

Updated in 2019

Get started with Docker in your projects

THROUGH EXAMPLES

by Mark Takacs



Build, Ship and Run examples in Python, Node.js, Go, PHP and more!

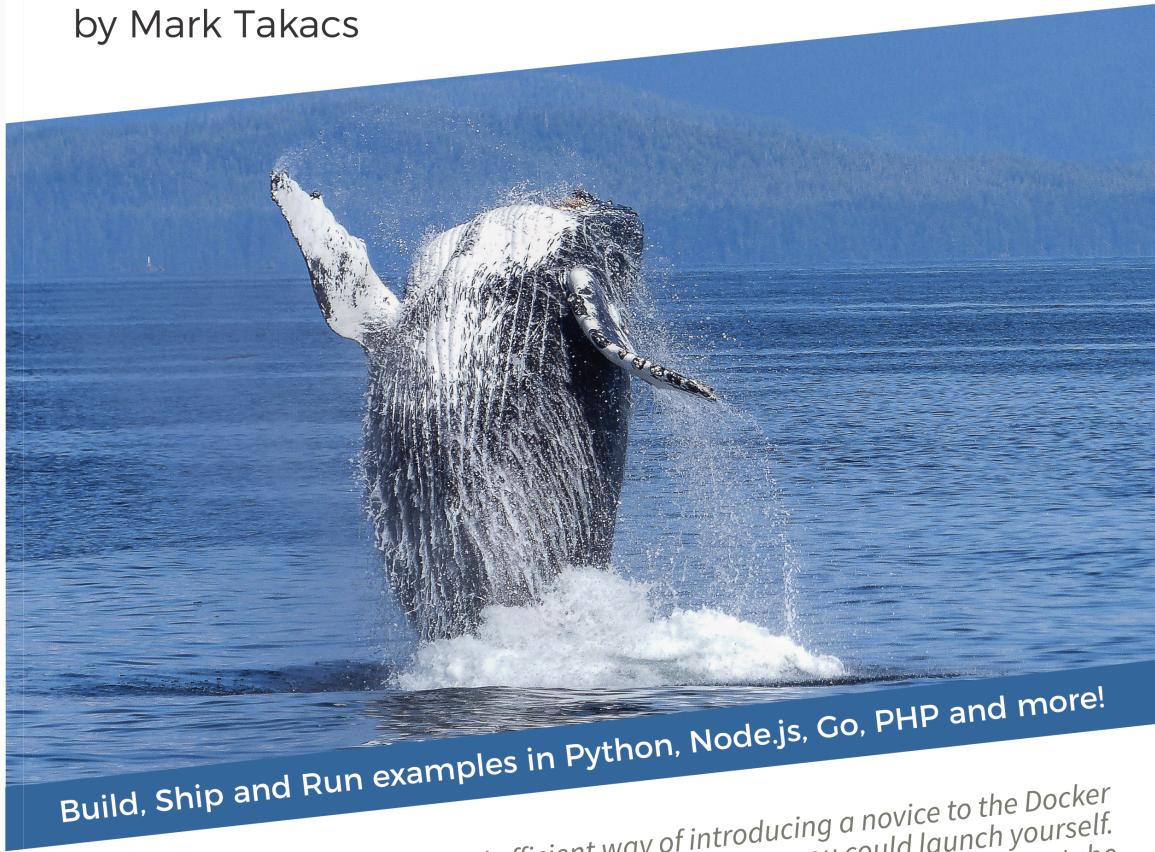
This is really a very crisp and efficient way of introducing a novice to the Docker world. This lays the stepping stone from where you could launch yourself.
- Alaka on Youtube

Updated in 2019

Get started with Docker in your projects

THROUGH EXAMPLES

by Mark Takacs



Build, Ship and Run examples in Python, Node.js, Go, PHP and more!

This is really a very crisp and efficient way of introducing a novice to the Docker world. This lays the stepping stone from where you could launch yourself.
- Alaka on Youtube

Copyright

Published by Savalera Kft.

Copyright © 2019 Márk Takács

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission in writing of the publisher.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Savalera Kft., and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Savalera Kft. has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Savalera Kft. cannot guarantee the accuracy of this information.

The publisher is not responsible for websites (or their content) that are not owned by the publisher.

Savalera Kft.
1112 Tóberek u. 8
Budapest, Hungary.

savalera.com, takacsmark.com

Preface

Dear Reader,

Welcome to “Get Started with Docker in your Projects!”

I’m sure you are as excited as I am about your new learning journey in the Docker world.

I wrote this book to give you a practical, step by step learning experience, where you start from the basics and develop your Docker skills building working examples that resemble real-life project situations.

First, in the introduction chapter, **I explain what Docker is and what problems it solves in software development projects.**

Next, we’ll install Docker on your machine and start building together. **I built every chapter in the book on code-along examples.**

I recommend you to read the book and build the examples in parallel. You can copy and paste the commands from the book to your command line and code editor, or you can use the code base that I supplied in a separate zip file.

We start the journey by running applications in Docker containers. I'll show you how to start and manage your applications with Docker. I added a chapter to demonstrate practices that are crucial for Dockerized project work.

As a next step, **we dive into Docker images**, and see how images define containers. Again, we'll have a project focus, and I'll show you the practices to find and select the best Docker images for your projects online.

As you progress with Docker image learning, you'll create your custom Docker images with the Dockerfile, learn Dockerfile best practices, and I'll show you my image building workflow.

Docker containers and Docker images are the foundation of Docker-based projects; therefore I have included *a long chapter called “Image building exercises” at this point, where we’ll build and run applications with shell scripts, Node.js, Python, PHP, C, and Golang. I have included crucial aspects of Docker into these examples that you’ll find valuable in project work.*

After completing the above exercises, you'll have a firm understanding of Docker concepts, Docker containers, and Docker images.

Next, we'll dive into multi-container applications, where multiple containers work together to provide application

functionality. I'll show you how you can define, build and run applications with the Docker Compose tool, plus, we'll learn about Docker Networks and Docker Volumes in this part of the book in greater detail. I also included a chapter to show you build practices with Compose.

The chapters so far in the book targeted build activities in Docker-based projects. You'll learn how to start and manage containers, how to build Docker images for projects, how to build and manage multi-container applications, manage networks and volumes.

I have included a “build summary” chapter at this point, where I give you an overview of project tasks in the build phase, and you'll get a graphical outline of the build workflow.

Now you'll be ready to **learn how to ship your applications once they are built**. We'll see how to ship your application's Docker images to the Docker Hub, we'll set up an automatic build workflow involving GitHub, and you'll set up your own Docker Registry, too. We'll also see a CI example later in the book.

Similarly to the “build summary” chapter, I have added a “ship summary” chapter to outline the shipping workflow in Dockerized projects.

Next, we'll learn how to run your applications in production. We will run applications in the first chapters, too, but in the “run” part of the book, we'll focus on high availability, scalable production run.

You'll learn how to **setup and manage a Docker Swarm cluster**, how to deploy your application stacks to the Swarm, and how to **scale and manage services** in a Swarm.

We'll start with cluster deployments on your local machine, **first with one machine, and then we'll set up two machines** locally and create a Swarm cluster. We'll learn how to deploy and manage applications on multiple nodes, and we'll learn how to manage nodes in a Swarm cluster.

We'll deploy our application to the cloud in the next chapters.

Again, you'll have a “run summary” chapter with an overview of typical run tasks in your projects.

At this point, you will have solid Docker skills. You shall be ready to address more complex challenges.

To make you more fluent in Docker, I have included an end to end example where we'll Dockerize a Flask application, and we'll use all the learning in the book, plus we'll add more tricks and best practices.

We'll take a popular Flask web application boilerplate project from GitHub and extend it so that we will use Docker to build, ship and run the application.

In this exercise, we'll use Docker containers to generate the application code. We'll use environment variables to drive various aspects of our project build, just like in real-life projects. **We'll automate most of the project tasks.**

We **set up the development environment and automated testing using Travis CI**. We also prepare the production Docker configuration for the project.

Once the project setup is complete, we'll deploy and run the web application on AWS in a Swarm cluster, and in a Kubernetes cluster, too.

I have set up the examples to be gradual with a gentle learning curve. I explain the steps in detail so that you can follow along.

I hope you'll enjoy the journey and acquire great skills.

Corrections

I have taken meticulous care to ensure the accuracy of the content. Still, you may find mistakes in the text or code. If

you see any errata, please contact me at
mark.takacs@savalera.com or mark@takacsmark.com.

Downloads

If you bought the book on Amazon or the iBooks Store,
please download the code files [here](#).

Introduction

Let me give you a short introduction to the big picture without examples now, and we'll go through all of the points again with examples in later chapters.

What is Docker?

Docker is a tool that lets you build, ship, and run your applications in containers.

You usually meet Docker when you are about to ship your application to a new machine, and you need a reliable tool to ship your application's technical environment along with your code.

You can also meet Docker when you need an easy way to set up a new technical environment for a project.

You usually use various technologies in your software development projects. You can have front-end components, back-end servers, databases, and many other elements like cache systems, log management, and others.

You build all these components using a programming language, platform or framework. You can, for example, develop your web server components with Python, Node.js, PHP or similar tools.

As you build your applications with these tools, you are creating source code that fulfills your functional requirements. Along the way, your solution needs more than the source code you created.

You'll install packages, create configuration files and add executable programs to your project's environment. These dependencies are also part of your application; your components won't work without them.

I think we all agree, that the ideal solution would be to encapsulate your program code and its dependencies into one bundle and run this bundle anytime on any machine the same way.

Docker containers do exactly this. Docker containers encapsulate your source code and its dependencies and run your application. Docker containers are portable; you can ship a Docker container to other machines and run the container with the same command anywhere.

In a typical project, you'll create separate containers for different application components. You can, for example,

create a container to run your front-end server and another container to run your back-end.

In this case, your front-end container will encapsulate your front-end code and its dependencies and will run your front-end server process(es).

Similarly, your back-end container will encapsulate your back-end code and its dependencies and will run your back-end server process(es).

The containers are isolated from each other, which means, that even if your Docker containers are running on the same machine, their processes, file systems, networking resources and users are isolated.

Docker containers encapsulate your code and its dependencies. These dependencies include the servers that are used to serve your applications.

If, for example, you are serving a web page with Nginx, your Docker container will include an Nginx web server installation, containing the Nginx runtime and the configuration. It'll also include your source code, once you add it to the container.

Similarly, Node.js containers will include a certain version of Node and npm; Python containers include some version of Python, and so on.

Containers are defined in Docker images. A Docker image is the file system snapshot of files and directories which includes all the files needed to run a container. We'll look inside containers and images in the examples.

You start Docker containers from Docker images with the `docker container run` command. I'll give you detailed examples in the "build" chapters.

A Docker image usually includes operating system programs like package managers (e.g., `apt` or `apk`), utility programs like `curl`, `zip` or `wget`, and system libraries. Images also feature a main technology component, like Node, Python, or PHP preinstalled. Your image will also include your own project files once you add them to the image.

You will find about 1.9 million Docker images online; I'll show you later how to pick the right ones for your project.

You'll find preinstalled images for all the different technologies you may want to use in your project. You'll have a dedicated image for Python, PHP, Go, C, C#, Node, MySQL, CouchBase, MongoDB, Nginx, Redis, Ubuntu, Memcached, Postgres, Ruby, Wordpress, Elasticsearch, Debian and many more.

On top of this, you can create your own images for your projects; I'll show you how.

When you are working with Docker on your projects, you create your own Docker images that include your application code and all the programs and configuration needed to run your applications.

Once your images are ready, you use Docker tools to ship your images to other environments and use Docker commands to start up containers and run your application components.

Your Docker containers will act as isolated technical environments, giving a high level of separation to your application components.

What problem does docker solve?

This is the fundamental question of everyone who wants to start out with Docker. Let's answer the question with everyday words, not Docker terminology.

What problem does Docker solve? Docker solves the problem of having an identical environment for your individual application components across various stages of delivery (e.g., development, testing, staging, and production), where every application has its own isolated environment, the environment is the same across stages, and the environment is as light-weight as possible.

Environment setup and management is a tedious task in every project. Our aim has always been to isolate applications from each other because it's much easier and less costly to resolve issues, re-configure settings, restart and upgrade applications when they are isolated; thus we decrease the probability of a cascading effect when we change one component.

Let's translate the above definition into Docker terms; I'll give you an example shortly, so don't worry if things are not crystal clear immediately.

A Docker container is a running instance of your application, or an application service.

When you're “dockerizing” your application, you are building a file system snapshot of your application and its dependencies (including system packages, binaries, libraries, configuration files, etc. that your application depends on). This snapshot is called a Docker image.

You use Docker commands to start up containers from Docker images. You can start up multiple containers from the same image. You can think of the Docker image as a file system image and containers as running application environments.

To get a better idea of the problem that Docker will solve for you in your projects, let's recap how we managed

environments in the past.

Our infrastructure teams used to create different environment scripts for different stages, like development, test, staging, and production. These environments were not identical, just mostly similar.

On top of all this, we used to do our local development and unit testing on Windows machines, while all other stages were run on Unix systems.

Working like this was not impossible, but it was a costly, time-consuming effort to manage these environments with a lot of inherent risks that caused quality issues in all stages.

Later on, virtualization was widely adopted. In this scenario, we set up virtual machines (VMs) on our physical servers and deployed our applications to virtual machines. This way we had a toolset to keep our applications separate by deploying each component to a dedicated VM.

Virtualization greatly improves maintainability and portability of software components, but it comes at a cost because we are adding a full-fledged virtual machine to our application runtimes.

Our goal is to assign dedicated resources to our applications, while keeping our applications as isolated as possible, and minimizing the additional resources required to achieve this.

Docker provides a solution to this problem with containerization. Docker containers provide a level of application isolation that is similar to VMs, but you don't need a dedicated VM to run your application. A container is more light-weight than a VM because containers share more system resources than VMs. We'll learn how this works in the next chapter.

Let me give you an example of some containers I have on my machine so that you get a better idea. I started up two containers right now:

CONTAINER ID	IMAGE	COMMAND
54ade73c494f	wordpress:4.9.2-php7.2-apache	"docker-entrypoint.sh apache2 &"
d8322333b489	mariadb:10.3.3	"docker-entrypoint.sh mysqld &"

As you can see I have two containers running; one of them is a Wordpress instance with an Apache web server, and the other one is MariaDB.

If you look at the ports, you can see that Wordpress is listening on port 80, which means that this is my active localhost server.

Similarly, MariaDB is listening on the default port (3306). I can connect to it with a MySQL client (like SequelPro) and use it just like I would use any MySQL server.

The Wordpress container comes from the official Wordpress image, which includes a snapshot of the Wordpress file structure, plus all the files of the Apache web server plus dependency libraries and binaries. It contains everything you need to serve Wordpress with Apache. Docker, the Docker Engine to be explicit, makes sure that the Apache processes run isolated from other processes on my machine and the container has the dedicated resources (processor, memory, and storage) to run.

I started the MariaDB container from the official MariaDB image, which contains all files that are needed to run MariaDB.

I do not need to install and run Apache or MySQL on my machine natively. I do my development in Docker containers. I can now replicate or re-create the same environments on another laptop, server or VM in the cloud using a few Docker commands.

The main source of Docker images is the [Docker Hub](#), a cloud registry where you can browse and pull Docker images for various technologies. There are about 1.9 million images on the Docker Hub today.

I pulled the specific version of the Wordpress and MariaDB images that I'm using from the Docker Hub.

A Docker container is a running instance of a Docker image. You'll see that Docker images play a vital part in

your development workflow.

With a few modifications of the official Wordpress image, you can create your project specific Wordpress image with your Wordpress plugins and themes included, and distribute it as one Docker image, code and environment together. You can run your Wordpress application in its self-contained environment (a container) on any machine the same way.

You can use a Docker image for development, and then you can distribute the same image to someone else's computer and start up a container, without worrying about the configuration of the other machine.

Docker images are light-weight, portable, reproducible and declarative.

Project teams usually highlight two main benefits when talking about Docker:

1. Docker solves the problem of “it worked on my machine”. For the reason that you can build your application with the same Docker image on developer laptops/desktops and use the same image on the development and test servers and later in production, you'll face fewer bugs caused by environmental issues and misconfiguration. This will save you and your team time and money by improving the quality of your application.

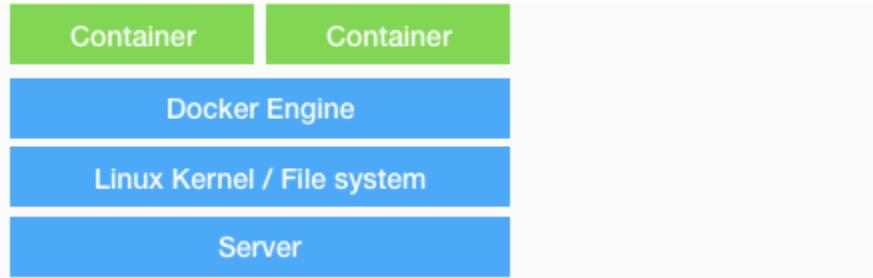
2. You can build an end to end “build, ship and run” workflow for your project with Docker. You can use Docker’s standard configuration files to define your images, and you can define how containers should be started from images, how containers should connect and cooperate, and how they should behave in production. You can define the entire application, environment, and behavior as code, and port it to your infrastructure of choice.

In this book, you’ll learn how to cover the entire “build, ship and run” workflow through examples.

How does Docker work?

Docker was created to manage containers on Linux servers. The base use-case is to enable applications to run in isolated containers, application and its dependencies bundled together, on Linux servers.

The main benefit of running applications in containers on Linux is that while the containers are separated, they share the Linux kernel of the server machine.

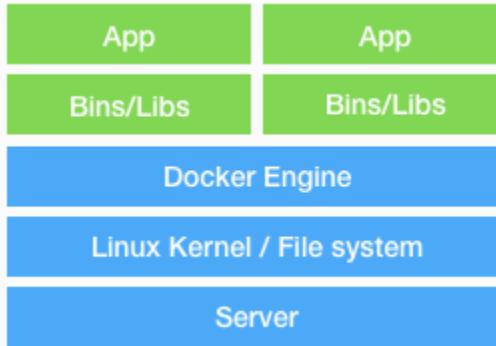


Docker containers aim to maximize resource sharing between containers and enable us to isolate those parts of the applications that are different and need their own space.

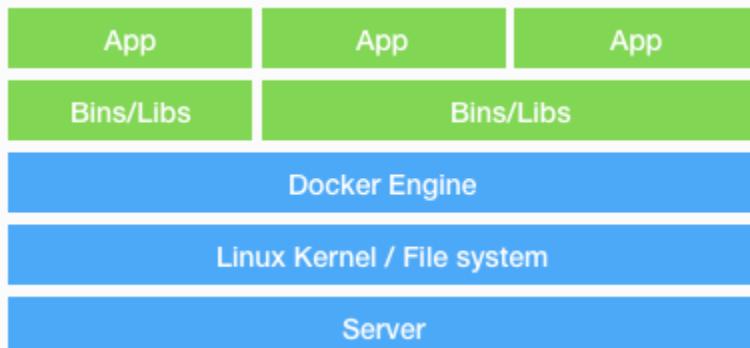
Because of high resource sharing, we can build Docker containers with a small footprint that makes application distribution easier and it makes container startup times faster.

A Docker image and thus the container started from the image, contains the files needed to execute the containerized application on top of the Linux kernel of the host machine it's running on. We call the Linux machine that runs the Docker Engine the host machine.

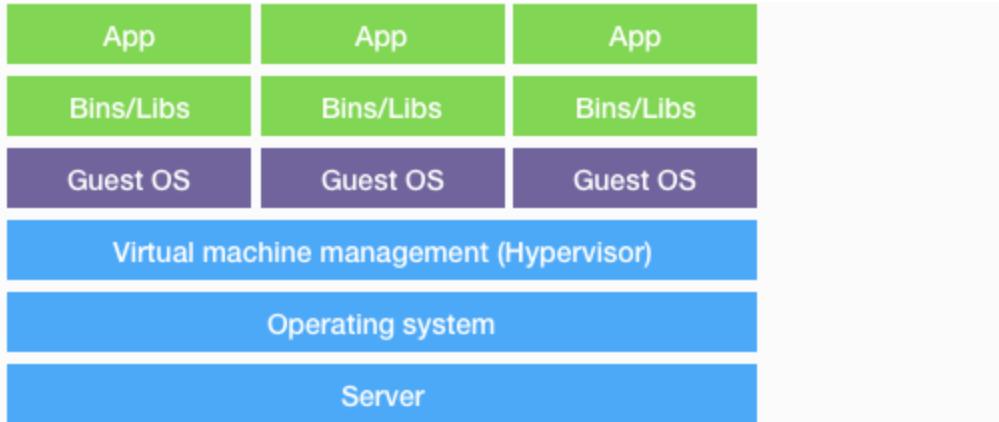
A Docker container contains the required binary files, libraries, configuration files and other application files in the Linux file system hierarchy.



Docker images and thus containers contain applications and their dependencies in a layered file system. This lets containers share their dependencies when appropriate, so we can achieve further resource efficiency on top sharing the host machine's kernel by sharing file system layers.



This is a major step forward compared to the practice of isolating applications with the use of virtual machines. You can achieve the same level of application isolation with virtual machines at a higher cost because you need to run a dedicated guest operating system with every isolated application if you're using virtual machines.



Due to higher resource sharing, containers are more portable and efficient than virtual machines. Due to higher resource sharing, containers need to address security and isolation concerns that are more uncomplicated to manage with virtual machines where the guest operating system is isolated from the host.

Docker containers on non Linux systems

We have seen that Docker containers share the Linux Kernel of the underlying host machine. This means that Docker containers need to run on a host machine that has the Linux Kernel.

Based on what we learned so far, we know that we can containerize applications on Linux servers and develop containers on Linux machines.

Two logical questions arise:

1. Can we build Docker containers for non Linux servers?
2. How can we develop Docker containers on Mac and Windows machines?

Can we build Docker containers for non Linux servers?

Docker started out as a container management solution for Linux systems released in 2013. In 2016 Docker and Microsoft announced that Docker containers are available on Windows servers and can also run natively on top of the Windows server kernel.

Today there are two types of Docker containers running on production servers:

- Linux containers running on Linux servers.
- Windows containers on Windows Server 2016 or Windows 10.

Linux containers and Windows containers are fundamentally different; we use Linux shell commands, binaries, and tools in Linux containers, while we use Windows Powershell commands, Windows binaries and tools in Windows containers. Linux containers run on Linux machines, Windows containers run on Windows machines.

In this book, we'll build Linux containers and will run them on Linux servers. From now on when I mention Docker containers in the book, I mean Docker Linux containers.

The majority of Docker applications run on Linux containers today.

How can we develop Docker containers on Mac and Windows machines?

Linux containers need the Linux kernel to run. This means that you cannot run Linux containers on Windows or Mac natively, because these operating systems do not have a Linux kernel.

You can indeed develop and run Docker containers on both Windows and Mac. When you install the Docker application on Windows or Mac, the installation package contains a minimal Linux virtual machine that will be run by Docker in the background.

The experience is seamless on Windows 10 and newer Macs, the Linux virtual machine runs hidden in the background, and it's fully managed by Docker.

On older Windows and Mac systems the experience is also smooth. You'll need to install a product called Docker Toolbox and use a special Terminal called Docker Quickstart Terminal that's part of the installation.

I would strongly recommend building the book examples on a Linux machine, or a newer Mac.

Install Docker on your machine

Docker is available in two main variants; Docker Community Edition (CE) and Enterprise Edition (EE). The Community Edition has everything to get you started, it's available for free, and is also used in production operations by businesses.

The Enterprise Edition is built on the same code base as CE, and it has some advanced features and enterprise-grade support. We'll use the Community Edition in the book.

Docker has different installation files and methods for different operating systems.

Docker supports 64bit operating systems, and if you have a 32bit system and you cannot upgrade right now, you can look at “building the examples in the cloud” option below.

Let's find out which is the best installation scenario for you using the below list; you'll find installation details later in this chapter.

If you are a Windows user:

- The latest and greatest: Docker has a desktop application for Windows 10, called Docker for Windows. It will work with 64bit Windows 10 Pro, Enterprise and Education (1607 Anniversary Update, Build 14393 or later). You can check the most recent requirements here:
<https://docs.docker.com/docker-for-windows/install/#what-to-know-before-you-install>.
- On other Windows systems, limited to a 64-bit operating system running Windows 7 or higher (this includes Windows 10 Home edition, too), you should use the Docker Toolbox.

If you are on a Mac:

- The latest and greatest: Docker has a MacOS desktop application, called Docker for Mac. It works with OS X El Capitan 10.11 or newer macOS releases running on a 2010 or newer Mac, with Intel's hardware support for MMU virtualization. You can check the most recent requirements here:
<https://docs.docker.com/docker-for-mac/install/#what-to-know-before-you-install>.
- If you are on an older Mac, you should use the Docker Toolbox. Your Mac must be running

macOS 10.8 “Mountain Lion” or newer to run Docker software. macOS Mountain Lion is a 64bit operating system, so you don’t need to worry about 64bit or 32bit if you are on a Mac.

If you are using Linux:

- CentOS, Debian, Fedora, and Ubuntu are the officially supported Linux distributions for Docker CE. You can run Docker on other distributions, too, but I recommend to use one of the below options to avoid any unnecessary headache.
 - CentOS: use a maintained version of CentOS 7.
 - Debian: use the 64bit version of Stretch 9 (stable), Jessie 8 (LTS) or Wheezy 7.7 (LTS).
 - Fedora: use the 64bit version of Fedora 26, 27 or 28.
 - Ubuntu: use the 64bit version of Bionic 18.04 (LTS), Xenial 16.04 (LTS) or Trusty 14.04 (LTS).

Using a virtual machine:

- If you have a strong computer, you can consider the option of building the examples in a virtual machine, like [Ubuntu 16.04.4 LTS Desktop](#) running in [VirtualBox](#). This way you can keep

your Docker playground isolated. (My experience is that this option would require at least 4GB of RAM for the virtual machine.)

Building the examples in the cloud:

- If the above options don't work out for you, you can build the examples on [Play with Docker](#), which is a Linux environment in the cloud, equipped with Docker tools out of the box. This option requires a stable internet connection with adequate bandwidth.
The only downside of Play with Docker is that your session has a 4 hours time limit and your files will be removed when you close your session. You can save your files, and I'll show you how.

I have tested the examples on the following desktop systems:

- MacOS Mojave 10.14.2 with Docker Version 2.0.0.0-mac81 (29211)
- MacOS High Sierra 10.13.3 (17D102) with Docker Version 18.03.1-ce-mac65 (24312)
- Ubuntu 16.04.1 LTS with Docker 18.03.0-ce, Docker Compose 1.20.1 Linux Kernel 4.13.0-37-generic, running in VMWare Fusion 8.5.10

- Tested most of the build chapters with Windows 10 PRO with Docker Version 18.03.0-ce-win59 (16762)

Install Docker for Windows

The Docker for Windows desktop app is the state of the art way of working with Docker on Windows.

Docker for Windows is available on 64bit Windows 10 Pro, Enterprise and Education (1607 Anniversary Update, Build 14393 or later). Docker is constantly evolving, so it is best to check all the prerequisites on <https://docs.docker.com/docker-for-windows/install/#what-to-know-before-you-install>.

The above link has all the necessary information you need, still, let me give you some additional explanation regarding a few points:

- Your version of Windows 10 64bit should have Microsoft Hyper-V in the system out of the box. Hyper-V is a hypervisor, and this means that it is a program that can create and run virtual machines. Hyper-V is required by Docker, to run the small virtual machine to run Linux containers on Windows.
All this is happening inside the Docker for Windows desktop application, and it will remain

hidden from you.

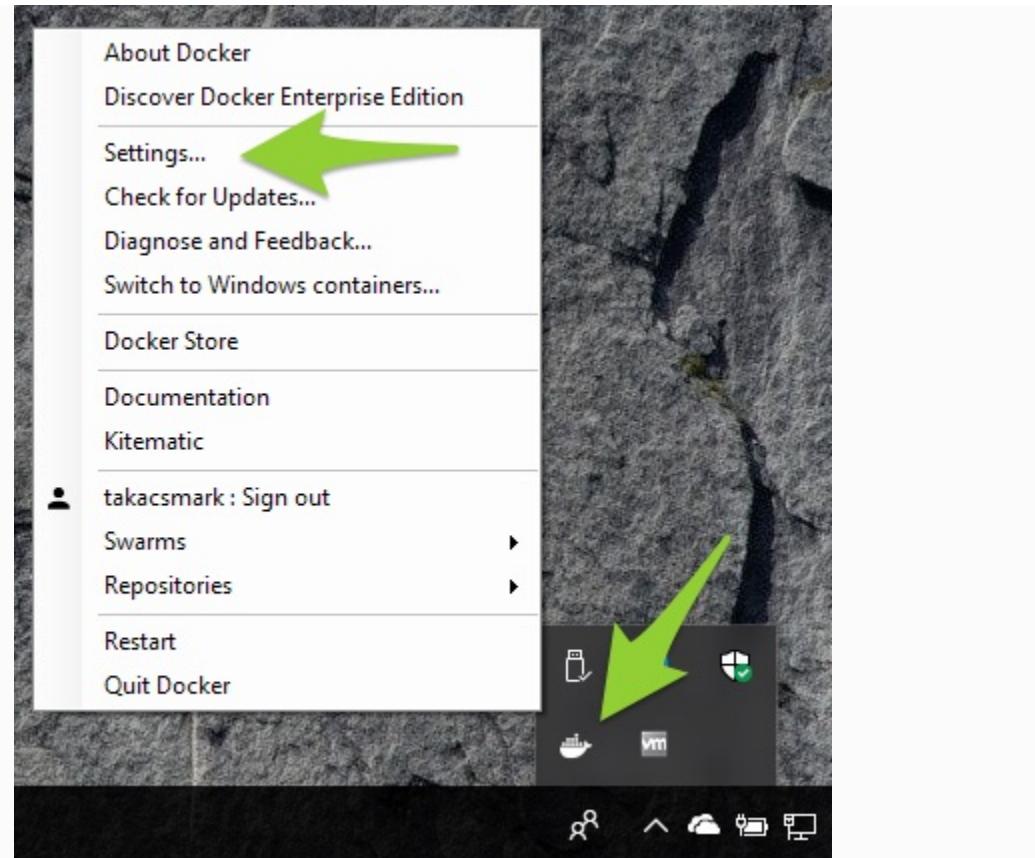
The Docker for Windows installer will enable Hyper-V automatically, if it's not enabled, and will restart your computer. **Just say yes if the installer asks you about this.**

- You must enable virtualization on your computer. This is additional to enabling Hyper-V, please follow the details here
<https://docs.docker.com/docker-for-windows/install/#what-to-know-before-you-install>.

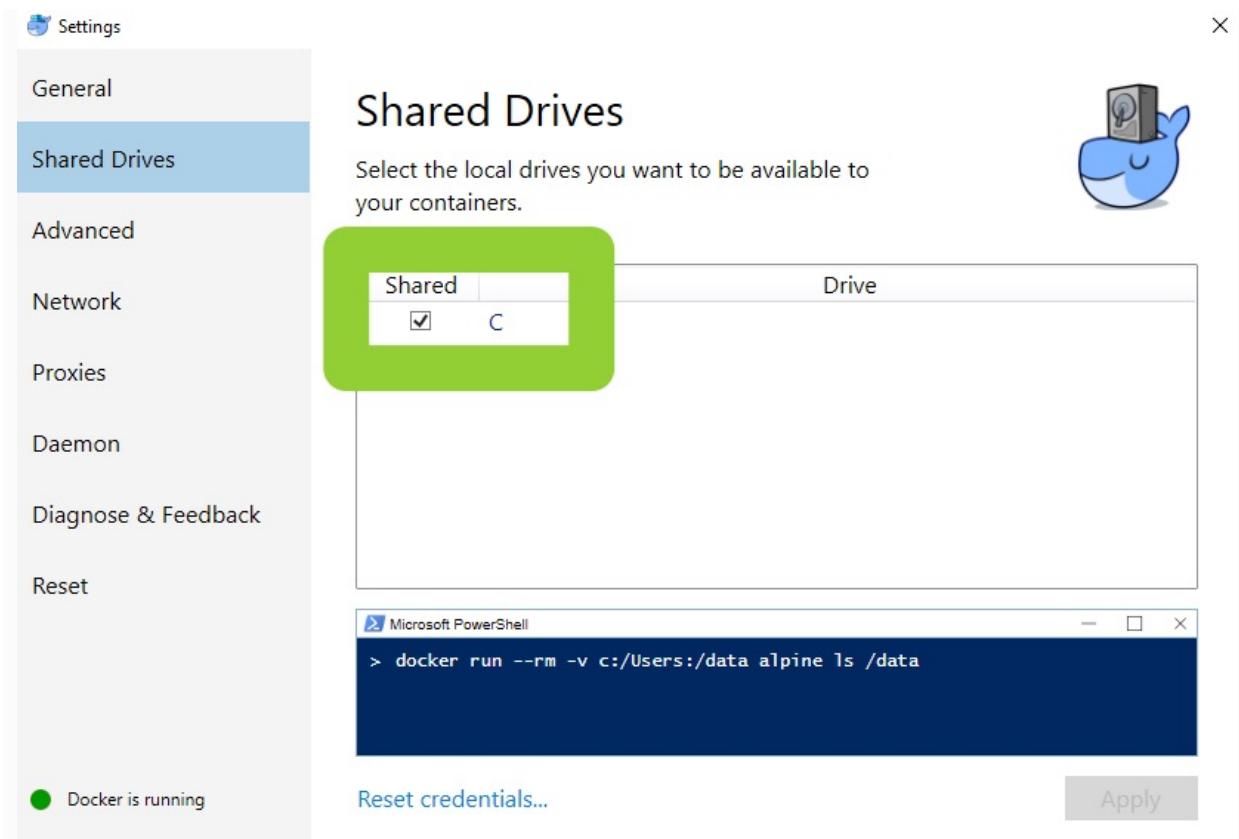
Please download the **stable version** (denoted by “Stable channel”) of Docker for Windows from the Docker Hub (<https://hub.docker.com/editions/community/docker-ce-desktop-windows>).

The download will give you an `exe` file, please double click the installer `exe` and follow the wizard. Start Docker as described on the docker site after installation (<https://docs.docker.com/docker-for-windows/install/#start-docker-for-windows>).

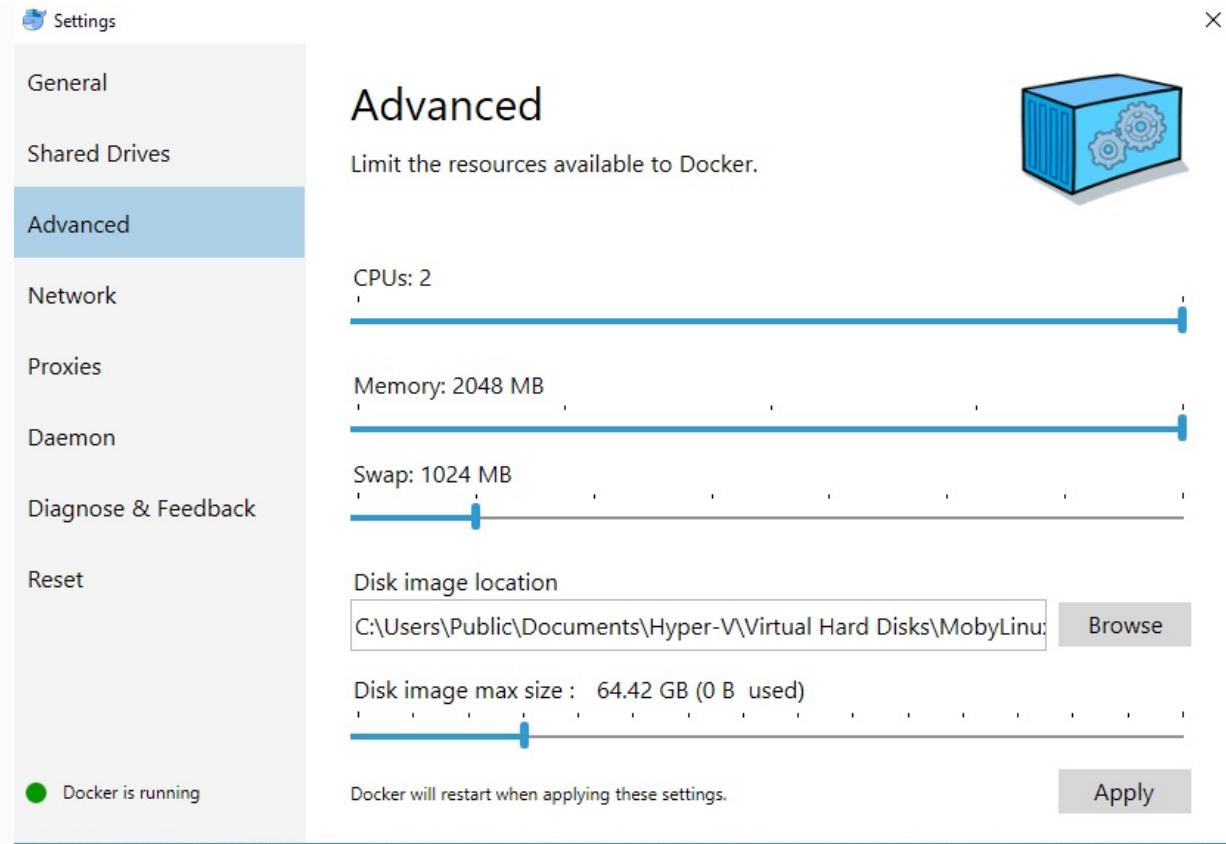
Once you started Docker, let's adjust the settings. Please right click the whale icon in the status bar and open up the settings menu.



