Container Initiative* (OCI) standardisiert und erleichtern anderen Anbietern, mit Docker kompatible Produkte anzubieten. Das für Linux-Anwender interessanteste Konkurrenzprodukt *Podman* stellen wir im Anhang dieses Buch vor.

Docker hat also einen Standard gesetzt, der aller Voraussicht nach unabhängig vom wirtschaftlichen Erfolg der Ursprungsfirma Bestand haben wird.

Wozu dieses Buch?

In diesem Buch geben wir eine Einführung in den Umgang mit Docker und einen Überblick über die wichtigsten Bausteine (Images), aus denen Sie eigene Container-Welten zusammensetzen können. Wir zeigen anhand mehrerer großer Beispiele, wie Sie Docker in der Praxis einsetzen, und gehen ausführlich auf das Deployment in der Cloud ein.

Wir haben das Buch in drei Teile gegliedert:

- **Teil I** stellt Docker vor. Sie lernen anhand vieler Beispiele, wie Sie die Kommandos `docker` und `docker-compose` sinnvoll einsetzen und wie die Syntax der Dateien `Dockerfile` und `docker-compose.yml` aussieht.
- **Teil II** präsentiert wichtige Images, die als Basis für eigene Projekte dienen können. Dazu zählen unter anderem:
  - Alpine Linux
  - die Webserver Apache und Nginx (inklusive Proxy-Setup und Let's-Encrypt-Konfiguration)
  - die Datenbankserver MySQL/MariaDB, MongoDB, PostgreSQL und Redis
  - die Programmiersprachen JavaScript (Node.js), Java, PHP, Ruby, Python und Swift
  - die Webapplikationen WordPress, Joomla und Nextcloud

- **Teil III** zeigt den Einsatz von Docker in der Praxis. Wir zeigen Ihnen sowohl, wie Sie moderne Webapplikationen mit Docker besonders effizient entwickeln, als auch, wie Sie vorhandene Projekte mit all ihren Altlasten in besser wartbare Docker-Projekte umwandeln.

  Zwei Kapitel zur Nutzung von GitLab mit Docker und zu *Continuous Integration* (CI) und *Continuous Delivery* (CD) demonstrieren Ihnen neue Paradigmen und Hilfsmittel für das Entwickeln von Software im Team.

  Auch das Deployment kommt nicht zu kurz: Mit Docker Swarm und Kubernetes bringen Sie Ihre Docker-Projekte in die Cloud und profitieren von den dort gegebenen Möglichkeiten zur Skalierung. Eine Sammlung von Tipps stellt sicher, dass dabei die Sicherheit nicht zu kurz kommt.

  Schließlich stellen wir Ihnen im Anhang das von Red Hat entwickelte Programm *Podman* kurz vor. Es übernimmt viele Ideen und die Syntax von Docker, beschreitet aber bei der Implementierung neue Wege. Podman ist aktuell kein vollwertiger Ersatz für Docker (insbesondere, weil macOS und Windows nicht unterstützt werden), zeigt aber, dass Docker-Know-how über die Grenzen des eigentlichen Produkts Gültigkeit hat.

Wer Docker einmal ausprobiert und kennengelernt hat, will nie wieder auf seine Funktionen verzichten. Lassen Sie sich von uns in eine neue Welt führen!

Bernd Öggl (*https://komplett.cc*)
Michael Kofler (*https://kofler.info*)

Kapitel 1

Hello World

Die drei in diesem Kapitel präsentierten Hello-World-Beispiele für Docker sollen etwas mehr leisten, als nur die viel zitierte Zeichenkette am Bildschirm auszugeben: Wir wollen jeweils einen Webserver starten, der eine Webseite ausliefert, auf der die aktuelle Uhrzeit und ein Wert für die Auslastung des Servers angezeigt wird. Dabei kommen drei unterschiedliche Programmiersprachen zum Einsatz.

Um den von Docker unabhängigen und für dieses Beispiel unwichtigen Programmcode möglichst kurz zu halten, verzichten wir auf die Trennung von Frontend- und Backend-Code, wie es eine moderne Webapplikation machen würde. Verstehen Sie dieses Kapitel mehr als *proof of concept*. Beispiele für moderne Webapplikationen finden Sie in Teil III dieses Buchs.

1.1 Docker-Schnellinstallation

Als ersten Schritt müssen Sie die Docker-Software auf Ihrem Computer installieren. Eine ausführliche Installationsanleitung finden Sie in Kapitel 2. Hier wollen wir einen Schnelleinstieg für all jene bieten, denen es bereits in den Fingern kribbelt.

Windows

Aktuell (im Dezember 2019) benötigen Sie zur Docker-Installation Windows 10 Pro oder Enterprise. Voraussichtlich ab Frühjahr 2020 wird auch Windows Home ausreichen. Docker verwendet dann WSL2 anstelle von Hyper-V als Fundament zur Ausführung der Container.

Zur Installation von Docker laden Sie einfach das Setup-Programm von folgender Adresse herunter und führen es aus:

https://hub.docker.com/editions/community/docker-ce-desktop-windows

Wenn Sie sich nicht im Docker Hub registrieren wollen, können Sie auch den direkten Link zum Download verwenden:

https://download.docker.com/win/stable/
Docker%20for%20Windows%20Installer.exe

macOS

Die Installationsdatei (DMG) für macOS finden Sie unter folgender Adresse:

https://hub.docker.com/editions/community/docker-ce-desktop-mac

Auch für macOS gibt es einen direkten Download-Link, ohne Registrierung im Docker Hub:

https://download.docker.com/mac/stable/Docker.dmg

Linux

Für eine gängige Linux-Distribution auf Debian-Basis (Debian, Ubuntu …) und einen Computer mit 64-Bit-Prozessor reicht es in der Regel aus, die folgenden vier Kommandos auf der Kommandozeile einzugeben. (Für die mit dem #-Prompt gekennzeichneten Kommandos benötigen Sie `root`-Rechte.)

```
sudo apt -y install apt-transport-https ca-certificates curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo apt-key add -
sudo add-apt-repository \
 "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable"
sudo apt update
sudo apt -y install docker-ce docker-ce-cli containerd.io
```

Für andere Linux-Distributionen (Fedora, CentOS …) werfen Sie bitte einen Blick in Kapitel 2.

Damit Sie die folgenden Abschnitte ausprobieren können, benötigen Sie außer der Installation auch noch die für Docker notwendigen Berechtigungen. Sollten Sie unter Linux Schwierigkeiten haben, so führen Sie die Docker-Kommandos einfach als `root` auf Ihrem Computer aus. Windows und macOS kümmern sich bei der Installation um die nötigen Rechte. Außerdem ist eine schnelle Internetverbindung hilfreich: Die Basis-Images, die Sie herunterladen müssen, sind mehrere Hundert Megabyte groß.

1.2 Apache mit PHP 7

Wir werden drei verschiedene Varianten beschreiben, um die Vorgaben für dieses Beispiel (eine HTML-Seite, die von einem Webserver ausgeliefert wird) mit Docker umzusetzen. Die erste Hello-World-Variante wird mit der bekannten Kombination aus Apache-Webserver und PHP als Programmiersprache programmiert.

Erstellen Sie am besten ein eigenes Verzeichnis für jeden Docker-Versuch. Für dieses Beispiel bietet sich als Projektverzeichnis `hello-world-php` an. Der PHP/HTML-Code

ist sehr übersichtlich. Im Folgenden sehen Sie die Datei `index.php`, die im Projektverzeichnis abgelegt wird:

```
<!DOCTYPE html>
<!-- Datei index.php -->
<html>
<head>
  <title>Hello world</title>
  <meta charset="utf-8" />
</head>
<body>
<h1>Hello world: apache/php</h1>
<?php
  $load = sys_getloadavg();
?>
  Serverzeit: <?php echo date("c"); ?><br />
  Serverauslastung (load): <?php echo $load[0]; ?>
</body>
</html>
```

Der PHP-Aufruf `sys_getloadavg` liefert ein Maß für die aktuelle Auslastung des Systems. Unter Unix-verwandten Betriebssystemen ist dies eine gängige Größe, die meist durch drei Durchschnittswerte dargestellt wird. Sie beschreiben die CPU-Auslastung als jeweils ein Mittel der letzten Minute, der letzten fünf Minuten beziehungsweise der letzten 15 Minuten. In dem Hello-World-Beispiel verwenden wir nur den ersten dieser drei Werte: die durchschnittliche Auslastung der letzten Minute (der Array-Index ist 0, daher `$load[0]`).

Damit diese Datei von einem Docker-Container mit Apache und PHP 7 ausgeliefert werden kann, müssen Sie zuerst ein Docker-Image mit der genannten Software erstellen. Von diesem Image wird dann ein Docker-Container abgeleitet, der die eigentliche Arbeit übernimmt.

> **Nomenklatur**
>
> Lassen Sie sich jetzt nicht durch vielleicht ungewohnte Begriffe wie *Docker-Image* und *Docker-Container* abschrecken. In Kapitel 2, »Installation und Grundlagen«, werden diese Begriffe ausführlich erklärt.

Um ein Docker-Image zu erstellen, legen Sie die folgende Datei `Dockerfile` an, die Sie ebenfalls im Projektverzeichnis speichern. Diese Datei enthält quasi die Anleitung zum Erstellen des Images.

```
# Datei: hello-world-php/Dockerfile (docbuc/hello-world-php)
FROM php:7-apache
ENV TZ="Europe/Amsterdam"
COPY index.php /var/www/html
```

Drei Zeilen, die nahezu selbsterklärend sind, sollen also genügen, um einen Apache Webserver und PHP in der aktuellen Version zu installieren und die `index.php`-Datei der Welt zur Verfügung zu stellen?

Drei Zeilen und zwei Befehle, um genau zu sein. Diese zwei Befehle müssen Sie jetzt noch auf der Kommandozeile eingeben, um den Zauber zu starten:

```
docker build -t hello-world-php .
```

Dadurch wird aus dem `Dockerfile` das gewünschte Docker-Image erzeugt und mit dem Tag `hello-world-php` versehen. Beim ersten Aufruf dieses Kommandos wird Docker versuchen, das Basis-Image `php:7-apache`, das Sie in der ersten Zeile des Dockerfiles referenziert haben, aus dem Internet zu laden. Genauer gesagt wird es vom *Docker Hub* heruntergeladen, einer Plattform, die von der Firma hinter Docker zur Verfügung gestellt wird und die eine Vielzahl vorbereiteter Images vorhält (siehe Abschnitt 2.2, »Grundlagen und Nomenklatur«). Alle zukünftigen `docker build`-Kommandos, die auf das `php:7-apache`-Image aufbauen, werden Ihre lokale Kopie verwenden. Abschließend starten Sie von dem Docker-Image einen Container, in dem die Applikation läuft:

```
docker run -p 8080:80 hello-world-php
```

Der Parameter `-p` veranlasst Docker, den Port 80 des Containers (das ist die 80 hinter dem Doppelpunkt) mit dem Port 8080 des ausführenden Systems (das ist die 8080 vor dem Doppelpunkt) zu verbinden. Sie können jetzt über *http://localhost:8080* auf die Applikation zugreifen (siehe Abbildung 1.1).