Click on the shared drives tab and select the hard drive(s) you'll be working on and click apply. This will let you share files between your machine and your containers.



On the advanced tab, you can fine-tune the resource limitations for Docker. The image shows my defaults settings.



Install Docker Toolbox on Windows

Please follow the Docker documentation here
https://docs.docker.com/toolbox/toolbox_install_windows/; it's informative and pragmatic. Make sure to conduct the steps to verify your installation. You'll need to use the Docker Quickstart Terminal application to build the examples.

Install Docker for Mac

Docker provides a `dmg` installer for Docker for Mac; please download the stable version (denoted by “Stable”) on the Docker Hub:

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

Follow the installation steps on the Docker site; you’ll be fine.

Install Docker Toolbox on Mac

Please follow the Docker documentation here

https://docs.docker.com/toolbox/toolbox_install_mac/; it’s informative and pragmatic. Make sure to conduct the steps to verify your installation. You’ll need to use the Docker Quickstart Terminal application to build the examples.

Install Docker on Linux

Docker provides detailed guides on the Docker site; please follow the one that applies to your distribution. I usually prefer to install Docker using the repository, but you’ll have more alternatives. You can find the guidelines here:

- CentOS:
<https://docs.docker.com/install/linux/docker-ce/centos/>

- Debian:
<https://docs.docker.com/install/linux/docker-ce/debian/>
- Fedora:
<https://docs.docker.com/install/linux/docker-ce/fedora/>
- Ubuntu:
<https://docs.docker.com/install/linux/docker-ce/ubuntu/>

Complete Linux post-installation steps

Once the installation is complete, please make sure to complete the Linux post-installation steps. Make sure to complete the first two steps, namely:

- manage Docker as a non-root user
- configure Docker to start on boot

You can skip the other steps for now. You can find the Linux post-installation guide here:

<https://docs.docker.com/install/linux/linux-postinstall/>.

Install Docker Compose

Windows and Mac installers come with Docker Compose included, on Linux however, you have to install Docker Compose manually. Please follow this guide to install Docker

Compose: <https://docs.docker.com/compose/install/#install-compose>.

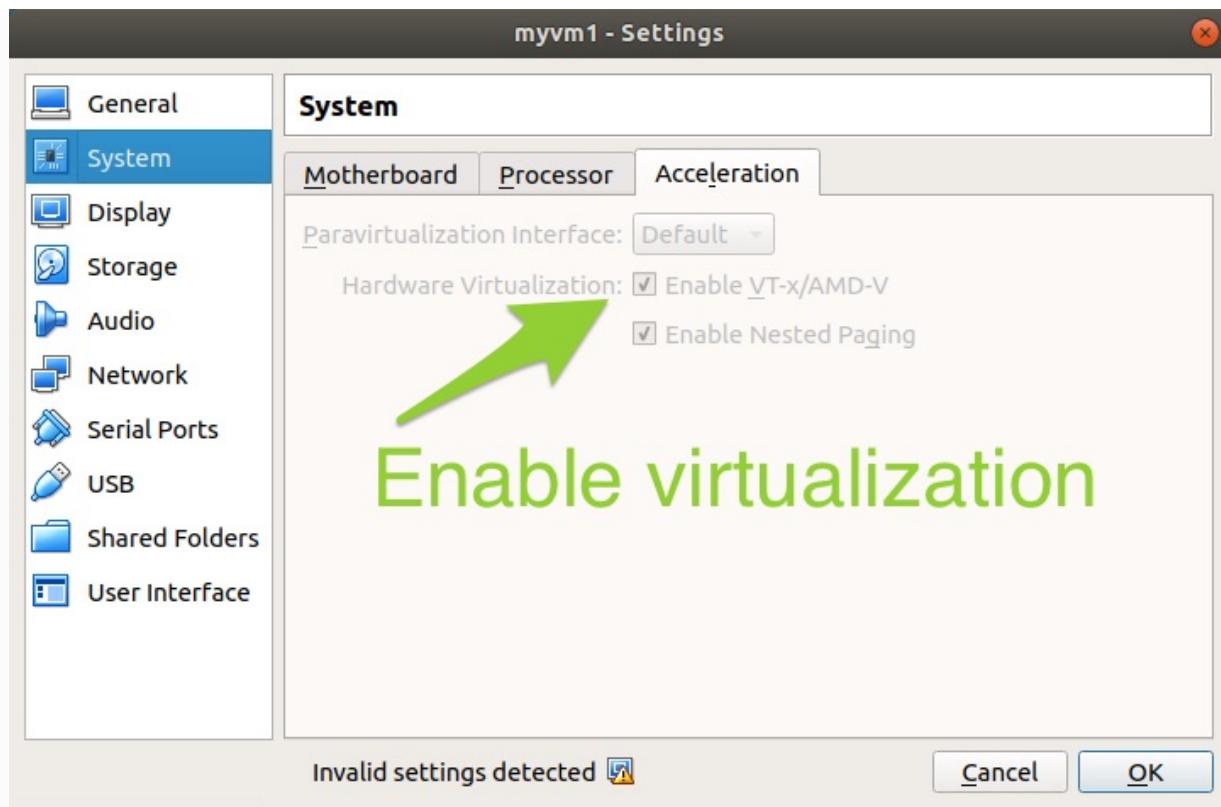
Use Docker in a virtual machine

If your computer has enough power, you can use a virtual machine to build the examples. You can use [VirtualBox](#), a free virtualization tool, or you can use a commercial product like VMWare.

You'll be able to build most of the examples in the book, but you might get difficulties when we run multiple virtual machines to build a cluster. I was able to build all examples in a VMWare Ubuntu virtual machine, even running a cluster in the VM.

If you decide to use a virtual machine, please enable virtualization inside the virtual machine, we'll need this feature later in the book.

You can enable virtualization in VirtualBox in the configuration menu by checking the “Enable VT-x/AMD-V” check box under `System -> Acceleration :`



You can enable virtualization in the VMware Fusion configuration pane under Processors -> Advanced -> Enable hypervisor applications in the virtual machine :

The screenshot shows the 'Processors & Memory' settings for a virtual machine named 'Ubuntu 17.10.1 64-bit'. At the top, there are buttons for 'Show All', 'Ubuntu 17.10.1 64-bit: Processors & Memory', and 'Add Device...'. Below this, the 'Processors' section shows '2 processor cores'. In the 'Memory' section, a slider is set to 4096 MB. Below the slider, resource limits are listed: 4 MB, 1024 MB (recommended), and 8192 MB. A green checkmark icon is next to the recommended value. A large green arrow points from the text 'Enable virtualization' to the checked checkbox for 'Enable hypervisor applications in this virtual machine'. The checkbox is described as enabling support for Intel VT-x/EPT. Another checkbox for 'Enable code profiling applications in this virtual machine' is also present. A yellow warning icon at the bottom indicates that changes cannot be made while the virtual machine is running.

Processors

2 processor cores

Memory

4096 MB

4 MB 1024 MB (recommended) 8192 MB

4096 MB remaining for your Mac

Enable virtualization

Enable hypervisor applications in this virtual machine

Enables running modern virtualization applications by providing support for Intel VT-x/EPT inside this virtual machine.

Enable code profiling applications in this virtual machine

Enables running modern code profiling applications by providing support for CPU performance monitoring counters inside this virtual machine.

The settings on this page cannot be changed until the virtual machine is shut down.

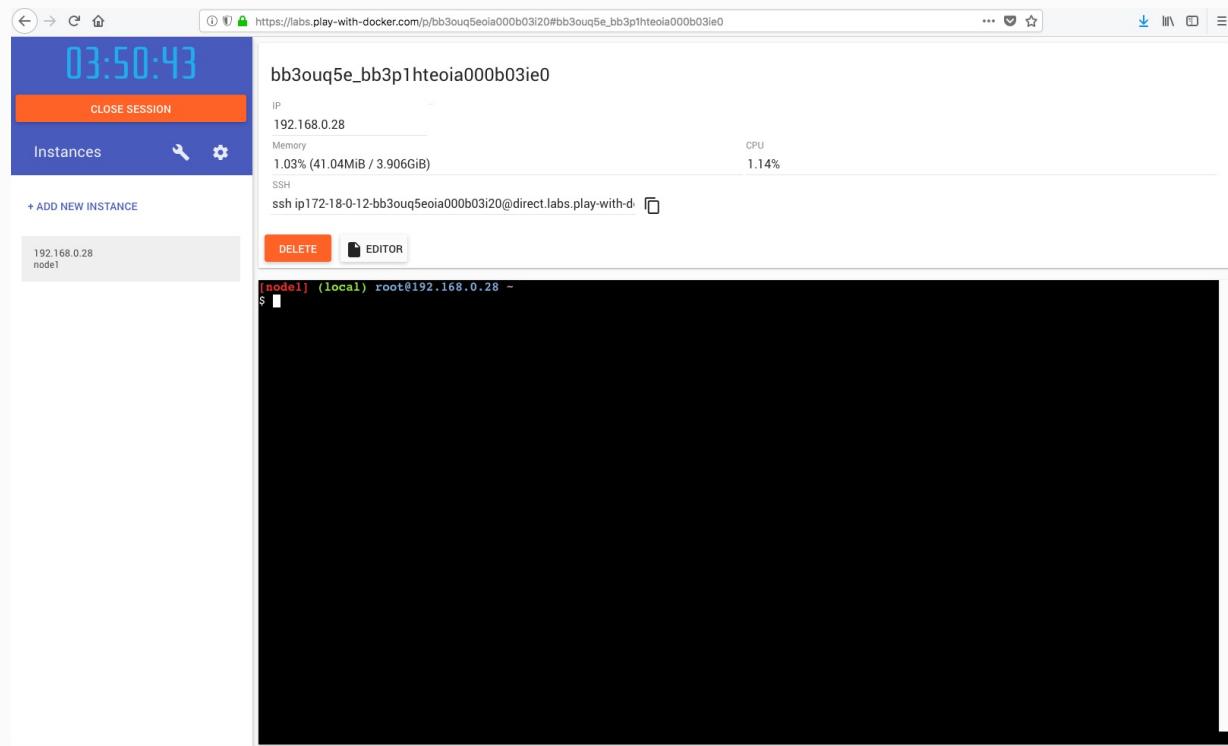
I have limited the resources of my virtual machine to 2 processor cores and 4096MB of RAM.

Build the examples in the cloud

If you decided to explore [Play with Docker](#), please visit the site at <https://labs.play-with-docker.com/>. You can log in with a Docker Hub account, and you'll find a "Create Account" link on the login window, you can use it if you don't have a Docker Hub account.

We'll use a Docker Hub account in the book, so we'll create one anyway, feel free to proceed.

Once you have logged in, you can use the “Add new instance” link on the left side of the screen to start a new machine. You can add multiple machines to build a sandbox cluster. You’ll be presented with a command prompt, and you are ready to go.



Please note that you should not use any personal credentials or secrets on Play with Docker, it's a sandbox environment with no guarantees. Also, note that Play with Docker may become unstable or your session might break unexpectedly.

You can connect to the playground instance with ssh using the connection information provided under “SSH” right above your Play with Docker Terminal.

You can use drag-and-drop or `scp` to copy files from your local machine to the Play with Docker sandbox using the ssh connect string. Below is an example to copy an entire directory on your local machine to Play with Docker, I’m using the connect string in the screenshot in the example:

```
scp -r . ip172-18-0-12-bb3ouq5eoia000b03i20@direct.labs.play-with-docker.com:/var/www/html
```

This way you can easily upload your project files from your local machine and start building.

Verify your installation

Let’s check if your installation is ready to proceed to the first example, please open up a Terminal window and issue the following commands:

```
$ docker --version  
Docker version 18.09.0, build 4d60db4  
  
$ docker-compose --version  
docker-compose version 1.23.2, build 1110ad01
```

You should see Docker version information similar to the above output. These are the versions that I'm using for this book.

Build with containers

Your first container

To build Docker based applications in your projects, you need to master the idea of containers and the basics of container management first. So we'll start our journey here.

Let's set up an Nginx web server as our first container example.

Please open up the Docker Hub, <https://hub.docker.com/>, in your browser. The Docker Hub is a cloud registry where you can search and pull Docker images for your projects. The Docker Hub is a Docker Registry, that is a Docker product to store and serve Docker images.

You can also push your own Docker Images to the Docker Hub and make them available for other members of your team or the general public. You can set up your own Docker Registry for your organization if you prefer. We'll see later in the book how to do this.

It is a crucial part of your project work to find the right Docker Images for your projects on the Docker Hub.

Therefore we'll dedicate a chapter to navigating and understanding Docker repositories on the Docker Hub later.

Let's focus on your first container now. Please go to the search bar on top of the Docker Hub page, enter "nginx" and hit Enter.

You'll get a long list of available Docker Images. Note that the first image on the list is marked as "official image".

Introducing the New Docker Hub: Combining the best of Docker Hub, Cloud and Store. [Learn more](#)

The screenshot shows the Docker Hub interface with a search bar containing 'nginx'. The results page displays five official Docker images for Nginx:

- nginx**: Updated 22 minutes ago. Official build of Nginx. Tags: Container, Linux, IBM Z, PowerPC 64 LE, 386, ARM 64, ARM, x86-64, Application Infrastructure. Metrics: 10M+ Downloads, 10K+ Stars.
- jwilder/nginx-proxy**: By jwilder • Updated 2 months ago. Automated Nginx reverse proxy for docker containers. Tags: Container, Linux, x86-64. Metrics: 10M+ Downloads, 1.5K Stars.
- jrcs/letsencrypt-nginx-proxy-companion**: By jrcs • Updated 14 days ago. LetsEncrypt container to use with nginx as proxy. Tags: Container, Linux, x86-64. Metrics: 10M+ Downloads, 465 Stars.
- centos/nginx-18-centos7**: By centos • Updated a month ago. Platform for running nginx 1.8 or building nginx-based application. Tags: Container, Linux, x86-64. Metrics: 5M+ Downloads, 10 Stars.
- bitnami/nginx**: By bitnami • Updated a day ago. Bitnami nginx Docker Image. Tags: Container, Linux, x86-64. Metrics: 5M+ Downloads, 59 Stars.

On the left sidebar, there are filters for Docker Certified, Images, Categories (Analytics, Application Frameworks, Application Infrastructure, Application Services, Base Images, Databases, DevOps Tools, Featured Images, Messaging Services, Monitoring, Operating Systems, Programming Languages, Security, Storage), Operating Systems (Linux, Windows), and Architectures (ARM, ARM 64, IBM POWER, IBM Z).

Official images are maintained by the community, and Docker has a dedicated team responsible for reviewing and publishing the official images. This team collaborates with software maintainers, experts, and the Docker community. New users are encouraged by Docker to use the official images. You can read more about this topic here: https://docs.docker.com/docker-hub/official_repos/.

The image list displays the number of downloads and the number of stars that images have. “Downloads” denotes how many times an image has been downloaded, “stars” are like “likes” on social media platforms and give you an idea of the quality or feature-richness of the image.

Click on the the official Nginx image and open up the details page. I’ll explain what you need to know about this page later. Let’s download the image now.

You can see a little box in the right half of the page that says “Copy and paste to pull this image”. The Docker command line interface (CLI) provides the `docker pull` command to pull down images from the Docker Hub.

Once we issue the command, Docker will download the image to your computer. To double check what will happen, let’s first list the images that are currently stored on your computer. If you have a new installation, this list should be empty.

Docker CLI provides a management command called `docker image` for image management. To list the images on your computer, let’s issue the `docker image ls` command in Terminal:

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATI
------------	-----	----------	--------

You'll get an empty list with the column headers highlighted like above. This means that you have no images on your machine.

Let's pull the Nginx image, we'll use the Docker Pull Command we've seen on the Docker Hub, issue the following in Terminal:

```
$ docker pull nginx

Using default tag: latest
latest: Pulling from library/nginx
e7bb522d92ff: Pull complete
6edc05228666: Pull complete
cd866a17e81f: Pull complete
Digest: sha256:285b49d42c703fdf257d1e2422765c4ba9d3e37768d6ea83d7
Status: Downloaded newer image for nginx:latest
```

It may take a while depending on your internet speed, but in the end, you should end up with an output that looks like mine, with different identifier numbers.

Let's list docker images on your computer again:

```
$ docker image ls

REPOSITORY      TAG      IMAGE ID      CREATED
nginx           latest   3f8a4339aadd  5 weeks ago
```

The Docker CLI provides the `docker container` management command to manage containers.

Now that you have your first Docker image on your machine, let's start an Nginx container with the `docker container run` command:

```
$ docker container run -p 80:80 -d nginx  
b0aa56c74f22d854498e778e8ca31ee8bfc06ffe616a3110ae9535fe3978e0bc
```

Please make sure that you are not running a server on your machine on port 80. Otherwise, you'll get an error that the port is already allocated. When you see the long number appear, it means that your container is running.

Please open your favorite browser and visit <http://localhost>. You should see a web page that says: "Welcome to nginx!".



You just started an Nginx web server in a container. You did not have to install Nginx on your machine and did not have to configure it to run. It's all encapsulated in the Docker image and the container that you started from it.

Let's analyze the command that we used to start the container:

```
$ docker container run -p 80:80 -d nginx
```

- `docker container run` - You can use this command to start up a new container from a Docker image.
- `-p 80:80` - You can tell Docker with the `-p` option that you want to map a port from the container to the host machine.

This is the setting that made Nginx available on <http://localhost>. Its form is `-p hostPort:containerPort`.

We wanted to access the web server on port 80 on your machine, so we put 80 as the first number. Nginx is listening on port 80 in the container, so we put 80 as the second number. (I'll explain later how we know that Nginx is listening on port 80 in the container.)

Without this mapping, we would not be able to access our web server on the host machine. You'll see that you will use this option a lot.

- `-d` tells docker to run the container in the background so that you'll get your command

prompt back after starting the container. `-d` stands for “detached”.

- `nginx` is the name of the image that you want to use to start the container.

We pulled an Nginx image from the Docker Hub and started a container on your machine. We are running an Nginx web server inside that container. I hope you think it's a great start.

Managing containers

Let's see, how we can manage basic aspects of our containers. We'll do the following tasks:

- list running containers
- look at container logs
- stop containers
- list stopped containers
- start and restart containers
- remove containers
- start a container with a custom name
- display live container resource usage

- display the running processes of a container
- run a container in the foreground

List running containers

You'll see that you'll have a couple of running containers on your machine most of the time. Issue the following command in Terminal to list running containers:

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND
e9fc0f6293bb	nginx	"nginx -g 'daemon off...'"

The result will show the details of the running containers. Let's note the fields in the output:

- CONTAINER ID - Docker gives a unique identifier to your containers, you can use this ID to refer to the container in Docker commands.
- IMAGE - the name of the image that was used to start the container
- COMMAND - the main command of the container
- CREATED - time when the container was created
- STATUS - the status of the container, right now it's "Up", which means it's running. Docker also

informs you about how long the container's been running.

- PORTS - the list of available ports in the container. The value “0.0.0.0:80->80/tcp” means that we have mapped port 80 in the container to port 80 on the host.

The first number denotes the host port. The second number denotes the port in the container.

0.0.0.0 means that your Nginx server is mapped to all network interfaces on your machine.

- NAMES - Docker assigns a unique human-readable name to all containers, you can specify a custom name in the `docker run` command, we'll see how. You can use the CONTAINER ID or the NAME to refer to a container in Docker commands.

There is another shorthand for `docker container ls` ; it's called `docker ps` , it will produce the same results:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
e9fc0f6293bb	nginx	"nginx -g 'daemon off;"

Look at container logs

Once you have a running container, you'll probably want to see what's going on inside the container. In order see the logs use this command (please replace container name with your container name from the output of the previous command):

```
$ docker container logs vigorous_jones
```

You can also use the shorthand version:

```
$ docker logs vigorous_jones
```

Alternatively, you can use the container id like so (again please replace the id with your container id):

```
$ docker container logs e9fc0f6293bb
```

This command will bring up the logs of your Nginx container. Depending on how many times you visited your Nginx server this log may be shorter or longer. Let's learn a few simple commands to enhance your log experience.

The above commands will also work with stopped containers by default. So you can read the logs with the same

commands even if your container has stopped. This is useful to understand why some containers exit unexpectedly.

You can limit the number of lines shown from the end of the log, let's display only the last 10 lines of the log:

```
$ docker container logs --tail 10 vigorous_jones
```

Display the log live as it changes:

```
$ docker container logs --follow vigorous_jones
```

This will display your log and wait for something to happen. If you go to your browser to visit <http://localhost>, you'll see the new items appear in the log live. You can leave the log with `ctrl-c`.

You can also combine the two options:

```
$ docker container logs --follow --tail 10 vigorous_jones
```

You can check out further options with `docker logs --help`.

Stop containers

Containers stay around on the host machine, even if they are stopped. So they can be restarted if you need them.

This is how to stop a container (again please replace the container name with your container name):

```
$ docker container stop vigorous_jones  
vigorous_jones
```

Or use the short version:

```
$ docker stop vigorous_jones  
vigorous_jones
```

List stopped containers

If you try to list the containers on your machine with `docker container ls` after stopping the running containers, you will get an empty list. Your container is still around, but it's not running, and `docker container ls` will display running containers only.

You can list all containers with:

```
$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND
e9fc0f6293bb	nginx	"nginx -g 'daemon off..."

You can also issue this command in the short form:

```
$ docker ps -a
```

Note how the status changed to “Exited” displaying the exit time. Also, note that the human-friendly name is still assigned to the container.

The `-a` option, or the long version `--all`, will expand the list to display all containers.

You can explore more listing options with `docker container ls --help`.

Start and restart containers

Let’s start our nginx container again. Please use your container name.

```
$ docker container start vigorous_jones
```

vigorous_jones

We can formulate the command also the short way:

```
$ docker start vigorous_jones  
vigorous_jones
```

Check the status of your running container again:

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND
e9fc0f6293bb	nginx	"nginx -g 'daemon off...'"

Now you see that I created the container 3 hours ago and started it 57 seconds ago.

Sometimes you need to restart a running container. You could stop it and start again, but Docker provides a dedicated command to restart containers. Let's run the following command in Terminal. Make sure you replace the container name in your command.

```
$ docker container restart vigorous_jones  
vigorous_jones
```

If you run `docker container ls` again, you'll see that the container is Up 11 seconds, which means it was restarted.

For convenience you can use a shorter version of the restart command, too:

```
$ docker restart vigorous_jones  
vigorous_jones
```

Explore restart options with `docker container restart --help`.

Remove containers

You'll want to remove containers for various reasons. The most important being that you need to change the configuration of your container.

This is a vital aspect of containers; you don't fine-tune them as they run, you destroy them when they are not needed and start new containers with improved options. You shall design your containers to be stateless to be able to replace them as needed.

To remove a container, first you need to stop it and then use the `docker container rm` command like this:

```
$ docker container stop vigorous_jones  
vigorous_jones  
  
$ docker container rm vigorous_jones  
vigorous_jones
```

Our container is now removed. You'll get an empty list if you run `docker container ls -a`.

Please note that you can use the short commands to remove containers like this:

```
$ docker stop vigorous_jones  
vigorous_jones  
  
$ docker rm vigorous_jones  
vigorous_jones
```

Note that you can remove running containers with the `-f` or `--force` flag if you need to.

Start a container with a custom name

Let's run an Nginx container, again. Since we just removed the previous container, we need to launch a new one. Let's give it a custom name this time. I'll name the container "docker-tutorial", feel free to use any custom name.

```
$ docker container run --name docker-tutorial -p 80:80 -d nginx
```

Alternatively, you can use the shorter form of:

```
$ docker run --name docker-tutorial -p 80:80 -d nginx
```

Check the list of running containers and see how the container name has been set to "docker-tutorial".

A custom container name may help you if you want to use the container name in a script to automate your workflow.

Display live container resource usage

Let me show you how you can monitor resources used by your running container. Here is the command:

```
$ docker container stats docker-tutorial
```

This command will give you a real time view of resource usage, where you can follow usage statistics. Open your browser, place it next to your Terminal window, and refresh your Nginx page a few times. You'll see the load and net traffic increase as you send requests to your server.

This command has the `-a` or `--all` flag, that lets you monitor all running containers at the same time.

Display the running processes of a container

Sometimes you'll want to look into the running processes of a container. To achieve this, use:

```
$ docker container top docker-tutorial
```

UID	PID	PPID	C
root	9022	9005	0
systemd+	9061	9022	0

You'll get an output similar to mine.

Run a container in the foreground

In the previous example, we have run our containers with the `-d` option. During development time it is useful to run

your container in the foreground and follow the output as it appears in Terminal.

To run your container in the foreground, do not add the `-d` option to your run command.

```
$ docker container run --name docker-tutorial -p 80:80 nginx
```

If you have the previous example still running, please make sure to remove the running container with the same name first.

Start building with containers

In this chapter, we will go through a number of typical development tasks with Docker containers. I'll show you how you can address some crucial challenges with core tools.

Please stop and remove the previous example, with the commands you learned and let's start building.

Mapping directories and files from the host machine to your container

As a first step, I think it would be nice to display a custom web page in our Nginx web server instead of the default “Welcome to nginx!” page.

We will do this by creating a simple HTML page on the host machine and mount it into a container.

Create a directory on your machine for this example. I usually create a “Development” directory in my home directory and create my project directories in the “Development” directory. I’ll call my new directory “example1”.

You can use your file manager or command prompt to do so. My project directory is located at

/home/takacsmark/Development/example1 . (I’m using a Linux desktop machine.)

Make this directory the active directory (`cd` into the project directory) and create a file called `index.html` . Open up your favorite code editor, copy the following simple HTML markup into the file and save it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
    <title>Docker Example</title>
</head>
```

```
<body>
  Hello, this is a Docker example!
</body>

</html>
```

We will serve this page from our Nginx container. To achieve this, we'll bind mount the local project directory on the host machine (in my case `/home/takacsmark/Development/example1`) into the container.

We will use the `-v` or `--volume` option of `docker container run` to bind mount the host directory into the container. This means that the contents of the host directory will be replicated in the container in a way that any changes done on the host machine or the container will immediately reflect on the other side.

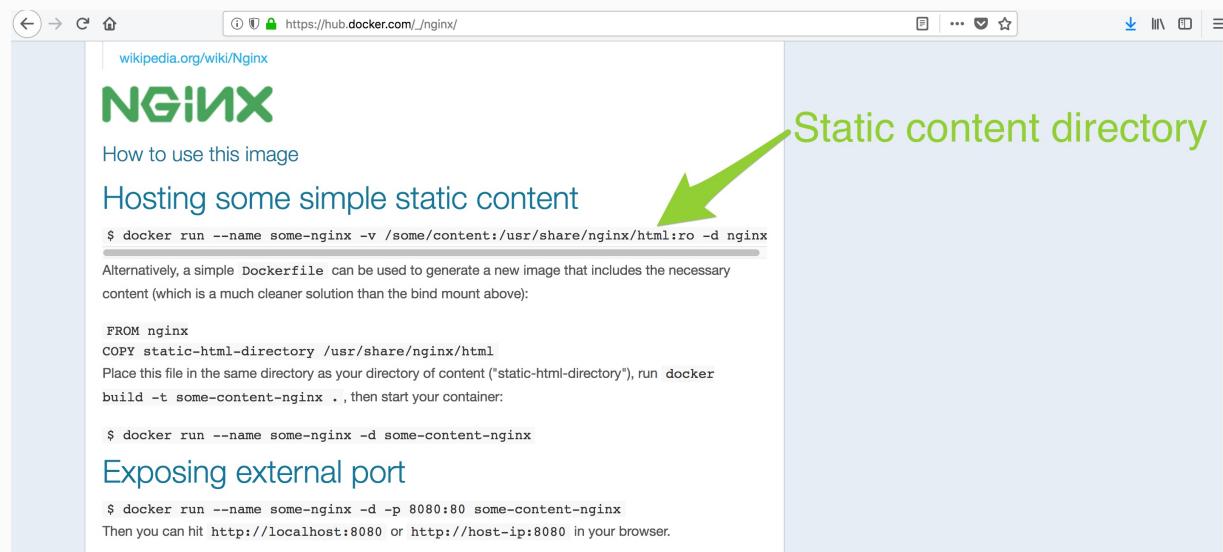
When you bind mount your code into a container, your development changes will be available in the container on the fly. This is how you can use a container for code development.

We'll use `-v` in the form `-v host-directory:container-directory`.

We have to specify the host directory first. We'll use the absolute path of our project directory (in my case `/home/takacsmark/Development/example1`). Please use the `pwd` command to print the current working directory in Terminal.

We also have to specify the directory in the container where we map the files from the host machine. We have to pick the directory that the Nginx container uses to serve web content. We can find this out by looking at the Nginx image documentation on the Docker Hub.

If we go to https://hub.docker.com/_/nginx/, reading the documentation we can find out that this directory is at `/usr/share/nginx/html` in the container. (You can find this around the usage example paragraph on the web page).



I prefer to re-map our Nginx server to port 8000 for development, so we'll apply this change, too.

Our command will be the following (please replace the host path with the absolute path of your project directory):

```
$ docker container run --name docker-example1 \
-v /home/takacsmark/Development/example1:/usr/share/nginx/html \
```

```
-p 8000:80 \
-d nginx
```

```
a518c6745b7825691082cb6819d4f976fe334a73cdf244630ab6e267bcef176c
```

You can create the same command using the results of print working directory, `pwd` so that you don't need to type in the full path with every command:

```
$ docker container run --name docker-example1 \
-v $(pwd):/usr/share/nginx/html -p 8000:80 \
-d nginx
```

If you are on Windows 10, it is better to use the `pwd` form to avoid issues of having spaces or special characters in the directory name. Please use curly braces in PowerShell, the exact form is `-v ${PWD}:/usr/share/nginx/`.

Visit <http://localhost:8000/> and you should see the message “Hello, this is a Docker example!”.

Let's go one step further. Open up `index.html` on the host machine and change the line

```
Hello, this is a Docker example!
```

into

```
Hello, I have changed this line!
```

Save the file and refresh your browser! The changes you made to the HTML file are visible in the container and thus in your browser.

Bind mounting volumes into containers is a common practice. **Bind mounting a directory into your container will create a directory in the container if it did not exist before. If the directory exists in the container, bind mounting will override its contents with the contents of the directory on your host machine.**

You can also bind mount files. You can, for example, override configuration files in the container this way. This is not the best practice, but right now the goal is to understand how bind mounting works, so we'll do it. We'll learn the proper way later when we build custom Docker images.

To build this, we'll use a little trick; we'll execute commands in a running container.

Execute commands inside running containers

Docker has another handy feature that you'll use very often. It is common that you need to see what's going on inside a container. We can, for example, check the contents of our `index.html` file inside the container.

We can do this by running a Linux command in a running container with `docker container exec`. Let's execute the following:

```
$ docker container exec docker-example1 \
cat /usr/share/nginx/html/index.html
```

The following shorter command works, too.

```
$ docker exec docker-example1 \
cat /usr/share/nginx/html/index.html
```

This line executes the Linux command `cat` `/usr/share/nginx/html/index.html` in the running container named “docker-example”. As a result, you'll see the contents of the `index.html` file. This time the file contents are coming from the container.

The Linux command `cat` lets you view the contents of files. As I mentioned in the introduction, we are working with Linux containers that use Linux commands and tools. Therefore we use the Linux command `cat` to view the

contents of a file in the container. Let's look inside a container.

Run an interactive shell in running containers

You can issue the `docker container exec` command in interactive mode, where you launch a shell right in a running container.

The Linux shell is a program that takes commands from you and passes them to the operating system for execution. In our example, we will launch `bash` which is a shell program used on many Linux machines, and we'll use it to look around inside the container.

Issue the following command:

```
$ docker container exec -ti docker-example1 /bin/bash  
root@82c2b2fb8a68:/#
```

You'll get a prompt similar to the above, stating that you are user "root" at ("@") the host called "82c2b2fb8a68". (You'll see your container id here.) Now we are sitting in front of the shell of the running container.

Let's note a few things in the command we used:

- We used the `docker container exec` command with the `-ti` option. This will execute the command in interactive mode (`-i`), where you can provide the input from your Terminal and get the program output on your Terminal (`-t`).
- We run the command `/bin/bash` to start the shell.

Now we can use Linux commands to look around. Let's try a few things, issue the following commands (Please copy the command after the `#` sign):

```
root@82c2b2fb8a68:/# whoami
```

```
root
```

We are logged in as user root. Root is the superuser of the system, so be careful you have the power to change anything in the container right now.

```
root@82c2b2fb8a68:/# pwd
```

```
/
```

The working directory is the root directory of the container.

```
root@82c2b2fb8a68:/# ls
```

```
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root
```

The root directory structure is a Linux root directory structure. Remember that Docker images contain the file system snapshot of the application and its dependencies. This is what you see now.

Let's see what kind of Linux distribution we are using:

```
root@82c2b2fb8a68:/# cat /etc/*-release

PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
NAME="Debian GNU/Linux"
VERSION_ID="9"
VERSION="9 (stretch)"
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

The output says that our container is equipped with Debian programs. To run something Debian specific, let's use Debian's package manager `apt` to list the installed Linux packages in our container.

Let's issue the command:

```
root@82c2b2fb8a68:/# apt list
```

The response will be a long list of installed packages in your container.

We can double check our `index.html` file inside the container, too. Issue the following command

```
root@82c2b2fb8a68:/# cat /usr/share/nginx/html/index.html
```

You should be seeing your HTML code in Terminal right now.

We can also test if bind mounting works both ways. Let's change our html code inside the container and see if the changes are visible on the host machine.

First, we need to install a text editor into the container. We'll install `nano` because it's easier to use for beginners. The following command will install `nano`:

```
root@82c2b2fb8a68:/# apt-get update && apt-get install -y nano
```

Edit the file with the following command:

```
root@82c2b2fb8a68:/# nano /usr/share/nginx/html/index.html
```

Let's change the line in the file

```
Hello, I have changed this line!
```

into

```
Hello, I have changed this line in the container!
```

You can move around in the `nano` editor with the arrow keys, and you can use the `del` key to delete text. Save your changes with `Control + o` and hit `Enter` when asked about the “File Name to Write”. Leave `nano` with the `Control + x` key combination.

Refresh your browser and check out the new message served by Nginx.

Leave the container with exit command like this:

```
root@82c2b2fb8a68:/# exit  
~/Development/example1$
```

You should see the prompt of your host machine now. Let's check the contents of `index.html` on the host machine:

```
~/Development/example1$ cat index.html
```

The changes that you applied to the file in the container are visible on the host machine because bind mounting works both ways.

Mount a configuration file

I promised before that we'll map a configuration file into our Nginx container. I hope now that we know a lot more about our container, and this exercise will be fun to accomplish.

Let's create a new directory inside our project directory for our website and another one for our configuration files. I'll call these directories `html` and `config`. You can do this from a file manager, code editor or the command line. This is how you can do it on Linux and macOS (make sure your working directory is your project directory, I check this with the `pwd` command:

```
~/Development/example1$ pwd  
/home/takacsmark/Development/example1
```

Create the directories:

```
~/Development/example1$ mkdir html config
```

Move `index.html` to the new directory:

```
~/Development/example1$ mv index.html html/
```

My directory structure looks like this right now:

```
~/Development/example1$ tree
.
├── config
└── html
    └── index.html
```

Let's create the Nginx configuration file. We need a sample Nginx configuration file to start from, so we will copy the default configuration file from the running container with the `docker container cp` command. The config file is located at `/etc/nginx/nginx.conf` in the container as described in the documentation on https://hub.docker.com/_/nginx/.

It is common practice to use a container to extract default settings and configuration files to your host machine, change the files and start a new container with the new settings.

The `docker container cp` command copies files from running containers to the host machine and vice versa. Your container must be running, so it's better to open up a new Terminal window, navigate to your project directory and issue the command:

```
~/Development/example1$ docker container cp \
docker-example1:/etc/nginx/nginx.conf ./config/
```

After issuing the above command, my project structure is the following:

```
~/Development/example1$ tree
.
├── config
│   └── nginx.conf
└── html
    └── index.html
```

We will turn on nginx's gzip compression for this example. Edit the config file (`config/nginx.conf`) with your text or code editor.

Go to the line that says

```
#gzip  on;
```

Remove the `#` sign at the beginning of the line. `#` means that this line is commented out, removing `#` means that `gzip on` will take effect once we map this configuration file into the container.

Your line should look like this now:

```
gzip  on;
```

Please save the file and exit the editor.

I'm sure you prefer to double check if we turn on gzip compression in Nginx. We can do this in our browser. I show

you how to do it in Firefox. You can similarly access this information in Chrome DevTools and Internet Explorer developer tools.

Please visit <http://localhost:8000/> in your browser and open up developer tools. You can do this if you right click on the web page and pick the “inspect” menu item from the context menu.

Go to the network tab in your developer tools, reload the page and click on the URL in developer tools. You’ll see response headers appear at the bottom of the page. Here is my screenshot from Firefox:

The screenshot shows the Network tab of the Chrome DevTools Network panel. At the top, there's a status message: "Hello, I have changed this line in the container!". Below it, the URL is listed as "localhost:8000". The Headers tab is selected, showing request and response headers. The Response headers section is expanded, displaying various header fields like Accept-Ranges, Connection, Content-Length, Content-Type, Date, ETag, Last-Modified, and Server. The status code 200 OK is also visible.

URL

Network tab

Headers tab

Response headers

Right now our response headers do not contain compression information, because gzip is not turned on. We'll change this now.

Please stop and remove the container that we've been working with so far. Use the commands we learned in the previous chapters. We'll run a new container now with the following command:

```
~/Development/example1$ docker container run \
--name docker-example1 \
```

```
-v $(pwd)/html:/usr/share/nginx/html \
-v $(pwd)/config/nginx.conf:/etc/nginx/nginx.conf \
-p 8000:80 -d nginx

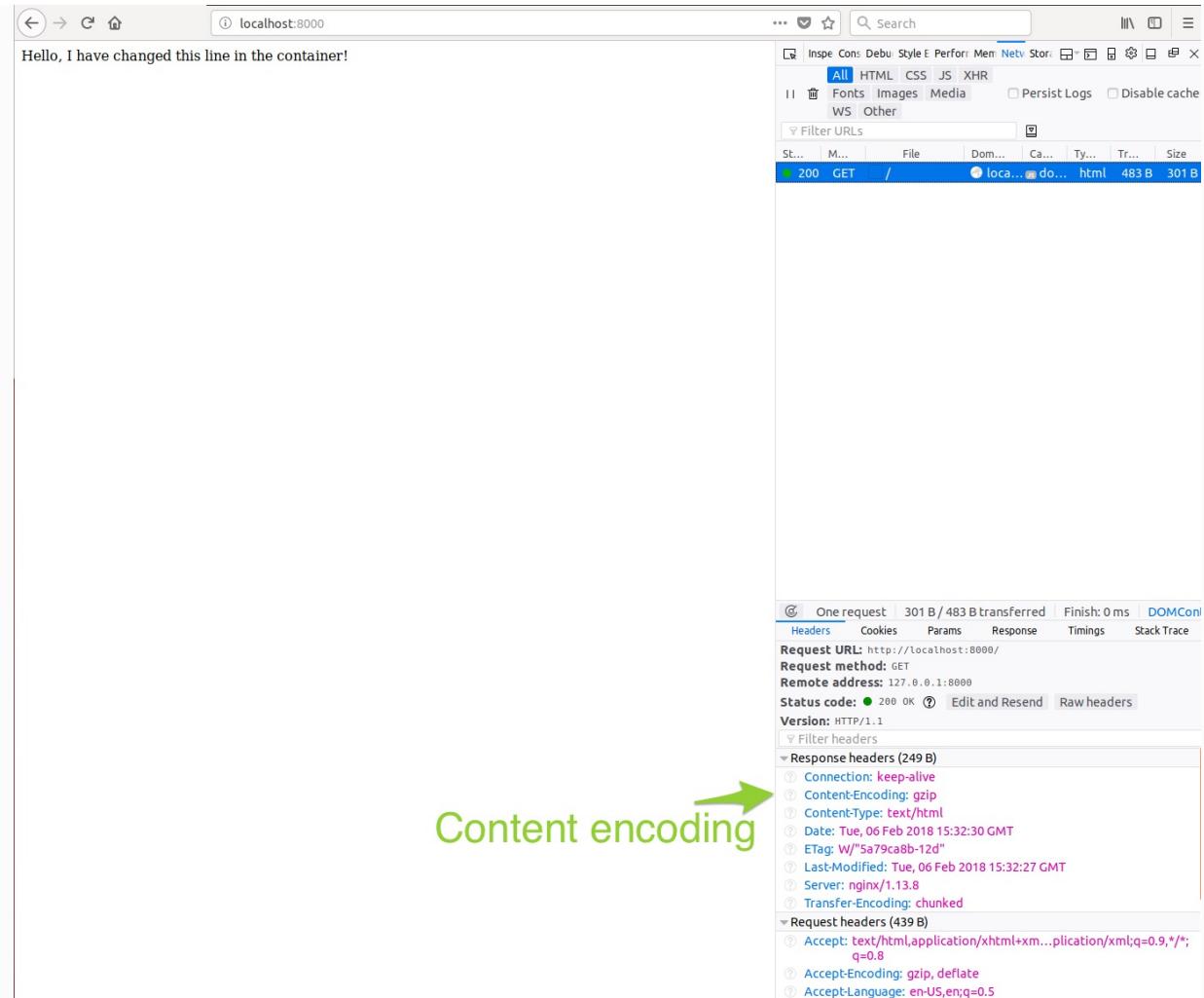
3cb8f23ba808bf8472698a20fcabfb8396d52914e6119e1b2cfa2fb6e77d1b6
```

Please note how I updated the host path of the HTML code that we are mounting into the container to reflect the new project structure, i.e., we moved `index.html` under the `html` directory.

The main change is that we added another bind mount to the command in the form of another `-v` option. We mapped our `nginx.conf` file on the host machine, to be the `nginx.conf` file in the container.

Remember, if the file already exists in the container, bind mounting will override the container file with the file from the host machine.

Let's visit <http://localhost:8000/> in our browser and check response headers again. Please clear browser cache, if you don't see the changes immediately. Mine looks like this:



As you can see, the following header appears in my response headers: `Content-Encoding: gzip`.

Congratulations, you just mapped your source code and your custom configuration file into a Docker container!

Build images

Identifying the right Docker images for your project

We used containers as the first step. We have learned how to pull and run application containers on your machine with Docker. We created an Nginx web server and served a simple HTML page that you created on your machine.

We also learned how to modify code on your machine and see the changes on the web page served by your Nginx container. We also modified the configuration file in the Nginx container. We used bind mounted volumes to map code files and the configuration file into the Nginx Docker container.

Most importantly, we have learned that Docker containers are based on the Linux operating system. Every container is an isolated Linux environment with its isolated settings, file system, libraries, system and user programs. (For the sake of completeness let me repeat that by now there are Windows-based containers, too, but let's leave this topic for another book.)

We have seen in the previous chapter that we can start up Docker containers from Docker images. A Docker image contains all files (configuration files, executables, programs, libraries and settings) that are required to run a certain application in a container. A Docker image contains all of the above in the form of files in a Linux file structure.

If you want to start a container, all you need to do is to pull the right image from the Docker Hub and issue the `docker container run ...` command.

When you work with Docker in your projects, it is an essential skill to identify the right Docker image on the Docker Hub for your project. Let's experiment with a few images to get used to the ways of the Docker Hub.

It happens quite often that I want to try Python code snippets to test ideas. Let's set up a container to run Python code on your machine. In this exercise, we'll work on the Docker Hub and see how to identify and run the right Python image for the task. We'll take the following steps:

1. Create your Docker ID to be used on the Docker Hub
2. Find and choose the right image to be used for this task
3. Pull the image to your local computer
4. Start the container with the right configuration

Create your Docker ID

As I said, Docker Images are stored in Docker Registries. The Docker Hub is a public registry, meaning there is a huge selection of images available for free for anyone on the Docker Hub.

Images are stored in repositories on the Docker Hub. Publicly available images are stored in public repositories, while you can also have private repositories where you can restrict the access according to your needs. Right now you can have one private repository for free.

A repository is the home of a Docker image and all of its versions. We will have a look at this in a minute.

You have the option to register yourself as a user of the Docker Hub by signing up for a Docker ID. As you have seen in the previous exercises, you don't need to be registered to pull public images from the Docker Hub, but you'll need to register to be able to push your images later.

You can use your Docker ID all across the Docker ecosystem, so it's time to become an official member of the community.

Please visit <https://hub.docker.com/> and use the form on the left side of the page to create your Docker ID.

The screenshot shows the Docker Hub homepage. At the top, there's a navigation bar with icons for back, forward, search, and user profile. The URL https://hub.docker.com is displayed. Below the bar, a banner reads "Introducing the New Docker Hub: Combining the best of Docker Hub, Cloud and Store. [Learn more](#)". The main heading "Build and Ship any Application Anywhere" is prominently displayed, with a subtext: "Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications." Below this are two buttons: "Sign up for Docker Hub" and "Browse Popular Images". A large green arrow points from the "Sign up for Docker Hub" button towards a "Sign up here!" call-to-action. The background features a blue and white 3D geometric graphic. At the bottom, a section states: "Docker Hub is the world's largest library and community for container images. Browse over 100,000 container images from software vendors, open-source projects, and the community."

Find and choose images on the Docker Hub

This step is less techie, but I think this is the most important step for you to learn now.

Please click on the “Explore” menu item on the Docker Hub. It’s the first item on the top menu bar , this is the place to explore public repositories.

I would like to add a side note here; if you look closer, you'll see a dark-blue bar at the top of the page with this message: "Introducing the New Docker Hub: Combining the best of Docker Hub, Cloud and Store. Learn more".

This means that Docker used to have three web pages for their cloud services, namely the Docker Hub, the Docker Store and Docker Cloud. Now they are all combined into the Docker Hub, so if you see references to the Docker Store or Docker Cloud online, you should know that these entities are not in function any more, your single point of contact is the Docker Hub.

Let's stay with our task and see how to pick the Python image from the official repositories list.

What are official repositories? Official repositories are maintained by community members on GitHub. You can also contribute to official repositories on GitHub here:
<https://github.com/docker-library/official-images>.

Official images are special because even though they are maintained by the community they follow very strict rules and according to the information from Docker, Docker sponsors a dedicated team to review the official images (as you can read here https://docs.docker.com/docker-hub/official_repos/).

Official images are a great source for learning. You can find Docker image creation and security best practices in the

official repositories. I recommend that you read official images. The point of the activity we are doing now is exactly this.

Take some time and have a look at the list of official images. Once you are ready to move on, use the search bar to find the Python repository. (Please type “Python” into the search bar at the top of the page.)

The screenshot shows the Docker Hub search results for "Python". A green arrow points to the search bar at the top left, which contains the text "Python". Another green arrow points to the first item in the search results, which is the "python" official image. The results page includes filters for Docker Certified, Images, Categories, and more. The "python" image card shows it was updated an hour ago, is an official image with 10M+ downloads and 3.8K stars, and is a Container image for Linux, Windows, x86-64, IBM Z, PowerPC 64 LE, 386, ARM 64, and ARM. It is categorized under Programming Languages. Below it are cards for "pypy" and "django", both also official images.

The first item on the list is the official repository, click the box to reach the page of the repository. Official repositories are usually well documented on the Docker Hub. They give the information that you need to get started, the following information is usually included.

Tags

The available versions of the image are marked with tags. In case of Python, you can see many-many tags. This has the following meaning:

The image itself is called “python”, you’ll use this name in commands. It is indicated in the page title, plus to be explicit you can check the “Copy and paste to pull this image” box on the right that we used in case of our Nginx container. On the Python page it says “docker pull python”, so we know that the name of the image must be “python”.

Tags indicate different versions of the image. We can refer to a specific version of an image using `:` after the image name like this: `python: 3.6.4-jessie` , `python: 2.7.14-alpine3.7` or `python:latest` , to give a few examples.

Versions depend on the technology used, they usually indicate the version number of the specific technology, plus they provide details about technical aspects of the image.

In case of Python, we see that tags start with numbers like `3.7.xxx` , `3.6.4.xx` , `3.5.xxx` , `2.7.14.xxx` , these numbers refer to the Python version included in the image. Python has two major versions existing in parallel; Python2 and Python3, and you’ll find Docker images featuring one specific minor version per image.

The other parts of tags give information about the image. This needs a bit of experience with Linux systems. In case of Python, we see words like `stretch`, `jessie`, `wheezy` and `alpine`. This refers to the Linux environment that the image provides. Our Nginx container in the previous chapter was based on Debian Linux, which means that you have access to a set of Debian programs in that image.

In case of Python `stretch`, `jessie` and `wheezy` refer to different versions of Debian. (Stretch refers to Debian 9, Jessie to Debian 8 and Wheezy to Debian 7, you can see Debian releases here <https://www.debian.org/releases/>.)

The word `alpine` refers to a Linux distribution called Alpine Linux (<https://alpinelinux.org/>). The definition on the Alpine website reads: “Alpine Linux is a security-oriented, lightweight Linux distribution based on musl libc and busybox.”

Most official Docker images have a version based on Alpine Linux now. The reason for this is that Alpine Linux has a very small footprint. The Alpine Docker image takes about 4MB space on your server, while Debian takes 100MB.

We’ll see later that it is essential to keep Docker image sizes as small as possible for portability and deployability.

In some tags, you’ll find that there are dedicated image versions built with different Alpine versions (like `2.7.14-`

`alpine3.7` , `2.7.14-alpine3.6` , `2.7.14-alpine3.4`) so that you'll have a very sophisticated selection of image variants for your work.

This abundance of options might be scary at first, but don't worry. You don't need to know much of these details for learning. Later, when you work on a project, you'll need to dedicate some time to find the right version to reach your project goals. Plan carefully and plan accordingly.

If you look at the tags, you'll see, that some images have additional information in between the Python version info and the Linux distro info. The words, "slim", "alpine" and "onbuild" indicate image variants. There is a dedicated section for image variants on the Python repository page you are looking at in your browser; we'll talk about these in a minute.

You can also see that some images are based on Windows.

Every Docker repository has the `latest` tag, which indicates the latest version of the image. If you don't specify any version in your `docker pull` command (i.e., `docker pull python`), Docker will pull the latest image for you.

Pulling latest is not best practice, because the latest tag may move as time passes. When I first published the book the latest tag was set to `3.6.4` in the Python repository.

Today when I'm updating the book for 2019, the latest tag is set to `3.7.2`. (You can check this if you search for the word "latest" on the page.) The `latest` tag is moving, so it's better to use an explicit version in your projects (like `3.7.2`) rather than `latest`.

It is always best practice to use a specific image version from the Docker Hub.

For the current activity we will use `3.6.4`, but let's talk about the rest of the repository page a little more.

Quick reference

The quick reference section of the page contains useful info and links for the given technology. I usually use the "Where to get help" and "Where to file issues" flags to check the documentation and browse or create issues when necessary.

These two sections mostly list links on StackOverflow and GitHub, so there is no big miracle here. You can turn to these sources and Google any time you are in need of help.

Every repository has a few paragraphs describing what the given technology is about. This is useful when you are in the process of choosing a solution for a unique, niche problem.

The part called "**How to use this image**" is the one that I use the most. Maintainers usually give examples and guidelines on how to use different images. In case of

Python, you see a Dockerfile example; this will play a very important role later in this book.

One of the usage examples for Python is as follows:

```
docker run -it --rm --name my-running-script -v "$PWD":/usr/src/myapp -w /usr/src/myapp python:3 python your-daemon-or-script.py .
```

This line shows us how to run a single Python script with Python 3. We can derive a very important piece of information from this example. Namely, it's common practice to place your app folder in the container under `/usr/src/myapp`, so we'll map our source folder here in the next example.

Image variants

The image variants section gives you a practical explanation of the different variants of the image you can find in this repository. As we've seen above, when we discussed tags, the repository has different versions of the Python image included.

Some part of the tags will describe the variants that were defined by the community that created and maintains the repository. In case of Python we see the following variants (please note that these variants are incorporated into the image tag as described before):

- `python:<version>` - we can find crucial information here to choose the right image for our work. The

tags in the form of `python:<version>` are based on Debian Linux and have several packages included. The idea here is that you won't need to install too many additional packages to run your applications with this image variant.

- `python:<version>-slim` - the slim variants are also based on Debian, but they contain only the minimum set of packages that are required to run Python.
- `python:<version>-alpine` - is based on Alpine Linux rather than Debian, resulting in a much smaller image. It's best to use this variant to minimize image size.
- `python:<version>-windowsservercore` - these images are based on Windows Server Core.

I usually read the variants section first, before diving into the tag jungle at the top of the page.

License

This is the part developers tend to ignore most because licenses are not technical. It is, however, a must to read the license part, especially if you want to use the image for commercial projects.

Considering all of the above information, it's time we choose the image we will use for this example.

I would avoid Alpine or slim variants for now, so that we will have all the packages we might probably need; therefore we'll pick a Debian variant.

Our chosen version is the current latest, which at the time of writing was [3.6.4](#).

Pull the image to your local computer

Once you know which image you are going to use, you can pull that image to your computer by adding the relevant tag to the name of the package, in our case your command will look like this:

```
$ docker pull python:3.6.4

3.6.4: Pulling from library/python
f49cf87b52c1: Pull complete
7b491c575b06: Pull complete
b313b08bab3b: Pull complete
51d6678c3f0e: Pull complete
09f35bd58db2: Pull complete
0f9de702e222: Pull complete
73911d37fcde: Pull complete
99a87e214c92: Pull complete
Digest: sha256:98149ed5f37f48ea3fad26ae6c0042dd2b08228d58edc95ef0
Status: Downloaded newer image for python:3.6.4
```

Pulling this image will take a little longer due to its size. Distribution is an important constraint of Docker images. That's why so many minimalist variants are created by image maintainers.

You have pulled the Python 3.6.4 image to your computer, and now you should have two Docker images on your machine. Let's check the list of images with the following command:

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATI
nginx	latest	3f8a4339aadd	6 wee
python	3.6.4	c1e459c00dc3	7 wee

We pulled 692MB from the Docker Hub.

Start the container with the right configuration

Please create a directory called `example2` in your development directory, `cd` into the `example2` directory and create a Python file called `app.py`. Your directory should look like this now:

```
~/Development/example2$ tree
```

```
.
```

```
└── app.py
```

```
0 directories, 1 file
```

Open your code editor and copy the following line into `app.py`:

```
print("Hello world")
```

Let's use the Python image now to run this small piece of code. We'll use Python 3.6.4 in the Python image to run our Python script.

We will base our command on the sample usage command from the Docker Hub. This is our base that we will modify:

```
docker run -it --rm --name my-running-script -v  
"$PWD":/usr/src/myapp -w /usr/src/myapp python:3 python your-  
daemon-or-script.py .
```

Most of the options on the `docker run` command are familiar already. Let's note a few peculiarities:

- `--rm` means that Docker will automatically remove the container after the script execution is complete. This way you will not have your unused container still around on your machine.
- `-w` sets the working directory in the container. As you see, we are mapping the current directory

to `/usr/src/myapp` into the container with the `-v` flag. We will make sure that the container's working directory will be the same directory (`/usr/src/myapp`) so that our script will be found and executed.

- “`$PWD`” means that we bind mount the current working directory into the container
- the example uses the `python:3` image, we'll change this to the version that we pulled
- we'll replace the command `python your-daemon-or-script.py` with `python app.py` to run our script

Our command looks like this, please execute it in Terminal:

```
~/Development/example2$ docker run -ti --rm \
--name python-example \
-v $(pwd):/usr/src/myapp \
-w /usr/src/myapp python:3.6.4 python app.py

Hello world
```

You should see “Hello world” as the output of the script.

Since we run this container in interactive mode (with `-ti`), we could add some interactivity to our script, let's modify `app.py` to look like this:

```
name = input("Enter your name: ")  
print("Hello ", name)
```

When you run this script with the same command as above, you should be asked for your name. Enter your name hit enter and you'll get a greeting message:

```
~/Development/example2$ docker run -ti --rm \  
--name python-example \  
-v $(pwd):/usr/src/myapp \  
-w /usr/src/myapp python:3.6.4 python app.py  
  
Enter your name: Mark  
Hello Mark
```

Managing images

We have already seen that we can manage containers and images with the `docker container` and `docker image` management commands respectively.

Management commands provide help information if you just put them on the command line and hit Enter. Let's issue this command:

```
$ docker image  
  
Usage: docker image COMMAND
```

```
Manage images
```

```
Options:
```

```
Commands:
```

build	Build an image from a Dockerfile
history	Show the history of an image
import	Import the contents from a tarball to create a file
inspect	Display detailed information on one or more images
load	Load an image from a tar archive or STDIN
ls	List images
prune	Remove unused images
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rm	Remove one or more images
save	Save one or more images to a tar archive (streamed +
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

```
Run 'docker image COMMAND --help' for more information on a command
```

This way you can get the list of further commands that can be used with `docker image` or in a similar way you can do this with all the management commands of the Docker CLI.

You can get even more help in the form of `docker image COMMAND --help`, for example, you can get help on the `ls` command by issuing `docker image ls --help` from the command line.

At this stage we will have a look at the following basic management commands from this list:

- Pull images
- List images
- Remove images
- Inspect images

Pull images

We have already pulled two images from the Docker Hub with the `docker image pull` command.

The command `docker pull` is an alias of `docker image pull` which does the same thing. `docker image pull` pulls an image from a Docker registry.

So far we have used the Docker Hub as our registry, we will set up our own registry in Chapter “Create your own Docker Registry”.

List images

Once you start your Docker projects, you’ll see a growing number of images on your machine. Listing images on your computer can be done with the command `docker image ls`.

`docker image ls` provides options for filtering images or listing image ids only. An exciting option is the `--all` or `-a` option

that lists all images on your computer including intermediary images. We'll learn about this later.

Remove images

You can remove images from your machine with the `docker image rm` command. You can use the image name or image id as an argument to this command.

The command `docker rmi` is a short alias for `docker image rm`, I prefer the longer form, because it is consistent with Docker's latest CLI logic.

Docker images take up space on your computer. It's a good idea to clean up the images that you don't use any more. You can remove an image once you stopped and removed all containers that used that image.

Inspect images

You can use the `docker image inspect` command to get detailed information about Docker images. Try and execute the below command in Terminal:

```
$ docker inspect nginx:latest
```

This command gives you information about the configuration of a Docker Image, including environment variables, exposed ports default image command, working directory to name the most critical ones.

You can use the `-f` option to filter the output of `docker image inspect`, this is useful if you want to use some of the values in a script. You can use Go templates (<https://golang.org/pkg/text/template/>) to filter the output.

You can, for example, do something like this:

```
$ docker inspect nginx:latest -f "{{ .Config.Env }}"
[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin]
```

Build images for your projects

It is time to learn another crucial concept of Docker that will change the way you use and plan with Docker for your future projects.

You have now pulled a couple of images from the Docker Hub and started containers using those images.

We have learned that containers are isolated, self-contained and portable run-time environments, mostly based on the

Linux operating system.

We have also learned how to browse and select images on the Docker Hub and how to use them to run our program code or how you can use them to start up a web server, for example.

You have also seen the base commands that you can use to start, stop or remove containers, follow logs or check resource usage. We have also seen how to list and remove images.

All the exercises we went through so far in the book were built around Docker images. Images contain all files, settings, and configuration required to run a particular application.

A very logical question at this stage is to find out where images come from.

A Docker image is defined in a file called Dockerfile. A Dockerfile defines the steps that need to be executed to build a Docker image. A Docker image is built with the `docker image build` command from a Dockerfile.

This means that maintainers of Docker repositories define their images in Dockerfiles. Our Python image was defined in a Dockerfile, just like our Nginx image was defined in another Dockerfile.

A Dockerfile is a step by step definition of building up a Docker image. The Dockerfile contains a list of instructions that Docker will execute when you issue the `docker image build` command. Your workflow is like this:

1. You create the Dockerfile and define the steps that build up your image.
2. You issue the `docker image build` command which will build a Docker image from your Dockerfile.
3. Now you can use this image to start containers with the `docker container run` command.

You usually start searching for available Docker images on the [Docker Hub](#). You'll also find images on GitHub included with a good number of repos (in the form of a Dockerfile), or you can share Docker images with your team or company by creating your Docker Registry (we'll do this later).

The definition also implies that the images that you'll find on the Docker Hub are defined in Dockerfiles. Technology vendors and developers usually provide one or more Dockerfile(s) with their specific technologies. They define the steps of building the image in the Dockerfile, and they use `docker image build` to create the Docker image.

What happens if you can't find the exact Docker image that you need for your project? Most images are generic and won't cover your exact case. **This is the point when you'll create your own Docker Images.**

When you'll be working on your projects in real life situations, you'll almost every time need a Docker image that addresses exactly your project's needs. You'll create your custom Dockerfiles for those projects and build your custom images.

All you need to do is to create a text file named `Dockerfile` (with no extension) and define your image build steps inside this file.

Even small projects usually involve more than one custom images, thus more than one Dockerfiles. We'll see later what are the best practices to structure a project like this.

Dockerfile basics

Let's check out an example and see what's in a Dockerfile. I chose the below Dockerfile from a later example in the book. Please browse through the file; it's enough for now if you note the critical structural elements, skim through the comments and see how the file is built up.

```
FROM python:3.6.4-alpine3.7

ARG app_name

RUN apk add --update --no-cache nodejs postgresql-dev gcc python3

WORKDIR /root/
COPY $app_name $app_name
```

```
WORKDIR /root/$app_name
RUN pip install -r requirements/dev.txt
RUN npm install

ENV FLASK_APP=autoapp.py
ENV FLASK_DEBUG=1

CMD npm start
```

At this point it's essential to understand the following key points based on the example file:

1. The Dockerfile is a text file that (mostly) contains the instructions that you would execute on the command line to create an image.
2. A Dockerfile is a step by step set of instructions.
3. Docker provides a set of standard instructions to be used in the Dockerfile, like `FROM`, `ADD`, `COPY`, `RUN`, `ENV`, `EXPOSE`, `CMD` to name the critical ones.
4. Docker will build a Docker image automatically (after you issue the `docker image build` command) by reading these instructions from the Dockerfile.

So from a developer, or tech user perspective, you'll be describing the build steps of your environment in the Dockerfile. Then you'll build your image from the Dockerfile and start up your containers.

This also implies that understanding Dockerfile instructions is not enough to create your Dockerfile because you also need to understand the context of the technology you are building.

The good news is that you can save a lot of time when starting out experimenting with a new technology because you can use a Docker image prepared by someone else. **Reading Dockerfiles prepared by others is a great way to learn about technology.**

Your first Dockerfile

Let's create your first custom Docker image by extending an existing image. We'll use the Alpine image for this activity so let's pull it with this command:

```
~$ docker image pull alpine:3.7

3.7: Pulling from library/alpine
ff3a5c916c92: Pull complete
Digest: sha256:7df6db5aa61ae9480f52f0b3a06a140ab98d427f86d8d5de0be
Status: Downloaded newer image for alpine:3.7
```

This is the list of images on my computer right now (please note that I have removed all previous images from my machine to give you a clearer picture, you don't need to remove the images you already have):

REPOSITORY	TAG	IMAGE ID	CREATED
alpine	3.7	3fd9065eaf02	4 weeks ago

Let's look into the image in an interactive shell:

```
~$ docker container run --rm -ti alpine:3.7 /bin/sh
/ # ls
bin dev etc home lib media mnt proc root run
```

You see that we have a Linux file system in the root directory, if you check the contents of `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin` you'll see the binaries (programs) that you use for user activities and administrative tasks.

We can see, for example, that `wget` comes bundled with Alpine in the `/usr/bin` directory, so you can use it to retrieve files via HTTP or FTP. `curl` however is not included:

```
/ # curl
/bin/sh: curl: not found
```

What if we want to add `curl` to this image?

We have the option to install `curl` while in the interactive shell in the container. This will install curl in the container, but it will only be available while the container is not removed.

It's more desirable to use a method that we can reproduce anytime automatically, and create a version of the Alpine image that we can reuse in our project team. So we will create a custom image.

Leave the shell with the `exit` command and let's create a new directory in our development directory and call it `example3`. Please create a file called `Dockerfile` in the `example3` directory. Your structure should look like this:

```
~/Development/example3$ tree
.
└── Dockerfile
0 directories, 1 file
```

Open the `Dockerfile` and copy and paste the following into the file:

```
FROM alpine:3.7
RUN apk add --update curl
```

Let's build our image from this Dockerfile and then I'll explain the steps in detail. Use this command (use your

Docker username from the Docker Hub in the image name in the form `--tag <your_docker_username>/alpine-smarter`:

```
~/Development/example3$ docker build \
--tag takacsmark/alpine-smarter .

Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM alpine:3.7
--> 3fd9065eaf02
Step 2/2 : RUN apk add --update curl
--> Running in 0644f7f5596d
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/community/x86_64/APKINDEX.tar.gz
(1/3) Installing ca-certificates (20171114-r0)
(2/3) Installing libcurl (7.58.0-r0)
(3/3) Installing curl (7.58.0-r0)
Executing busybox-1.27.2-r7.trigger
Executing ca-certificates-20171114-r0.trigger
OK: 5 MiB in 14 packages
Removing intermediate container 0644f7f5596d
--> 7f6ee00f65d9
Successfully built 7f6ee00f65d9
Successfully tagged takacsmark/alpine-smarter:latest
```

We have just built a new image that appears on our image list:

```
~/Development/example3$ docker image ls

REPOSITORY          TAG      IMAGE ID
takacsmark/alpine-smarter    latest   7f6ee00f65d9
alpine              3.7     3fd9065eaf02
```

Please note how I used my Docker Hub username and `alpine-smarter` as the name of the image. This naming convention will let you push your custom image to the Docker Hub later. I did not specify any version tag, so the image got the `latest` tag by default.

My image is slightly larger than the original Alpine image that we used as the base image, apparently `curl` has been added to the image.

Let's try `curl` with our new image now:

```
~/Development/example3$ docker container run --rm \
-ti takacsmark/alpine-smarter /bin/sh

# curl --version

curl 7.58.0 (x86_64-alpine-linux-musl) libcurl/7.58.0 LibreSSL/2.6
Release-Date: 2018-01-24
Protocols: dict file ftp ftps gopher http https imap imaps pop3 p
Features: AsynchDNS IPv6 Largefile NTLM NTLM_WB SSL libz UnixSocket
```

We just created a standalone Docker image that is based on Alpine and has `curl` included out of the box.

Let's add `vim` to the image, and I explain every step in detail.

1. Define the base image with FROM

Every Dockerfile must start with the `FROM` instruction. The idea behind this is that you need a starting point to build your image. You can start `FROM scratch`. Scratch is an explicitly empty image on the Docker Hub that is used to build base images like Alpine, Debian and so on.

You can start your Docker images from any valid image that you pull from public or private registries, or you can start from `scratch`. The image you start from is called the parent image. In our case, it's `FROM alpine:3.7` in the Dockerfile.

Note: I said that every Dockerfile must start with the `FROM` instruction, which is generally true, with one exception. Dockerfiles can receive arguments through the build command. Arguments, noted by the `ARG` instruction, may precede `FROM` in the Dockerfile. We'll use this in a later example.

Note: Docker provides a mechanism called multi-stage builds, where you can have more than one `FROM` instructions in a single Dockerfile, we will look at this later, too.

2. Add the lines to install packages

Please modify the file to install vim and curl like this:

```
FROM alpine:3.7
```

```
RUN apk update
```

```
RUN apk add vim  
RUN apk add curl
```

The `RUN` instruction executes Linux commands that you put in the Dockerfile after `RUN`.

In the example, we use `RUN` to update `apk` package information and install `vim` and `curl` using the `apk` package manager.

You can add one instruction per line as shown in the example, or you can combine the commands to one line, as we will see later.

3. Build your image

Please run the following in the Terminal:

```
~/Development/example3$ docker image build \  
-t takacsmark/alpine-smarter:1.0 .
```

This command is structured as follows:

- `docker image build` is the command to build a Docker image from a Dockerfile
- `-t takacsmark/alpine-smarter:1.0` defines the tag (hence `-t`, or `--tag`) of the image, this will be the

name of the image.

As the first part, I put my own Docker username `takacsmark` because I plan to publish my image on the Docker Hub under my username.

Then I gave it a human-readable name `alpine-smarter` and provided a version number `1.0`.

- please note the dot (.) at the end of the line. You need to specify the directory where `docker build` should be looking for the Dockerfile. The `.` tells `docker image build` to look for the Dockerfile in the current directory.

You should see a similar output in Terminal now:

```
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM alpine:3.7
--> 3fd9065eaf02
Step 2/4 : RUN apk update
--> Running in 97a797457152
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/community/x86_64/APKINDEX.tar.gz
v3.7.0-74-g1540930789 [http://dl-cdn.alpinelinux.org/alpine/v3.7/main]
v3.7.0-73-g5f006fb2a0 [http://dl-cdn.alpinelinux.org/alpine/v3.7/community]
OK: 9047 distinct packages available
Removing intermediate container 97a797457152
--> c6d82fa461e7
Step 3/4 : RUN apk add vim
--> Running in e51fa5c055c4
(1/5) Installing lua5.2-libs (5.2.4-r4)
(2/5) Installing ncurses-terminfo-base (6.0_p20171125-r0)
(3/5) Installing ncurses-terminfo (6.0_p20171125-r0)
```

```
(4/5) Installing ncurses-libs (6.0_p20171125-r0)
(5/5) Installing vim (8.0.1359-r0)
Executing busybox-1.27.2-r7.trigger
OK: 38 MiB in 16 packages
Removing intermediate container e51fa5c055c4
---> 633a475e2240
Step 4/4 : RUN apk add curl
---> Running in 676afeed72a1
(1/3) Installing ca-certificates (20171114-r0)
(2/3) Installing libcurl (7.58.0-r0)
(3/3) Installing curl (7.58.0-r0)
Executing busybox-1.27.2-r7.trigger
Executing ca-certificates-20171114-r0.trigger
OK: 39 MiB in 19 packages
Removing intermediate container 676afeed72a1
---> 3b61c44fadbb
Successfully built 3b61c44fadbb
Successfully tagged takacsmark/alpine-smarter:1.0
```

4. Enjoy the results

Docker created an image from your Dockerfile. You should see a new image in your image list issuing `docker image ls` again. This is my list now:

```
~/Development/example3$ docker image ls

REPOSITORY          TAG      IMAGE ID
takacsmark/alpine-smarter  1.0      3b61c44fadbb
takacsmark/alpine-smarter  latest   7f6ee00f65d9
alpine               3.7      3fd9065eaf02
```

Note that our alpine-smarter image with tag `latest` and `1.0` ended up being two different images on the list.

Let's check what's inside our new image, let's run the following command and check out vim and curl:

```
~/Development/example3$ docker container run --rm \
-ti takacsmark/alpine-smarter:1.0 /bin/sh
```

Right now you should be in the shell of your running container, let issue the following commands:

`vim --version` and `curl --version`. You should be seeing the version of vim and curl in your Terminal.

We have successfully added two packages to the Alpine base image. Let's not stop here; there is more!

Image layering

If you look at the output of the `docker image build` command again, you can notice that `docker image build` provided the build output in 4 steps, namely Step 1/4, Step 2/4, Step 3/4 and Step 4/4.

At the headline of each step, you can see the corresponding line in your Dockerfile. This is because `docker image build`

executes the lines in the Dockerfile one step at a time.

What is more important, with every line with a `RUN` instruction in the build process, Docker will create an intermediary image for the specific step. This means that Docker will take the base image (`alpine:3.7`), then execute `RUN apk update`, and then Docker will add the resulting files from that step as another layer on top of the base image.

You can follow the concept by following the lines in the output that start with `----> <image id>` these lines denote the image ids of intermediary images. Please note that the lines starting with `----> Running in <container id>` denote the ids of intermediary containers that Docker is using to execute the steps. The intermediary containers will be removed at the end of each step, and an intermediary image will be created.

This means that the final Docker image consists of 4 layers and the intermediary layers are also available on your system. Intermediary images contain the difference in the image's file system, that is the changes compared to the underlying layer.

This is useful because Docker will use the intermediary images as image cache, which means your future builds will be much faster for those Dockerfile steps that you do not modify.

Let's first see all the images that were created. Please issue the command

```
docker image ls -a in Terminal.
```

```
~/Development/example3$ docker image ls -a
```

REPOSITORY	TAG	IMAGE ID
takacsmark/alpine-smarter	1.0	3b61c44fadbb
<none>	<none>	633a475e2240
<none>	<none>	c6d82fa461e7
takacsmark/alpine-smarter	latest	7f6ee00f65d9
alpine	3.7	3fd9065eaf02

We used `-a` to list all images on your computer including intermediary images. Please note how the intermediary image ids are equivalent to the image ids you see during the build process after the `---->` signs.

Only the `RUN`, `COPY` and `ADD` Dockerfile instructions create layers to improve build performance. Other Dockerfile instructions will not create intermediary image layers.

Note: the Docker ecosystem changes rapidly, I've last checked on January 10, 2019. It's always a good idea to check the [official documentation](#) if the above statement is still valid.

The main advantage of image layering lies in image caching.

If you build your Dockerfile again now, with the same command `docker image build -t takacsmark/alpine-smarter:1.0 .`, you'll notice that the build was almost instantaneous and the output for every step says that the build was done from cache.

This behavior makes our lives a lot easier. Since image layers are built on top of each other, Docker will use the image cache during the build process up to the line where the first change occurs in your Dockerfile. Every later step will be re-built.

Image cache example

Let's play with the cache a little bit. Let's change our Dockerfile to see the behavior. Let's change the line from adding curl to adding git. This is the resulting file:

```
FROM alpine:3.7

RUN apk update
RUN apk add vim
RUN apk add git
```

Let's issue our build command again: `docker image build -t takacsmark/alpine-smarter:1.0 .`.

You'll see that the first 3 steps run using cache and only the last step will be re-run, as shown below.