Abbildung 1.1 Der Hello-World-Webserver mit dem PHP-Container im Browser

Dockerfile

Zuletzt noch eine kurze Erklärung zu den drei Zeilen des auf der Vorseite abgedruckten `Dockerfiles` (siehe auch Kapitel 3, »Eigene Docker-Images (Dockerfiles)«):

- `FROM php:7-apache` – Sie verwenden das von den PHP-Entwicklern vorgelegte Docker-Image für PHP 7 als Basis.

- `ENV TZ="Europe/Amsterdam"` – Damit die Uhrzeit in der korrekten Zeitzone ausgegeben wird, setzen Sie die Umgebungsvariable (*ENVironment*) `TZ` (für *TimeZone*) auf den entsprechenden Wert.
- `COPY index.php /var/www/html` – Der Apache-Webserver in Ihrem Image hat das Standardverzeichnis für Dokumente auf den Pfad `/var/www/html` gesetzt. Genau an diese Position im Image kopieren Sie die Datei `index.php`.

Damit ist Ihr erstes Docker-Image fertig. Sie können es auf jedem Computer verwenden, auf dem eine aktuelle Docker-Version läuft.

Das war verblüffend einfach, oder? Sie mussten nicht die gerade aktuelle Apache-Version im Internet suchen und herunterladen, das dazu passende PHP-7-Modul suchen und installieren und dann die `index.php` an die richtige Stelle auf Ihrem Computer kopieren. Außerdem haben Sie auch nicht gerade unzählige Dateien an unterschiedlichen Orten im Dateisystem Ihres Computers verteilt, die bei einer Deinstallation einer der Komponenten schwer zu finden sind. Docker hilft Ihnen also unter anderem bei einem schnellen Start und dabei, einen *sauberen* Computer zu behalten.

1.3 Node.js

Die zweite Hello-World Variante wird mit der populären JavaScript-Runtime *Node.js* umgesetzt. Erstellen Sie ein neues Verzeichnis mit dem Namen `hello-world-node`. Das `Dockerfile` ist wiederum sehr kurz gehalten:

```
# Datei: hello-world-node/Dockerfile (docbuc/hello-world-node)
FROM node:12
ENV TZ="Europe/Amsterdam"
COPY server.js /src/
USER node
CMD ["node", "/src/server.js"]
```

Bei diesem Beispiel baut Ihr Docker-Image auf `node:12` auf. Das ist wiederum ein offizielles Image, das von den Node.js-Entwicklern gewartet wird. Gleich wie im ersten Beispiel wird die Standardzeitzone gesetzt, um für eine korrekte Uhrzeit zu sorgen. Anschließend wird die `server.js`-Datei vom lokalen Dateisystem in Ihr Image kopiert.

Da Sie den Server nicht als `root`-Benutzer ausführen wollen, wechseln Sie im nächsten Schritt zum unprivilegierten Benutzer `node`. Neu ist hier auch die letzte Zeile mit der `CMD`-Instruktion. Diese startet den Node.js-Interpreter mit dem Parameter `server.js`. Warum der Aufruf als Array-Konstrukt geschrieben wird, erfahren Sie in Abschnitt 3.1, »Dockerfiles«.

Die server.js-Datei enthält den Code zum Start eines Webservers, der auf eingehende Anfragen antwortet:

```
// Datei hello-world-node/server.js
const http = require("http"),
  os = require("os");

http.createServer((req, res) => {
  const dateTime = new Date(),
    load = os.loadavg(),
    doc = `<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Hello world: node</h1>
    Serverzeit: ${dateTime}<br />
    Serverauslastung (load): ${load[0]}
  </body>
</html>`
  res.setHeader('Content-Type', 'text/html');
  res.end(doc);
}).listen(8080);
```

Auch die Node.js-Runtime hat einen Aufruf für die Unix-Load, os.loadavg(). Dieser Aufruf funktioniert analog zu der im vorigen Abschnitt beschriebenen PHP-Funktion (mit dem kleinen Unterschied, dass die Ausgabe von Node.js mit 12 Nachkommastellen sehr detailliert ist – siehe Abbildung 1.2).

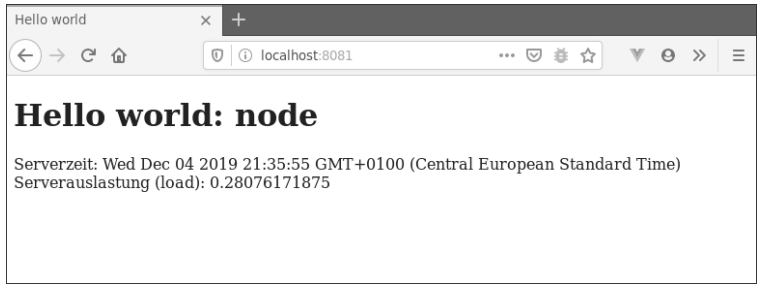

Abbildung 1.2 Der Hello-World-Webserver mit dem Node.js-Container im Browser

Um diese Variante zu starten, müssen Sie zuerst ein neues Docker-Image erzeugen:

```
docker build -t hello-world-node .
```


Wenn bei dem Vorgang keine Fehler auftreten, können Sie einen Container auf Basis Ihres neuen hello-world-node-Images starten:

```
docker run -p 8080:8080 hello-world-node
```

Sollte das docker-Kommando mit einer etwas kryptischen Fehlermeldung quittiert werden, die beispielsweise mit

```
Bind for 0.0.0.0:8080 failed: port is already allocated.
```

endet, so läuft höchstwahrscheinlich noch der Container aus dem ersten Beispiel und belegt den Port 8080. Linux-Systeme erlauben nur einen Dienst pro Port zu verwenden. Wer sollte sonst auch die Anfrage beantworten: der PHP-Container oder der Node.js-Container? Um den Node.js-Container erfolgreich zu starten, können Sie entweder den PHP-Container beenden (Strg+C im Terminal), oder Sie verbinden den Port 8080 des Node-Containers mit einem anderen freien Port auf dem lokalen System:

```
docker run -p 8081:8080 hello-world-node
```

Der Node.js-Container liefert unter der Adresse *http://localhost:8081* eine Testseite (siehe Abbildung 1.2). Parallel können Sie den noch laufenden PHP-Container unter der bisherigen Adresse *http://localhost:8080* erreichen. Die Angabe der Ports beim Parameter -p erfolgt vom lokalen zum entfernten System, Sie verbinden also den lokalen Port 8081 auf Ihrem System mit dem Port 8080 des Containers.

Wenn Sie den Node.js-Container beenden wollen, werden Sie feststellen, dass dieser hartnäckiger ist als der PHP-Container. Die Tastenkombination Strg+C hilft hier nicht weiter. Um den Container zu beenden, ohne den Computer neu zu starten, müssen Sie sich zuerst die eindeutige Kennung des Containers ausgeben lassen. Die Docker-Kommandozeile kann auch hier helfen:

```
docker ps
```

listet alle laufenden Container auf (Sie benötigen dazu eine neue Shell). Die Ausgabe könnte so ähnlich aussehen.

```
CONTAINER ID    IMAGE             COMMAND
8b7c3ff0a405    hello-world-node  "node /src/server.js"

CREATED                          PORTS
21 seconds ago Up 20 seconds     0.0.0.0:8081->8080/tcp

NAMES
heuristic_roengen
```

Die Ausgaben wurden im obigen Listing aus Platzgründen über mehrere Zeilen verteilt. Zum aktuellen Zeitpunkt ist nur die erste Spalte, `CONTAINER ID`, interessant. Mit dieser hexadezimalen Zeichenkette können Sie den Container stoppen:

```
docker stop 8b7c3ff0a405
```

Für produktive Arbeiten würde man wohl keinen Webserver auf diese Art und Weise programmieren. Alle Anfragen, egal an welche URL auf diesem Server, werden immer mit der gleichen Antwort quittiert. Im Node.js-Umfeld werden Webserver in der Regel mit der gut entwickelten `express`-Bibliothek erstellt. (Details dazu folgen in Kapitel 7, »Webserver und Co.«).

1.4 Python

Damit auch die Python-Community nicht zu kurz kommt, zeigen wir hier eine dritte Variante des Hello-World-Beispiels in Python. Egal welche Python-Version Sie auf Ihrem Computer installiert haben (oder ob Sie überhaupt Python installiert haben), mit diesem `Dockerfile` starten Sie `python` in der aktuellen Version 3:

```
# Datei: hello-world-python/Dockerfile (docbuc/hello-world-python
FROM python:3
ENV TZ="Europe/Amsterdam"
COPY server.py /src/
USER www-data
CMD ["python", "/src/server.py"]
```

Der Python-Code für den Webserver ist dem vorangegangenen Node.js-Beispiel sehr ähnlich:

```
#!/usr/bin/env python3
from http.server import BaseHTTPRequestHandler, HTTPServer
import os, datetime

class myServer(BaseHTTPRequestHandler):
  def do_GET(self):
    load = os.getloadavg()
    html = """<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Hello world: python</h1>
    Serverzeit: {now}<br />
```



```
    Serverauslastung (load): {load}
  </body>
</html>""".format(now=datetime.datetime.now().astimezone(),
                  load=load[0])

    self.send_response(200)
    self.send_header('Content-type','text/html')
    self.end_headers()
    self.wfile.write(bytes(html, "utf8"))
    return

def run():
  addr = ('', 8080)
  httpd = HTTPServer(addr, myServer)
  httpd.serve_forever()

run()
```

Python-Freunde werden sich gleich auskennen: In der `run`-Funktion wird der Webserver auf Port 8080 gestartet; die `do_GET`-Funktion in der `myServer`-Klasse behandelt den `HTTP GET`-Aufruf, den der Browser beim Aufruf der Adresse *http://localhost* auslöst. Das HTML-Gerüst in der `html`-Variablen wird mit dem aktuellen Datum und der *load* gefüllt, die Ihnen aus den vorigen Abschnitten schon bekannt ist. Nach dem Setzen der Kopfzeilen wird die `html`-Variable in `utf8` konvertiert und an den Browser gesendet (`self.wfile.write`).