```
~/Development/example3$ docker image build \
-t takacsmark/alpine-smarter:1.0 .

Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM alpine:3.7
--> 3fd9065eaf02
Step 2/4 : RUN apk update
--> Using cache
--> c6d82fa461e7
Step 3/4 : RUN apk add vim
--> Using cache
--> 633a475e2240
Step 4/4 : RUN apk add git
--> Running in 3faee1ce5262
(1/5) Installing ca-certificates (20171114-r0)
(2/5) Installing libcurl (7.58.0-r0)
(3/5) Installing expat (2.2.5-r0)
(4/5) Installing pcre2 (10.30-r0)
(5/5) Installing git (2.15.0-r1)
Executing busybox-1.27.2-r7.trigger
Executing ca-certificates-20171114-r0.trigger
OK: 52 MiB in 21 packages
Removing intermediate container 3faee1ce5262
--> 1d5349914234
Successfully built 1d5349914234
Successfully tagged takacsmark/alpine-smarter:1.0
```

Let's see what happens if you change an early step in the Dockerfile, for example, if you add one line after `apk update` like this:

```
FROM alpine:3.7

RUN apk update
RUN apk add curl
```

```
RUN apk add vim  
RUN apk add git
```

In this case, every step after the change will be re-built. Which means that the steps to install curl, vim and git will be re-run, no caching will be available beyond the point where the change occurred.

Dangling images

If you execute `docker image ls` now in Terminal, you'll see something strange.

```
~/Development/example3$ docker image ls
```

REPOSITORY	TAG	IMAGE ID
takacsmark/alpine-smarter	1.0	1d5349914234
<none>	<none>	3b61c44fadbb
takacsmark/alpine-smarter	latest	7f6ee00f65d9
alpine	3.7	3fd9065eaf02

Our newly built image is ready to use, but the previous image that we built is still hanging around, and it does not have a proper tag or name right now (`<none>:<none>` on the list). The reason behind this is that we have built the image with the same tag `takacsmark/alpine-smarter :1.0`, but with a different Dockerfile. The new image was tagged `takacsmark/alpine-smarter :1.0`, and the image that used to

have this tag in the previous build has lost its tag, and it is unnamed.

Docker calls such images dangling images. You can use the following command to list dangling images:

```
$ docker image ls --filter "dangling=true"
```

Here is how you can remove them:

```
~/Development/example3$ docker image prune
```

```
WARNING! This will remove all dangling images.
```

```
Are you sure you want to continue? [y/N] y
```

```
Deleted Images:
```

```
deleted: sha256:3b61c44fadbb96d14ca42b91b7766f6c5c49f22c0e290d4297
```

```
deleted: sha256:e00a5d02725031e8cbd9e55a68e5e611e0e76c5d5910cab578
```

```
Total reclaimed space: 1.35MB
```

Another way to remove dangling images is the following command:

```
$ docker image rm $(docker images -q --filter "dangling=true")
```

Understanding image layering

The idea of maximum resource sharing and reusability applies to Docker images, too. In every Docker image, we take a base image and add our changes to the base image in the form of file system layers.

Files will be copied to a new layer only if they are modified or created. If the files remain unchanged, they will be used from the underlying layer. Docker will only mark the differences in the layers that are built on top of each other.

This strategy is called copy on write. Let's see the layers of our `example3` image, the `docker image history` command will help us here.

```
~/Development/example3$ docker image history \
takacsmark/alpine-smarter:1.0

IMAGE              CREATED          CREATED BY
1d5349914234      31 minutes ago   /bin/sh -c apk add git
633a475e2240      About an hour ago /bin/sh -c apk add vim
c6d82fa461e7      About an hour ago /bin/sh -c apk update
3fd9065eaf02      5 weeks ago       /bin/sh -c #(nop)  CMD ["/"
<missing>          5 weeks ago       /bin/sh -c #(nop) ADD file
```

This output shows that our `alpine-smarter:1.0` image has 4 layers plus the base image listed bottom up. The base image, `alpine:3.7`, takes up 4.15 MB of space. You can follow the size each layer adds on top of this base image.

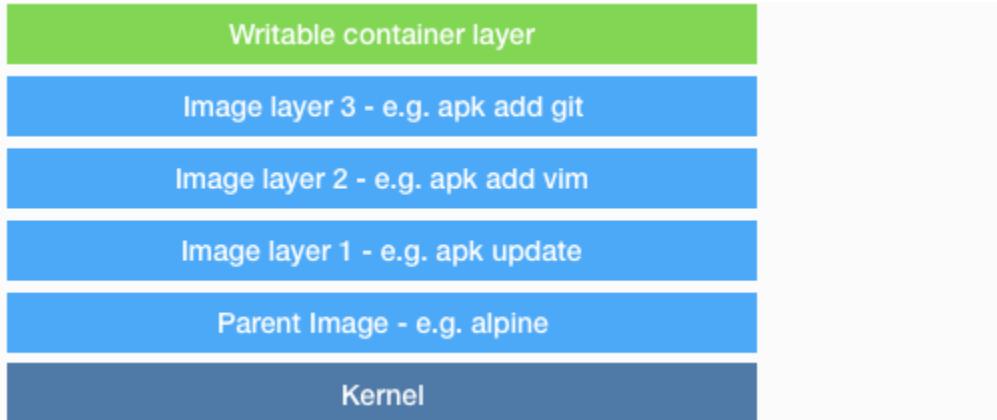
- `apk update` added 1.22MB

- apk add vim added 26.5MB
- apk add git added 14.8MB

All these layers add up to the overall size of 46.7MB that you see on the output of `docker image ls`.

The layers in images are read-only. Docker will not modify the layers. It will add changes with the copy on write strategy in the form of image layers. When you start a container from the image, Docker will add another, this time read-write, layer to the image. This read-write layer will be the home of any runtime information written in the container. So the main difference between the image and the container behind the scenes is a read-write file system layer on top.

If you start multiple containers from one single image, every container will have a read-write layer on top, and all containers will share the underlying image layers. If a container needs to change some file, it will copy those files into its top read-write layer and use its own copy. All other containers will still use the copy from the underlying image.



Main benefits of the copy on write (CoW) strategy are:

- resource sharing
- faster container start-up time
- optimized disk usage

Docker uses a union mount file system called Overlay2 by default on Linux machines. If your project has special storage needs, or special file system requirements, you can change the underlying file system, by changing the Docker storage driver. This is an advanced topic; we'll not discuss this in detail here.

Dockerfile key instructions

The official Docker documentation provides an in-depth description of Dockerfile instructions here:

<https://docs.docker.com/engine/reference/builder/>.

We'll cover the following instructions to get you started:

- `FROM` - every Dockerfile starts with `FROM`.
- `COPY` and `ADD` - copy files and directories into the image from the host machine.
- `ENV` - is used to set environment variables.
- `RUN` - runs operating system commands.
- `VOLUME` - defines volumes to store files outside the container file system.
- `USER` - specifies a user other than root of your image.
- `WORKDIR` - sets the working directory.
- `EXPOSE` - exposes ports.
- `CMD` and `ENTRYPOINT` .

FROM

As we've already learned, every Dockerfile must start with the `FROM` instruction in the form of `FROM <image>[:tag]` . This will set the parent image for your Dockerfile, which means that subsequent instructions will be applied to this parent image.

The `tag` value is optional. If you don't specify the `tag` , Docker will use the tag `latest` and will try and use or pull

the latest version of the base image during the build.

On the little bit more advanced side, let's note the following:

- There is one instruction that you can put before `FROM` into your Dockerfile. This instruction is `ARG`. `ARG` is used to specify arguments for the `docker image build` command with the `--build-arg <varname>=<value>` flag. This way you can pass the parent image version as an argument to Docker build, for example. We'll see examples of this later.
- You can have more than one `FROM` instructions in your Dockerfile. You will want to use this feature, for example, when you use one image to build your app and another image to run it. It's called a multi-stage build, and you can read about it [here](#). We will build an example together. This is why every section that starts with `FROM` in your Dockerfile is called a build stage (even in the simple case of having only one `FROM` instruction).

COPY vs ADD

Both `ADD` and `COPY` are designed to add directories and files to your Docker image in the form of `ADD <src>... <dest>` or `COPY <src>... <dest>`. Most resources suggest using `COPY`.

The `ADD` instruction was added to Docker first, and `COPY` was added later to provide a straightforward, rock-solid solution for copying files and directories into your image's file system because `ADD` was unpredictable in some cases.

`ADD` can pull files from URL sources, while `COPY` cannot. `ADD` can extract compressed files assuming it can recognize and handle the format. You cannot extract archives with `COPY`. The extraction feature of `ADD` depends on the archive format that it has to extract and this makes its behavior sometimes unpredictable. This is why most users prefer `COPY`.

If you want to pull files from the web into your image, it's better to use `RUN` and `curl` and uncompress your files with `RUN` and compression commands you would use on the command line.

ENV

`ENV` is used to define environment variables. The exciting thing about `ENV` is that it does two things:

1. You can use it to define environment variables that will be available in your container. So when you build an image and start up a container with that image, you'll find that the environment variable is available and is set to the value you specified in the Dockerfile.

2. You can use the variables that you specify by `ENV` in the Dockerfile itself. So in subsequent instructions, the environment variable will be available.

RUN

`RUN` will execute commands, so it's one of the most-used instructions. I would like to highlight two points:

1. You'll use a lot of `apt-get` type of commands to add new packages to your image. It's always advisable to put `apt-get update` and `apt-get install` commands on the same line.
This is important because of layer caching.
Having these on two separate lines would mean that if you add a new package to your install list, the layer with `apt-get update` will not be invalidated in the layer cache and you might end up in an inconsistent state.
2. `RUN` has two forms; `RUN <command>` (called shell form) and `RUN ["executable", "param1", "param2"]` called exec form. Please note that `RUN <command>` will invoke a shell automatically (`/bin/sh -c` by default), while the exec form will not invoke a command shell.

VOLUME

You can use the `VOLUME` instruction in a Dockerfile to tell Docker that the directories and files that you store in that specific directory should be stored on the host file system, not in the container file system. This implies that information stored in the volume will persist and be available also after you destroy the container.

In other words, it is best practice to create a volume for your data files, database files, or any file or directory that your users will change when they use your application.

The data stored in the volume will remain on the host machine even if you stop the container and remove the container with `docker container rm`. (The volume will be removed on exit if you start the container with `docker run --rm`.)

We'll learn more about volumes later.

USER

Don't run your programs as root in your containers, use the `USER` instruction to specify the user in the Dockerfile. This user will be also used to run any subsequent `RUN`, `CMD` AND `ENDPOINT` instructions in your Dockerfile.

WORKDIR

A convenient way to define the working directory, it will be used with subsequent `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions in the Dockerfile. You can specify `WORKDIR` multiple times in a Dockerfile.

If the directory does not exist, Docker will create it for you.

EXPOSE

An important instruction to inform your users about the ports your application is listening on. `EXPOSE` will not publish the port, you need to use `docker container run -p...` to do that when you start the container.

CMD and ENTRYPOINT

`CMD` is the instruction to specify what main process is to be run by your image in the following form: `CMD ["executable", "param1", "param2"...]`.

You can override `CMD` when you're starting up your container by specifying your command after the image name like this:

```
$ docker container run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND]  
[ARG...].
```

You can only specify one `CMD` in a Dockerfile (if you specify more than one, only the last one will be used).

It is good practice to specify a `CMD` even if you are developing a generic image. In this case, an interactive shell is a good `CMD` entry. So you can add `CMD ["python"]` or `CMD ["php", "-a"]` to give your users something to work with.

What is `ENTRYPOINT`? When you specify an entry point, your image will work a bit differently. You use `ENTRYPOINT` as the main executable of your image. In this case, whatever you specify in `CMD` will be added to `ENTRYPOINT` as parameters.

```
ENTRYPOINT ["git"]
CMD ["--help"]
```

The main use case is to add your custom script as the `ENTRYPOINT` of your Docker image, this way you can control the behavior of your container during runtime. You can use an entry point script, for example, to check whether a database has been initiated already and execute commands based on the results. We'll build an example of this soon.

Dockerfile best practices

Clean up steps in the Dockerfile

Some steps like installing Linux packages may add temporary files to your image. The best practice is to delete the temporary files in the same instruction in the Dockerfile so that they will not be added to your image.

We can reformulate our Dockerfile like this:

```
FROM alpine:3.4

RUN apk update && apk add \
    curl \
    vim \
    git \
    && rm -rf /var/cache/apk/*
```

After building this Dockerfile the usual way, you'll also find, that this time it has only taken 2 steps instead of 4.

```
~/Development/example3$ docker image build \
-t takacsmark/alpine-smarter:1.0 .

Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM alpine:3.4
--> c7fc7faf8c28
Step 2/2 : RUN apk update && apk add curl vim git && rm -rf /var/cache/apk/*
--> Running in 1fa93930b033
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/community/x86_64/APKINDEX.tar.gz
v3.4.6-299-ge10ec9b [http://dl-cdn.alpinelinux.org/alpine/v3.4/main]
v3.4.6-160-g14ad2a3 [http://dl-cdn.alpinelinux.org/alpine/v3.4/community]
OK: 5973 distinct packages available
(1/12) Installing ca-certificates (20161130-r0)
(2/12) Installing libssh2 (1.7.0-r0)
```

```
(3/12) Installing libcurl (7.59.0-r0)
(4/12) Installing curl (7.59.0-r0)
(5/12) Installing expat (2.2.0-r1)
(6/12) Installing pcre (8.38-r1)
(7/12) Installing git (2.8.6-r0)
(8/12) Installing lua5.2-libs (5.2.4-r2)
(9/12) Installing ncurses-terminfo-base (6.0_p20171125-r0)
(10/12) Installing ncurses-terminfo (6.0_p20171125-r0)
(11/12) Installing ncurses-libs (6.0_p20171125-r0)
(12/12) Installing vim (7.4.1831-r3)
Executing busybox-1.24.2-r14.trigger
Executing ca-certificates-20161130-r0.trigger
OK: 54 MiB in 23 packages
Removing intermediate container 1fa93930b033
--> ededa5b94924
Successfully built ededa5b94924
Successfully tagged takacsmark/alpine-smarter:1.0
```

Sort multi-line instructions

It's a good idea to sort multiline instructions in a human-readable manner. My example above is not optimal, because I'm installing packages in a random order. I should write a file like this instead, where I order packages in alphabetical order. This is very useful when you work with a long list.

```
FROM alpine:3.4

RUN apk update && apk add \
    curl \
    git \
```

```
vim \
&& rm -rf /var/cache/apk/*
```

Start your Dockerfile with the steps that are least likely to change

Your images will stabilize after a while, and changes will be less likely. The best practice is to structure your Dockerfile according to the following:

1. Install tools that are needed to build your application.
2. Install dependencies, libraries, and packages.
3. Build your application.

Clean up your Dockerfile

Review your steps in the Dockerfile and only keep the minimum set of steps that are needed by your application and remove unnecessary components.

Use a `.dockerignore` file

The directory where you issue the `docker build` command is called the build context. Docker will use all of the files and directories in your build directory as part of the build context. If you have files in your directory that are not needed by your build, you'll have an unnecessarily larger build context that results in a larger build.

You can remedy this situation by adding a `.dockerignore` file that works similarly to `.gitignore`. You can specify the list of folders and files that should be ignored in the build context.

If you want to have a look at the size of your build context, check out the first line of your `docker image build` output. My alpine build output, for example, says: `Sending build context to Docker daemon 2.048kB`.

Containers should be ephemeral

It is your best interest to design and build Docker images that can be destroyed and recreated/replaced automatically.

Which means that you should create Dockerfiles that define stateless images. Any state should be kept outside of your containers. We'll see how.

One container should have one concern

Think of containers as entities that take responsibility for one aspect of your project. Design your application in a way that your web server, database, in-memory cache and other components have their dedicated images.

You'll see the benefits of such a design when scaling your app horizontally. We'll look into interoperability of containers and container networking in later chapters.

It's a good idea to check out [the official Dockerfile best practices page](#) for more information.

Image building workflow

Now that we looked at the toolset and best practices, you might be wondering, what is the best way of building your Dockerfile.

Well, I think everybody has their own ways, let me show you mine. I use a fairly straightforward 4 step approach to build my Dockerfiles iteratively. It is the following:

1. **Pick the right base image** - in this step I experiment with the base images available online for the technology in question. I usually check out different flavors, like an image based on Debian Jessie and another one based on Alpine.

2. Use the shell to build your environment -

as a next step, I go with a try-and-fail approach. I pull my chosen images to my computer and start a container in interactive mode with a shell. Like we did with `docker container run --rm -ti takacsmark/alpine-smarter:1.0 /bin/sh`.

I start manually executing the steps in the container that I would put on the Dockerfile, and see how things work out. Once a step seems to be OK, I add it to my Dockerfile.

If something goes wrong, I change the course, and I update the Dockerfile immediately.

3. Add the steps to your Dockerfile and build your image - I keep adding steps continuously as I make progress with my setup in the container.

Now and then I stop and build my image from the Dockerfile to make sure that it produces the same results every time.

I use the newly built image to start a container with a shell and go on with my installation and set-up steps.

4. Repeat steps 2 and 3 - I keep repeating steps 2 and 3 until I reach the stage that I like.

Image building exercises

I hope you are happy with your newly acquired Docker toolkit. I would like to make you more comfortable, so we'll create a number of standalone image examples.

In this chapter, we will be building Docker images. We'll create Dockerfile examples and start up containers from them. We'll create and run shell scripts, Python applications, PHP and a MySQL database instance.

Run a shell script in a container

Let's call this `example4`, please create a directory in your development directory and create two files in that directory, named `myscript.sh` and `Dockerfile`; `myscript.sh` will be a Linux shell script, and `Dockerfile` is our Dockerfile.

My directory looks like this:

```
~/Development/example4$ tree
.
├── Dockerfile
└── myscript.sh
```

Let's add two lines to `myscript.sh`:

```
#!/bin/sh
echo 'Hello, World from my Docker Container!'
```

Let's create a Dockerfile as follows:

```
FROM alpine:3.7

COPY ./myscript.sh /root/
RUN chmod 764 /root/myscript.sh

CMD /root/myscript.sh
```

We'll use alpine:3.7 as our base image.

On line 3, `COPY ./myscript.sh /root/`, we are copying our script file from the host machine into the home directory of the root user, which is `/root` on Alpine Linux.

`RUN chmod 764 /root/myscript.sh` means that we are running the `chmod` Linux command to change file permissions on our script to make it executable.

The last line on the Dockerfile is `CMD /root/myscript.sh`. It means that our shell script is the main command of the Docker image, it will be executed when a container starts up.

Let's build the image and start a container to see what happens.

```
~/Development/example4$ docker image build \
--tag takacsmark/example4:1.0 .
```

```
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM alpine:3.7
 ---> 3fd9065eaf02
Step 2/4 : COPY ./myscript.sh /root/
 ---> e68142b3732d
Step 3/4 : RUN chmod 764 /root/myscript.sh
 ---> Running in 808ee6de8f17
Removing intermediate container 808ee6de8f17
 ---> 8926b5ea0944
Step 4/4 : CMD /root/myscript.sh
 ---> Running in b80a16294359
Removing intermediate container b80a16294359
 ---> 26200d65a67b
Successfully built 26200d65a67b
Successfully tagged takacsmark/example4:1.0
```

And run:

```
~/Development/example4$ docker run --rm takacsmark/example4:1.0
Hello, World from my Docker Container!
```

Add a USER

You should not run your programs as the root user. Therefore we'll learn how to create a user in your image and make it active. We'll stay with our shell script example. Let's modify our `Dockerfile` in `example4`.

Let's add a user called `example4` to our `Dockerfile` and set it as the user of our image. Please add the following lines to your `Dockerfile`, and I'll explain what's happening:

```
FROM alpine:3.7

RUN adduser -D example4

COPY ./myscript.sh /home/example4/bin/
RUN chmod 764 /home/example4/bin/myscript.sh

RUN chown -R example4:example4 /home/example4/bin
USER example4

CMD /home/example4/bin/myscript.sh
```

The `FROM` instruction stays the same as before.

We added the line `RUN adduser -D example4` to run the Linux command `adduser`. This will add a user to our system with the username `example4`. Please note that Linux will add a group with the same name (`example4`) and make the `example4` user the only member of that group. This way you can manage the user and group level file permissions for the user's own files. `-D` means that we are not setting a password for that user.

Note that we added the user at the beginning of the Dockerfile because we expect that it will not change often.

We changed the `COPY` instruction to copy our script file into the directory `/home/example4/bin/`, where `/home/example4` is the home directory of `example4` user that we just created. We specified that the file should be copied into a `bin`

subdirectory of the example4 user home directory. This way our script will be placed into a specific directory.

We change file permissions the same way we did before, to make our script executable.

The next line contains something new. `'' RUN chown -R example4:example4 /home/example4/bin` runs the `chown` command, which will change the ownership of the `/home/example4/bin` directory and its contents to user `example4` and group `example4`. We need to do this because `COPY` will by default copy our files into the image with owner user and owner group of `root:root`. This is not convenient for us, because we will run the container as user `example4` and we need to have our file permissions set accordingly.

At the line where we are running the `chown` command in the Dockerfile, we are still acting as the `root` user, which is practical, because we need superuser access to run `chown`.

The `COPY` instruction does not respect the `USER` instruction in the Dockerfile, so putting `USER` before `COPY` in the Dockerfile would not have the expected effect on `COPY`.

On the next line after `chown`, we set the user with `USER example4`. This means that every `RUN` instruction and the `CMD` instruction will be run as user `example4`. `COPY` and `ADD` will ignore `USER`, there is a handy feature to make it a bit better looking, and I'll show it to you in the next step.

Our `CMD` line reflects the new location of `myscript.sh`. Let's build the image and run the container.

```
~/Development/example4$ docker image build \
--tag takacsmark/example4:1.0 .

Sending build context to Docker daemon 3.072kB
Step 1/7 : FROM alpine:3.7
--> 3fd9065eaf02
Step 2/7 : RUN adduser -D example4
--> Using cache
--> d07806c07f0a
Step 3/7 : COPY ./myscript.sh /home/example4/bin/
--> Using cache
--> eb0dde5d5ab6
Step 4/7 : RUN chmod 764 /home/example4/bin/myscript.sh
--> Using cache
--> 86cd06a0e7a1
Step 5/7 : RUN chown -R example4:example4 /home/example4/bin
--> Using cache
--> 53167fb1d9c4
Step 6/7 : USER example4
--> Using cache
--> 37ee64276186
Step 7/7 : CMD /home/example4/bin/myscript.sh
--> Using cache
--> 86086641f2c9
Successfully built 86086641f2c9
Successfully tagged takacsmark/example4:1.0
```

And the run output is the following:

```
~/Development/example4$ docker run --rm takacsmark/example4:1.0
```

```
Hello, World from my Docker Container!
```

COPY with chown

In the previous section, we have seen that we need to use `chown` to change the ownership of the script that we copy into the image. I wanted to show you the base problem before I give you a more elegant solution to it.

Recently Docker added `chown` functionality to the `COPY` instruction in the Dockerfile. Let's change your Dockerfile like this:

```
FROM alpine:3.7

RUN adduser -D example4
USER example4

COPY --chown=example4:example4 ./myscript.sh /home/example4/bin/
RUN chmod 764 /home/example4/bin/myscript.sh

CMD /home/example4/bin/myscript.sh
```

As you can see, I have moved `USER example4` to the beginning of the Dockerfile, because I do not need superuser access to run `RUN chown ...`, because we're using the new form of `COPY --chown` to do `chown` for us.

The line `COPY --chown=example4:example4 ./myscript.sh /home/example4/bin/` has everything we need to copy the file

into the image and make our `example4` user the owner of the file at the same time.

If you build and run this Dockerfile, you'll be getting the same results as before.

Set the working directory

You have probably noticed that we had to type the path `/home/example4/bin` a couple of times in our Dockerfile. If we change this path, we'll have to change several lines in the Dockerfile. This takes time and effort and is error-prone.

It is always a good idea to eliminate redundant code, so we'll use `WORKDIR` to specify our working directory and point it to `home/example4/bin`. After we specify `WORKDIR`, we can refer to it as the current directory with `./`.

Our Dockerfile will look like this:

```
FROM alpine:3.7

RUN adduser -D example4
USER example4

WORKDIR /home/example4/bin

COPY --chown=example4:example4 ./myscript.sh .
RUN chmod 764 ./myscript.sh

CMD ./myscript.sh
```

Please build and run the project; you should get the same results as before.

Provide arguments to your script

Let's modify our shell script to take an argument and instead of echoing "Hello World" let's echo "Hello <your name>", where <your name> is the argument of our shell script.

I have changed our script to take one argument. `myscript.sh` looks like this now:

```
#!/bin/sh
if [ $# -eq 0 ]; then
    echo "Please provide your name as an argument!"
else
    echo "Hello $1 from our Docker container!"
fi
```

We check if the script received an argument with `if [$# -eq 0]; then`, if the number of arguments is 0, we echo a message to provide an argument. If at least one argument was supplied, then we echo the message with the first argument (`$1`).

Let's build our image again.

```
~/Development/example4$ docker image build \
--tag takacsmark/example4:1.0 .
```

Let's run the container.

```
~/Development/example4$ docker run \
--rm takacsmark/example4:1.0
```

Please provide your name as an argument!

It says we have to provide an argument. Let's see what happens if I provide a name argument to the `docker container run` command:

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0 ./myscript.sh Cedric
```

Hello, Cedric from our Docker container!

This works as expected, but I would prefer to give the argument without having to specify the executable on the command line since the executable is always `myscript.sh`.

The solution is to use `ENTRYPOINT` in the Dockerfile. Let's modify our Dockerfile like this:

```
FROM alpine:3.7

RUN adduser -D example4
USER example4

WORKDIR /home/example4/bin
```

```
COPY --chown=example4:example4 ./myscript.sh ./
RUN chmod 764 ./myscript.sh

ENTRYPOINT [ "./myscript.sh" ]
```

Let's build the image with `docker image build --tag takacsmark/example4:1.0 .` and run it:

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0
```

Please provide your name as an argument!

This is the expected behavior, let's give it an argument like this:

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0 Harbinder
```

Hello, Harbinder from our Docker container!

Let's also give our message a default value, in case no argument is specified. The value of `CMD` will be automatically appended to `ENTRYPOINT` and overridden by `docker container run ...` arguments. Let's update the `Dockerfile`:

```
FROM alpine:3.7

RUN adduser -D example4
USER example4

WORKDIR /home/example4/bin

COPY --chown=example4:example4 ./myscript.sh ./
RUN chmod 764 ./myscript.sh

ENTRYPOINT [ "./myscript.sh"]

CMD ["World"]
```

Let's build the image again and run a container without and with a command line argument:

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0
```

```
Hello, World from our Docker container!
```

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0 Lisa
```

```
Hello, Lisa from our Docker container!
```

Please note how we used the `["..."]` form, called exec form, in the `ENTRYPOINT` and `CMD` instructions. Arguments will be picked up in exec form only. We could write `ENTRYPOINT ./myscript.sh`, which is called shell form, but in this case our `docker container run ...` arguments won't be picked up.

We created a Docker image that has our custom shell script as its entry point. This is widely used practice to influence the behavior of containers programmatically. Just imagine the potential of checking various conditions at startup of the container in your shell script and act accordingly. Let's see an example!

Using environment variables to control container behavior

As a next step, we will add an environment variable to define whether we are in the development environment or production environment.

If we are in development, we will run `myscript.sh` in trace mode; if we are in production, we will run `myscript.sh` the same way as before.

We will create a new shell script that will be the new entry point of our container. We'll check the environment variable in this script. We'll call the environment variable `script_env` with two potential values `dev` and `prod`.

Let's create the script and call it `setup.sh` with the below content:

```
#!/bin/sh
if [ "$script_env" == "dev" ]; then
    echo "You are running in development mode";
    sh -x ./myscript.sh @@
elif [ "$script_env" == "prod" ]; then
```

```
        exec ./myscript.sh $@  
fi
```

We are checking the value of the environment variable `$script_env` in this script. We'll define the default value of this variable in the Dockerfile in a minute.

If we are in the `dev` environment, we'll execute `myscript.sh` in trace mode with `sh -x`. `$@` will pass the arguments to `myscript.sh` that were given to `setup.sh`.

If we are in the `prod` environment, we'll execute `myscript.sh` without trace turned on.

Let's define the default value of the `script_env` environment variable in the Dockerfile:

```
FROM alpine:3.7  
  
RUN adduser -D example4  
USER example4  
  
WORKDIR /home/example4/bin  
  
COPY --chown=example4:example4 ./setup.sh ./myscript.sh ./  
RUN chmod 764 ./setup.sh ./myscript.sh  
  
ENV script_env=dev  
  
ENTRYPOINT [ "./setup.sh" ]  
  
CMD [ "World" ]
```

The line `ENV script_env=dev` defines the environment variable. We set it to `dev` for development by default.

Please note how I added `setup.sh` to the `COPY` instruction and the `chmod` command. I have also changed the `ENTRYPOINT` to `setup.sh`.

Let's build it with the command `docker image build --tag takacsmark/example4:1.0`. Let's test the container with the default `dev` setting:

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0
```

```
You are running in development mode
Hello, World from our Docker container!
+ [ 1 -eq 0 ]
+ echo Hello, World from our Docker container!
```

```
~/Development/example4$ docker container run \
--rm takacsmark/example4:1.0 Arnold
```

```
You are running in development mode
+ [ 1 -eq 0 ]
+ echo Hello, Arnold from our Docker container!
Hello, Arnold from our Docker container!
```

You see the trace info of the first run without a command line argument, that echoes "Hello World from our Docker container!" and the second run with an argument echoes the appropriate message.

Let's run the container in production mode; we'll override the value of the environment variable from the command line like this:

```
~/Development/example4$ docker container run --rm \  
--env script_env=prod takacsmark/example4:1.0
```

```
Hello, World from our Docker container!
```

```
~/Development/example4$ docker container run --rm \  
--env script_env=prod takacsmark/example4:1.0 Rahul
```

```
Hello, Rahul from our Docker container!
```

This time there is no trace info.

You may be wondering how we can run a shell against this image. The command `docker container run --rm -ti takacsmark/example4:1.0 /bin/sh` will not work, because the argument will be passed to the script, so we'll get the following output:

```
~/Development/example4$ docker container run --rm \  
-ti takacsmark/example4:1.0 /bin/sh
```

```
You are running in development mode  
+ [ 1 -eq 0 ]  
+ echo Hello, /bin/sh from our Docker container!  
Hello, /bin/sh from our Docker container!
```

We have to override the entry point to get into the shell. It works as follows:

```
~/Development/example4$ docker container run --rm -ti \
--entrypoint /bin/sh takacsmark/example4:1.0

~/bin $
```

In this shell script example we have built a Docker image using `USER`, `COPY`, `ENTRYPOINT`, `WORKDIR` and `ENV`, we reviewed how to build a Docker image with a custom user, how to set file permissions. We have also discussed how to create a setup script for a Docker container.

How I built the shell script example

In the previous chapter we have built a shell script, and we've developed a Docker image to run it. When we built the example in the book, I've explicitly told you what to copy and paste into individual files.

I think it would be a good idea to share with you how I built that example on my machine, so when you start building your own, you have some tips how to get started.

There is a high chance that your laptop runs Windows, macOS or a version of Linux that is not Alpine. I think in any

case the best idea is to build your shell script on a system that is identical to your target environment.

In this case, we were building the shell script for Alpine Linux, so it's best to use Alpine.

How do you start experimenting with Alpine? The easiest way is to start up an Alpine container in shell mode and map your development project directory into it. This way your changes will be saved to the source files on your host machine.

Let's take our `example4` directory and start up a container like this:

```
~/Development/example4$ docker container run --rm -ti \
-v $PWD:/usr/src/example4 \
-w /usr/src/example4 alpine:3.7 sh

/usr/src/example4 #
```

The command binds your `example4` directory into `/usr/src/example4` in the container and makes this directory the working directory with `-w /usr/src/example4`.

The only difference compared to our image is, that we have not created the `example4` user in this container; therefore we need to be careful because right now we are `root`. That's why we are not mapping our code to the home directory of the `example4` user, because it does not exist.

We can create the user with one command now and switch to it to make the environment more like the final image.

```
/usr/src/example4 # adduser -D example4

/usr/src/example4 # chown -R example4:example4 .

/usr/src/example4 # su example4

/usr/src/example4 $ ls -l

total 12
-rw-r--r--    1 example4 example4      253 May 30 15:16 Dockerfile
-rw-r--r--    1 example4 example4      143 May 30 15:12 myscript
-rw-r--r--    1 example4 example4      188 May 30 15:16 setup.sh

/usr/src/example4 $ whoami

example4
```

Now we have a shell to develop the shell script. Let's change `myscript.sh`. First we need to add execute permissions to the user `example4`.

```
/usr/src/example4 $ chmod 744 myscript.sh

/usr/src/example4 $ ls -l

total 24
drwxr-xr-x    8 example4 example4     4096 Feb 16 14:46 .git
-rw-r--r--    1 example4 example4      244 Feb 16 14:06 Dockerfile
-rwxr--r--    1 example4 example4      133 Feb 16 14:03 myscript
-rw-r--r--    1 example4 example4      181 Feb 16 14:07 setup.sh
```

You have probably noticed that I have executed the same steps that we have in the Dockerfile, `adduser`, `chown`, and `chmod`.

When you create a Docker image, you can build it like this. You execute the steps in a container to get to your desired state, and you add the steps to your Dockerfile along the way. Once your work is done, you can destroy the container with the shell, and you can recreate a container with your Dockerfile. If results are the same, you have done an excellent job; you can reproduce this environment anytime and anywhere.

Now we can run the script right here:

```
/usr/src/example4 $ ./myscript.sh  
Please provide your name as an argument!  
/usr/src/example4 $ ./myscript.sh World  
Hello, World from our Docker container!
```

Working with Docker has advantages in every stage of the development lifecycle. You can launch a sandbox with your exact production target environment to start experimenting.

This will reduce the risk of having to rework your code in later stages of development. Thus Docker minimizes the

cost of rework and improves the overall quality of your development process and your software product.

Python Flask server with a volume for logs

In this example, let's call it `example5`, we will build a Python web application, and **we'll showcase an essential aspect of Docker, namely Docker volumes**.

We'll use a Python microframework, called Flask (<http://flask.pocoo.org/>) to create this example.

Let's create a Flask application that displays "Hello World". Please create a directory in your development directory and call it `example5`. Create your Python file, name it `app.py` with the below content:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

On the first line we import `Flask` from `flask`, then we create an instance of the `Flask` class that will be our application.

`@app.route("/")` defines the root route of our web application. If you run your app on `http://example.com` this means the route `http://example.com`.

`def hello():` is the definition of a function called `hello` that takes no arguments. `return "Hello World!"` is the body of the function, it'll return the string "Hello World". Please note how this line is indented, Python uses strict indentation to identify code segments.

The last two lines say:

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

These lines mean that if we are running our script directly (i.e., its name is `__main__`) then run our app on port 5000. `host='0.0.0.0'` makes our app available on all network interfaces.

Now that we have this simple example script, all we need is a Docker container to run it. So let's build a Docker image. We'll need Python and Flask inside that image.

For the base image, let's pick something that is small and has Python latest inside. At the time of writing this image is

```
python:3.6.4-alpine3.7 , so let's go with it.
```

```
FROM python:3.6.4-alpine3.7

RUN pip install Flask
WORKDIR /usr/src/flaskapp
COPY ./app.py .

CMD python app.py
```

We `RUN pip install Flask` as described on the Flask site, <http://flask.pocoo.org/>. `pip` is a tool to install Python packages.

We create the working directory for our application and copy `app.py` into the image. We'll run the script with `python app.py`.

Let's build the image:

```
~/Development/example5$ docker build \
--tag takacsmark/example5:1.0 .
```

Let's run our server like this:

```
~/Development/example5$ docker container run --rm \
-p 5000:5000 takacsmark/example5:1.0

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Now if you visit <http://localhost:5000> in your browser; you'll see the following output:



Let's change this little app now so that it writes log messages into a log file. We'll use Python's logging functionality

(<https://docs.python.org/3/howto/logging.html#logging-basic-tutorial>).

We'll create a log file in our container at

`/var/log/example5/app.log`. This is going to be the file where we write log messages. We set the log level to `DEBUG`, which provides the most log messages. We also log the date and time with every message.

To do this, we import `logging` and `FileHandler`, which handles writing log messages to a file. We also add code to our `hello()` function to display log messages. Our updated `app.py` is the following:

```
import logging
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    app.logger.debug('debug message')
```

```
app.logger.info('info message')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
app.logger.critical('critical message')
return "Hello World!"

if __name__ == "__main__":
    handler = logging.FileHandler('/var/log/example5/app.log')
    handler.setLevel(logging.DEBUG)
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(level
    handler.setFormatter(formatter)
    app.logger.addHandler(handler)
    app.logger.setLevel(logging.DEBUG)
    app.run(host='0.0.0.0', port=5000)
```

Our file has become a bit more lengthy, but fundamentally it's still very simple. We create and set up the logger after `if __name__ == "__main__"` and added it to our `app`. Then we added log messages to the `hello()` function.

As you see we have specified that we'll save our log messages to `/var/log/example5/app.log` in our container. To make it work, we have to create the directory `/var/log/example5` in our Dockerfile. Let's update our Dockerfile to look like this:

```
FROM python:3.6.4-alpine3.7

RUN pip install Flask
WORKDIR /usr/src/flaskapp
COPY ./app.py .

RUN mkdir /var/log/example5
```

```
CMD python app.py
```

It's time to build and run again with these commands:

```
~/Development/example5$ docker image build \
--tag takacsmark/example5:1.0 .

~/Development/example5$ docker container run --rm \
-p 5000:5000 takacsmark/example5:1.0

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

If you visit the site again, you'll see that error and critical messages appear on the console:

```
~/Development/example5$ docker container run --rm \
-p 5000:5000 takacsmark/example5:1.0

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
[2018-02-21 10:46:19,443] ERROR in app: An error occurred
[2018-02-21 10:46:19,443] CRITICAL in app: critical message
172.17.0.1 - - [21/Feb/2018 10:46:19] "GET / HTTP/1.1" 200 -
```

Let's have a look at our log file. It is inside the container, so we have to have a look inside the running container. Let's list the running containers on our machine first, to find out the name of the container:

```
~/Development/example5$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND
0caa9e7df05b	takacsmark/example5:1.0	"/bin/sh -c 'python

My container is called `reverent_roentgen`, so I'll use this name in the following command, you can also use the container id. Let's execute an interactive shell:

```
~/Development/example5$ docker exec -ti reverent_roentgen sh
```

```
/usr/src/flaskapp #
```

Let's see the contents of the log file:

```
/usr/src/flaskapp # cat /var/log/example5/app.log
```

```
2018-02-21 10:46:19,442 - __main__ - DEBUG - debug message
2018-02-21 10:46:19,442 - __main__ - INFO - info message
2018-02-21 10:46:19,442 - __main__ - WARNING - A warning occurred
2018-02-21 10:46:19,443 - __main__ - ERROR - An error occurred
2018-02-21 10:46:19,443 - __main__ - CRITICAL - critical message
```

All the messages are displayed neatly in our log file.

This solution leaves us with two concerns:

1. The log is written into the container's file system. If you remember, our container is a

layered file system, writing to it is costly. It's better to write outside the container's file system.

2. The other concern is that our log file will be lost when we remove the container. If you want to retain the log file for the future, you'll need to find a way to store it outside the container.

Docker has a mechanism for storing files outside your container's file system; it's called Docker volumes.

Volumes are locations on your host machine outside the container. Docker will store your volume data at these locations. You can use volumes in two ways:

1. You can use Docker's built-in mechanism to create volumes for your containers. These volumes are stored in Docker's internal file structure.

On Linux machines this location is at /var/lib/docker/volumes. In case of Docker for Mac, you can use the command screen ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty to attach to a Terminal and browse /var/lib/docker in the virtual machine included with Docker. Docker's proprietary files are located at C:\Users\Public\Documents\Hyper-V\Virtual hard disks\MobyLinuxVM.vhdx under Windows.

The volumes will stay around after you remove the container, you can give volumes a name, and you can also use them to share data between containers.

We'll learn more about volumes in the "Docker volumes" chapter.

2. You can use volumes to bind mount a directory from your local host into the container. When we used `-v` in the earlier examples, we used volumes to bind mount our source directory from your host machine into the container.

Let's define a Docker volume for our log file. Docker provides tools to define a volume in the Dockerfile itself. This means, that when you start a container, Docker will create a volume automatically in `/var/lib/docker/volumes`. The volume will stay around also after you stop and remove the container, so that you can review the files later.

The downside of defining the volume in the Dockerfile is that Docker will give the volume a random name, so you cannot refer to it later and use the same volume in future containers. Let's start with this and later I'll show you how to create volumes that have a name.

We'll add one line to the Dockerfile to define a volume:

```
FROM python:3.6.4-alpine3.7

RUN pip install Flask
WORKDIR /usr/src/flaskapp
COPY ./app.py .

RUN mkdir /var/log/example5
VOLUME /var/log/example5

CMD python app.py
```

Docker will automatically create a volume when we start up a container. To check the results, let's first verify the current volumes on your machine with the following command:

```
~/Development/example5$ docker volume ls
DRIVER          VOLUME NAME
```

As you can see, I have no volumes on my machine. If you strictly followed the book and did not create other images, you'll also have an empty list as the output.

Let's build and run our container:

```
~/Development/example5$ docker image build \
--tag takacsmark/example5:1.0 .

~/Development/example5$ docker container run \
-p 5000:5000 takacsmark/example5:1.0

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Note that our `run` command does not contain the `--rm` flag. This is because using `--rm` will remove the volume when the container is stopped, but in this case, we want to examine the volume after the container has been stopped.

Now that the container is running, let's list the volumes again:

```
~/Development/example5$ docker volume ls  
  
DRIVER      VOLUME NAME  
local        a8b868b5f66babfabbf8b85f5624480b19ad10ae42749c
```

Docker has created a volume with a long random name. You can also inspect the volume like this:

```
~/Development/example5$ docker volume inspect \  
a8b868b5f66babfabbf8b85f5624480b19ad10ae42749cbf84d2d732811dbadd  
  
[  
 {  
   "CreatedAt": "2018-02-21T14:54:37+01:00",  
   "Driver": "local",  
   "Labels": null,  
   "Mountpoint": "/var/lib/docker/volumes/a8b868b5f66babfabbf8b85f5624480b19ad10ae42749cbf84d2d732811dbadd/_data",  
   "Name": "a8b868b5f66babfabbf8b85f5624480b19ad10ae42749cbf84d2d732811dbadd",  
   "Options": {},  
   "Scope": "local"  
 }  
]
```

The “Mountpoint” entry points to the volume location on the host machine. Let’s see what’s inside the volume. You can issue the following command on Linux:

```
~/Development/example5$ sudo ls -al /var/lib/docker/volumes/a8b868b5-  
[sudo] password for takacsmark:  
total 8  
drwxr-xr-x 2 root root 4096 febr 21 14:54 .  
drwxr-xr-x 3 root root 4096 febr 21 14:54 ..  
-rw-r--r-- 1 root root 0 febr 21 14:54 app.log
```

You need administrator access to read this location, that’s why I used `sudo` in this command. Our `app.log` is inside the directory, its size is 0, which means it’s empty.

Let’s visit <http://localhost:5000> and let’s check the log again.

```
~/Development/example5$ sudo ls -al /var/lib/docker/volumes/a8b868b5-  
total 12  
drwxr-xr-x 2 root root 4096 febr 21 14:54 .  
drwxr-xr-x 3 root root 4096 febr 21 14:54 ..  
-rw-r--r-- 1 root root 322 febr 21 14:57 app.log
```

The file size has changed, we can use `cat` to look inside:

```
~/Development/example5$ sudo cat /var/lib/docker/volumes/a8b868b5-  
2018-02-21 13:57:30,956 - __main__ - DEBUG - debug message
```

```
2018-02-21 13:57:30,956 - __main__ - INFO - info message
2018-02-21 13:57:30,956 - __main__ - WARNING - A warning occurred
2018-02-21 13:57:30,957 - __main__ - ERROR - An error occurred
2018-02-21 13:57:30,957 - __main__ - CRITICAL - critical message
```

The messages are also present in the file. Remember the file is on the host machine now, it's not inside the container's file system. This means, if we stop the container with `docker container stop <container_name>` and remove the container with `docker container rm <container_name>`, the volume will be still around, and the log file will be inside the volume containing all the log information.

Please open up a new Terminal window and issue the command:

```
~/Development/example5$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND
da36445070ff	takacsmark/example5:1.0	"/bin/sh -c 'python

Take the container name (`wizardly_engelbart` in my example) and issue the commands:

```
~/Development/example5$ docker stop wizardly_engelbart
wizardly_engelbart

~/Development/example5$ docker container rm wizardly_engelbart
wizardly_engelbart
```

Your container has been removed; your container list is empty:

```
~/Development/example5$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED
[REDACTED]			

However, if you go to your volumes list, you'll see that the volume is still there:

```
~/Development/example5$ docker volume ls
```

DRIVER	VOLUME NAME
local	a8b868b5f66babfabbf8b85f5624480b19ad10ae42749c

And its contents are unchanged:

```
~/Development/example5$ sudo cat /var/lib/docker/volumes/a8b868b5f66babfabbf8b85f5624480b19ad10ae42749c/_log/main.log
```

```
2018-02-21 13:57:30,956 - __main__ - DEBUG - debug message
2018-02-21 13:57:30,956 - __main__ - INFO - info message
2018-02-21 13:57:30,956 - __main__ - WARNING - A warning occurred
2018-02-21 13:57:30,957 - __main__ - ERROR - An error occurred
2018-02-21 13:57:30,957 - __main__ - CRITICAL - critical message
```

What happens if we start another container from the same image? Let's try:

```
~/Development/example5$ docker container run \
-p 5000:5000 takacsmark/example5:1.0
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Use a new Terminal window to list the volumes with:

```
~/Development/example5$ docker volume ls
```

DRIVER	VOLUME NAME
local	a8b868b5f66babfabff8b85f5624480b19ad10ae42749c
local	da9f39f6cce0c22e7fd4046f8d029ba010621b13e92a90

As you can see, we have two volumes now. Each run command produces a new volume in the Docker file hierarchy. This is a good solution to retain files from various runs, but it's not a good solution if you want to append to an already existing log file.

Let's change the solution so that we append to the log that we already had.

Before we start, it would be best to clean up the volumes that we created so that they don't take up space. Stop and remove the containers and use the following command to clean up volumes (we'll learn about volume management details later):

```
~/Development/example5$ docker volume prune

WARNING! This will remove all volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
a8b868b5f66babfabbf8b85f5624480b19ad10ae42749cbf84d2d732811dbadd
da9f39f6cce0c22e7fd4046f8d029ba010621b13e92a90f41f5406a9b36639a7

Total reclaimed space: 322B

~/Development/example5$ docker volume ls
```

DRIVER	VOLUME NAME

To use the same volume, again and again, we need to give it a name; this is called a **named volume**. You cannot, however, create a named volume in the Dockerfile; you can only create a named volume at runtime. The reason for this is to avoid hardcoded naming inside images that are supposed to be generic and portable. We have to create our named volume when we start up the container.

We can simply do it in our `docker run` command, like this:

```
~/Development/example5$ docker container run \
-p 5000:5000 \
-v example5log:/var/log/example5 \
takacsmark/example5:1.0

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

We used the `-v` option to specify the volume name, which is the first part of the expression, while the second part specifies the location inside the container. The list of volumes looks like this right now:

```
~/Development/example5$ docker volume ls
```

DRIVER	VOLUME NAME
local	example5log

Let's inspect the volume:

```
~/Development/example5$ docker volume inspect example5log
```

```
[  
 {  
   "CreatedAt": "2018-02-21T15:12:51+01:00",  
   "Driver": "local",  
   "Labels": null,  
   "Mountpoint": "/var/lib/docker/volumes/example5log/_data",  
   "Name": "example5log",  
   "Options": {},  
   "Scope": "local"  
 }  
]
```

After visiting <http://localhost:5000> our log file contains the following lines:

```
~/Development/example5$ sudo cat \  
 /var/lib/docker/volumes/example5log/_data/app.log
```

```
2018-02-21 14:15:46,285 - __main__ - DEBUG - debug message
2018-02-21 14:15:46,285 - __main__ - INFO - info message
2018-02-21 14:15:46,285 - __main__ - WARNING - A warning occurred
2018-02-21 14:15:46,286 - __main__ - ERROR - An error occurred
2018-02-21 14:15:46,286 - __main__ - CRITICAL - critical message
```

Stop and remove the container as we did before, and start a new container with the same command:

```
~/Development/example5$ docker container run \
-p 5000:5000 \
-v example5log:/var/log/example5 \
takacsmark/example5:1.0

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

After visiting `http://localhost:5000` again the log contains these lines:

```
~/Development/example5$ sudo cat \
/var/lib/docker/volumes/example5log/_data/app.log

2018-02-21 14:15:46,285 - __main__ - DEBUG - debug message
2018-02-21 14:15:46,285 - __main__ - INFO - info message
2018-02-21 14:15:46,285 - __main__ - WARNING - A warning occurred
2018-02-21 14:15:46,286 - __main__ - ERROR - An error occurred
2018-02-21 14:15:46,286 - __main__ - CRITICAL - critical message
2018-02-21 14:18:12,352 - __main__ - DEBUG - debug message
2018-02-21 14:18:12,352 - __main__ - INFO - info message
2018-02-21 14:18:12,352 - __main__ - WARNING - A warning occurred
2018-02-21 14:18:12,353 - __main__ - ERROR - An error occurred
2018-02-21 14:18:12,353 - __main__ - CRITICAL - critical message
```

As you can see, we have appended the log entries from the new run into the original file in the `example5log` volume.

The volume is persisted even if you remove the containers using it. This will be very useful when, for example, you want to store your MySQL database files on the host machine.

We have seen that Docker has a built-in mechanism for storing data outside the container file system. This mechanism is called Docker volumes. Volumes are by default stored in the Docker file structure on the host machine (under `/var/lib/docker/volumes` on Linux).

It is a good idea and best practice to store your volumes in the Docker structure for portability reasons. You can specify a directory on the host machine other than `/var/lib/docker/volumes`, but then you need to make sure that the directory exists on the host. This may give you a portability headache in the long run.

Note that the volume will be created on the host where the container is run. The volume will not be available on other host machines.

We'll learn more about volumes in the "Docker volumes" chapter.

PHP Slim example with Composer

In this chapter, we'll see an example where we use a package management system in our project to install dependencies. Many projects use solutions like Node's npm, PHP's Composer or Python's pip to manage dependencies.

You'll learn how to prepare a Docker image for this scenario and learn to run the project. For this example, we'll use a PHP microframework called SLIM and the PHP package manager, Composer.

Node.js and the npm package manager are very similar to this scenario; we'll build a Node project in the next example.

Please create a directory called `example6` in your usual development directory and let's see what we have to do.

Composer is a package manager for PHP; it lets you install packages like the SLIM framework. If you remember the previous example, we installed the Flask Python microframework in the Dockerfile directly with `RUN pip install Flask`. In this example, we'll create a more sophisticated project and install project dependencies (meaning the PHP packages that are needed for our project) using a dependency configuration file.

In case of Composer, we need to describe our dependencies in a file called `composer.json`. This is Composer's convention

as described on <https://getcomposer.org/doc/01-basic-usage.md>. We need to create a file in our `example6` directory and name it `composer.json`.

We can browse Composer packages by clicking “Browse packages” on <https://getcomposer.org/>. This link takes us to <https://packagist.org/>. Here you can search for the Slim package; the details page can be found here <https://packagist.org/packages/slim/slim>.

Let’s add the Slim dependency lines to our `composer.json`.

```
{  
    "require": {  
        "slim/slim": "^3.0"  
    }  
}
```

Now we have a configuration file, and we need a way to install the dependencies that we listed in `composer.json`. To be able to install the dependencies, we need an environment that has PHP and Composer installed.

We can achieve this by taking a PHP parent image and install Composer as described on <https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx>. So let’s do this.

First let’s review the available PHP image variants on the Docker Hub in the official PHP repository,

https://hub.docker.com/_/php/. At the time of writing, the latest version of the PHP image is 7.2.2, and there are different variants available. We would need more than running a PHP script, which would be easily feasible with the images tagged `cli`, but we are going to build a web page, so we would need something with a web server.

The images tagged `apache` include an Apache web server as their main process, you can check this in the Dockerfile on the Docker Hub (<https://github.com/docker-library/php/blob/fd8e15250a0c7667b161c34a25f7916b01f72367/7.2/stretch/apache/Dockerfile>). Checking this Dockerfile, you can also see that the web server runs on port 80 and the working directory is `/var/www/html` so this is where we will copy our project code files.

Let's use `php:7.2.2-apache` for this example. Let's create the `Dockerfile` and let's start building:

```
FROM php:7.2.2-apache
```

The next step is to install Composer. The steps described on <https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx> describe the manual installation, we'll use the programmatic installation steps described here: <https://getcomposer.org/doc/faqs/how-to-install-composer-programmatically.md>. We translate these steps into one `RUN` instruction:

```
RUN EXPECTED_SIGNATURE=$(wget -q -O - https://composer.github.io/composerSetup.php | sed -e "s/\\r\\n/\\n/g") && \n    php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" && \n    $EXPECTED_SIGNATURE=$(php -r "echo hash_file('SHA384', 'composer-setup.php');") && \n    if [ \"$EXPECTED_SIGNATURE\" != \"$ACTUAL_SIGNATURE\" ]; then >&2\n        rm composer-setup.php && \\n        mv composer.phar /usr/local/bin/composer
```

We make our Composer installation global by adding the below line to the Dockerfile at the end of the `RUN` instruction (as described on <https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx>):

```
mv composer.phar /usr/local/bin/composer
```

Once Composer is installed we'll need to copy `composer.json` into our Docker image and run `composer install`. Our entire `Dockerfile` should look like this:

```
FROM php:7.2.2-apache\n\nRUN apt-get update && apt-get install -y \\n    wget \\n    && rm -rf /var/lib/apt/lists/*\n\nRUN EXPECTED_SIGNATURE=$(wget -q -O - https://composer.github.io/composerSetup.php | sed -e "s/\\r\\n/\\n/g") && \\n    php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" && \\n    $EXPECTED_SIGNATURE=$(php -r "echo hash_file('SHA384', 'composer-setup.php');") && \\n    if [ \"$EXPECTED_SIGNATURE\" != \"$ACTUAL_SIGNATURE\" ]; then >&2\n        rm composer-setup.php && \\n        mv composer.phar /usr/local/bin/composer
```

```
then >&2 echo 'ERROR: Invalid installer signature' ; \
rm composer-setup.php; exit 1; fi && \
php composer-setup.php --quiet && \
rm composer-setup.php && \
mv composer.phar /usr/local/bin/composer

COPY ./composer.json /var/www/html/

RUN composer install
```

Let's build it with the following command (please feel free to change the tag from `takacsmark` to your Docker username:

```
~/Development/example6$ docker build \
--tag takacsmark/example6:1.0 .
```

If you build the image now, you'll get errors. I know, I know, I could have done the final Dockerfile immediately, but building Docker images is usually a repetitive process in real life, so let's fix these issues now:

1. Don't run composer as user root/super user is a warning, but let's take care of it. The best place to start is to check what user is used in the Dockerfile is at <https://github.com/docker-library/php/blob/fd8e15250a0c7667b161c34a25f7916b01f72367/7.2/stretch/apache/Dockerfile>. We can learn here, that the default user in the PHP image is `www-data`, so let's use this user in the Dockerfile to run the installation. To run

apache as a non-root user, we'll need to change the apache, configuration, too.

2. Git is not installed; therefore Composer installation fails. We can fix this by installing git in the Dockerfile.
3. The zip extension and unzip command are both missing, skipping - we can fix this issue by installing zip with apt-get install.

Let's update the Dockerfile to look like this:

```
FROM php:7.2.2-apache
RUN apt-get update && apt-get install -y \
    git \
    wget \
    zip \
    && rm -rf /var/lib/apt/lists/*

RUN EXPECTED_SIGNATURE="$(wget -q -O \
    - https://composer.github.io/installer.sig)\" && \
php -r \"copy('https://getcomposer.org/installer', \
    'composer-setup.php');\" && \
ACTUAL_SIGNATURE="$(php -r "echo hash_file('SHA384', \
    'composer-setup.php');")\" && \
if [ \"$EXPECTED_SIGNATURE\" != \"$ACTUAL_SIGNATURE\" ]; \
    then >&2 echo 'ERROR: Invalid installer signature' ; \
    rm composer-setup.php; exit 1; fi && \
php composer-setup.php --quiet && \
rm composer-setup.php && \
mv composer.phar /usr/local/bin/composer

RUN sed -i 's/VirtualHost *:80/VirtualHost *:8080/' \
    /etc/apache2/sites-enabled/000-default.conf && \
sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf
```

```
USER www-data

COPY ./composer.json /var/www/html/

RUN composer install
```

Note that I added the `sed` commands to replace the port numbers in the Apache configuration files so that Apache will be started on port 8080. We need this because only the root user has permissions to bind to the default port 80.

I used the Linux command called `sed`, which is a stream editor, you can use it to edit files inline in the Dockerfile, you can read more about it here:

<https://en.wikipedia.org/wiki/Sed>.

For simple projects, you can use a command like `sed` to edit configuration files in place in your Dockerfile. For complex configuration tasks, it's better to create your own configuration files in your source directory and copy them into the Docker image during the build.

Building this image will give one more warning, namely “ Cannot create cache directory `/var/www/.composer/cache/repo/https---packagist.org/`, or directory is not writable. Proceeding without cache ”. This means that Composer cannot write package data into a cache folder in our image.

We want to keep the image size small, so it is a good thing that Composer does not write cache files into our image. We could leave it as it is, because the behavior does not cause a bug, but we will do this correctly.

Since `composer install` does not have a `--no-cache` option or similar, we need to add a couple of steps to fix it. We'll reconfigure composer to use a cache directory where the `www-data` user has write access. We specify this directory in `composer.json` like this:

```
{  
    "config": {  
        "cache-dir": "/var/www/html/composer-cache"  
    },  
    "require": {  
        "slim/slim": "^3.0"  
    }  
}
```

Please note that I have given the directory the name `composer-cache`, but you can give it any name you like. The main point is that the user `www-data` needs to have write access to the directory where we create our cache directory.

The `composer install` step in the Dockerfile will automatically use the new cache location because we specified it in `composer.json`. We don't want that cache files hang around in our Docker image, so we change our `Dockerfile` to remove the cache directory right after `composer install`, our `Dockerfile` looks as follows:

```
FROM php:7.2.2-apache
RUN apt-get update && apt-get install -y \
    git \
    wget \
    zip \
    && rm -rf /var/lib/apt/lists/*

RUN EXPECTED_SIGNATURE=$(wget -q -O \
    - https://composer.github.io/installer.sig) && \
php -r "copy('https://getcomposer.org/installer', \
'composer-setup.php');" && \
ACTUAL_SIGNATURE=$(php -r "echo hash_file('SHA384', \
'composer-setup.php');")" && \
if [ "$EXPECTED_SIGNATURE" != "$ACTUAL_SIGNATURE" ]; \
then >&2 echo 'ERROR: Invalid installer signature' ; \
rm composer-setup.php; exit 1; fi && \
php composer-setup.php --quiet && \
rm composer-setup.php && \
mv composer.phar /usr/local/bin/composer

RUN sed -i 's/VirtualHost \*:80/VirtualHost \*:8080/' \
    /etc/apache2/sites-enabled/000-default.conf && \
sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf

USER www-data

COPY ./composer.json /var/www/html/

RUN composer install && \
    rm -rf /var/www/html/composer-cache
```

The Slim environment is ready with this step. We can start building our Slim web application now, let's implement the getting started example from

<https://www.slimframework.com/>. Please create a file called `index.php` in your project directory and add the following:

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require 'vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Response $response) {
    $name = $args['name'];
    $response->getBody()->write("Hello, $name");

    return $response;
});
$app->run();
```

This code is similar to the Flask code we used in the previous example, in the sense that we rely on an app instance (`$app`) that represents our Slim application and we use a routing mechanism (`$app->get(...)`) to respond to HTTP request.

The line `require 'vendor/autoload.php';` loads all the dependencies automatically that we installed with Composer. This file was generated by Composer during `composer install` and is very useful when you work with PHP and Composer.

We'll add `./index.php` to the already existing `COPY` instruction in the Dockerfile in a minute.

To make routing work, we need to tell the Apache web server not to serve requests based on the directory structure, but to let the Slim routing mechanism handle the URL. For that, we need to create a `.htaccess` file in our project directory as described in the web server section of the Slim user guide here:

<https://www.slimframework.com/docs/v3/start/web-servers.html>.

Please create a file called `.htaccess` in the project directory, note that the filename starts with a `.` on purpose. Copy the contents into the file from the user guide:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ index.php [QSA,L]
```

We need to enable URL rewriting in Apache, we can do this from the command line with `a2enmod rewrite`, so we add this command to the `Dockerfile`, and we copy `index.php` and `.htaccess` to the image, as well:

```
FROM php:7.2.2-apache
RUN apt-get update && apt-get install -y \
    git \
    wget \
    zip \
    && rm -rf /var/lib/apt/lists/*
RUN a2enmod rewrite
```

```
RUN EXPECTED_SIGNATURE=$(wget -q -O \  
    - https://composer.github.io/installer.sig) && \  
  php -r "copy('https://getcomposer.org/installer', \  
    'composer-setup.php');" && \  
  ACTUAL_SIGNATURE=$(php -r "echo hash_file('SHA384', \  
    'composer-setup.php');") && \  
  if [ "$EXPECTED_SIGNATURE" != "$ACTUAL_SIGNATURE" ]; \  
  then >&2 echo 'ERROR: Invalid installer signature' ; \  
  rm composer-setup.php; exit 1; fi && \  
  php composer-setup.php --quiet && \  
  rm composer-setup.php && \  
  mv composer.phar /usr/local/bin/composer

RUN sed -i 's/VirtualHost *:80/VirtualHost *:8080/' \  
    /etc/apache2/sites-enabled/000-default.conf && \  
  sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf

USER www-data

COPY ./composer.json index.php .htaccess /var/www/html/

RUN composer install && \  
  rm -rf /var/www/html/composer-cache
```

Let's re-build and run our image.

```
~/Development/example6$ docker build \  
--tag takacsmark/example6:1.0 .

~/Development/example6$ docker container run --rm \  
-p 80:8080 takacsmark/example6:1.0
```

Now visiting <http://localhost/hello/mark> should produce this result:



It is a best practice to isolate your public HTML code from the rest of the application, so let's move our `index.html` under a subdirectory called `public`. Please create a directory with the name `public` and move your `index.html` and `.htaccess` files under the `public` directory.

My project structure is like this right now:

```
~/Development/example6$ tree -a
.
├── composer.json
├── Dockerfile
└── public
    ├── .htaccess
    └── index.php
```

Let's update the path in our `index.php` where we require the `vendor/autoload.php` file because the file's relative location had changed when we moved `index.php`. Our updated `index.php` is like this:

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;
```

```
require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Response $response) {
    $name = $args['name'];
    $response->getBody()->write("Hello, $name");

    return $response;
});
$app->run();
```

Let's tell the Apache web server that the document root of our application is at `/var/www/html/public`. We can do this by adding one line to the Dockerfile. We also need to change the `Dockerfile` to copy the public directory into the image like this:

```
FROM php:7.2.2-apache

RUN apt-get update && apt-get install -y \
    git \
    wget \
    zip \
&& rm -rf /var/lib/apt/lists/*

RUN a2enmod rewrite

RUN EXPECTED_SIGNATURE=$(wget -q -O \
    - https://composer.github.io/installer.sig)" && \
php -r "copy('https://getcomposer.org/installer', \
'composer-setup.php');" && \
ACTUAL_SIGNATURE=$(php -r "echo hash_file('SHA384', \
'composer-setup.php');")" && \
if [ "$EXPECTED_SIGNATURE" != "$ACTUAL_SIGNATURE" ]; \
```

```
then >&2 echo 'ERROR: Invalid installer signature' ; \
rm composer-setup.php; exit 1; fi && \
php composer-setup.php --quiet && \
rm composer-setup.php && \
mv composer.phar /usr/local/bin/composer

RUN sed -i 's/DocumentRoot.*$/DocumentRoot \
    \'/var/www/html/public/' \
    /etc/apache2/sites-enabled/000-default.conf && \
sed -i 's/VirtualHost *:80/VirtualHost *:8080/' \
    /etc/apache2/sites-enabled/000-default.conf && \
sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf

USER www-data

COPY ./composer.json /var/www/html/

RUN composer install && \
rm -rf /var/www/html/composer-cache

COPY ./public /var/www/html/public
```

The line `RUN sed -i 's/DocumentRoot.*$/DocumentRoot \'/var/www/html/public/' /etc/apache2/sites-enabled/000-default.conf` will replace the line `Documentroot.*` with `DocumentRoot \'/var/www/html/public/` in the file `/etc/apache2/sites-enabled/000-default.conf` which is the default configuration file of the Apache web server.

Now let's rebuild and run our image:

```
~/Development/example6$ docker build \
--tag takacsmark/example6:1.0 .
```

```
~/Development/example6$ docker container run --rm \
-p 80:8080 takacsmark/example6:1.0
```

We get the same results from our restructured project.

Congratulations! You have built a Docker image that takes a project package configuration `composer.json` and uses this file to install project dependencies into the Docker image. You'll see that this pattern comes back with other technologies like Python or Node.js.

Create a generic base image for a Composer based project with ONBUILD

Let's see now how we can create a more generic image that we can reuse with various Composer based projects. The idea is to move common tasks like installing composer, enabling mod rewrite and installing Linux packages into a parent image and extend it later for other projects.

We'll use the `ONBUILD` instruction in the Dockerfile to execute context specific steps when the project image is built.

We can see more and more `ONBUILD` image variants being deprecated in Docker Hub repositories. Their usage is discouraged, and they will be phased out. The use of Python `ONBUILD` variants is discouraged, this is the case with the

official Node.js `ONBUILD` variants, too. Docker gives the following explanation:

While the `onbuild` variant is really useful for "getting off the ground running" (zero to Dockerized in a short period of time), it's not recommended for long-term usage within a project due to the lack of control over when the `ONBUILD` triggers fire (see also [docker/docker#5714](#), [docker/docker#8240](#), [docker/docker#11917](#)).

I think it's a good idea to give `ONBUILD` to your toolset, but make sure you use it more as an experimental feature.

I'll call this `example7`. We'll extend `example6`, so create a copy of `example6` with the name `example7`.

Let's see how to build the base image. We extract every generic step into a new Dockerfile that we call `Dockerfile-base`:

```
FROM php:7.2.2-apache

RUN apt-get update && apt-get install -y \
    git \
    wget \
    zip \
&& rm -rf /var/lib/apt/lists/*

RUN a2enmod rewrite

RUN EXPECTED_SIGNATURE=$(wget -q -O \
```

```
- https://composer.github.io/installer.sig)" && \
php -r "copy('https://getcomposer.org/installer', \
'composer-setup.php');" && \
ACTUAL_SIGNATURE=$(php -r "echo hash_file('SHA384', \
'composer-setup.php');")" && \
if [ "$EXPECTED_SIGNATURE" != "$ACTUAL_SIGNATURE" ]; \
then >&2 echo 'ERROR: Invalid installer signature' ; \
rm composer-setup.php; exit 1; fi && \
php composer-setup.php --quiet && \
rm composer-setup.php && \
mv composer.phar /usr/local/bin/composer

RUN sed -i 's/DocumentRoot.*$/DocumentRoot \
\var\www\html\public/' \
/etc/apache2/sites-enabled/000-default.conf && \
sed -i 's/VirtualHost *:80/VirtualHost *:8080/' \
/etc/apache2/sites-enabled/000-default.conf && \
sed -i 's/Listen 80/Listen 8080/' /etc/apache2/ports.conf

USER www-data

ONBUILD COPY ./composer.json /var/www/html/

ONBUILD RUN composer install && \
rm -rf /var/www/html/composer-cache

ONBUILD COPY ./public /var/www/html/public
```

We added the project specific steps at the end of the Dockerfile with `ONBUILD` instructions. This means that these steps will not be executed when this base image is built, they will only fire when we use this image as the parent image of another image. We do this in a minute.

Our Dockerfile has a custom name; it's called `Dockerfile-base` rather than `Dockerfile`. We can build this image with `docker image build` using the `-f` option that lets us specify the Dockerfile to be used for the build. We use this command:

```
~/Development/example7$ docker image build \
--tag takacsmark/example7-base:1.0 -f Dockerfile-base .
```

The build will result in an image called `takacsmark/example7-base :1.0`. We'll use this as the base image in our `Dockerfile`:

```
FROM takacsmark/example7-base:1.0
```

We can build the Dockerfile with the command:

```
~/Development/example7$ docker image build \
--tag takacsmark/example7:1.0 .