Um diesen Service als Docker-Container zu starten, müssen Sie wie bei den vorangegangenen Beispielen zuerst das Image erzeugen:

```
docker build -t hello-world-python .
```

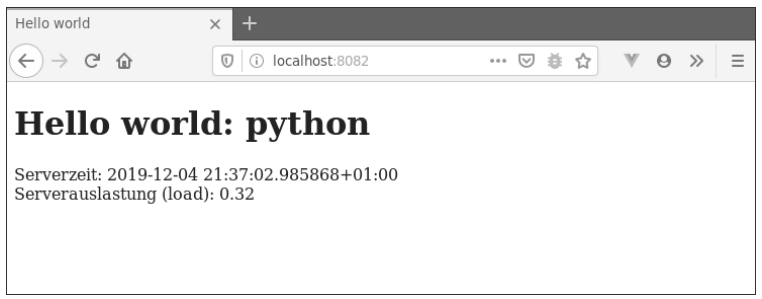

Abbildung 1.3 Der Hello-World-Webserver mit dem Python-3-Container im Browser

Anschließend erzeugen und starten Sie den Container, hier mit dem Port-Mapping auf 8082 auf dem lokalen Port (siehe Abbildung 1.3):

```
docker run -p 8082:8080 hello-world-python
```

1.5 Fazit

In diesem Kapitel haben Sie erlebt, wie einfach es ist, unterschiedliche Programmierumgebungen in Docker zu starten. Viele renommierte Softwareprojekte betreuen ihre Docker-Images auf Docker Hub, von wo wir auch die Images in diesem Kapitel geladen haben (siehe auch Kapitel 2, »Installation und Grundlagen«).

Außerdem haben Sie eines der stärksten Features von Docker gesehen: Egal auf welchem System Sie arbeiten und welche Version Ihre lokal installierten Bibliotheken oder Programmiersprachen haben, mit Docker können Sie genau die gewünschte Version einer Software starten, ohne sich mit der Installationsprozedur zu quälen und ohne Ihren Computer mit möglicherweise inkompatiblen Bibliotheken zu belasten.

Kapitel 6

Alpine Linux

Als Basis für Docker-Container kommen alle möglichen Linux-Distributionen zum Einsatz: CentOS, Debian, Ubuntu etc. Im Mittelpunkt dieses Kapitels steht aber eine Linux-Distribution, die außerhalb der Docker-Welt (und eventuell im Umfeld von *Embedded Linux*) weitgehend unbekannt ist. Alpine Linux ist im Hinblick auf Sicherheit und den sparsamen Umgang mit Ressourcen optimiert und unterscheidet sich in vielen technischen Details von anderen Linux-Distributionen. Das wichtigste Unterscheidungsmerkmal geht aber aus Tabelle 6.1 hervor: Der Platzbedarf des Images ist im Vergleich zu dem anderer Distributionen verschwindend klein!

Distribution	Docker-Image-Größe
Alpine Linux	ca. 4 MByte
CentOS 7	ca. 210 MByte
Debian 9	ca. 100 MByte
Debian 10	ca. 115 MByte
Ubuntu 18.04	ca. 80 MByte

Tabelle 6.1 Docker-Image-Größe von wichtigen Linux-Distributionen

Generell gehen wir in diesem Buch davon aus, dass Sie Linux zumindest in seinen Grundzügen kennen und mit seinen wichtigsten Funktionen und Kommandos vertraut sind. (Sollte das nicht der Fall sein, empfehlen wir Ihnen natürlich die Lektüre von »Linux – Das umfassende Handbuch« von Michael Kofler, Rheinwerk Verlag!)

Aber selbst viele Linux-Profis haben noch nie von Alpine Linux gehört. Deswegen fassen wir in diesem Kapitel ganz kurz zusammen, in welchen Details sich Alpine Linux von anderen Distributionen unterscheidet und wie es zu bedienen ist. Ein Hauptaugenmerk legen wir dabei auf die Paketverwaltung, die beim Zusammenstellen eigener Images von großer Bedeutung ist.

6.1 Merkmale

Um Alpine Linux möglichst schlank zu halten, haben seine Entwickler andere Komponenten ausgewählt, als in »großen« Linux-Bibliotheken üblich. Beispielsweise nutzt Alpine Linux die C-Standardbibliothek `musl` anstelle von `glibc` oder das Init-System OpenRC anstelle von systemd. Auch die grundlegenden Linux-Kommandos stehen nicht in ihren Vollversionen zur Verfügung, sondern stammen in abgespeckter Form aus dem Paket *BusyBox*.

Beim interaktiven Arbeiten bietet Alpine Linux deswegen vergleichsweise wenig Komfort. Aber Alpine Linux reicht aus, um einen Server-Dienst wie z. B. Apache oder Nginx mit minimalem Overhead auszuführen – und darauf kommt es im Docker-Umfeld an. Weitere Informationen können Sie hier nachlesen:

https://alpinelinux.org/about
https://de.wikipedia.org/wiki/Alpine_Linux
http://gliderlabs.viewdocs.io/docker-alpine

Alpine Linux ausprobieren

Auch wenn Alpine Linux nicht für die interaktive Nutzung gedacht ist, sollten Sie sich zum Ausprobieren der Grundfunktionen einen Container erstellen. Am schnellsten gelingt dies mit dem folgenden Kommando:

```
docker run -it  -h alpine --name alpine alpine
```

Damit gelangen Sie in eine interaktive root-Shell, in der Sie die Versionsnummer von Alpine Linux ergründen können:

```
cat /etc/os-release
  NAME="Alpine Linux"
  ID=alpine
  VERSION_ID=3.9.3
  PRETTY_NAME="Alpine Linux v3.9"
  HOME_URL="https://alpinelinux.org"
  BUG_REPORT_URL="https://bugs.alpinelinux.org"
```

Shell

In Alpine Linux wird standardmäßig die Shell `/bin/sh` ausgeführt, die Teil von BusyBox ist (siehe die nächste Überschrift). Diese Shell kann nicht mit dem Komfort der `bash` mithalten, die bei anderen Linux-Distributionen zum Einsatz kommt. Deswegen kann es sich lohnen, zur Erkundung von Alpine Linux die viel größere `bash` zu installieren. (Details zum Kommando `apk` folgen in Abschnitt 6.2, »Paketverwaltung mit apk«.)


```
apk add --update bash bash-completion

/bin/bash
  bash-4.4#
```

Wenn Sie nun in einem zweiten Terminal `docker ps -s` ausführen, werden Sie sehen, dass die Container-Größe von wenigen Bytes auf ca. 6 MByte angewachsen ist. Mit anderen Worten: Allein die `bash` beansprucht um 50 % mehr Speicherplatz als das gesamte Alpine-Image!

BusyBox

BusyBox ist ein für Alpine-Verhältnisse relativ großes Programm (0,8 MByte). Dafür enthält es aber Implementierungen von rund 140 Standardkommandos wie `cat`, `echo`, `grep`, `gzip`, `hostname`, `ip`, `ls`, `mount`, `ping`, `rm`, `route` oder `su`. Diese Kommandos stehen in Form von symbolischen Links auf `busybox` zur Verfügung:

```
ls /bin /sbin -l
  /bin
  ...        12  ash -> /bin/busybox
             12  base64 -> /bin/busybox
             12  bbconfig -> /bin/busybox
         805024  busybox
             12  cat -> /bin/busybox
                 ...

  /sbin
             12  acpid -> /bin/busybox
             12  adjtimex -> /bin/busybox
         206472  apk
             12  arp -> /bin/busybox
                 ...
```

Der offensichtliche Vorteil von BusyBox besteht darin, dass der Platzbedarf für diese Standardkommandos gering ist. Der Nachteil von BusyBox ist, dass die Kommandos zum Teil in einer vereinfachten Form realisiert sind und diverse Optionen fehlen, die unter Linux gebräuchlich sind. Eine Zusammenfassung aller Kommandos und Optionen, die mit BusyBox verwendet werden können, finden Sie hier:

https://busybox.net/downloads/BusyBox.html

Init-System und Logging

Das Init-System ist unter Linux dafür zuständig, beim Hochfahren des Rechners Hintergrund- und Netzwerkdienste zu starten. Dazu besteht bei der Anwendung

unter Docker aber keine Notwendigkeit. (Das gilt auch für die meisten anderen Linux-Images für Docker.)

Insofern trifft es sich gut, dass Alpine Linux anstelle des sonst üblichen riesigen `systemd` das minimalistische System *OpenRC* vorsieht. OpenRC ist mit dem Init-V-System kompatibel, das Ihnen wahrscheinlich aus der Urgeschichte von Unix und Linux vertraut ist. Die zentrale Steuerung erfolgt durch `/etc/inittab`, Init-Scripts können in `/etc/init.d` gespeichert werden. Beachten Sie, dass OpenRC im Alpine-Image für Docker zwar installiert, aber standardmäßig gar nicht ausgeführt wird!

Ähnlich sieht es beim Logging aus: Standardmäßig ist im Alpine-Image kein Logging vorgesehen. Bei Bedarf können Sie das Paket `rsyslog` installieren und müssen sich um dessen Start kümmern.

Die musl-Bibliothek

Die in Alpine Linux eingesetzte Bibliothek `musl` ist eine schlankere `libc`-Implementierung als die weitverbreitete `glibc`. Allerdings führt diese Bibliothek mitunter zu Problemen. Eines besteht darin, dass die Auswertung von `/etc/resolv.conf` vereinfacht wurde. So ignoriert `musl` die Schlüsselwörter `domain` und `search`. Sollten Sie Probleme beim Auflösen von Domainnamen haben, werfen Sie unbedingt einen Blick auf die folgende Webseite:

http://gliderlabs.viewdocs.io/docker-alpine/caveats

Schwierigkeiten machen auch Binärdateien, die nicht für Alpine Linux kompiliert wurden. Sie verwenden mitunter Symbole, die es zwar in der `glibc` gibt, nicht aber in `musl`. Bei der Fehlersuche hilft das Kommando `ldd <binary>`.

Ein weiteres Problem ist die fehlende Unterstützung von Lokalisierungsdateien (*locales*). Ein möglicher Ausweg besteht darin, eine für Alpine Linux optimierte Variante der `glibc` zu installieren. Details können Sie hier nachlesen:

https://github.com/gliderlabs/docker-alpine/issues/144#issuecomment-339906345
https://github.com/sgerrand/alpine-pkg-glibc

Dokumentation

In Alpine Linux sind standardmäßig weder das `man`-Kommando noch die dazugehörenden Hilfetexte installiert. Abhilfe schaffen diese Kommandos, die auch gleich `less` aktivieren, damit Sie bequem durch die Hilfetexte navigieren können:

```
apk add --update man man-pages mdocml-apropos less less-doc
export PAGER=less
```

Beachten Sie, dass Sie damit zwar die `man`-Seiten zu diversen Standardkommandos erhalten, nicht aber solche zu extra installierten Paketen. Unter Alpine Linux ist es

üblich, dass sich Dokumentationsdateien in einem getrennten Paket befinden, das den Namen `<paketname>-doc` hat. Wenn Sie also die `man`-Seiten zur Shell `bash` lesen wollen, müssen Sie zusätzlich das Paket `bash-doc` installieren:

```
apk add --update bash-doc
```

Eine gute Alpine-spezifische Dokumentation finden Sie zudem im Wiki der Projektseite. Ein guter Startpunkt ist hier:

https://wiki.alpinelinux.org/wiki/Tutorials_and_Howtos

Sicherheitsprobleme

Im Mai 2019 wurde bekannt, dass die Docker-Images von Alpine Linux ohne `root`-Passwort ausgeliefert wurden. Zum Glück klingt das dramatischer, als es tatsächlich ist: Im Docker-Umfeld ist es eher ungewöhnlich, dass innerhalb eines Containers eine Benutzerverwaltung aktiv ist. Standardmäßig ist dies weder in Alpine Linux noch in unzähligen darauf aufbauenden Images der Fall. Umgekehrt ist es aber eben nicht auszuschließen, dass einzelne Docker-Anwender Zusatzpakete für die Benutzerverwaltung mit `apk` installieren – und dann wird die Möglichkeit eines `root`-Logins ohne Passwort natürlich wirklich zum Sicherheitsrisiko.

Das Problem wurde mittlerweile behoben. `cat /etc/shadow` zeigt, dass in der Hash-Code-Spalte für das `root`-Passwort ein Ausrufezeichen im Sinne von »ungültiges Passwort« gespeichert ist.

```
cat /etc/shadow
  root:!::0:::::
  bin:!::0:::::
  daemon:!::0:::::
  ...
```

6.2 Paketverwaltung mit apk

Unter Debian und Ubuntu verwenden Sie zur Installation von Paketen `apt`, unter Fedora `dnf`, unter CentOS `yum`. Das äquivalente Kommando unter Alpine Linux heißt `apk`. Die beiden wichtigsten Kommandos `apk update` zum Einlesen der Paketquellen sowie `apk add <name>` zur Installation eines Pakets haben Sie im vorigen Abschnitt ja schon kennengelernt. Einige weitere Kommandos fasst Tabelle 6.2 zusammen. Noch mehr Details und Optionen können Sie hier nachlesen:

https://wiki.alpinelinux.org/wiki/Alpine_Linux_package_management

Kommando	Funktion
`apk add <name>`	installiert das angegebene Paket.
`apk del <name>`	entfernt das angegebene Paket.
`apk info`	listet die installierten Pakete auf.
`apk search <name>`	sucht nach Paketen in den Paketquellen.
`apk stats`	zeigt an, wie viele Pakete installiert sind.
`apk update`	ermittelt, welche Pakete aktuell verfügbar sind.
`apk upgrade`	aktualisiert alle installierten Pakete.

Tabelle 6.2 Die wichtigsten Kommandos zur Paketverwaltung

Standardmäßig sind im Docker-Image von Alpine Linux nur die folgenden Pakete installiert:

```
apk info | sort
  alpine-baselayout
  alpine-keys
  apk-tools
  busybox
  ca-certificates-cacert
  libc-utils
  libcrypto1.1
  libssl1.1
  libtls-standalone
  musl
  musl-utils
  scanelf
  ssl_client
  zlib
```

Paketquellen

Die Paketquellen für Alpine Linux sind in der Datei `/etc/apk/repositories` definiert:

```
cat /etc/apk/repositories
  http://dl-cdn.alpinelinux.org/alpine/v3.9/main
  http://dl-cdn.alpinelinux.org/alpine/v3.9/community
```

Das Paketangebot ist nicht so riesig wie unter Debian oder Ubuntu, aber mit fast 10.000 Paketen doch sehr beachtlich:


```
apk update
  ...
  OK: 9770 distinct packages available
```

Die Index-Dateien der Paketquellen werden in `/var/cache/apk` gespeichert und beanspruchen ca. 1 MByte Platz. Um Speicherplatz zu sparen, können Sie die dort befindlichen Dateien nach Abschluss der Installationsarbeiten wieder löschen.

Mit `apk search` können Sie in den Paketquellen nach Paketen suchen:

```
apk search php7 | sort
  cacti-php7-1.1.38-r1
  php7-7.2.18-r0
  php7-apache2-7.2.18-r0
  php7-bcmath-7.2.18-r0
  ...
```

Alternativ können Sie zur Paketsuche auch einen Blick auf die folgende Seite werfen:

https://pkgs.alpinelinux.org/packages

Vergessen Sie »apk update« nicht!

Bevor Sie das erste Paket installieren können, müssen Sie mit `apk update` einen lokalen Index erstellen, der alle aktuell in den Paketquellen verfügbaren Pakete enthält. Alternativ können Sie `apk add` mit der Option `--update` ausführen: Dann wird `apk update` automatisch ausgeführt.

Wenn Sie `apk` in einem `Dockerfile` verwenden, um ein neues Image zu erstellen, das zusätzliche Pakete nutzt, sollten Sie anstelle der gerade erwähnten Option `--update` die Option `--no-cache` verwenden. Auch diese Option bewirkt ein `apk update`, allerdings wird der heruntergeladene Paketindex nach der Installation sofort aus dem Cache gelöscht. Das verhindert, dass das Image durch unnötige Daten aufgebläht wird.

Ebenfalls empfehlenswert ist es, nur vorübergehend benötigte Pakete sofort wieder zu löschen, wie dies im folgenden Beispiel zu sehen ist:

```
# Datei Dockerfile
...
RUN apk add --no-cache \
    build-base \
    python-dev \
    jpeg-dev \
    zlib-dev \
    ffmpeg \
    && pip install sigal \
```

```
    && pip install cssmin \
    && apk del build-base python-dev jpeg-dev zlib-dev
```

Pakete und ihre Dateien

`apk info <name>` fasst die wichtigsten Informationen zu einem Paket zusammen:

```
apk info musl
  musl-1.1.20-r4 description:
    the musl c library (libc) implementation
  musl-1.1.20-r4 webpage:         http://www.musl-libc.org/
  musl-1.1.20-r4 installed size:  602112
```

Mit `apk info -L <name>` liefert das Kommando eine Liste aller Pakete, die zu einem Paket gehören:

```
apk info -L musl
  musl-1.1.20-r4 contains:
  lib/libc.musl-x86_64.so.1
  lib/ld-musl-x86_64.so.1
```

Umgekehrt können Sie mit `apk info --who-owns <datei>` feststellen, zu welchem Paket die angegebene Datei gehört:

```
apk info --who-owns /bin/ls
  /bin/ls symlink target is owned by busybox-1.29.3-r10
```

Kapitel 12

Grafana

Im IT-Umfeld gibt es unzählige Daten, die analysiert werden müssen. Die meisten Menschen – mal abgesehen von Statistikern – tun sich schwer, Zahlen in Tabellen auf einen Blick sinnvoll zu interpretieren. Wir verwenden dazu gern Grafiken, und wenn diese Grafiken auch noch gut aussehen, dann ist das umso besser.

Zwar gibt es für alle möglichen Programmiersprachen eine große Anzahl von Bibliotheken, mit denen man Grafiken zeichnen kann. Sie erfordern aber meist etwas Programmieraufwand, was es dem Endanwender nicht ermöglicht, die Ausgabe maßgeblich zu beeinflussen.

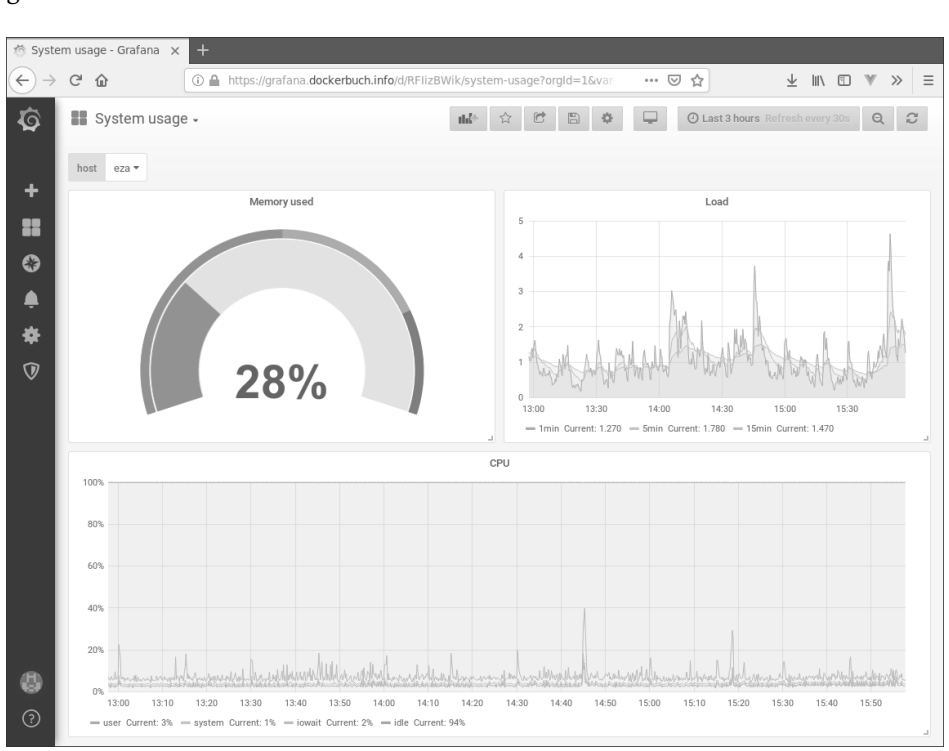

Abbildung 12.1 Informationen zur Systemauslastung in Grafana (hier mit dem hellen Theme)

Grafana ist angetreten, um grafisch schöne und inhaltlich flexible Dashboards zu verwalten. Das soll auch für Nichtprogrammierer möglich sein, und zwar über eine Weboberfläche.

Warum stellen wir Grafana in diesem Buch vor? Wenn Sie Grafana einmal ausprobieren wollen, helfen Ihnen die für alle Plattformen zur Verfügung gestellten Pakete nur begrenzt weiter: Sie benötigen einen Dienst, der Daten sammelt, und einen weiteren Dienst, der Daten in einem kompatiblen Format speichert, um sie mit Grafana zu visualisieren. Sie merken schon: Mit einer einzelnen Windows-EXE-Datei ist es hier nicht getan, und auch unter Linux werden Sie verschiedene Pakete mit mehreren Abhängigkeiten installieren müssen, um zu einem lauffähigen Grafana-System zu gelangen.

Mit Docker können Sie diese Aufgabe quasi in einem Handgriff erledigen: Sie benötigen drei verschiedene Docker-Images für die drei angesprochenen Dienste in einem `docker-compose`-Setup sowie minimalen Konfigurationsaufwand, um ein Testsystem zu starten. Wir sprechen hier von wenigen Minuten Aufwand, und wenn Sie nach Ihren Tests nicht zufrieden sind, löschen Sie Container, Images und Volumes von Ihrem Computer (`docker-compose --rmi all -v down`), und Ihr System bleibt *sauber*.