Sending build context to Docker daemon 7.168kB
Step 1/1 : FROM takacsmark/example7-base:1.0
# Executing 3 build triggers
---> Running in dbaa9082795f
Do not run Composer as root/super user! See https://getcomposer.org
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 6 installs, 0 updates, 0 removals
- Installing psr/container (1.0.0): Downloading (100%)
- Installing container-interop/container-interop (1.2.0): Downloading (100%)
- Installing nikic/fast-route (v1.3.0): Downloading (100%)
- Installing psr/http-message (1.0.1): Downloading (100%)
- Installing pimple/pimple (v3.2.3): Downloading (100%)
```

```
- Installing slim/slim (3.9.2): Downloading (100%)
Writing lock file
Generating autoload files
Removing intermediate container dbaa9082795f
    ---> eb1035338160
Successfully built eb1035338160
Successfully tagged takacsmark/example7:1.0
```

As you can see, all the steps that were specified with `ONBUILD` in the parent image have been executed during the build of the child image.

This technique assumes that the structure of your project is in line with the structure assumed in the parent image. In this example we assumed the following constraints:

1. Your project has a `composer.json` file in the project directory
2. The source code of your public files is stored in a subdirectory in the project directory, and it's called `public`.

If your project is not laid out like this, your Docker build will result in an error. You must document such constraints in your Docker repository to make your team members aware of the conventions you used in the parent image.

Run the project with the below command.

```
~/Development/example7$ docker run --rm \
-p 80:8080 takacsmark/example7:1.0
```

It will produce the same results as in example 6 (check <http://localhost/hello/Han>).

Node.js, npm, Express, and Pug

I hope that you found the examples so far useful and you're feeling more confident building Docker images and starting Docker containers.

I also hope that you are still patient enough to build a few examples before we change gears towards more complex applications because in this chapter I would like to build an example with you using the popular Node.js platform.

We build a web application using Node.js. We'll make use of the npm package manager, the Express framework and a templating engine to display application information.

On top of all this, we'll do some things we have not done so far in the book; we'll simulate a development scenario and use our image for development automation, we'll generate some of our source files and project structure with a Docker container. In this example, I would like to showcase how you can think of Docker as a Swiss army knife to accomplish any

task in a software building, shipping, and running scenario.

Picking the node image

We start our workflow with the usual task of picking the right Docker image or images for our project. Let's go to the official repository of Node.js on the Docker Hub (https://hub.docker.com/_/node/) and review the options.

We have many tags at our disposal. It's a good idea to check the LTS (long-term support) version number of Node.js on <https://nodejs.org/en/>. Let's stay conservative and not use the latest features, let's go for LTS. The current LTS version is 8.9.4, so we'll look for the related image on the Docker Hub.

Let's be a bit progressive and start with the alpine variant, my tag of choice is `8.9.4-alpine`.

It's a good idea at this stage to explore the Dockerfile of the official image to find out crucial information about any volumes, users, and ports in the image. The Dockerfile is located here: <https://github.com/nodejs/docker-node/blob/8498ff5302c19506c9edc3ce152f2bd4aa6b26b3/8/alpine/Dockerfile>.

Looking at the Dockerfile, we can see that a user called `node` is created with a user id of `1000`, that is specifically hardcoded into the Dockerfile. This is important because

Linux systems maintain file system information based on the user id (uid) not the username.

Create the project files

Let's create a new project directory called `example8` and build up the boilerplate of our project. We'll create the following files to support our project:

- A `package.json` file with the project dependencies.
- A `Dockerfile` for the environment.
- An `index.js` file for the application server code.

You can find an enormous number of packages created for Node.js that you can use to build applications. These packages are available in an online package registry on <https://www.npmjs.com/>. You can use the `npm` package manager to install Javascript packages in your project.

Similarly to Composer's `composer.json`, npm has a file called `package.json`. We use this file to specify properties of our Node.js project, and we use it to define the list of packages to be used in your project. We'll create this `package.json` file now and specify dependencies.

We'll also create a `Dockerfile` to set up the first runtime environment of our example, and we'll add an `index.js` file with the code to start a simple server and display "Hello World!".

We'll use a handy little trick to create `package.json`. Namely, we'll use the node image in shell mode to generate `package.json`. First of all, let's pull the node image with the below command.

```
~/Development/example8$ docker pull node:8.9.4-alpine
```

Start up a container in interactive mode and start building.

```
~/Development/example8$ docker run --rm -ti \
-v $(pwd):/home/node/app node:8.9.4-alpine /bin/sh
/ #
```

We started a node container bind mounting the project directory on the host to `/home/node/app` in the container. We'll use `npm` to generate `package.json` from within the container.

I'm bind mounting the host directory to `/home/node/app` in the container. This was a conscious decision that I've taken. The Dockerfile of the official node image (<https://github.com/nodejs/docker-node/blob/8498ff5302c19506c9edc3ce152f2bd4aa6b26b3/8/alpine/Dockerfile>) shows that a new user is created with the name `node` and I decided to use this user and run my application as `node`, not `root`. The directory `/home/node` is the home directory of the `node` user. Adding a subdirectory called `app` to `/home/node` was my decision, you can give this directory any other name.

The good thing about the home directory of user `node` (`/home/node`) is that this directory is owned by the `node` user and we'll have the necessary permissions to run our application in this directory. So it's a practical choice.

You can use other directories to run your application, but you need to make sure to set ownership and permissions accordingly.

Right now we are sitting in front of a shell in the running node container, and we have the host project directory mapped to `/home/node/app`. This means that whatever file changes we apply to this directory in the container, those changes will be available on the host machine. This is a good setup to generate some of the project files with Node tools.

Let's change to the app directory and initialize our project following the wizard after `npm init`. The command `npm init` is a command line tool to initialize a Node project in a straightforward wizard. You can see my answers to the wizard questions below:

```
/ # cd /home/node/app/  
/home/node/app # npm init
```

This utility will walk you through creating a `package.json` file. It only covers the most common items, and tries to guess sensible

See ``npm help json`` for definitive documentation on these fields and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and save it as a dependency in the package.json file.

Press ^C at any time to quit.

```
package name: (app) example8
version: (1.0.0)
description: Example node app with Docker
entry point: (index.js)
test command:
git repository:
keywords:
author: Mark Takacs
license: (ISC)
About to write to /home/node/app/package.json:
```

```
{
  "name": "example8",
  "version": "1.0.0",
  "description": "Example node app with Docker",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Mark Takacs",
  "license": "ISC"
}
```

Is this ok? (yes) yes

This will generate the `package.json` file in the container and also on the host because we bind mounted the project directory into the container. Let's install Express with npm from the command line like this:

```
/home/node/app # npm install --save express

npm notice created a lockfile as package-lock.json. You should cor
npm WARN example8@1.0.0 No repository field.

+ express@4.16.2
added 49 packages in 6.019s
```

You can ignore the warning about “No repository field.”, this means that we have not specified a git repository for our project in our `package.json`, which is fine for now.

Leaving the container with `exit` will take us back to our host machine, and we can check the contents of our directory and the `package.json` file.

The directory structure looks like this right now:

```
~/Development/example8$ tree -L 1
.
├── node_modules
└── package.json
└── package-lock.json

1 directory, 2 files
```

The `package.json` file has the overall information about our package, and the `package-lock.json` file is the detailed list of all the packages that were installed when we installed the Express framework with `npm install`.

`package-lock.json` contains the entire package tree installed with specific version numbers so that you can reproduce the same package environment when you move your app to another machine. This is what we need in our Docker image for this project.

If you have another look into the directory tree, you'll see that `npm` generated another directory called `node-modules`. This directory is the standard location for packages installed with `npm` in your project. If you list the contents of the directory, you'll find all packages neatly organized into subdirectories under `node-modules`.

The `node-modules` directory is something that we re-generate in every environment. The definition of its contents is stored in `package.json` and `package-lock.json`, these two files are enough to reproduce an identical version of `node-modules`.

For the above reason we do not check-in `node-modules` into source control, plus we do not need to store `node-modules` on our local machine. What we'll do is, that we'll re-generate `node-modules` in our Docker image, we'll add this step to our Dockerfile. Right now we have the directory on the host machine because we installed Express inside the container. We can delete the directory from the host, so let's do it:

```
~/Development/example8$ sudo rm -rf node_modules/
```

My generated `package.json` file is like this right now:

```
{  
  "name": "example8",  
  "version": "1.0.0",  
  "description": "Example node app with Docker",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Mark Takacs",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.16.2"  
  }  
}
```

I will not list `package-lock.json`, because it's too lengthy, please check yours on your machine and see what's inside.

Create the Dockerfile

Let's copy `package.json` and `package-lock.json` into a Docker image and install the dependencies using a Dockerfile with the following code:

```
FROM node:8.9.4-alpine  
  
USER node  
  
RUN mkdir /home/node/app  
WORKDIR /home/node/app  
COPY --chown=node:node package*.json index.js ./  
RUN npm install  
  
EXPOSE 3000
```

```
CMD node index.js
```

Let me explain what we are doing in the Dockerfile:

- `node:8.9.4-alpine` is the parent image.
- We set the user to be `node` with the `USER` instruction, we'll not run our application as root.
- We create the working directory with `RUN mkdir /home/node/app`. The `WORKDIR` instruction would create the directory for us, but if we create the directory with `WORKDIR` it will be owned by root and not the `node` user, which will not give us the right permissions. Creating the directory with `RUN mkdir ...` will make the `node` user the owner of `/home/node/app`.
- We set the working directory with `WORKDIR /home/node/app`.
- We copy the necessary project files into the image with `COPY --chown=node:node package*.json index.js ./`. Note how we are using `chown` again, because `COPY` will give file ownership to `root` by default, so we have to explicitly change ownership to the `node` user.
The file `index.js` is our simple server code that we'll create in the next step, so that file does not exist yet.

- Running `RUN npm install` will take `package.json` and `package-lock.json` and install the packages that we listed under “dependencies”. This step will generate the `node-modules` directory inside the image.
- We expose port 3000, you’ll see in `index.js` that we’ll start our server on port 3000.
- We specify the command of our Docker image with `CMD node index.js`, this means that we will run our `index.js` Javascript code with the `node` command line interface.

Let’s create our `index.js` file with the “Hello World!” sample code from <https://expressjs.com/en/starter/hello-world.html>.

```
const express = require('express')
const app = express()

app.get('/', (req, res) => res.send('Hello World!'))

app.listen(3000, () => console.log('Example app listening on port'))
```

First build and run

We can build our image with the below command.

```
~/Development/example8$ docker image build \
--tag takacsmark/example8:1.0 .
```

Let's run the project for the first time and see what it does:

```
~/Development/example8$ docker container run \
-p 3000:3000 takacsmark/example8:1.0
```

Visiting <http://localhost:3000> in your browser will display the message “Hello World!”.

Generate an Express app

We have set up a simple Node.js project with Docker tools in this example so far. Let's go a step further and use the Express application generator tool,

<https://expressjs.com/en/starter/generator.html>, to generate our application with a template engine.

Let's open a new folder called `example9`. We will create this project in two main steps:

1. First, we'll use a container running in interactive mode to generate the application skeleton.
2. We set up a Dockerfile to define the image to run the project.

The idea is to use Docker as a development tool for source file generation and then use also Docker to run the project. This way you'll use Docker in different roles in the same project.

We are following the steps described on <https://expressjs.com/en/starter/generator.html> with the below logic:

- We install the express generator in our image globally with `npm install express-generator -g`. The `-g` flag tells `npm` to install the package globally, that's why we are running these steps as root.
- We change the working directory to `/home/node`. We use `/home/node` because we'll use the express generator to generate the `app` directory and set up the entire project structure inside `/home/node`.
- The line `express --view=pug app` will create the project directories and files with the Pug template engine and place it in a directory called `app`.

The execution of the above commands worked out for me like this:

```
~/Development/example9$ docker run --rm \
-ti node:8.9.4-alpine /bin/sh

/ # npm install express-generator -g

/usr/local/bin/express -> /usr/local/lib/node_modules/express-gene
+ express-generator@4.15.5
added 6 packages in 1.026s

/ # cd /home/node/
```

```
/home/node # express --view=pug app

create : app
create : app/package.json
create : app/app.js
create : app/public
create : app/routes
create : app/routes/index.js
create : app/routes/users.js
create : app/views
create : app/views/index.pug
create : app/views/layout.pug
create : app/views/error.pug
create : app/bin
create : app/bin/www
create : app/public/javascripts
create : app/public/stylesheets
create : app/public/stylesheets/style.css

install dependencies:
$ cd app && npm install

run the app:
$ DEBUG=app:* npm start

create : app/public/images
```

We created the project structure inside the container. The only thing we have to do now is to copy the source files to our host machine. Let's use `docker cp` in a new Terminal window to do this (please find the name of your running container with `docker container ls`, mine is called `hungry_beaver`):

```
~/Development/example9$ docker cp \  
hungry_beaver:/home/node/app app
```

Docker `cp` is a utility that copies files and folder between containers and the host file system. As a result, you'll see the `app` directory copied to your host machine into the project directory. The first part of our work is finished, we have generated our source code base in a few easy steps.

Let's create a Dockerfile to run our code in a container. Our `example9` directory has a subdirectory right now that's called `app`, we shall place our Dockerfile at the same level right into the `example9` folder like this:

```
~/Development/example9$ tree -L 1  
.  
└── app  
└── Dockerfile
```

Let's add the following code to our `Dockerfile`:

```
FROM node:8.9.4-alpine  
  
USER node  
  
WORKDIR /home/node  
COPY --chown=node:node app app  
  
WORKDIR /home/node/app  
RUN npm install
```

```
EXPOSE 3000
```

```
CMD npm start
```

In our Dockerfile we are using the `node` user to execute commands, we copy our source code from the `app` directory on the host machine to `/home/node/app` in the container. We change the working directory to `/home/node/app` and run `npm install` to install the dependencies listed in our `package.json` file.

We expose port 3000 because our sample code runs on port 3000, you can read this information in the file called `app/bin/www`.

Reading the `script` section in `app/package.json` you can also find out that this project has a start script that we can run with `npm start`; hence this is the command we specify as `CMD` in the Dockerfile.

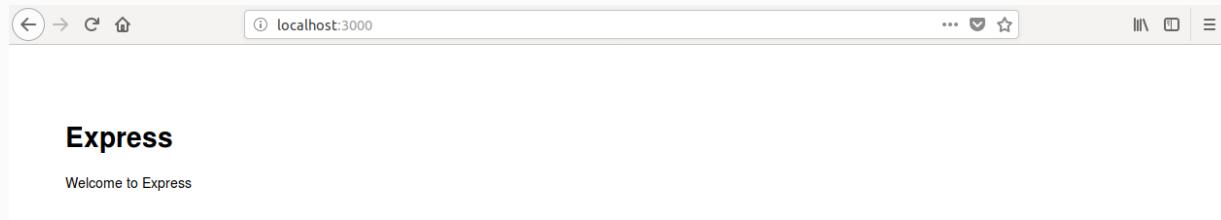
It's time to build our project and test drive it with the command:

```
~/Development/example9$ docker build \  
--tag takacsmark/example9:1.0 .
```

You can check the end result by starting a container like this:

```
~/Development/example9$ docker run --rm \
-p 3000:3000 takacsmark/example9:1.0
```

Visiting <http://localhost:3000> will give you the below result:



Let's have a look at our project directory and see what's inside:

```
~/Development/example9$ tree -L 3
```

```
.
```

```
├── app
│   ├── app.js
│   ├── bin
│   │   └── www
│   ├── package.json
│   ├── public
│   │   ├── images
│   │   ├── javascripts
│   │   └── stylesheets
│   ├── routes
│   │   ├── index.js
│   │   └── users.js
│   └── views
│       ├── error.pug
│       ├── index.pug
│       └── layout.pug
└── Dockerfile
```

The directory structure reflects all the key components of the web application framework, including routes, views, public components and the entry point of the application under `bin/www`.

Looking at the source code you'll see that the example serves another route apart from <http://localhost:3000>, you can try <http://localhost:3000/users> in your browser, too.

Set up for development

In the current setup, we have to re-build our Docker image after every code change. We need a more flexible setup for development. We have to be able to see our changes as we work.

Let's start up a container for development!

We need to consider a couple of points to make this work:

1. We need a mechanism to reflect changes in our code files in the running server. There is a handy npm package to do this; it's called `nodemon` (<https://github.com/remy/nodemon>). Nodemon will monitor the source files and restart the server when a file is changed. We install `nodemon` globally into our Docker container.
2. We enable debug messages by setting an environment variable called `DEBUG`. We'll do this

in our Docker container.

3. We need to start our Docker container with the `nodemon` command to monitor changes.
4. We need to bind mount our source files from the host machine into our development container, to be able to edit the code and run the server at the same time.

Considering all of the above, it seems to be reasonable to create a dedicated Docker image for development. We'll restructure our project to reflect this change. Let's create a directory called `docker` in our project folder and create two subdirectories called `production` and `development`. We'll create a Dockerfile in each of these subdirectories. It'll look like this:

```
~/Development/example9$ tree -L 3
.
├── app
└── docker
    ├── development
    │   └── Dockerfile
    └── production
        └── Dockerfile
```

Let's move our current Dockerfile into `docker/production` with the command:

```
~/Development/example9$ mv Dockerfile docker/production/
```

This means that the Dockerfile we created so far will be our production Dockerfile. We'll use this image as the parent image in our development Dockerfile. Let's create the file `docker/development/Dockerfile` with the following code:

```
FROM takacsmark/example9:1.0

USER root
RUN npm install -g nodemon

USER node

ENV DEBUG="app:* npm start"

CMD nodemon bin/www
```

We installed `nodemon` globally as the `root` user, then we changed the user back to `node` and set the environment variable `DEBUG` as described in the express generator documentation,

<https://expressjs.com/en/starter/generator.html>.

We have changed the structure of our project, which means that we need to update the paths in our production Dockerfile to reflect the new directory structure like this:

```
FROM node:8.9.4-alpine
```

```
USER node

WORKDIR /home/node
COPY --chown=node:node ../../app app

WORKDIR /home/node/app
RUN npm install

EXPOSE 3000

CMD npm start
```

Note that I have updated the line with the `COPY` instruction to copy our app from `../../app` which is two levels higher in the directory hierarchy compared to the location of the Dockerfile (`docker/production/Dockerfile`).

We need to fine-tune our `docker build` command to reflect the changes. We'll keep building our project from the project directory, in my case `~/Development/example9`. The reason for this is that we need to have our `app` folder in the context of our Docker build because we are copying the contents of the `app` folder into our image.

We cannot run our build from the `docker/production` subdirectory, because this way the `app` directory would be outside the build context.

We'll use the `-f` flag of `docker build` to specify the location of the Dockerfile we want to build. Our command will look like this:

```
~/Development/example9$ docker build \
--tag takacsmark/example9:dev -f docker/development/Dockerfile .
```

Please note how I tagged this image `:dev` to note that this is the development variant.

Now we need to bind mount our source code into the container when we run it. We have done bind mounting before, so this task seems to be easy, but there is a catch.

The first thing that comes to our mind is to mount the `app` directory on the host into `/home/node/app` in the container, so all our source files will be mounted into the container and it should be all OK. Let's try what happens:

```
~/Development/example9$ docker run --rm -p 3000:3000 \
-v $(pwd)/app:/home/node/app takacsmark/example9:dev

[nodemon] 1.17.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node bin/www`
module.js:540
    throw err;
^

Error: Cannot find module 'express'
    at Function.Module._resolveFilename (module.js:538:15)
    at Function.Module._load (module.js:468:25)
    at Module.require (module.js:587:17)
    at require (internal/module.js:11:18)
    at Object.<anonymous> (/home/node/app/app.js:1:77)
    at Module._compile (module.js:643:30)
```

```
at Object.Module._extensions..js (module.js:654:10)
at Module.load (module.js:556:32)
at tryModuleLoad (module.js:499:12)
at Function.Module._load (module.js:491:3)
[nodemon] app crashed - waiting for file changes before starting.
```

We get an error that says; “Cannot find module express”. This is because our `app` directory on the host contains all of our source files, but it does not contain the `node_modules` directory. Therefore node cannot find Express when starting up the server.

The `node_modules` directory is created when we run `npm install` in our image. We have the `node_modules` directory inside our image; we used `npm install` in our Dockerfile to create it.

However, the `node_modules` directory in the container is currently not visible, because we have overridden the contents of `/home/node/app` with the bind mount.

We can explore two solutions here:

1. Find a way to re-mount the underlying `/home/node/app/node_modules` directory from the image to the container. There is a workaround for this, that I'll show in a minute.
2. Find a way to install packages and generate `node_modules` on the host, too. We'll check out this solution too.

The handy workaround to re-mount the `node_modules` directory is the following:

```
~/Development/example9$ docker run --rm -p 3000:3000 \
-v $(pwd)/app:/home/node/app \
-v /home/node/app/node_modules takacsmark/example9:dev
```

What we did here is that we added another volume that points to `/home/node/app/node_modules` in the Docker image. This will mount the directory from the underlying image layer into our container. You can notice as a side effect, that the `node_modules` directory also becomes visible on your host machine under `app`, but it seems to be empty, meaning you cannot list directories and files underneath.

In the container, however, the full contents of the `node_modules` directory will be available. Your server is running in development mode now. You can use a code editor to change any file in your project, and your changes will be reflected in the browser.

You can, for example, change the file `app/routes/users.js`, to send a different message by changing the line:

```
res.send('respond with a my cool message');
```

Once you save the file, you'll see the below output from the container:

```
[nodemon] restarting due to changes...
[nodemon] starting `node bin/www`
Tue, 06 Mar 2018 15:59:46 GMT app:server Listening on port 3000
```

Refresh the browser to see the changes on <http://localhost:3000/users>. My output has changed to the following:



Let's see the other option to make our bind mount work. We will run our container with the original command (`docker run --rm -p 3000:3000 -v $(pwd)/app:/home/node/app takacsmark/example9:dev`), but before we do that, we'll install node modules into the host project folder with the command:

```
~/Development/example9$ docker run --rm \
-v $(pwd)/app:/home/node/app takacsmark/example9:dev npm install
```

With this command, we are mounting our `app` folder and install the packages with `npm install` as user root. The solution will work, you can test it by running

```
~/Development/example9$ docker run --rm -p 3000:3000 \
-v $(pwd)/app:/home/node/app takacsmark/example9:dev
```

Compiled binary based on scratch in multi-stage build

We have come to the last exercise in our image building scenarios. Going through the examples so far you have learned a lot about Docker image building practices.

In this example, I'll show you a compelling use case that gives you insight into the nature of Docker containers.

We'll build two Docker images from `scratch`, the empty Docker image. We'll not take any existing Docker image as a parent image in our Dockerfile but will start from nothing.

Why is this interesting? In the previous examples, we always started from an existing image like `php`, `node` or `alpine`. These parent images already had programs and configuration files in their file system and we built our environments on top of these tools.

What happens behind the scenes in Docker is, that the parent images contain the user and system programs, but they do not contain the Linux kernel. Every Docker container uses the kernel of the host machine to run applications.

The kernel is the core of your operating system; it's a program that controls your computer hardware including CPU and memory management. You can read more about

the kernel on [Wikipedia](#), let me quote the part that is the most relevant for us now:

Role of the kernel

The kernel is the most fundamental part of an operating system. It can be thought of as the program which controls all other programs on the computer. When the computer starts, it goes through some initialization (booting) function, such as checking memory. It is responsible for assigning and unassigning memory space which allows software to run.

It provides services so programs can request the use of the network card, the disk or other piece of hardware (the kernel forwards the request to special programs called device drivers which control the hardware), manages the file system and sets interrupts for the CPU to enable multitasking. Many kernels are also responsible for ensuring that faulty programs do not interfere with the operation of others, by denying access to memory that has not been allocated to them and restricting the amount of CPU time they can consume.

You can find the Linux kernel on a Linux host under the `/boot` directory, here is the kernel on my Ubuntu host:

```
~/Development$ ll /boot/
```

```
total 376744
drwxr-xr-x  3 root root    4096 febr 23 14:11 .
drwxr-xr-x 24 root root    4096 febr 23 14:10 ..
-rw-r--r--  1 root root 1500286 okt 11 19:33 abi-4.13.0-16-gene
-rw-r--r--  1 root root 1500314 nov  1 16:17 abi-4.13.0-17-gene
-rw-r--r--  1 root root 1500387 dec  4 12:27 abi-4.13.0-19-gene
-rw-r--r--  1 root root 1500049 jan  8 20:13 abi-4.13.0-25-gene
-rw-r--r--  1 root root 1500342 jan 23 09:13 abi-4.13.0-32-gene
-rw-r--r--  1 root root 1501359 febr 16 18:49 abi-4.13.0-36-gene
-rw-r--r--  1 root root 213028 okt 11 19:33 config-4.13.0-16-gene
-rw-r--r--  1 root root 213028 nov  1 16:17 config-4.13.0-17-gene
-rw-r--r--  1 root root 213040 dec  4 12:27 config-4.13.0-19-gene
-rw-r--r--  1 root root 213156 jan  8 20:13 config-4.13.0-25-gene
-rw-r--r--  1 root root 213156 jan 23 09:13 config-4.13.0-32-gene
-rw-r--r--  1 root root 213212 febr 16 18:49 config-4.13.0-36-gene
drwxr-xr-x  5 root root    4096 márc 4 15:36 grub/
-rw-r--r--  1 root root 50730511 febr 13 09:01 initrd.img-4.13.0-16-gene
-rw-r--r--  1 root root 50725779 febr 13 09:01 initrd.img-4.13.0-17-gene
-rw-r--r--  1 root root 50727781 febr 13 09:01 initrd.img-4.13.0-19-gene
-rw-r--r--  1 root root 50727429 febr 13 09:00 initrd.img-4.13.0-25-gene
-rw-r--r--  1 root root 50727486 febr 13 09:00 initrd.img-4.13.0-32-gene
-rw-r--r--  1 root root 50951611 febr 23 14:11 initrd.img-4.13.0-36-gene
-rw-r--r--  1 root root 182704 jan  28 2016 memtest86+.bin
-rw-r--r--  1 root root 184380 jan  28 2016 memtest86+.elf
-rw-r--r--  1 root root 184840 jan  28 2016 memtest86+_multiboot
-rw-r--r--  1 root root 2860 febr 16 18:49 retpoline-4.13.0-16-gene
-rw-----  1 root root 3878463 okt 11 19:33 System.map-4.13.0-16-gene
-rw-----  1 root root 3881480 nov  1 16:17 System.map-4.13.0-17-gene
-rw-----  1 root root 3881597 dec  4 12:27 System.map-4.13.0-19-gene
-rw-----  1 root root 3876016 jan  8 20:13 System.map-4.13.0-25-gene
-rw-----  1 root root 3877797 jan 23 09:13 System.map-4.13.0-32-gene
-rw-----  1 root root 3880918 febr 16 18:49 System.map-4.13.0-36-gene
-rw-r--r--  1 root root 7812896 okt 29 09:36 vmlinuz-4.13.0-16-gene
-rw-----  1 root root 7816976 nov  1 16:17 vmlinuz-4.13.0-17-gene
-rw-----  1 root root 7821072 dec  4 12:27 vmlinuz-4.13.0-19-gene
-rw-----  1 root root 7829264 jan  8 20:13 vmlinuz-4.13.0-25-gene
```

```
-rw----- 1 root root 7833360 jan 23 09:13 vmlinuz-4.13.0-32
-rw----- 1 root root 7870224 febr 16 18:49 vmlinuz-4.13.0-36
```

We can find the kernel file on this list by checking the kernel in use with the following command:

```
~/Development$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.13.0-36-generic root=UUID=4890bc8e-1e7e
```

This means that on my Ubuntu machine the system is using the `/boot/vmlinuz-4.13.0-36-generic` kernel.

Let's check the kernel in a running container now, to see the relationship between the host kernel and the container kernel. Let's start up an alpine container:

```
~/Development$ docker run --rm -ti alpine:3.7 sh
/ #
```

Let's check whether the container has the `/boot` directory:

```
/ # ls -al /boot
ls: /boot: No such file or directory
```

There is no `/boot` directory in our container. Let's check what kernel is in use in the container:

```
/ # cat /proc/cmdline  
  
BOOT_IMAGE=/boot/vmlinuz-4.13.0-36-generic root=UUID=4890bc8e-1e7e
```

We learned that containers running on a Linux host will use the Linux kernel of the host machine. We've seen how an Alpine container uses the kernel of the Ubuntu host. I pulled down a Debian image to show you that it works the same way:

```
~/Development$ docker run --rm -ti debian:stretch /bin/bash  
  
root@73b93bbd4d14:/# cat /proc/cmdline  
  
BOOT_IMAGE=/boot/vmlinuz-4.13.0-36-generic root=UUID=4890bc8e-1e7e
```

Now you might ask what is inside the containers? We know the kernel is not included, we are using the kernel of the host machine. What is really included in Docker images and containers are the system programs, user programs and configuration files that come with Linux distributions and technology stacks.

This means that we can create an empty container, meaning with literally no files inside, and copy a single executable binary program inside and run it. This way we can create a minimalist container that contains a single executable program.

We will also learn how to do a multi-stage build, in this example. Let's get started.

We will create a compiled and executable program. You can compile your code in various programming languages with various tools. First, we'll use the C programming language to create a binary.

I created a directory under my development directory and called it `example-cscratch`, because we'll code in C and we'll run our program in a Docker container built from scratch, literally using `FROM scratch`, the empty image, as the parent image in our Dockerfile.

Let's add a file called `example.c` to our project with the following code:

```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello %s!\n", argv[1]);
    return 0;
}
```

We include a header file called `stdio.h`, so that we can use `printf`. We have one function called `main`, the main function of our program.

Our program accepts command line arguments, and it will print the first argument that it receives. So that you can

pass your name to this program, and it will print “Hello <your_name>”. We return with 0, which is the return code of successful execution.

We need to compile this code now into an executable program that can run standalone. We’ll use Docker to do this. We’ll set up our first multi-stage build in a Dockerfile.

A multi-stage build in Docker means that we have more than one `FROM` instructions in the Dockerfile. This means that in every build stage in the Dockerfile, we build up a Docker image for a particular build task, we can then copy the end result from a previous stage to the next stage.

The whole Docker build will result in one final Docker image from the last build stage, and all the previous stages will be discarded, they were only used for the build. An example may shed some light on this process:

```
FROM alpine:3.7 as builder

WORKDIR /root/
COPY example.c .

RUN apk --update --no-cache add build-base && \
    gcc -static -o example example.c

FROM scratch

COPY --from=builder /root/example /
ENTRYPOINT ["/example"]
CMD ["World"]
```

We use the above Dockerfile to build and run our C program. The first stage is using an Alpine image with build-base tools to build the binary. Please note that we named this stage `builder` in the `FROM` instruction like this: `FROM alpine:3.7 as builder`.

We use `gcc` to build our source code with the `-static` option so that our compiled binary file contains all dependencies and can run standalone as a single executable file in our container.

We want to make our final image as small as possible, so we won't need Alpine programs and the build tools (the build-base Linux packages) to run our program. We need a single compiled file in the final image.

We can achieve this by building an image for execution from the empty image `scratch`. We start building this image on the line `FROM scratch`.

The key to our multi-stage build is the line where we copy the end result from the builder stage to the final image like this: `COPY --from=builder /root/example /`. This line is pretty self-explanatory, it copies from the builder stage the file `/root/example`, which is our compiled binary to the root (`/`) of the image that we are building from scratch.

All we need to do now is to set our binary as the entry point, and we also give a default argument with the `CMD`

instruction.

We can build this image with one command:

```
~/Development/example-cscratch$ docker build \
--tag takacsmark/example-cscratch:1.0 .
```

Listing our image on the host machine, you can see that we managed to build a minimal image that's 120kB in size.

```
~/Development/example-cscratch$ docker image ls
```

REPOSITORY	TAG	IMAGE ID
takacsmark/example-cscratch	1.0	9f6260a93eaa

We can run our program with Docker like this:

```
~/Development/example-cscratch$ docker run \
--rm takacsmark/example-cscratch:1.0
```

Hello World!

Alternatively, we can run it with an argument:

```
~/Development/example-cscratch$ docker run \
--rm takacsmark/example-cscratch:1.0 Mira
```

Hello Mira!

I gave you this example to learn three things:

1. Docker containers share the Linux kernel of the host machine.
2. You can create minimal images that contain only one executable program and nothing else; this may be useful at some point.
3. Use multi-stage builds to minimize image size and create a smooth build workflow.

Scratch example with Golang

Let's quickly create another multi-stage build with the popular Go programming language.

Let's create a new directory called `example-goscratch` and let's use the official getting started example code in a file called `example.go` :

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello World!\n")
}
```

Let's create the multi-stage build in our Dockerfile:

```
FROM golang:1.10.0 as builder

WORKDIR /go/src/app
COPY example.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o example

FROM scratch

COPY --from=builder /go/src/app/example /
ENTRYPOINT ["/example"]
```

Let's build it:

```
~/Development/example-goscratch$ docker build \
--tag takacsmark/example-goscratch:1.0 .
```

And run it:

```
~/Development/example-goscratch$ docker run \
--rm takacsmark/example-goscratch:1.0
```

```
Hello World!
```

This is how easy it is to get started with Golang using Docker.

Build multi-container applications

Most applications consist of a number of different components that work together to provide the required functionality. In most projects, you'll have more than one Docker images for your various components, like a database, a back-end server, and a front-end server, for example.

When you start building up your project, you usually depict the architecture and define how components work together.

Based on what we have learned so far in the book, you can identify and create Docker images for your individual components.

In this chapter, you'll learn how to use Docker tools, to create complex applications that are composed of different containers and how you can define the interoperability of these containers.

Docker Compose

We have already run several containers in the book, and I'm sure you noticed that we usually define similar runtime properties every time we start a service. We usually specify port mappings, volumes, the image that we use, commands, environment variables and other options.

I'm sure you agree, that we don't want to manually enter parameters every time we start up a new container. So we need a way to define the runtime parameters of our containers that we can use over and over again to start up the same container the same way.

This is when Docker Compose comes into the picture. Docker Compose is a command line tool to define and run multi-container applications. You can specify your application services and their behavior in a file called Compose file or `docker-compose.yml`.

In `docker-compose.yml` you'll have one service per functional component you use in your project. One service can use one Docker image to run containers. So if you use, for example, three images for front-end, back-end, and database respectively, you should define three services in your `docker-compose.yml`.

Once you defined your application in `docker-compose.yml`, you can use the `docker-compose` command to start up and manage your application. To start up your containers, you need to issue a single command called `docker-compose up`.

Your build workflow looks like this:

1. Define and build the right images for your application using the practices you learned in the previous chapters.
2. Define your application services and their interoperability in `docker-compose.yml`.
3. Use `docker-compose up` and further `docker-compose` commands to manage your application.

You can use the `docker-compose` command to start up multiple containers that work together. You can use `docker-compose` in every stage of development including build, test, and production, too.

It is very important to note that the `docker-compose` command does not have built-in features for managing scaled, high availability distributed applications. In real life, you'll probably use it for development and quick testing or production run of personal and hobby projects.

It is nonetheless essential to learn about `docker-compose` because we can use the same `docker-compose.yml` file that we use with `docker-compose` to ship and run applications to high availability computer clusters with other Docker commands. We'll learn how to do this in later chapters.

Your first docker-compose.yml file

The Wordpress with MariaDB stack is one of the most wanted Docker examples online. Let's build it with Docker Compose.

In this example, we'll use two Docker images to create a Wordpress website with a MySQL database. For Wordpress, we'll use the official Wordpress image, and for MySQL, we'll use MariaDB, which is the community edition of MySQL.

We'll pick the image versions on the Docker Hub as usual.

- For Wordpress, we'll use the latest version from the official repository (https://hub.docker.com/_/wordpress/), which is 4.9.4-php7.2-apache as of today. This includes Wordpress 4.9.4, PHP 7.2 and an Apache web server. The Wordpress image also has an FPM variant based on Alpine. If you are seriously into Wordpress, I would suggest to build up your stack with the FPM version as a hobby project after finishing the book.
- MariaDB has fewer variants in the official repository (https://hub.docker.com/_/mariadb/), the latest version is 10.3.3 as of today, and we will use this one for this example.