12.1 Grafana-Docker-Setup

In diesem Abschnitt wollen wir ein Docker-Setup vorstellen, in dem Grafana mit einer Datenbank und einem Collector arbeitet. Das Setup liest Performance-Daten von Ihrem Computer aus und erzeugt daraus Grafiken (siehe Abbildung 12.1).

> **Beispielcode auf GitHub**
>
> Wie zu den meisten Beispielen in diesem Buch finden Sie auch den vollständigen Quellcode zu diesem Kapitel auf GitHub:
>
> *https://github.com/docbuc/grafana*

Daten mit Kollektoren erzeugen (Telegraf)

Die Daten, die wir visualisieren wollen, werden wir selbst erzeugen, indem wir sie vom laufenden Computer abgreifen. Performance-Daten zur Auslastung Ihres Computers eignen sich hervorragend zur Darstellung mit Grafana.

In klassischer Microservice-Architektur werden wir für jeden Dienst einen Docker-Container verwenden. Für die Datensammlung verwenden wir *Telegraf*. Telegraf ist ein in der Programmiersprache *Go* geschriebener Dienst, der mithilfe von Plugins verschiedenste Daten erheben kann. Die Liste der *Input-Plugins* ist sehr lang. Hier folgt nur ein kurzer Auszug einiger der bekannteren Dienste:

- Apache
- AWS cloudwatch
- Docker
- Dovecot
- iptables
- Kubernetes
- MongoDB
- MySQL
- ping
- …

Nachdem wir die Daten erhoben haben, können wir sie transformieren und aggregieren, bevor wir sie an ein *Output-Plugin* weiterleiten. Für die Ausgabe wird meist eine zeitreihenoptimierte Datenbank verwendet, wie etwa *InfluxDB*, die aus dem gleichen Softwareprojekt entstanden ist. Mehr dazu finden Sie im nächsten Abschnitt.

In unserem Setup wollen wir Statistiken zur Systemauslastung (CPU und Speicher), zur Netzwerkverfügbarkeit (*ping*) und zu Docker selbst visualisieren. Der entsprechende Ausschnitt aus der `docker-compose.yml`-Datei sieht folgendermaßen aus:

```
telegraf:
  image: telegraf:1.10
  hostname: telegraf
  volumes:
    - ./telegraf.conf:/etc/telegraf/telegraf.conf:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
  restart: always
```

Wir verwenden das offizielle Docker-Image für Telegraf in der aktuellen Version `1.5` und binden zwei Volumes ein: die Konfigurationsdatei `telegraf.conf` aus dem aktuellen Verzeichnis und das Docker-Socket. Beides wird nur lesend eingebunden, was Sie an der abschließenden Zeichenkette `:ro` erkennen. Während Statistiken zur CPU und zum Speicher des Host-Computers auch innerhalb des Containers abgefragt werden können, benötigen wir für die Docker-Statistiken das eingebundene Socket. Da Telegraf selbst keine Daten speichert, verwenden wir hier kein separates Docker-Volume.

> **Docker unter Windows**
>
> Das Docker-Socket `/var/run/docker.sock` steht Ihnen unter Windows nicht zur Verfügung. Wenn Sie die vorgestellte Konfiguration mit der aktuellen Version von Docker unter Windows starten möchten, ist es am einfachsten, wenn Sie diese Zeile auskommentieren. Sie erhalten dann natürlich keine Statistiken zum Docker-Daemon.

Die (hier leicht gekürzte) Telegraf-Konfigurationsdatei ist nicht weiter kompliziert:

```
[agent]
  interval = "10s"
[[outputs.influxdb]]
  urls = ["http://influx:8086"]
  database = "telegraf"
  timeout = "5s"
[[inputs.cpu]]
  percpu = true
  totalcpu = true
[[inputs.mem]]
[[inputs.system]]
[[inputs.ping]]
  urls = ["www.google.com"]
  count = 1
[[inputs.docker]]
  endpoint = "unix:///var/run/docker.sock"
...
```

Wie Sie erkennen können, gibt es einen *globalen* Abschnitt `[agent]`, in dem Sie das Abfrage-Intervall einstellen. Die weiteren Abschnitte definieren jeweils Input- beziehungsweise Output-Plugins. Das Docker-Input-Plugin benötigt die Angabe eines `endpoint`, in unserem Fall ist dies das eingebundene Unix-Socket. Das Plugin könnte auch einen Docker-Daemon auf einem entfernten System überwachen, wenn dieser über das Netzwerk erreichbar ist. Die Zeile müsste dann entsprechend die IP-Adresse enthalten, zum Beispiel `endpoint = "tcp://1.2.3.4:2375"`.

Wenn Sie sich für weitere Input-Plugins interessieren, kopieren Sie sich am besten die originale Konfigurationsdatei aus dem Telegraf-Container. Sie ist sehr ausführlich kommentiert und erklärt zahlreiche Einstellungen. Zum Kopieren können Sie beispielsweise folgendes Kommando verwenden:

```
docker run --rm -v ${PWD}:/src telegraf:1.10 \
  cp /etc/telegraf/telegraf.conf /src/example.conf
```

Dabei binden Sie das lokale Verzeichnis im Container unter dem Ordner `/src` ein und kopieren die Standardkonfigurationsdatei dorthin. Der Container wird daraufhin beendet und gelöscht (`--rm`).

Telegraf hat auch eine Option, die die aktuelle Konfiguration auf der Kommandozeile ausgibt. Sie können einen Container starten und die Ausgabe mit der Standardkonfigurationsdatei in eine Datei auf Ihrem Computer umleiten:

```
docker run --rm telegraf:1.10 telegraf config > telegraf.conf
```


Konfiguration ohne Kommentare

Wenn Sie nur die aktiven Konfigurationseinstellungen ohne Kommentare oder leere Zeilen sehen möchten, führen Sie unter Linux folgendes Kommando aus:

```
docker run -t telegraf:1.10 telegraf config | egrep -v '(^ *^M$|^ *#)'
```

In dem regulären Ausdruck hinter `egrep` müssen Sie das `^M` als Strg+V, gefolgt von Strg+M, auf der Tastatur eingeben. (Hierbei handelt es sich um das unterschiedliche Zeilenende unter Windows.)

Eine andere Möglichkeit, den regulären Ausdruck zu formulieren, wäre, statt des Windows-Zeilenendes ein beliebiges Steuerzeichen zu suchen. Die Syntax lautet dann so:

```
egrep -v '(^ *[[:cntrl:]]$|^ *#)'
```


Daten speichern mit InfluxDB

Wie schon in der Telegraf-Konfigurationsdatei zu sehen war, verwenden wir *InfluxDB* zum Speichern der Daten. Es handelt sich dabei um eine *Time Series Database*, einen speziellen Typ Datenbank, der auf zeitbezogene Inhalte spezialisiert ist. Vor allem aggregierte Abfragen über einen längeren Zeitraum können mit solchen Datenbanken effizient beantwortet werden.

Mehr zu InfluxDB

Weitere Informationen und Statistiken zur aktuellen Verwendung von *Time Series Databases* finden Sie unter:

https://www.influxdata.com/time-series-database

Sie können InfluxDB gemäß der freien MIT-Lizenz nutzen. InfluxDB funktioniert als Container im `docker-compose`-Setup wunderbar unkompliziert. Für den Anfang benötigt das Image keine weiteren Konfigurationseinstellungen. Um die gesammelten Daten auch bei einem Neustart von Containern nicht zu verlieren, verwenden wir ein benanntes Volume:

```
# Datei: grafana-manual/docker-compose.yml (Auszug)
...
influx:
  image: influxdb
  restart: unless-stopped
  volumes:
    - influx:/var/lib/influxdb
```

```
volumes:
  influx:
...
```

Das Volume `influx` wird mit dem Ordner `/var/lib/influxdb` verbunden, dem Standardordner für die Datenbankdaten. Viel mehr wollen wir zu InfluxDB in diesem Zusammenhang auch nicht sagen – es funktioniert einfach.

Daten visualisieren mit Grafana

Jetzt, da die Daten in der speziellen Datenbank vorhanden sind, geht es noch darum, sie zu visualisieren. Hier kommt Grafana ins Spiel. Im Grafana-Image von Docker Hub wird eine Konfigurationsdatei verwendet, deren Werte mit Umgebungsvariablen überschrieben werden können – ideal für unser Docker-Setup.

Wenn Sie lieber mit einer Konfigurationsdatei arbeiten, können Sie diese auch aus einem laufenden Container kopieren, entsprechend anpassen und mit einem Bind-Mount in Ihren Container einbinden. Um die Standardkonfigurationsdatei mit einem Kommando aus dem Image zu kopieren, müssen Sie bei Grafana den `entrypoint` überschreiben, zum Beispiel mit diesem Kommando:

```
docker run --rm -v ${PWD}:/src -u $UID:$GID --entrypoint=cp \
  grafana/grafana /etc/grafana/grafana.ini /src/
```

Der Trick dabei ist, dass der `entrypoint` für den Container das `cp`-Kommando ist und dass die Parameter für das Kommando (Quelldatei und Zielverzeichnis) nach der Bezeichnung des Images aufgeführt werden. Weil wir `${PWD}` unter `/src` in den Container einhängen, landet die Konfigurationsdatei im aktuellen Verzeichnis.

Für die erste Version werden wir aber gar nicht viele Einstellungen in der Konfigurationsdatei vornehmen; es reicht, mit der Variablen `GF_SECURITY_ADMIN_PASSWORD` ein Passwort für die Weboberfläche zu setzen. Außerdem richten wir ein Volume für die veränderlichen Daten in Grafana ein und verbinden den Port `3000` des Containers mit dem Host. Hier sehen Sie die vollständige `docker-compose.yml`-Datei, mit der wir die erste Version von Grafana starten:

```
# Datei: grafana-manual/docker-compose.yml
version: '3'
services:
  grafana:
    image: grafana/grafana:latest
    restart: always
    ports:
      - 3000:3000
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=geheim
```



```
    volumes:
      - grafana:/var/lib/grafana
  telegraf:
    image: telegraf:1.10
    hostname: telegraf
    volumes:
      - ./telegraf.conf:/etc/telegraf/telegraf.conf:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
    restart: always
  influx:
    image: influxdb
    restart: always
    volumes:
      - influx:/var/lib/influxdb
volumes:
  influx:
  grafana:
```


Starten Sie jetzt das Setup mit `docker-compose up -d`. Anschließend können Sie sich mit dem Benutzernamen `admin` und dem Passwort `geheim` unter der Adresse *http://localhost:3000* einloggen (siehe Abbildung 12.2).