Let's create a directory on your machine, I call mine `example10`. Let's pull the images to your computer.

```
~/Development/example10$ docker pull \
wordpress:4.9.4-php7.2-apache

~/Development/example10$ docker pull mariadb:10.3.3
```

The key of multi-container Docker applications is the `docker-compose.yml` file. Let's create a file in your project directory with the exact name `docker-compose.yml` and add the following content:

```
version: '3'

services:
  wordpress:
    image: wordpress:4.9.4-php7.2-apache
    environment:
      - WORDPRESS_DB_USER=wpadmin
      - WORDPRESS_DB_PASSWORD=gafs24rvaf4536sd
    ports:
      - 80:80

  mysql:
    image: mariadb:10.3.3
    environment:
      - MYSQL_ROOT_PASSWORD=acers1608gtv3
      - MYSQL_USER=wpadmin
      - MYSQL_PASSWORD=gafs24rvaf4536sd
      - MYSQL_DATABASE=wordpress
```

Let's review what we have in our `docker-compose.yml` :

- `version: '3'`: you have to specify the Docker Compose file specification version when you are starting your `docker-compose.yml` file. The current major version of the Docker Compose file format is 3, and there are new features added with every release of Docker. The latest version of the file format is '3.6', some new features will require you to specify the subversion, as well.
- `services`: The Services section contains the services configuration for each of our containers. Every service has exactly one image specified that is used to start up containers. We are using official images now, but you can use any image, including your custom images. Docker Compose respects the options that you define in your Dockerfile, so you don't need to redefine settings such as environment variables or CMD here.
You can, of course, define more options or override options like the CMD, if you need to. In the example, we are adding the environment variables required by Wordpress to run. Please note that there are more secure ways to provide passwords, we'll learn about these later.
In the Wordpress service, we are mapping port 80 in the container to port 80 on the host machine so that our website will be available on <http://localhost>.

We can launch our containers with `docker-compose up`, or you can start your containers in the background with `docker-compose up -d`:

```
~/Development/example10$ docker-compose up -d  
  
Creating network "example10_default" with the default driver  
Creating example10_mysql_1 ... done  
Creating example10_wordpress_1 ... done
```

Something very interesting has happened here. Docker Compose created two containers as expected, and it also created a network called `example10_default` for our application.

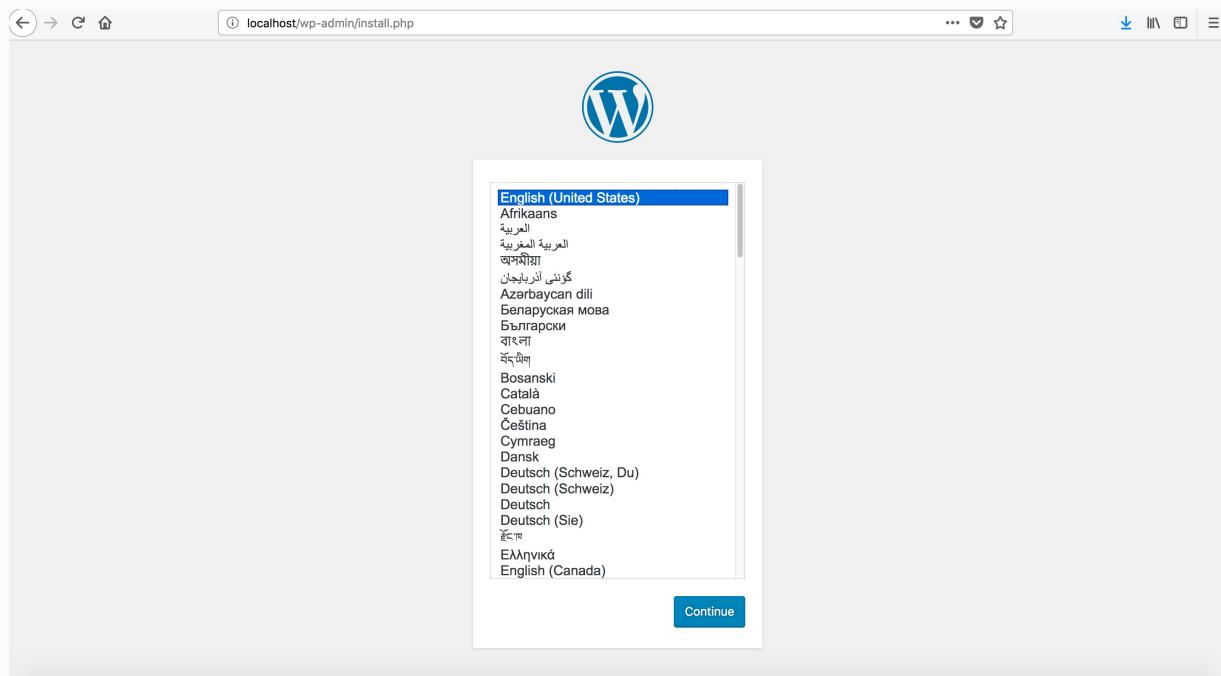
Our containers use this network to connect to each other. Our services can access each other on this network by using the service name as hostname.

On the default network, `example10_default`, the MariaDB container is available by the hostname `mysql` because Docker Compose uses the service name from `docker-compose.yml` as the hostname to access the container.

The Wordpress container can access MariaDB by the hostname `mysql`. In the `wp-config.php` file in the container the value of `DB_HOST` is set to `mysql` by default so Wordpress will be looking for the host by the name `mysql`.

I have intentionally named our MariaDB service `mysql` because I knew that the database hostname is set to `mysql` in Wordpress by default. If I used a different service name in `docker-compose.yml`, I would have to change the `DB_HOST` setting. Otherwise, Wordpress would not be able to connect to MariaDB.

Let's visit <http://localhost>; you should see the following installation page. (If you don't see the page immediately, wait a few seconds and refresh your browser, it takes some time for the application to start up.)



We have used a Docker Compose file (`docker-compose.yml`) to define two services that work together. We can use the Docker Compose file to specify further options for our containers. Moreover, we'll use the file to define our

network for container communications and volumes for storing data in our containers.

The Docker Compose file has a well-written reference on the Docker Website (<https://docs.docker.com/compose/compose-file/>), this is the resource that I use the most during everyday work besides the Dockerfile file reference (<https://docs.docker.com/engine/reference/builder/>).

Let's see how we can manage our running containers first, and then we'll dive into Docker Compose networks and volumes.

Managing multi-container applications

Docker Compose provides a set of commands that you can use to manage the services in your compose file. Let's see how you can accomplish key management tasks with Docker Compose commands:

- Start containers
- Stop and remove containers
- Stop and start existing containers
- Restart containers

- List containers
- Display running processes
- See logs for all containers
- Execute commands in a running container
- Run a one-off command
- Kill containers

I have not listed those commands that I'm not using frequently. You can use the `docker-compose` command in your Terminal to list all commands and see what they do.

Once you start up your containers with `docker-compose`, they will be running on your system as regular containers. You can manage these containers with `docker-compose` commands, and you can also use the `docker container` commands that we learned earlier in the book.

The advantage of using `docker-compose` commands is that you can manage all services at the same time and Docker Compose will help you keep all your containers aligned to the Compose configuration file.

Start containers

You can use the `docker-compose up` command to create containers, networks, and volumes described in your Docker

Compose file. We used this command in the example. As a result, Docker Compose started up our containers and the default network.

The `docker-compose up` command will create the components described in your Compose file if they do not exist. If your components, like containers, networks, and volumes already exists, `docker-compose up` will stop and recreate them if the configuration of a component has changed.

You can think of your Compose file as the description of the desired state of your multi-container application. Docker Compose will make sure that your containers run in the desired state when you run `docker-compose up`.

If we change the `docker-compose.yml` file, and issue `docker-compose up` again without touching the containers, Docker Compose will update our containers with our changes. Let's try this!

I have changed the port mapping of the Wordpress service in my `docker-compose.yml` to use port 8080 instead of 80. Update the ports section under the Wordpress service like this:

```
ports:  
  - 8080:80
```

Let's apply the changes with the following command and check the output:

```
~/Development/example10$ docker-compose up -d  
  
example10_mysql_1 is up-to-date  
Recreating example10_wordpress_1 ... done
```

Docker Compose has recreated our Wordpress container and changed the port mapping. You'll find your Wordpress site at <http://localhost:8080> now, and <http://localhost> will give you a connection error message.

It is often required to start a single service in your Compose file, for example, to test that your database functions properly before connecting to it from another service. You can provide the service name as a parameter to `docker-compose up` to achieve this (To test this, please stop the running stack with `docker-compose down`, before proceeding with the below command.):

```
~/Development/example10$ docker-compose up -d mysql  
  
Creating network "example10_default" with the default driver  
Creating example10_mysql_1 ... done
```

If you need to start multiple services, you can list their names specifically. (To test this, please stop the running

stack with `docker-compose down`, before proceeding with the below command.)

```
~/Development/example10$ docker-compose up -d mysql wordpress  
Creating network "example10_default" with the default driver  
Creating example10_wordpress_1 ... done  
Creating example10_mysql_1      ... done
```

Stop and remove containers

If you want to stop and remove all components of your application, you can use `docker-compose down`, that is the opposite of `docker-compose up`.

If I'm removing our Wordpress and MariaDB containers, Docker Compose removes the default network, as well:

```
~/Development/example10$ docker-compose down  
Stopping example10_wordpress_1 ... done  
Stopping example10_mysql_1      ... done  
Removing example10_wordpress_1 ... done  
Removing example10_mysql_1      ... done  
Removing network example10_default
```

Stop and start existing containers

Docker Compose provides further commands to stop and start running containers. If you start your containers with `docker-compose up -d`, you can use `docker-compose stop` to stop all containers and `docker-compose start` to start all containers.

Alternatively, you can stop and start individual services in your Compose file, so you can, for example, stop your Wordpress container like this:

```
~/Development/example10$ docker-compose stop wordpress  
Stopping example10_wordpress_1 ... done
```

You can start Wordpress again with `docker-compose start` or `docker-compose start wordpress` or `docker-compose up`.

Restart containers

You can use the `docker-compose restart` command to restart all containers in your Compose file, and you can use the same command to restart an individual service (for example `docker-compose restart wordpress`).

List containers

The `docker-compose ps` command lists the containers in your multi-container application. The output of the command

looks like this:

Name	Command	State
example10_mysql_1	docker-entrypoint.sh mysqld	Up
example10_wordpress_1	docker-entrypoint.sh apach ...	Up

The `docker-compose ps` command lists both running and stopped containers, so it's different from `docker container ls` that lists only running containers.

Display running processes

The command `docker-compose top` will list running processes in all containers across your services:

example10_mysql_1			
PID	USER	TIME	COMMAND
31321	999	0:00	mysqld
example10_wordpress_1			
PID	USER	TIME	COMMAND
31646	root	0:00	apache2 -DFOREGROUND
31810	xfs	0:00	apache2 -DFOREGROUND
31811	xfs	0:00	apache2 -DFOREGROUND

```
31812  xfs    0:00  apache2 -DFOREGROUND  
31813  xfs    0:00  apache2 -DFOREGROUND  
31814  xfs    0:00  apache2 -DFOREGROUND
```

See logs for all containers

Docker Compose provides a way to see the combined logs of your containers, you can also use `-f` to follow the logs and `--tail` to tail the output. Here is an example:

```
~/Development/example10$ docker-compose logs -f --tail 2  
  
Attaching to example10_wordpress_1, example10_mysql_1  
mysql_1      | 2018-04-21 15:12:34 0 [Note] mysqld: ready for con  
mysql_1      | Version: '10.3.3-MariaDB-10.3.3+maria~jessie' socl  
wordpress_1  | [Sat Apr 21 15:17:32.865168 2018] [mpm_prefork:not:  
wordpress_1  | [Sat Apr 21 15:17:32.865505 2018] [core:notice] [p:
```

Execute commands in a running container

Similarly to `docker container exec` or `docker exec`, you can use `docker-compose exec` to execute a command in a running container.

For example, if you want to check the `DB_HOST` setting in your Wordpress container you can use the below command:

```
~/Development/example10$ docker-compose exec wordpress \  
cat wp-config.php | grep DB_HOST  
  
define('DB_HOST', 'mysql');
```

The key difference between `docker-compose exec` and `docker container exec` is that you provide the service name as command parameter in the former case and you provide the container name or id as a command argument in the latter.

Run a one-off command

Docker Compose has the `docker-compose run` command, which takes a service name as its parameter and it will create a new container for your service. The container that you create with `docker-compose run` will ignore the port mappings in your Compose file to avoid port conflicts.

The main use case for `docker-compose run` is when you want to run a specific command on a new container and not in the running container you started with `docker-compose up`.

Such an example may be to list the environment variables of our Wordpress service with `docker-compose run wordpress env`.

Kill containers

When a container becomes unresponsive, you may want to force stop it. The `docker-compose kill` command will stop all containers in your application.

Docker Networks

In this chapter, we'll look into Docker networking in more detail and see what's going on in the background. We start by Docker networking in general, so this information applies to every container, not only to the ones started with Docker Compose.

On a clean installation of Docker, there are three networks created by default. You can use the `docker network ls` command to list them (if example 10 is still running on your machine, you'll see the `example10_default` network also on the list):

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
83b95126041a   bridge    bridge    local
a6a42ff114bd   host      host     local
6e7df6acd7a4   null     null     local
```

The 3 networks created by default in every Docker environment are the following:

- The default **bridge network**: Containers connect to this network by default. We used this network in every example before example 10. This network is isolated from the host machine's network, and you can make ports accessible on the host machine with the `-p` option of `docker run`. Containers that are connected to the default bridge network cannot reference each other by

container name, but they can access each other by IP address.

- Using the **host network** means that we are removing the isolation between container and host, and we're running the container on the host machine's network directly. This option will work on Linux machines as expected, but will not work on Docker for Mac and Windows. You can follow this issue to see if there are changes: <https://github.com/docker/for-mac/issues/68>.
- Using the **none** network means to disable networking for the container.

The default bridge network

Let's demonstrate the behavior of the default bridge network. Use the below commands to start up two containers. They will be connected to the default bridge network by default:

```
$ docker run --rm -ti -d --name c1 alpine:3.7 /bin/sh $ docker run --rm -ti -d --name c2 alpine:3.7 /bin/sh
```

Now you have two containers, c1 and c2 running on the host machine. Let's see the default bridge network details with the command:

```
$ docker network inspect bridge [ { "Name": "bridge", "Id": "50c34c6b38f11346d63f97c92c4a81c45815e0cc65e630aff1b83580ba885cd8" ,
```

```

"Created": "2018-04-07T16:25:46.379910968Z", "Scope": "local",
"Driver": "bridge", "EnableIPv6": false, "IPAM": { "Driver": "default", "Options": null, "Config": [ { "Subnet": "172.17.0.0/16", "Gateway": "172.17.0.1" } ] }, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": { "Network": "" }, "ConfigOnly": false, "Containers": {
{ "Name": "c2", "EndpointID": "5be49a9866f2649de0aeb7e003a419c261423bb4471efd1724750351d11d7430", "MacAddress": "02:42:ac:11:00:03", "IPv4Address": "172.17.0.3/16", "IPv6Address": "" },
"efe9161c3c98010f4489bf7cb37dd4db04822674371cff8660654cf2c05aeb4c": { "Name": "c1", "EndpointID": "f73973e2c2896f532aca0a94e5a796fa6864b7fc0e5b41a847c381e4eaf3287d", "MacAddress": "02:42:ac:11:00:02", "IPv4Address": "172.17.0.2/16", "IPv6Address": "" } }, "Options": {
"com.docker.network.bridge.default_bridge": "true",
"com.docker.network.bridge.enable_icc": "true",
"com.docker.network.bridge.enable_ip_masquerade": "true",
"com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
"com.docker.network.bridge.name": "docker0",
"com.docker.network.driver.mtu": "1500" }, "Labels": {} } ]

```

Let's use the IP address of c2 (172.17.0.3 in my case) and see if we can access it from c1:

```
$ docker attach c1 / # ping -c 2 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes 64 bytes
from 172.17.0.3: seq=0 ttl=64 time=0.156 ms 64 bytes from
172.17.0.3: seq=1 ttl=64 time=0.173 ms --- 172.17.0.3 ping
statistics --- 2 packets transmitted, 2 packets received, 0% packet
loss round-trip min/avg/max = 0.156/0.164/0.173 ms
```

The ping by IP succeeded. Let's see if we can ping c2 from c1 by name.

```
/ # ping -c 2 c2  
ping: bad address 'c2'
```

This fails, you cannot access a container on the default bridge network by its name from another container.

Your first user-defined network

Let's create a user-defined network and check its behavior with the Alpine containers first, and then we'll change our Wordpress example to use a user-defined network.

You can create a user-defined network with the `docker network create` command. Docker provides the following drivers to create different network types with `docker network create`:

- **Bridge network driver:** creates a private network on your host machine. The containers that you connect to a user-defined bridge network can communicate with each other by the container name as hostname. This service discovery functionality is provided by the bridge

network.

The bridge network isolates resources from your host machine's network and other networks. You can map ports to your host machine's network with `docker run -p` or the `ports` option in your Compose file to make your containers accessible from outside the network.

The bridge network is a local scope network, which means that it's limited to the host machine where you create the network, you cannot use it to connect resources on several hosts. We have the overlay driver for this.

- **Overlay network driver:** The overlay network driver is used in multi-host deployments. You'll see in later chapters when we learn about cluster deployment of Dockerized applications, that the overlay network provides service discovery, load balancing, IP address management, and TLS encryption out of the box.
- **MACVLAN network driver:** The MACVLAN driver allows you to assign a MAC address to a container, so it will appear as a physical device on your network. This means that your containers can be addressed with dedicated IP addresses on the external network, i.e., your containers can directly communicate with resources outside your host. This is beneficial

when connecting legacy systems with dockerized applications.

- **Host network driver:** Removes the isolation between the host network and containers. As we stated before, this only applies on Linux host machines.
- **None:** Disables container networking.

If your project needs a fine-grained networking solution, you can install 3rd party network plugins from the Docker Store (<https://store.docker.com/search?category=network&q=&type=plugin>).

Let's create your first user-defined network now: `$ docker network create mynet`

This command will create a new bridge network on your host machine. The default driver is `bridge`, and the previous command is equivalent to `docker network create --driver bridge mynet`.

Let's start up two Apache containers that connect to `mynet`:

```
$ docker run --rm -ti -d --name c1 \ --network mynet alpine:3.7  
/bin/sh $ docker run --rm -ti -d --name c2 \ --network mynet  
alpine:3.7 /bin/sh
```

The only change we applied is that we added the `--network` option.

Let's check service discovery on our new bridge network: \$

```
docker attach c1 / # ping -c 2 c2 PING c2 (172.18.0.3): 56 data  
bytes 64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.086 ms 64 bytes  
from 172.18.0.3: seq=1 ttl=64 time=0.165 ms --- c2 ping statistics  
--- 2 packets transmitted, 2 packets received, 0% packet loss  
round-trip min/avg/max = 0.086/0.125/0.165 ms
```

The key difference between the default bridge network and user-defined bridge networks is that in case of user-defined networks service discovery works by container name. This is why pinging c2 from c1 did not work on the default bridge network, but it works now on mynet .

Managing Docker networks

You can use the docker network commands to manage Docker networks. These are the key tasks you can perform with

```
docker network :
```

- list networks
- create a new network
- inspect a network
- connect to and disconnect from a network
- remove a network
- prune networks

List networks

You use the `docker network ls` command to list Docker Networks. Note that your user-defined network also appears on the list:

```
$ docker network ls
NETWORK ID NAME DRIVER SCOPE
50c34c6b38f1 bridge bridge local b14410585a68 host host local
47ed301528de mynet bridge local 6f7f82dc82cd none null local
```

Create a new network

We have already created a user-defined network and listed the different network drivers. If you need more advanced networking, you can deep dive into more options with `docker network create --help`.

Inspect a Docker network

You can inspect a Docker network with the `docker network inspect` command. We have used this command before, so I'll not copy the output here again.

Connect to and disconnect from a network

You can connect an existing container to a network, and you can also disconnect a container from a network. So I can, for example, disconnect `c2` from `mynet` with the command: `$ docker network disconnect mynet c2`

Now if you attach to the `c1` container and ping `c2`, you'll get an error.

Let's reconnect `c2` to `mynet` again with: `$ docker network connect mynet c2`

Now if you attach to `c1`, you can ping `c2` again.

Remove a network

You can remove a network if you don't need it any more. You need to make sure that no containers are connected to it before removal. After stopping `c1` and `c2`, you can remove `mynet` with this command: `$ docker network rm mynet mynet`

Prune networks

If you have several networks on the host, you can remove all unused networks with a single command: `$ docker network prune`

User-defined networks in Compose files

Let's go back to our `example10` project and add a user-defined network to our Compose file. If you remember, right now we are using the default network that is created automatically by Docker Compose, which is also a user-defined network, btw. This solution is adequate, but sometimes you'd want to create your own network or networks with custom configuration. So let's see how to do this.

You can create a user-defined network in the Docker Compose file by adding the `networks` tag to the root of your `docker-compose.yml`. A Simple two liner is the shortest way to define a new network called `wordpressnet`:

```
networks:  
  wordpressnet:
```

Then you need to specify which services should connect to this user-defined network. You use the `networks` option under your services to do this. My entire `docker-compose.yml` looks like this now:

```
version: '3'  
services:  
  wordpress:  
    image: wordpress:4.9.4-php7.2-apache  
    environment:  
      WORDPRESS_DB_USER=wpaadmin  
      WORDPRESS_DB_PASSWORD=gafsf24rvaf4536sd  
    ports:  
      - 8080:80  
    networks:  
      - wordpressnet  
  mysql:  
    image: mariadb:10.3.3  
    environment:  
      MYSQL_ROOT_PASSWORD=acers1608gtv3  
      MYSQL_USER=wpaadmin  
      MYSQL_PASSWORD=gafsf24rvaf4536sd  
      MYSQL_DATABASE=wordpress  
    networks:  
      - wordpressnet
```

If you use `docker-compose up -d`, you'll see that your Wordpress application still works on <http://localhost:8080>, plus you'll find your user-defined network under `docker network ls`.

You can further configure your user-defined network as described in the [Docker Compose file reference under Network Configuration](#).

Docker volumes

We have already used bind mounted directories and volumes on the host machine to map source code, and configuration files into Docker containers and store log files.

This time we'll see how to store persistent data in Docker containers properly.

**The concern we have to address is the following:
data stored in a container's file system will be lost
once the container is removed.**

Another reason to use Docker volumes is writing speed. The layered file system of Docker containers is less performant than external directories on the host, that's why it's beneficial to store your application data in a host directory.

In case of our Wordpress + MariaDB example, we should not store MariaDB data inside the MariaDB container, because we'll lose all of our data when we remove the container.

Containers are supposed to be removable, reproducible and portable. Docker is a solution that you can use to run your application anytime, anywhere. You should build your containers so that you can port them to a new infrastructure or upgrade them to a new version quickly.

This requires that any user generated data, data that may change or data that represents application state is stored outside your containers, so that you can destroy and recreate your containers as needed.

The Docker way of storing container data is with the use of Docker volumes.

Let me explain, how bind mounts are different from volumes. We used the `-v` or `--volume` option of `docker run` to create the bind mount. We use the same option to use a Docker volume in a container, but despite this, the two concepts are much different.

The key difference between a bind mount and a Docker volume is that a bind mount can point to any directory on the host machine. The bind mounted directory is a user-defined directory, and we assume that the bind mounted directory exists on the host machine. It's the developer's or operator's task to make sure that the bind mounted directory exists on the host machine in every environment.

Docker volumes are managed by Docker. They are directories, just like any other directory, but they live inside Docker's proprietary file structure on the host machine, and they are managed by Docker.

Defining volumes in the Dockerfile

You already have Docker volumes on your machine. We created volumes in our Flask example, and the Wordpress and MariaDB official images also created some volumes. You can check the Dockerfile of these Wordpress and MariaDB images online at <https://github.com/docker-library/wordpress/blob/59f3b513af128d12da1403721e4f9be8d882ec54/php7.2/apache/Dockerfile> and <https://github.com/docker-library/mariadb/blob/506cd7c6b9d6c53d7b514a2c94db5d3daee832fd/10.3/Dockerfile>.

Both Dockerfiles feature the `VOLUME` instruction, which means that they instruct the Docker Engine to store the files in the specific directories on the host machine outside the container.

The Wordpress Dockerfile features the line `VOLUME /var/www/html`, which means that your entire Wordpress installation will be stored outside the container. This makes a lot of sense because Wordpress files may change due to user activity, just think about your Wordpress configuration or your uploads directory, for example.

The MariaDB Dockerfile specifies the volume `VOLUME /var/lib/mysql`, which is the location of the database files.

If the Dockerfile includes the `VOLUME` instruction, Docker will create a Docker volume for every container started from that image and will give the volume a random name.

Let's list the volumes on your host machines with the below command:

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	4a2faac3e931fa478cff8a41d09f0e854db13b706ea6e6
local	acfd453659b9d00d100274178008e11ff63c8078fcaab

As you can see, I have two volumes with random names here, because I have a Wordpress container and a MariaDB container running (I have removed all previous volumes). If I stop the container and start up new ones, Docker will create new volumes for the new containers.

This means that these volumes will not be reused in this form across consecutive runs.

Your first named volume

The situation changes if you create a named volume, that is a volume with a specific name. You can map a named volume to your containers even if you destroy and re-create your container and preserve your precious data. This way

you can start up a new container over an existing database, for example.

You can create a named volume with the `docker volume create` command or in your Docker Compose file.

The named volume will be created locally on the host machine by default, and the default volume driver is `local`.

There are also other volume drivers that you can use in more complex scenarios. You can use such drivers to store your data on the network in networked file systems. These drivers are installed with Docker Plugins, and we'll not discuss them in this book. You can read more about them here if you need them in the future:

https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins.

Let's create a named volume like this:

```
$ docker volume create mydata  
mydata
```

Another volume appears on our volumes list:

```
$ docker volume ls  
DRIVER      VOLUME NAME  
local      4a2faac3e931fa478cff8a41d09f0e854db13b706ea6e6
```

```
local          acfd453659b9d00d100274178008e11ff63c8078fcaab:  
local          mydata
```

Let's start up a MariaDB container to use this volume:

```
$ docker run --rm -d -v mydata:/var/lib/mysql \  
-e MYSQL_ROOT_PASSWORD=pwd mariadb:10.3.3
```

This command has created a MariaDB container that stores its database files in the `mydata` volume. You can inspect the files in your volume on a Linux host at

`/var/lib/docker/volumes`. This location is part of Docker's proprietary file system on the host machine. On Mac and Windows, you can access this location differently.

In case of Docker for Mac, you can use the command `screen ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty` to attach to a Terminal and browse `/var/lib/docker` in the virtual machine included with Docker for Mac. The files are located at `C:\Users\Public\Documents\Hyper-V\Virtual hard disks\MobyLinuxVM.vhdx` on Windows 10.

Now if you stop and remove the container, your data will be still available on the host in the `mydata` volume. You can start up another container with the same command again to connect to the same volume and use the same data.

```
$ docker run --rm -d -v mydata:/var/lib/mysql \  
-e MYSQL_ROOT_PASSWORD=pwd mariadb:10.3.3
```

If you want to backup your data from the volume, you can copy the files from `/var/lib/docker/volumes` or a good approach is to use a container with the right toolkit to do so. For example:

```
$ docker run --rm -ti -v mydata:/usr/src/data debian /bin/bash
```

Managing Docker volumes

You can use the `docker volume` command family to manage Docker volumes. These are the typical activities:

- create a volume
- list volumes
- inspect volumes
- remove a volume
- prune volumes

Create a volume

We have already seen how to create a named volume with `docker create`. To complete the picture, let me mention that

you can also create a named volume with `docker run` by defining a volume mapping with `-v`. So you can create a named volume like this:

```
$ docker run --rm -d -v runcreate:/usr/src/app alpine:3.7
```

The above command creates a named volume called `runcreate` on the host machine.

List volumes

You can use the `docker volume ls` command to list volumes on the host.

Inspect volumes

You can inspect a volume or more volumes with `docker volume inspect`. The output will give detailed information about the volume, including the location of the volume directory in the Docker file structure.

Remove a volume

You can remove a volume with `docker volume rm`. You need to make sure that all containers that are using the volume are

removed.

Prune volumes

I use the `docker volume prune` command a lot because images create a lot of volumes in the background, plus I also create my named volumes, and old volumes have a tendency to stay on your computer taking up valuable storage space.

You can remove all unused volumes with `docker volume prune` and save up disk space.

User-defined volumes in Compose files

Let's continue with our `example10` project and add named volumes to our Compose file. We need to add one volume for Wordpress mapping to `/var/www/html` and one for MariaDB mapping to `/var/lib/mysql`.

You can define named volumes in the Docker Compose file by adding the `volumes` tag to the root of your `docker-compose.yml`:

```
volumes:  
  wordpressdata:
```

```
mariadbdata:
```

After the volumes are defined, you need to map the volumes to individual services with the `volumes` option under each service as shown below:

```
version: '3'

services:
  wordpress:
    image: wordpress:4.9.4-php7.2-apache
    environment:
      - WORDPRESS_DB_USER=wpadmin
      - WORDPRESS_DB_PASSWORD=gafs24rvaf4536sd
    ports:
      - 8080:80
    networks:
      - wordpressnet
    volumes:
      - wordpressdata:/var/www/html

  mysql:
    image: mariadb:10.3.3
    environment:
      - MYSQL_ROOT_PASSWORD=acers1608gtv3
      - MYSQL_USER=wpadmin
      - MYSQL_PASSWORD=gafs24rvaf4536sd
      - MYSQL_DATABASE=wordpress
    networks:
      - wordpressnet
    volumes:
      - mariadbdata:/var/lib/mysql

networks:
  wordpressnet:
    driver: bridge
```

```
volumes:  
  wordpressdata:  
  mariadbdata:
```

If you use `docker-compose up -d`, you'll see that your Wordpress application still works on <http://localhost:8080>, plus you'll find your user-defined network under `docker network ls` and the named volumes with `docker volume ls`.

It is important to note that your data is still vulnerable in a named volume because your named volume is stored on the host machine. If you change your host machine, for example, if you destroy your cloud virtual machine host, your data will be lost. You need to either set up a backup mechanism for your data or use a volume plugin to create a more robust storage solution.

Build practices with Docker Compose

With Dockerfile and Docker Compose on board, you can build an automated build process for your multi-container applications. In this chapter, we'll check out such practices.

Build custom images with Docker Compose

Let's update `example10` by creating a custom Wordpress theme and copying our source files into a custom image using a Dockerfile.

Let's add a directory named `src/mytheme` under our `example10` project directory and create 3 files as shown below:

```
. └── src  
    └── mytheme └── index.php └── screenshot.png └── style.css
```

You can leave `index.php` empty. Feel free to add any `png` file with the name `screenshot.png`. This image will be used as the image of your custom theme in the theme list. Add the following lines to `style.css`:

```
/* Theme Name: Custom theme  
placeholder for docker example Theme URI: n/a. Author: Mark Takacs  
Author URI: http://takacsmark.com Description: This is just a
```

```
placeholder, not a real theme. Version: 1.0 License: GNU General  
Public License v2 or later License URI:  
http://www.gnu.org/licenses/gpl-2.0.html Tags: placeholder, notheme  
Text Domain: docker This theme, like WordPress, is licensed under  
the GPL. Use it to make something cool, have fun, and share what  
you've learned with others. */
```

Don't worry about the details much; the above is the way to define an empty Wordpress theme, it is Wordpress convention. Feel free to replace my personal info with your data.

Let's add a Dockerfile to the `example10` directory to define a custom Wordpress image with this code:

```
FROM wordpress:4.9.4-  
php7.2-apache COPY src/mytheme /usr/src/wordpress/wp-  
content/themes/mytheme
```

The above builds on the official Wordpress image, we've been using so far. The second line copies our code into the image under `/usr/src/wordpress/wp-content/themes/mytheme`.

The installation Wizard copies the Wordpress files from `/usr/src/wordpress` into `/var/www/html`, so if we copy our custom theme into `/usr/src/wordpress`, it will be copied with the rest of the Wordpress files to the right location during installation.

If you are wondering how I know about this detail, the answer is, that I used the interactive shell with `docker exec` to explore the image and learned about its internal behavior.

The next step is to change the `docker-compose.yml` file to use the custom image. We can use Docker Compose to build the image for us. Let's change `docker-compose.yml` to the following:

```
version: '3'
services:
  wordpress:
    build: .
    image: takacsmark/wordpress:4.9.4-php7.2-apache
    environment:
      WORDPRESS_DB_USER=wpadmin
      WORDPRESS_DB_PASSWORD=gafs24rvaf4536sd
    ports:
      - 8080:80
    networks:
      - wordpressnet
    volumes:
      - wordpressdata:/var/www/html
    mysql:
      image: mariadb:10.3.3
      environment:
        MYSQL_ROOT_PASSWORD=acers1608gtv3
        MYSQL_USER=wpadmin
        MYSQL_PASSWORD=gafs24rvaf4536sd
        MYSQL_DATABASE=wordpress
      networks:
        - wordpressnet
      volumes:
        - mariadbdata:/var/lib/mysql
    networks:
      - wordpressnet
      - mariadbdata
    driver: bridge
    volumes:
      - wordpressdata
      - mariadbdata
```

I have replaced the image name in the Wordpress service, plus I have added the `build: .` section, which instructs Docker Compose to look for a Dockerfile in the current directory (`.`) to build the image used for the service.

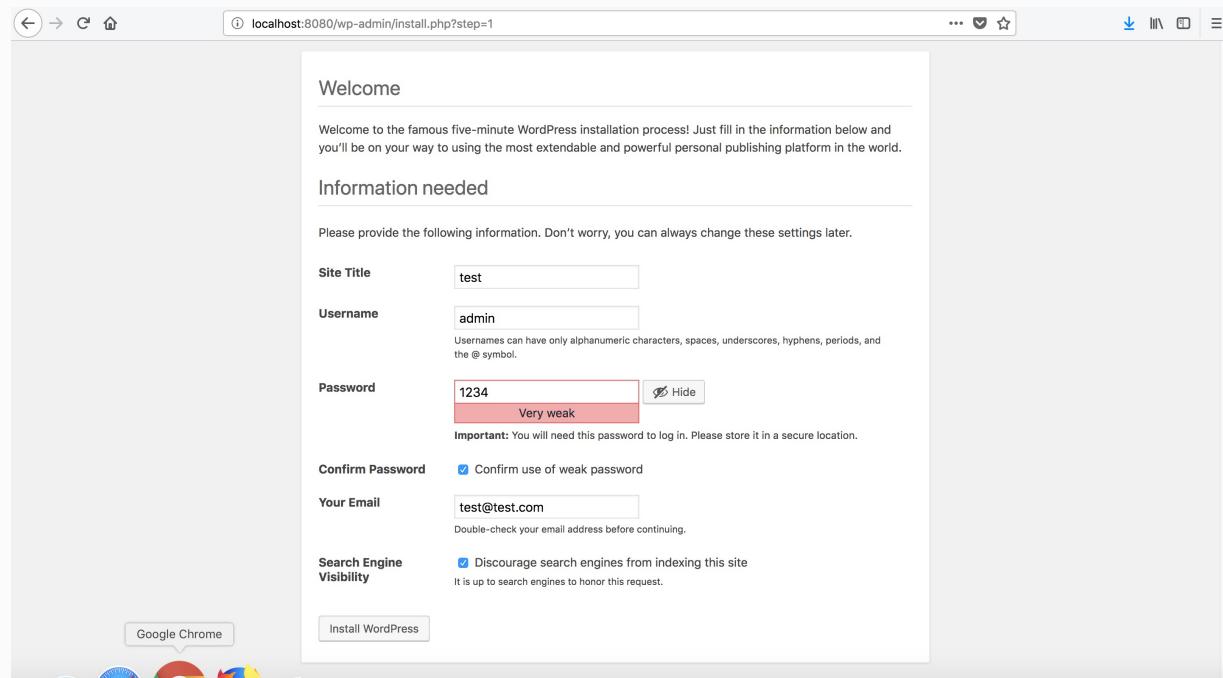
Let's clean up your system before proceeding, it's important to remove the existing named volumes, because they may persist data that we do not need. Please stop and remove all containers and run `docker volume prune`.