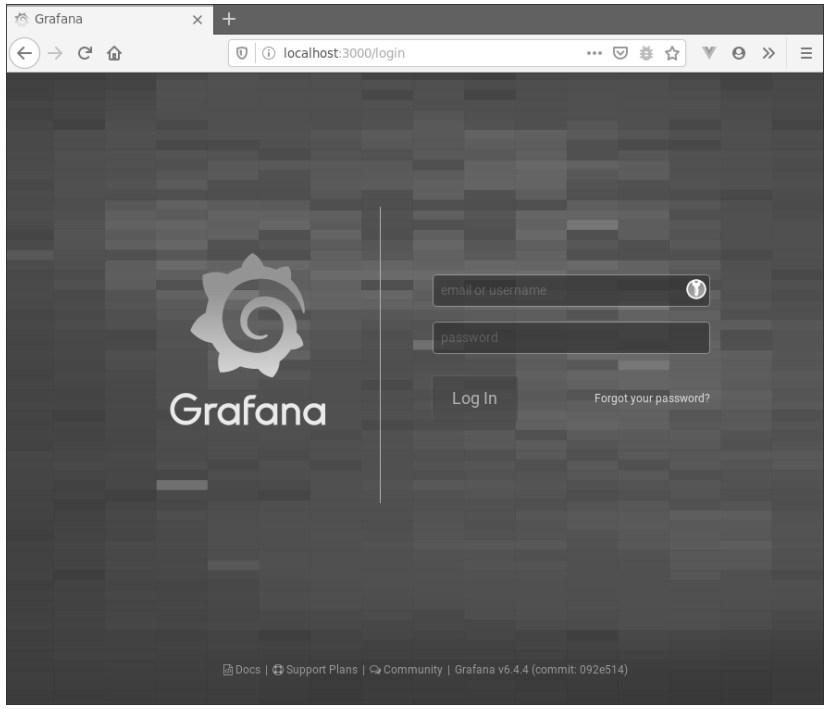

Abbildung 12.2 Der Login-Bildschirm von »Grafana«

Datenquelle und Dashboard erstellen

Nach dem erfolgreichen Login fügen Sie als Erstes eine Datenquelle (*Data Source*) hinzu. Wählen Sie dabei unter TYPE *InfluxDB* aus der Auswahlliste, vergeben Sie einen sprechenden Namen (wir verwenden *Influx*), und stellen Sie unter HTTP • URL *http://influx:8086* ein. Wichtig ist auch, dass die Einstellung HTTP • ACCESS auf *Server (Default)* steht (siehe Abbildung 12.3).

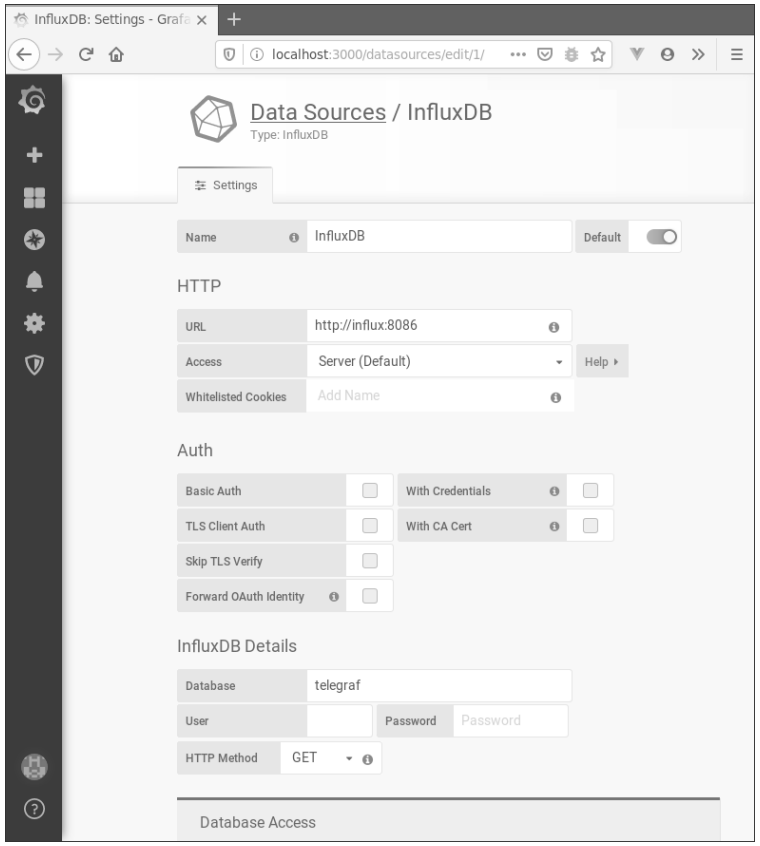

Abbildung 12.3 Die Einstellungen zur Datenquelle »InfluxDB«

Der Grafana-Container greift über die URL *http://influx:8086* auf den Influx-Container zu. Da diese URL außerhalb des Docker-Netzwerks nicht erreichbar ist, muss der Grafana-Container diese Adresse mithilfe eines Proxys vor dem Webbrowser verbergen.

Als weiteren Eintrag auf dieser Seite müssen Sie noch den Namen der Datenbank einstellen. Verwenden Sie dabei den Namen, den Sie in der Datei `telegraf.conf` im Abschnitt `[[outputs.telegraf]]` als `database` eingestellt haben (bei uns war das `telegraf`). Stellen Sie abschließend noch das minimale Zeitintervall auf 10 Sekunden

ein (>10s). Da Sie Telegraf mit einem Intervall von 10 Sekunden gestartet haben, macht es keinen Sinn, wenn InfluxDB Abfragen beantworten würde, die in kürzeren Abständen als 10 Sekunden eingehen.

Grafana verwaltet die grafische Ausgabe in sogenannten *Dashboards*. Diese lassen sich einfach erstellen (über die Weboberfläche) und noch einfacher verbreiten (als JSON-Strings), was für unser Vorhaben, ein flexibles, lauffertiges Docker-Setup zu entwickeln, noch wichtig sein wird.

Sobald die Datenquelle funktioniert, können Sie Ihr erstes Dashboard erstellen (siehe Abbildung 12.4). Klicken Sie im neuen *Panel* auf ADD QUERY um die Abfrage zu erstellen, die die Daten aus der Datenbank holt.

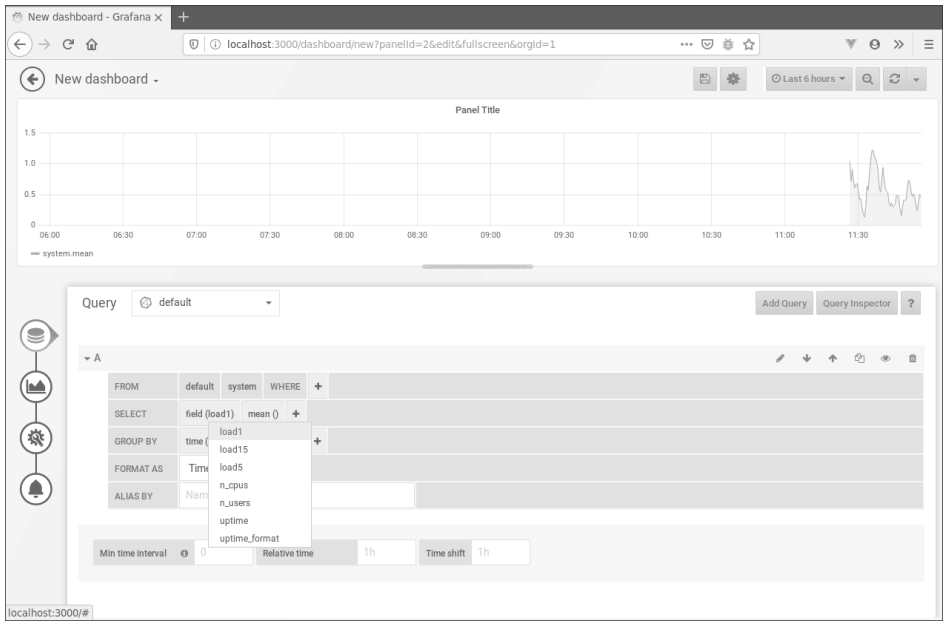

Abbildung 12.4 Das erste Diagramm mit Daten zur Systemauslastung in »Grafana«

Die vorgeschlagene Abfrage hilft schon sehr beim Erzeugen der ersten Grafik. Die Syntax lautet:

```
SELECT mean("value") FROM "measurement"
WHERE $timeFilter
GROUP BY time($__interval) fill(null)
```

Wenn Sie mit der Datenbanksprache *SQL* vertraut sind, werden Sie sich hier gleich auskennen. Aber auch, wenn Sie sich mit SQL nicht auskennen, ist es ein Leichtes, mit dem grafischen Editor die Syntax anzupassen. Wählen Sie einfach unter SELECT MEASUREMENT *system* aus und anschließend bei dem Feld FIELD (VALUE) *load1* (siehe Abbildung 12.4). Schon erscheint die erste Liniengrafik in der Anzeige.

Grafana-Dashboards

Wir wollen hier nicht allzu sehr in die Konfiguration von Grafana-Dashboards eintauchen. Die Gestaltungsmöglichkeiten sind vielfältig und, wenn man einmal das Konzept verstanden hat, auch sehr einfach über die Weboberfläche einzustellen. Lesen Sie mehr zu Grafana auf der gut dokumentierten Webseite:

http://docs.grafana.org/guides/basic_concepts

Sie müssen aber nicht bei null beginnen, wenn Sie ein Dashboard erstellen. Auf der Website von Grafana finden Sie eine Menge Dashboards, die von der Community erstellt wurden und die Sie sehr einfach in Ihre eigene Grafana-Installation importieren können.

Filtern Sie nun die Liste der verfügbaren Dashboards unter *https://grafana.com/dashboards* nach *InfluxDB* und *Telegraf*, und suchen Sie sich ein ansprechendes Dashboard aus. Zum Importieren kopieren Sie einfach die *ID* des Dashboards im Menü + Create • Import in die leere Textzeile. Im nächsten Schritt werden Sie nach der Datenquelle gefragt. Wählen Sie hier die zuvor konfigurierte InfluxDB aus (siehe Abbildung 12.5).

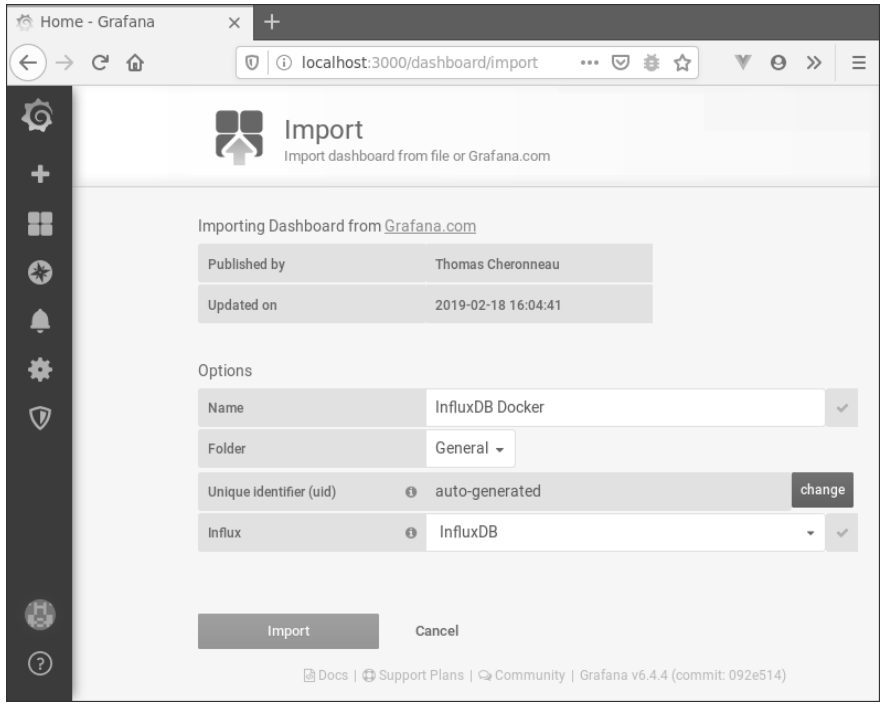

Abbildung 12.5 Import eines Dashboards von der Grafana-Webseite

Nach dem erfolgreichen Import können Sie die Grafiken in dem neuen Dashboard nach Belieben verändern und speichern. Bei manchen Dashboards werden Sie eine Variable namens *Host* oder *Server* finden. Das bisher vorgestellte Setup verwendet nur einen Collector; mehr zu einem verteilten Setup finden Sie in Abschnitt 12.3.

12.2 Provisioning

Die Standardinstallation von Grafana enthält keine vorkonfigurierten Datenquellen oder Dashboards, weshalb Sie diese üblicherweise als Erstes über die Weboberfläche einrichten müssen. Seit Grafana 5 gibt es aber die Möglichkeit, Dashboards und Datenquellen für eine Installation im Voraus bereitzustellen (*Provisioning*), was wir im Folgenden auch machen wollen. Der Vorteil dabei ist, dass wir eine funktionierende Grafana-Instanz ausliefern können, ohne irgendwelche Konfigurationen in der Weboberfläche vorzunehmen.

In der aktuellen Version kann Grafana sowohl Dashboards als auch Datasources vorkonfigurieren. Die Angaben dazu werden in dem Ordner `/etc/grafana/provisioning` als YAML-Dateien erwartet. Die Vorgabe für die InfluxDB-Datenbank sieht in unserem Fall so aus:

```
# Datei: grafana/provisioning/datasources/influx.yml
apiVersion: 1
datasources:
- name: InfluxDB
  type: influxdb
  access: proxy
  orgId: 1
  url: http://influx:8086
  database: telegraf
  isDefault: true
  version: 1
  jsonData:
    timeInterval: ">10s"
  editable: true
```

Wichtig ist der Eintrag `access: proxy`, damit Abfragen, die Grafana an die Datenbank richtet, vor dem Browser verborgen bleiben. Das minimale Zeitintervall für Abfragen an die Datenbank sehen Sie in der Struktur `jsonData - timeInterval`. Mit der abschließenden Einstellung `editable` stellen Sie ein, ob die Datenquelle in der Weboberfläche bearbeitet werden kann oder schreibgeschützt ist.

Die Vorgaben zu Dashboards sehen etwas anders aus. Auch hier werden im Ordner `/etc/grafana/provisioning/dashboards/` YAML-Dateien ausgewertet. Diese enthalten

aber nur den Pfad zu dem Ordner, in dem die Dashboards als JSON-Dateien abgelegt werden. (Sie können auch mehrere Ordner angeben.) Speichern Sie zum Beispiel eine Datei `default.yml` wie folgt in diesem Ordner ab:

```
# Datei: grafana/provisioning/dashboards/default.yml
apiVersion: 1
providers:
- name: 'default'
  orgId: 1
  folder: ''
  type: file
  disableDeletion: false
  editable: false
  options:
    path: /var/lib/grafana/dashboards
```

Grafana sucht jetzt nach Dashboards im Ordner `/var/lib/grafana/dashboards`. Damit Ihr Docker-Setup korrekt funktioniert, müssen Sie jetzt die entsprechenden Ordner in die Datei `docker-compose.yml` einbinden:

```
# Datei: grafana/docker-compose.yml (Auszug)
grafana:
  image: grafana/grafana:latest
  [...]
  volumes:
    - ./dashboards:/var/lib/grafana/dashboards
    - ./provisioning:/etc/grafana/provisioning
[...]
```

Damit das Setup automatisch funktioniert, müssen Sie noch einen Trick in der Konfiguration des Dashboards einbauen. Grafana speichert die Datenquelle in der Dashboard-Definition als Variable ab. Damit das Dashboard die Verknüpfung mit der Datenquelle automatisch herstellen kann, müssen Sie die Variable im Dashboard korrekt definieren.

Die Benennung der Variablen basiert auf dem Namen, den Sie der Datenquelle gegeben haben. Heißt, wie in unserem Fall, die Datenquelle `InfluxDB`, so lautet der Name der Variablen `DS_INFLUXDB`. Definieren Sie diese Variable in den Dashboard-Einstellungen (VARIABLES • NEW • GENERAL • TYPE • DATASOURCE), wobei Sie die Variable ruhig verstecken können (VARIABLES • EDIT • GENERAL • HIDE-VARIABLE); andernfalls wird sie im Dashboard ganz oben angezeigt (siehe Abbildung 12.6).

Exportieren Sie Ihr Dashboard jetzt mit der Funktion SHARE DASHBOARD • EXPORT, und legen Sie die JSON-Datei dann im Ordner `dashboards` ab.

Der Dateisystembaum in Ihrem Projektverzeichnis sollte ähnlich aussehen wie hier:

```
.
|-- dashboards
|   `-- System usage-1524297070270.json
|-- docker-compose.yml
|-- provisioning
|   |-- dashboards
|   |   `-- default.yml
|   `-- datasources
|       `-- influx.yml
`-- telegraf.conf
```

Am besten legen Sie dieses Setup auch in einem Git-Repository ab, wie wir es auf GitHub getan haben:

https://github.com/docbuc/grafana

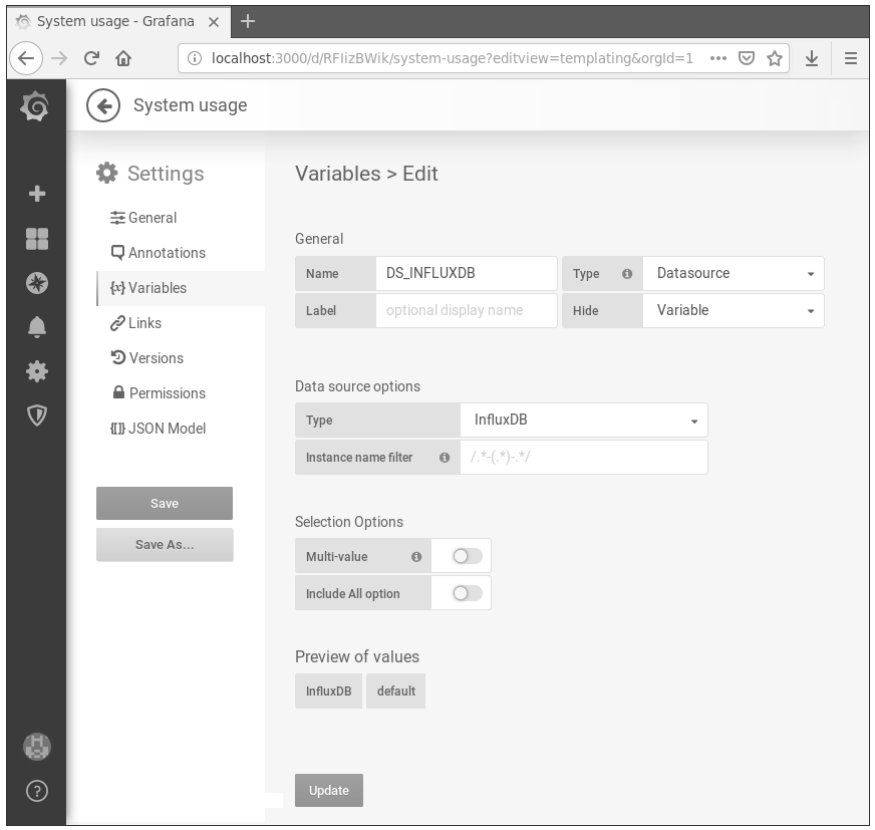

Abbildung 12.6 Variablendefinition in »Grafana«

12.3 Ein angepasstes Telegraf-Image

Das Setup mit Grafana, InfluxDB und Telegraf läuft jetzt auf einem Computer und sammelt fleißig Daten. Im nächsten Schritt wollen wir ein Docker-Image auf der Basis von Telegraf erzeugen, das auf einem anderen Computer gestartet werden kann und Performance-Daten zu der laufenden InfluxDB sendet.

Das Image soll ganz ohne weitere Dateien auskommen. Es soll also nur mit einem Aufruf in der Form

```
docker run -d docbuc/telegraf
```

gestartet werden können, woraufhin der Dienst direkt Daten an die InfluxDB sendet. Dies ist eine einfache Übung, wenn Sie die Daten in die `telegraf.conf`-Datei eintragen und diese in das Image kopieren:

```
# Datei: grafana/telegraf/Dockerfile (docbuc/telegraf)
FROM telegraf
COPY telegraf.conf /etc/telegraf
```

Die Lösung ist aber ein wenig unflexibel, da Sie das Image nur genau mit diesem InfluxDB-Server verwenden können. Eine bessere Möglichkeit besteht darin, Server-Adresse, Server-Port sowie Benutzername und Passwort für die Datenbank als Umgebungsvariablen beim Start zu setzen. Im Container brauchen wir dann eine Möglichkeit, diese Variablen in der Konfigurationsdatei zu ersetzen.

Da die Variablen erst zur Laufzeit im Container verfügbar sind, können Sie diese Ersetzung nicht im Dockerfile vornehmen. Sie müssen ein `ENTRYPOINT`-Script erstellen, das diese Aufgabe übernimmt.

Prinzipiell kann als `ENTRYPOINT` jedes ausführbare Programm verwendet werden, das im Image vorhanden ist. Für unseren Zweck verwenden wir ein kurzes *Bash-Script*:

```
#!/bin/bash
# Datei: grafana/telegraf/entrypoint.sh
set -e

INFLUXDB_HOST=${INFLUXDB_HOST:-influx}
INFLUXDB_PORT=${INFLUXDB_PORT:-8086}
INFLUXDB_USER=${INFLUXDB_USER:-telegraf}

sed -i "s/influx:8086/$INFLUXDB_HOST:$INFLUXDB_PORT/g;
  s/username = \"telegraf\"/username = \"$INFLUXDB_USER\"/g;
  s/password = \"influxpass\"/password = \"$INFLUXDB_PASS\"/g" \
    /etc/telegraf/telegraf.conf

if [ "${1:0:1}" = '-' ]; then
```



```
    set -- telegraf "$@"
fi

exec "$@"
```

In diesem Script sind einige interessante Bash-Tricks versteckt, die im Docker-Umfeld sehr nützlich sind:

- `set -e` beendet das Script mit einem Fehlerstatus, sobald ein Schritt in dem Script fehlschlägt. Da der `ENTRYPOINT` der Prozess ist, der mit dem Container-Status verbunden ist, wird auch der Container beendet, wenn das Script fehlschlägt.
- **Variablendefinition:** In den folgenden Zeilen werden die Variablen `INFLUXDB_*` definiert. Dabei wird der Inhalt der übergebenen Variablen mit dem gleichen Namen ausgewertet, und wenn diese nicht gesetzt sind, wird ein Standardwert festgelegt (hinter dem -).
- **Variablenersetzung:** Das `sed`-Kommando ersetzt bestimmte Zeichenketten in der Konfigurationsdatei `/etc/telegraf/telegraf.conf`. Der Parameter `-i` bewirkt die Ersetzung, ohne eine Kopie der Datei anzulegen (*in place*). Bei der Ersetzung wird der Inhalt der Variablen und nicht die Bezeichnung der Variablen in der Datei gespeichert.
- **Überprüfung von CMD:** Das folgende `if`-Konstrukt haben wir direkt aus dem Original-Dockerfile des `telegraf`-Images übernommen. Es überprüft, ob das Kommando, das beim Containerstart übergeben wurde, mit dem Zeichen - beginnt. Dabei werden die erweiterten Bash-Variablenfunktionen verwendet, um das erste Zeichen der Variablen `$1` zu extrahieren (`${1:0:1}`). `$1` entspricht in Bash-Scripts dem ersten Parameter, der dem Script übergeben wird. Trifft die Bedingung zu, so wird die Zeichenkette `telegraf` vor alle anderen übergebenen Parameter gesetzt (`set -- telegraf "$@"`).
- **Ausführung von CMD:** Abschließend werden alle Script-Parameter mit dem `exec`-Kommando ausgeführt. Damit gibt das Shell-Script die Kontrolle an das Programm ab, das als `$1` in der Liste der Parameter steht. Dieses Programm bekommt jetzt die Prozess-ID 1, und Signale an den Container werden an dieses Programm weitergeleitet.

Im `Dockerfile` müssen Sie jetzt noch das Script kopieren und den `ENTRYPOINT` setzen:

```
# Datei: grafana/telegraf/Dockerfile (docbuc/telegraf)
FROM telegraf
COPY telegraf.conf /etc/telegraf
COPY entrypoint.sh /
ENTRYPOINT ["/bin/bash", "/entrypoint.sh"]
CMD ["telegraf"]
```

Mit dem beschriebenen `ENTRYPOINT`-Script in der Kombination mit `CMD` ist Ihr Docker-Image sehr flexibel geworden. Sie können einen Container starten, der Telegraf mit der vorgegebenen Konfiguration betreibt:

```
docker run -d -v /var/run/docker.sock:/var/run/docker.sock:ro \
  docbuc/telegraf
```

Der Container startet jetzt mit den Einstellungen aus der `telegraf.conf`-Datei, wobei die Ersetzungen im `entrypoint.sh`-Script mit den Standardwerten durchgeführt werden. Das Passwort bleibt leer. Da Sie kein Kommando beim `docker run`-Befehl übergeben haben, wird das `CMD` aus dem `Dockerfile` verwendet, und es startet `telegraf`. Wenn Sie, wie oben angegeben, das Docker-Socket einbinden, werden auch die Statistiken zu Docker protokolliert.

Sie können sich aber auch einfach die Hilfeseite zu `telegraf` anzeigen lassen:

```
docker run --rm docbuc/telegraf --help

  Telegraf, The plugin-driven server agent for collecting and
  reporting metrics.

  Usage:
  telegraf [commands|flags]
  ...
```

Hier kommt die `if`-Abfrage ins Spiel: Das übergebene Kommando startet mit dem Zeichen -, also wird die Parameterliste neu zusammengesetzt. Sie besteht anschließend aus `telegraf --help`.

Außerdem funktioniert die Variablenersetzung in der Konfigurationsdatei im Container, was folgendes Beispiel zeigt:

```
docker run --rm -e INFLUXDB_HOST=influx.dockerbuch.info \
  -e INFLUXDB_PORT=8086 -e INFLUXDB_PASS=iijineeZ9iet \
  docbuc/telegraf cat /etc/telegraf/telegraf.conf

  ...
  [[outputs.influxdb]]
    urls = ["http://influx.dockerbuch.info:8086"] # required
    database = "telegraf" # required
    username = "telegraf"
    password = "iijineeZ9iet"
  ...
```

Als weiteren Bonus dieser Variante können Sie jedes andere Programm starten, das in dem Image installiert ist:

```
docker run --rm -it docbuc/telegraf bash
```


Wenig überraschend wird eine Shell gestartet, und Sie können sich in dem Container umsehen. Wenn Sie sich im Container alle laufenden Prozesse anzeigen lassen (`ps xa`), werden Sie feststellen, dass Ihre Shell die Prozessnummer 1 hat:

```
root@9f7c30fca7df:/# ps xa
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss     0:00 bash
   13 pts/0    R+     0:00 ps xa
```

GitHub und Docker Hub

Der Quellcode für das Docker-Image liegt auf GitHub. Auf Docker Hub haben wir unter unserem Account den GitHub-Account verlinkt und einen *automated build* eingerichtet. Sobald wir etwas am Image ändern und eine neue Version auf GitHub stellen (`git push`), startet automatisch ein Build auf Docker Hub.

GitHub: *https://github.com/docbuc/telegraf*
Docker Hub: *https://hub.docker.com/r/docbuc/telegraf*

Starten Sie nun das Docker-Image auf einigen verschiedenen Computern, und lassen Sie die Daten zu der InfluxDB senden:

```
docker run -v /var/run/docker.sock:/var/run/docker.sock:ro \
  --name telegraf -d -e INFLUXDB_HOST=influxdb.dockerbuch.info \
  -e INFLUXDB_PORT=80 -e INFLUXDB_PASS=iijineeZ9iet \
  --hostname laptop@home docbuc/telegraf
```

Wir haben zu diesem Zweck einen InfluxDB-Container auf dem Host `INFLUXDB_HOST=influxdb.dockerbuch.info` gestartet, der auf Port 80 Anfragen aus dem Internet entgegennimmt. Die explizite Angabe des Parameters `--hostname` ist sinnvoll, weil Telegraf beim Senden der Metriken auch den Hostnamen des Computers mitsendet, auf dem das Programm ausgeführt wird. Innerhalb eines Docker-Containers wäre der Hostname eine automatisch generierte Kennung, zum Beispiel `eeafca0dc73c`. Im Dashboard ist es angenehmer, wenn Sie einen sprechenden Namen für Ihre Computer sehen.

Anpassungen im Dashboard

Nachdem Sie das Telegraf-Image auf verschiedenen Computern gestartet haben und Ihre Performance-Daten nun an eine zentrale Influx-Datenbank gesendet werden, müssen Sie das Dashboard noch ein wenig anpassen.

Fügen Sie dem Dashboard eine Variable mit dem Namen *server* hinzu. Sie erreichen das Menü über SETTINGS • VARIABLES • NEW in der Dashboard-Ansicht. Die Variable muss vom Typ *Query* sein, und die entsprechende Abfrage für InfluxDB lautet:

```
SHOW TAG VALUES WITH KEY = "host"
```

Wenn Sie alles richtig eingestellt haben, erhalten Sie bereits eine Vorschau auf die verfügbaren Hosts (siehe Abbildung 12.7).

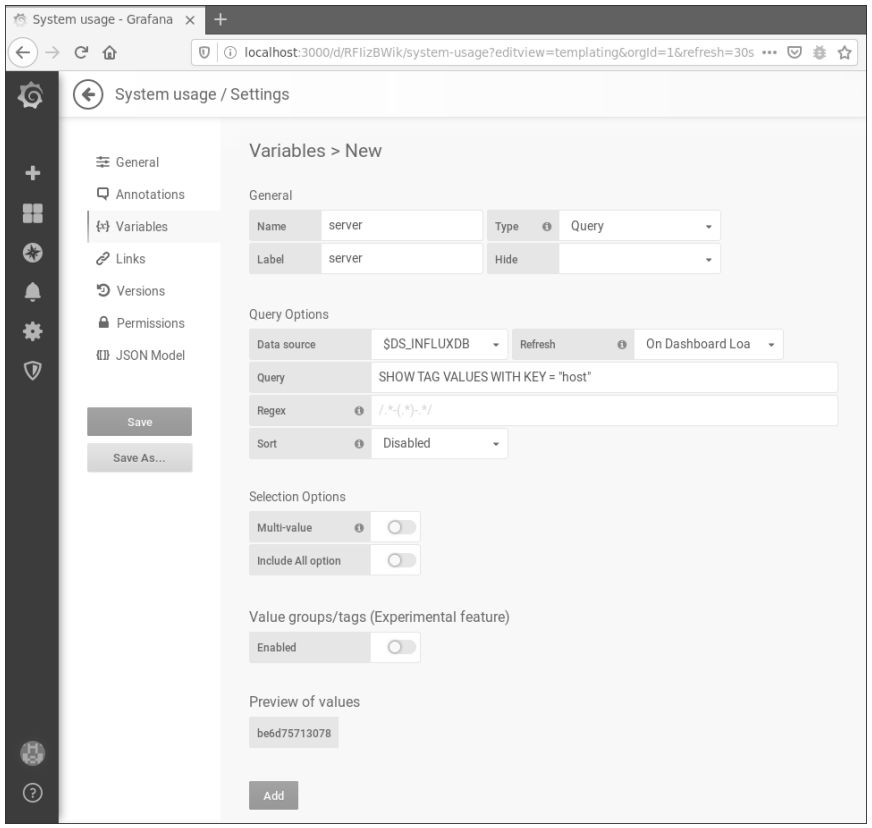

Abbildung 12.7 Fügen Sie die Server-Variable hinzu, um zwischen den unterschiedlichen Statistiken umschalten zu können.

Zurück im Dashboard, sehen Sie bereits das neue Auswahlmenü. Aber leider ändern sich die Werte noch nicht entsprechend. Damit die einzelnen Grafiken mit den korrekten Werten für den Host angezeigt werden, müssen Sie die einzelnen *Panels* bearbeiten. Fügen Sie bei jeder Abfrage in dem METRICS-Tab die Anweisung

```
WHERE ("host" =~ /^$server$/)
```

hinzu. Der praktische *Query builder* in der Weboberfläche hilft hier enorm und schlägt gleich die richtigen Werte vor.

Auf einen Blick

Inhalt

TEIL II Werkzeugkasten

TEIL III Praxis