```
~/Development/example10$ docker-compose down
~/Development/example10$ docker volume prune
~/Development/example10$ docker volume ls
DRIVER          VOLUME NAME
```

Now you can use the `docker-compose build` command to build all images used in the Compose file. Alternatively, you can use `docker-compose up`, it will build the image if it is missing.

You can use `docker-compose up -d --build` to start up your services rebuilding the images in the Compose file.

Please complete the Wordpress installation wizard on <http://localhost:8080>. You can use the below settings as a simple reference for testing:

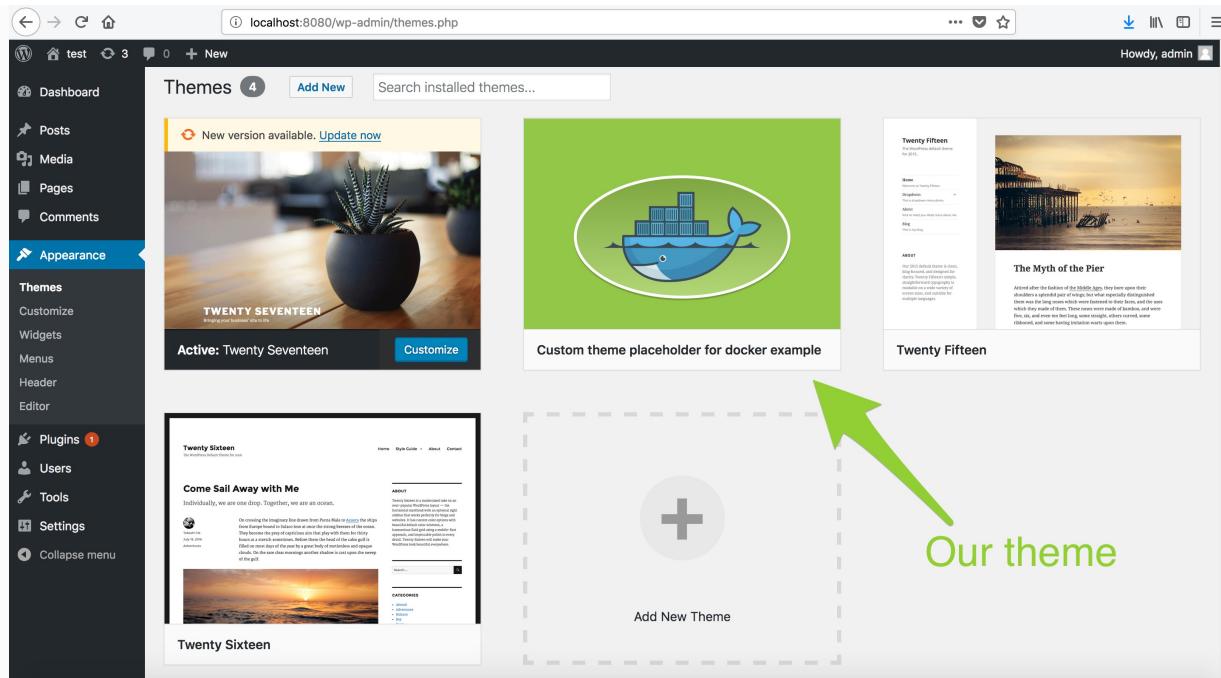


The screenshot shows the WordPress installation wizard page at `localhost:8080/wp-admin/install.php?step=1`. The page title is "Welcome". It says: "Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world." Below this is a section titled "Information needed" with the following fields:

Setting	Value
Site Title	test
Username	admin
Password	1234 Very weak
Confirm Password	<input checked="" type="checkbox"/> Confirm use of weak password
Your Email	test@test.com
Search Engine Visibility	<input checked="" type="checkbox"/> Discourage search engines from indexing this site It is up to search engines to honor this request.

At the bottom are two buttons: "Google Chrome" and "Install WordPress".

After logging in to the admin page and navigating to the Appearance -> Themes menu, you'll find our custom theme on the theme list:



Overriding the Compose file

Your project will probably need a way to override or extend Compose file settings for development.

The best practice in simple projects is to use the file called `docker-compose.yml` for production and use a file called `docker-compose.override.yml` for development.

Docker Compose will automatically use `docker-compose.override.yml` if it exists, you don't need to specify its name explicitly as a command argument to `docker-compose`.

Docker Compose will apply the settings in `docker-compose.override.yml` on top of `docker-compose.yml`, so you only

need to specify the delta in the override file.

Let's create a `docker-compose.override.yml` file in the `example10` directory and map the MySQL port to the host machine so that we can connect to it with a MySQL client for development purposes. The file contents will be as short as this:

```
version: '3' services: mysql: ports: - 3306:3306
```

You can update your application to the desired state with

```
docker-compose up : ~/Development/example10$ docker-compose up -d  
example10_wordpress_1 is up-to-date Recreating example10_mysql_1  
... done
```

The MySQL port is now mapped to the host:

```
~/Development/example10$ docker-compose ps
```

Name	Command	State	Ports
<hr/>			
example10_mysql_1	docker-entrypoint.sh mysqld	Up	0.0.0.0:3306->3306
example10_wordpress_1	docker-entrypoint.sh apache2	Up	0.0.0.0:80->80

Now you can use a regular MySQL client like MySQL Workbench to connect to your database.

Custom Compose file name

In more complex projects you may want to use more than two Compose files, or you may want to stack them to apply multiple overrides. You can achieve this by using the `-f` option of `docker-compose`. This flag lets you specify the Compose file name.

If, for example, you want to apply `docker-compose.yml` in `example10` without the override, you can tell `docker-compose` that you want to use `docker-compose.yml` like this:

```
~/Development/example10$ docker-compose \ -f docker-compose.yml up  
-d
```

This command removes the effects of your override file from the `mysql` service. Docker will automatically pick up the override file if it's present in your project and you do not specify a file name with `-f`, so it will apply both `docker-compose.yml` and `docker-compose.override.yml` automatically. You can achieve the same effect by stacking file parameters like this: `~/Development/example10$ docker-compose \ -f docker-compose.yml \ -f docker-compose.override.yml up -d`

You can use `-f` with any file name, and you can use more than two files that extend each other. This gives you a lot of flexibility building applications with Compose. We'll use this in our end to end example.

Compose environment and variable substitution

We have seen how to specify environment variables in Compose files. You don't want to hardcode environment variable values into configuration files, and you should not check in sensitive information into source control. On top of this, you'll often want to use a different environment variable set for different environments.

Docker has a handy feature for this. **Docker Compose will pick up any variable present in a .env file under the directory where the docker-compose command is executed.**

So we can move our environment variable value assignment into a file called `.env` in the `exmaple10` project directory. Let's add these values:

```
MYSQL_ROOT_PASSWORD=acers1608gtv3  
MYSQL_USER=wpadmin MYSQL_PASSWORD=gafs24rvaf4536sd  
MYSQL_DATABASE=wordpress
```

You can use variable substitution in the Compose file to make it more generic:

```
version: '3'  
services:  
  wordpress:  
    build:  
      .  
      image: takacsmark/wordpress:4.9.4-php7.2-apache  
    environment:  
      - WORDPRESS_DB_USER=${MYSQL_USER}  
      - WORDPRESS_DB_PASSWORD=${MYSQL_PASSWORD}  
    ports:  
      - 8080:80  
    networks:  
      - wordpressnet  
    volumes:  
      - wordpressdata:/var/www/html  
    mysql:  
      image: mariadb:10.3.3  
      environment:  
        - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}  
        - MYSQL_USER=${MYSQL_USER}  
        - MYSQL_PASSWORD=${MYSQL_PASSWORD}  
      MySQL_DATABASE=${MYSQL_DATABASE}  
    networks:  
      - wordpressnet  
    volumes:  
      - mariadbdata:/var/lib/mysql  
    networks:  
      - wordpressnet  
    driver: bridge  
    volumes:  
      - wordpressdata:mariadbdata
```

Keeping your `.env` file out of source control will help you manage sensitive information. Docker provides a more advanced way of managing secrets in its cluster orchestration solution. We'll talk about this later.

Please update your application with `docker-compose up -d` and check the results in your browser.

Build summary

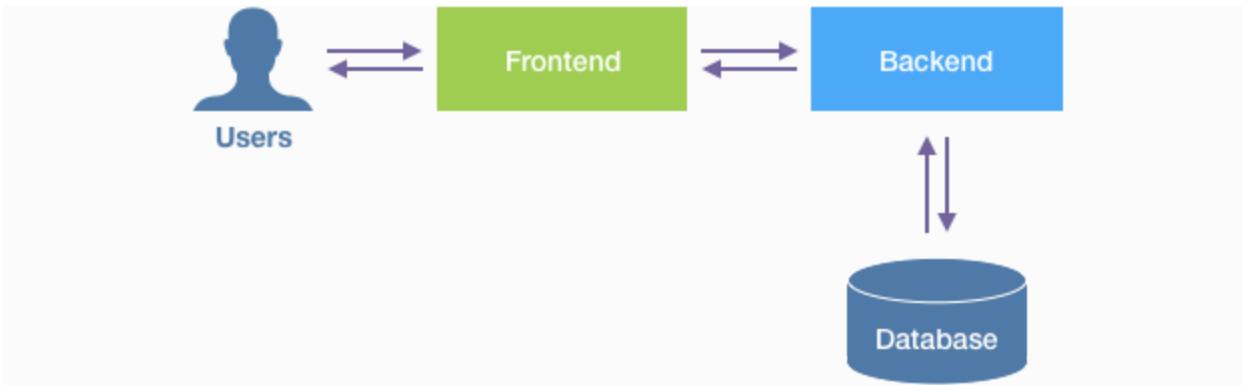
In the first part of the book, we went through the build steps of dockerized software delivery. **We have seen how Docker Containers, Images, Docker Compose, Docker Networks, and Volumes play together to create multi-container applications.**

In this summary chapter, I'll give you an overview of your typical build tasks and workflow in a project. Let's draw a simple, hypothetic web application stack to depict the idea.

Let's assume you are building a web application with the following components:

- User frontend based on Javascript served by a Node.js server
- API backend written in Python using the Flask microframework
- A Postgres SQL database for the API backend

Your stack looks like this:



Typically, as a first step, you'd pick the Docker images to start building your components. Similarly to previous examples, you'd go online and look for a suitable Docker image in the cloud.

You'll have to pick the preferred version of Docker images for Node.js, Python, and Postgres in the example. These images will serve as the basis for your containers.

The next step is to customize your images to your project. The Dockerfile gives you tools to start from public images online and add your customization. Customization includes additional system packages, environment variables, users, your source code and other settings needed by your application.

The definition of your custom images comes in the form of the Dockerfile. This means that you can rebuild your Docker images the same way over and over again. This makes your technical environments reproducible and repeatable anytime and anywhere.

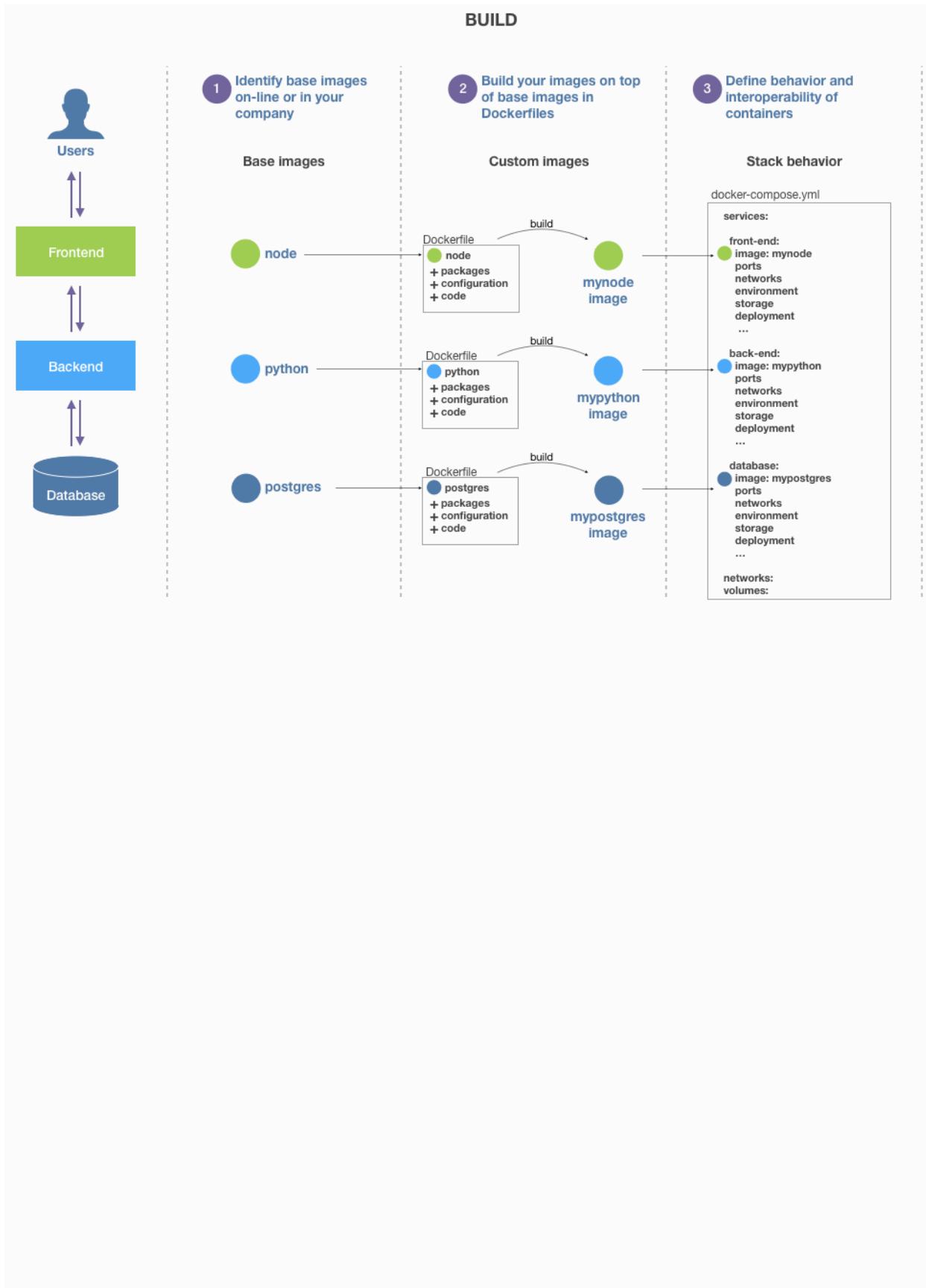
Once your images are ready, you usually define how to start up containers from your project's Docker images and how those containers should work together. This step is declarative, too, which means that we use a file to describe the behavior and interoperability of containers.

Docker provides a tool, called Docker Compose to start up a composition of different containers from Docker images that work together. You'll provide this definition in a file called `docker-compose.yml` and use a command called `docker-compose` to start up containers with the desired behavior.

You can use the `docker-compose.yml` file to start your containers with the same behavior anytime and anywhere with the same commands. We define our architecture components like front-end, back-end, and databases as "services".

In the hypothetical example stack, you'd define three services in your `docker-compose.yml` file, one for the front-end, one for the back-end and one for the database. You can then define how the containers should be started from the Docker images that you built for your project in `docker-compose.yml`.

The below figure shows the main build steps and artifacts of the example stack:



Ship dockerized applications

In the first part of the book, you've learned how to use Docker containers, how to build your custom Docker images and how to create applications that are composed of multiple services.

In this chapter, we'll see how to ship your applications with Docker tools.

As the first step, we'll create an example to ship your project's Docker images to the Docker Hub, and we'll see how you can set up automatic image building integrated with source control for your projects.

We'll also set up your Docker Registry that's similar to the Docker Hub, but it serves your project or organization exclusively.

Later on in the book, we'll build an end to end example where we'll integrate our Docker based project with CI tools. We'll do this in the chapter called "Build, ship and run a Flask web app end 2 end".

Ship your image to the Docker Hub

To share Docker images with your team, your organization or with a broader community, you can publish your Docker images to the Docker Hub.

After you build your Docker images from your project's Dockerfile(s), as we learned in the previous chapters, you'll want to ship your images to other environments. The best practice is to encapsulate your source code and its dependencies in your Docker images and ship them in one package, in the form of the Docker image.

People new to Docker often think of the Docker image as the environment, and many people try to ship their source code independently, outside the image. This is technically feasible, but shipping your code inside the image makes your application more portable and easy to scale.

Publishing your image to the Docker Hub is a straightforward process. You created your Docker Hub account already, and now we'll use these credentials to push an image to the Docker Hub.

1. Make sure you're logged in to Docker

Open up a Terminal window and use the `docker login` command to log in to Docker. Please make sure to use your username that you provided when you signed up.

```
~/Development $ docker login  
  
Login with your Docker ID to push and pull images from Docker Hub  
Username: YOUR_USERNAME  
Password: YOUR_PASSWORD  
Login Succeeded
```

2. Push the image to the Docker Hub

Let's push the parent image that we built in `example7` to the Docker Hub. My image is called `takacsmark/example7-base:1.0`, but this is not a very friendly name for the public, so we'll rename it first. The command is simple, what we do in fact is, that we re-tag the image like this:

```
~/Development$ docker image tag \  
takacsmark/example7-base:1.0 \  
takacsmark/php-composer:7.2.2-apache
```

I have re-tagged the image with the `docker image tag...` command. I decided to use version tagging similar to the tags of the PHP image, and specified `php-composer` as the image name.

Now we can push the image to the Docker Hub like this:

```
~/Development$ docker push takacsmark/php-composer:7.2.2-apache  
The push refers to repository [docker.io/takacsmark/php-composer]  
502c29b658cf: Pushed  
f6c37f16d82c: Pushed  
a0266bc1bc94: Pushed  
f0955f3c0d33: Mounted from library/php  
f8565cac8b50: Mounted from library/php  
89441ae9b4e3: Pushed  
2fe65268e1ea: Pushed  
f52d3c49f6fd: Pushed  
475ba53e3b68: Mounted from library/php  
b62ad04621bf: Pushed  
b5ae5b5b7481: Pushed  
a5a3cedf1e72: Pushed  
c1f5c17cbd1e: Mounted from library/php  
2dc758f90f4e: Mounted from library/php  
81f69a91d59f: Mounted from library/php  
b747d11dbfaf: Mounted from library/php  
014cf8bfcb2d: Mounted from library/php  
7.2.2-apache: digest: sha256:a81f2ef2ac4943e3f8fad491f31220630ea40
```

It takes a few minutes to push. Sometimes the operation would time out, if this happens, just rerun the above command. Once the operation is finished, we can go to the Docker Hub and see the results. Visit your dashboard at <https://hub.docker.com>. Mine looks like this:

The screenshot shows a web browser window displaying the Docker Hub repository list for the user 'takacsmark'. The URL in the address bar is <https://cloud.docker.com/repository/list>. The page header includes the Docker Hub logo, navigation links for Explore, Repositories, Organizations, Get Help, and a user profile for 'takacsmark'. A message at the top reads 'Introducing the New Docker Hub: Combining the best of Docker Hub, Cloud and Store. [Learn more](#)'. Below the header, there's a search bar with the placeholder 'Filter by repository name...' and a dropdown menu showing 'takacsmark'. A button labeled 'Create Repository +' is visible. The main content area displays a table of repositories:

REPOSITORY	DESCRIPTION	LAST MODIFIED
takacsmark / flask-sql-ci-web	This repository does not have a description...	① a day ago
takacsmark / flask-mysql-ci-web	This repository does not have a description...	① 21 days ago
takacsmark / e5	This repository does not have a description...	① 24 days ago
takacsmark / e3	This repository does not have a description...	① 24 days ago
takacsmark / flask-redis	This repository does not have a description...	① 3 months ago
takacsmark / onepagewebsite-nginx-bootstrap...	This repository does not have a description...	① 4 months ago
takacsmark / php-composer	Docker image with PHP and Composer.	① 7 months ago

You can further enhance your repository by providing the usual information that you see in other repositories. If you click the repository box, you can edit the short description, full description, tags, contributors and more.

Build a new project with your image from the Docker Hub

You worked hard to push your image to the Docker Hub, so let's use our new image to build a new example with PHP and Composer.

Let's create our file structure based on example 7:

```
~/Development/example7$ tree -a
.
├── composer.json
├── Dockerfile
└── public
    ├── .htaccess
    └── index.php

1 directory, 4 files
```

Let's copy the `example7` directory with the name `example11`:

```
~/Development$ cp -r example7 example11
```

Clean up Docker images on our machine.

```
~/Development/example11$ docker image prune

WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B
```

Let's delete the parent image locally, so that we can see when it's pulled from Docker Hub:

```
~/Development/example11$ docker image rm \
takacsmark/php-composer:7.2.2-apache

Untagged: takacsmark/php-composer:7.2.2-apache
Untagged: takacsmark/php-composer@sha256:a81f2ef2ac4943e3f8fad491
```

Let's keep the files as they are, let's change the Dockerfile so that we use the image that we pushed to the Docker Hub.

```
FROM takacsmark/php-composer:7.2.2-apache
```

Let's build our project with

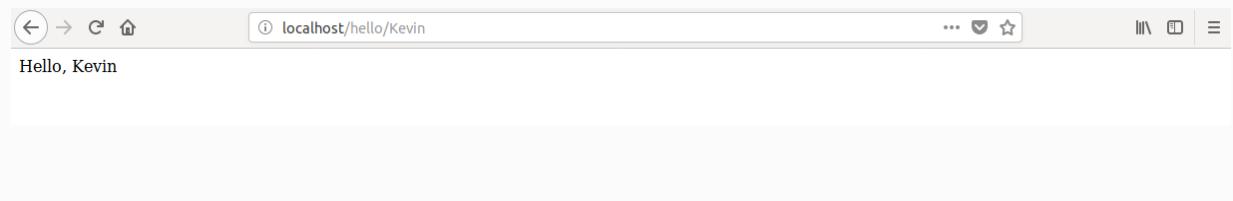
```
~/Development/example11$ docker build \
--tag takacsmark/example11:1.0 .
```

Let's run our project with the command

```
~/Development/example11$ docker run --rm \
-p 80:8080 takacsmark/example11:1.0
```

```
AH00558: apache2: Could not reliably determine the server's fully
AH00558: apache2: Could not reliably determine the server's fully
[Tue Feb 27 09:45:29.054753 2018] [mpm_prefork:notice] [pid 1] AH00094: 
[Tue Feb 27 09:45:29.054852 2018] [core:notice] [pid 1] AH00094: 
```

This is the result we get when visiting
<http://localhost/hello/Kevin>:



Automatically build your image from GitHub

I promised to show you how to build your Docker image automatically when you check in your changes into source control, specifically GitHub.

1. Sign up for a free GitHub account

You'll need a GitHub account to complete this exercise. You'll get an unlimited number of public repositories with the free account.

Please visit <https://github.com/> and complete the sing up process.

2. Create a new repository

Click the big green button that says “New repository” on <https://github.com/> after you signed up or signed in. You'll be presented with a form like the one below:

The screenshot shows the GitHub interface for creating a new repository. At the top, the URL https://github.com/new is visible. The main heading is 'Create a new repository'. Below it, a sub-instruction reads 'A repository contains all the files for your project, including the revision history.' The 'Owner' dropdown is set to 'takacsmark'. The 'Repository name' field contains 'docker-php-composer' with a green checkmark icon. A note below suggests names like 'studious-spork'. The 'Description (optional)' field contains the text 'Definition of a Docker image with composer included.' Under 'Visibility', the 'Public' option is selected, with the note 'Anyone can see this repository. You choose who can commit.' The 'Private' option is also shown. There is a checkbox for 'Initialize this repository with a README', which is unchecked. Below this are dropdowns for '.gitignore' (set to 'None') and 'Add a license' (set to 'GNU General Public License v3.0'). At the bottom is a large green 'Create repository' button.

I specified the name of my repository to be `docker-php-composer`, I provided a short description and chose to leave the repository public. I also changed the license setting and choose GNU GPL3, which means

The GNU GPLv3 is a copyleft license that requires anyone who distributes your code or a derivative work to make the source available under the same terms, and also provides an express grant of patent rights from contributors to users.

(As described on <https://choosealicense.com/>.)

Now click “Create repository”, and you’ll arrive at your repository page.

3. Clone the repository to your machine

Let’s clone your repository to your machine from GitHub. You’ll need to install git on your machine if you do not have it yet. Please follow the installation steps on this page: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

Once the installation is done, go to your repository page and click the green button that says “Clone or download” and click “use https” on the popup window. Select the text and copy it to the clipboard (you can see the text highlighted in the screenshot below.)

Definition of a Docker image with composer included.

Add topics

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request

Create new file Upload files Find file Clone or download

takacsmark Initial commit LICENSE Initial commit

Clone with SSH Use HTTPS
Use an SSH key and passphrase from account.
git@github.com:takacsmark/docker-php-composer

Download ZIP

© 2018 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

Open Terminal, go to your development directory and copy the contents of your clipboard after the command `git clone`. It should look similar to this command:

```
~/Development$ git clone \
https://github.com/takacsmark/docker-php-composer.git

Cloning into 'docker-php-composer'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

4. Add the Dockerfile to your repo

Our plan is to generate our parent Docker image from exercise 7 automatically that we described in the file `Dockerfile-base`. So let's copy this file into our repo as `Dockerfile`:

```
~/Development/docker-php-composer$ cp \
..../example7/Dockerfile-base Dockerfile
```

We do not need to change this file now, let's move on and push it to GitHub.

5. Push the file to GitHub

First, we need to add the `Dockerfile` to our repo with the command:

```
~/Development/docker-php-composer$ git add Dockerfile
```

Then we need to commit our changes to the repository locally:

```
~/Development/docker-php-composer$ git commit -am "Dockerfile added"
[master 07a10bc] Dockerfile added
 1 file changed, 22 insertions(+)
 create mode 100644 Dockerfile
```

Now we push the file to GitHub:

```
~/Development/docker-php-composer$ git push origin master

Username for 'https://github.com': takacsmark
Password for 'https://takacsmark@github.com':
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 765 bytes | 765.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/takacsmark/docker-php-composer.git
  21d0920..07a10bc  master -> master
```

If you are new to GitHub, you'll be asked for your username and password as I was in the above example. You can make your workflow smoother by adding your ssh key to your GitHub account. In my case, the ssh key was not added because I'm working on a clean Ubuntu virtual machine that I set up for the examples.

If you want to upload your ssh keys, please follow this guide: <https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/>. After you uploaded your ssh key to GitHub, please make sure to change the URL of the repository from the command line to the ssh variant shown in your GitHub repo. This way you won't need to enter your password when pushing to the remote repository. Your command should be similar to this one:

```
~/Development/docker-php-composer$ git remote set-url \
origin git@github.com:takacsmark/docker-php-composer.git
```

Now you should see your `Dockerfile` on GitHub in your browser, too.

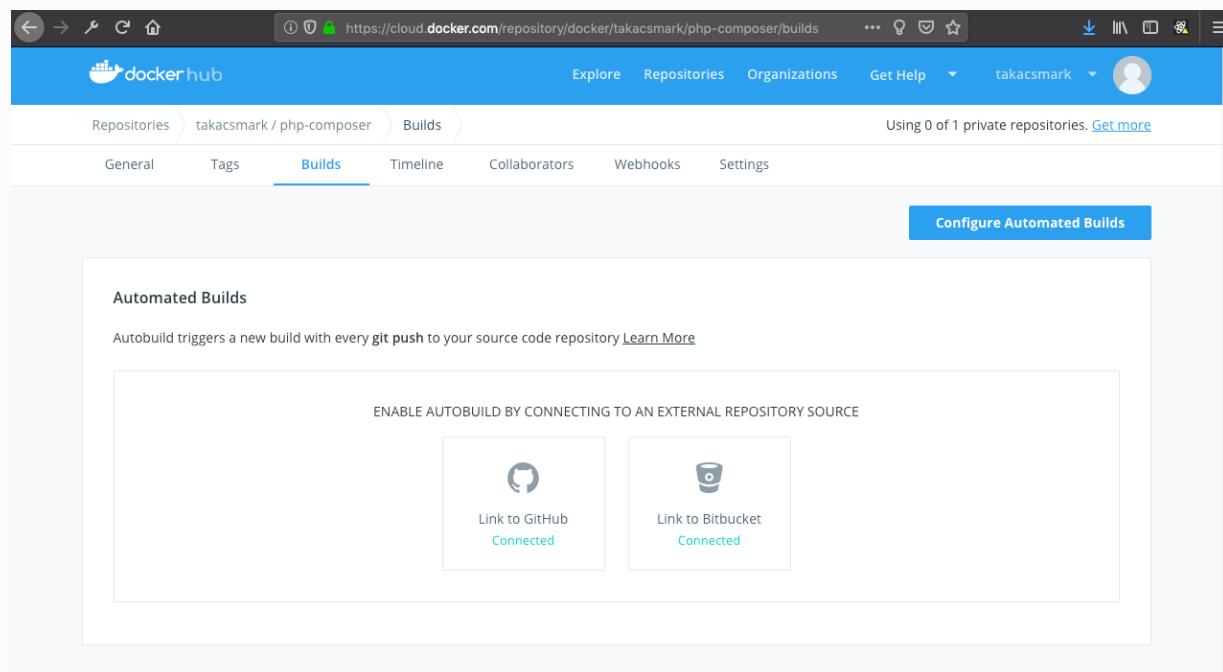
6. Adding the build automation to your Docker repository

Open the `php-composer` repository on the Docker Hub and click on `Builds`.

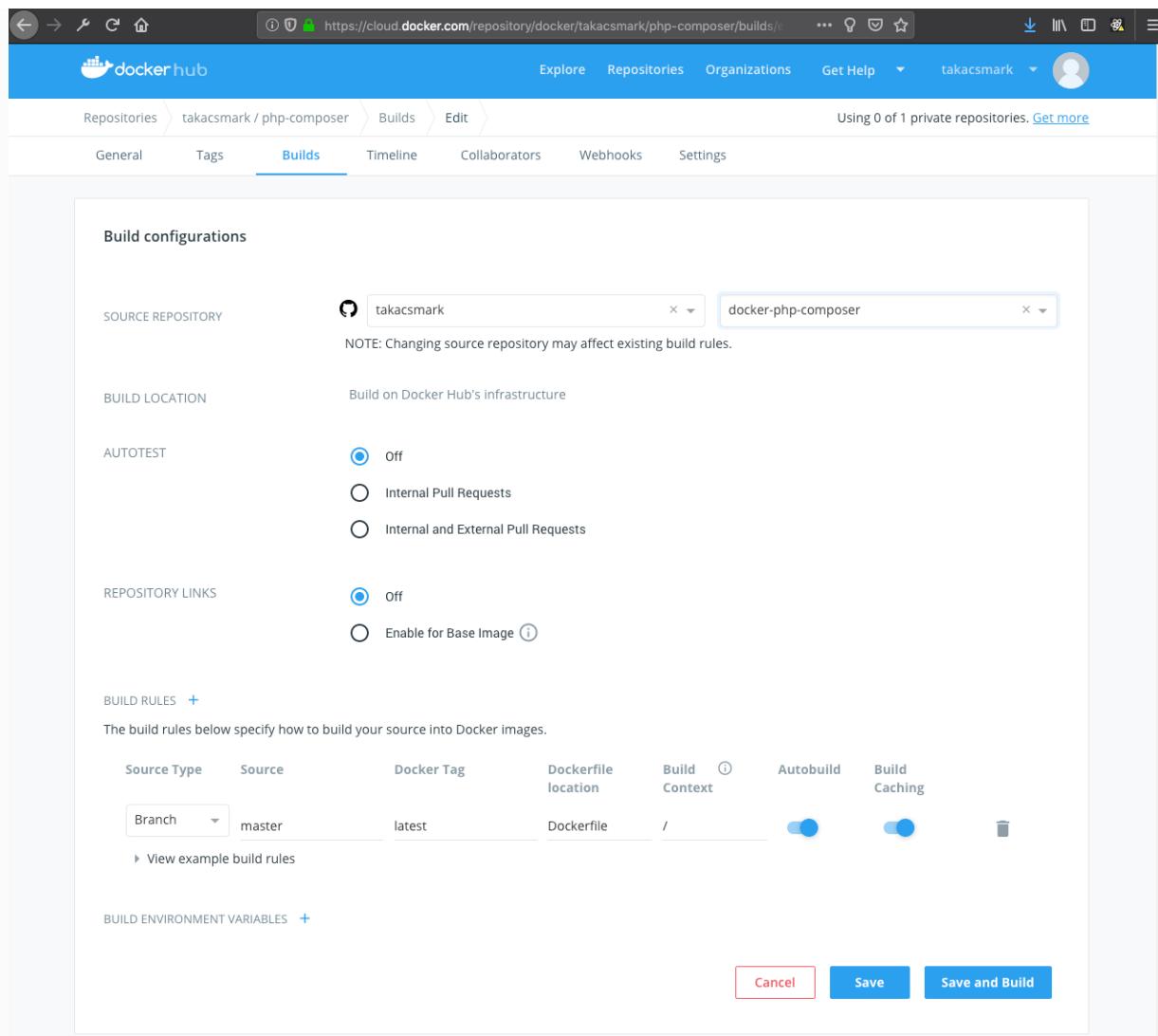
The screenshot shows a web browser displaying the Docker Hub interface. The URL in the address bar is <https://cloud.docker.com/repository/docker/takacsmark/php-composer/builds>. The page title is "Introducing the New Docker Hub: Combining the best of Docker Hub, Cloud and Store. [Learn more](#)". The header includes the Docker Hub logo, navigation links for Explore, Repositories, Organizations, Get Help, and a user account for "takacsmark". Below the header, there are tabs for Repositories, takacsmark / php-composer, and Builds. The Builds tab is highlighted with a blue underline. A green arrow points to this tab. To the right of the tabs, it says "Using 0 of 1 private repositories. [Get more](#)". Under the tabs, there are links for General, Tags, Builds, Timeline, Collaborators, Webhooks, and Settings. The main content area is titled "Automated Builds" and contains the sub-section "ENABLE AUTOBUILD BY CONNECTING TO AN EXTERNAL REPOSITORY SOURCE". It features two boxes: one for "Link to GitHub" (Disconnected) and one for "Link to Bitbucket" (Disconnected). There is also a blue button labeled "Configure Automated Builds".

Link your account to GitHub by pushing the “Link to GitHub” icon. This will take you to your linked accounts settings on the account settings page. Click “connect” next to GitHub.

Once the connection is set up, go back to the repository, now your GitHub account should be “connected”.



Clicking the GitHub icon again will take you to the build automation setup page. Please pick the GitHub organization and repository from the dropdown and you'll see all the settings like this:



The screenshot shows the Docker Hub build configuration interface for a repository named 'takacsmark / php-composer'. The 'Builds' tab is selected. The configuration includes:

- SOURCE REPOSITORY:** takacsmark / docker-php-composer
- BUILD LOCATION:** Build on Docker Hub's infrastructure
- AUTOTEST:** Off (radio button selected)
- REPOSITORY LINKS:** Off (radio button selected)
- BUILD RULES:** A table showing build rules:

Source Type	Source	Docker Tag	Dockerfile location	Build Context	Autobuild	Build Caching
Branch	master	latest	Dockerfile	/	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

View example build rules
- BUILD ENVIRONMENT VARIABLES:** (button)

At the bottom right are three buttons: Cancel, Save, and Save and Build.

Click save and build.

The build will take a while. You can follow the progress on the screen.

The screenshot shows the Docker Hub interface for the repository `takacsmark/php-composer`. The **Builds** tab is selected. The **Build Activity** section displays three recent builds with times: 2.1 min, 1.4 min, and 0.7 min. The **Automated Builds** section shows an Autobuild configuration for the `takacsmark/docker-php-composer` repository, which triggers on every git push to the master branch. The build status is **SUCCESS**. The **Recent Builds** section lists two entries: a successful build in 'master' and a Github Ping.

Once the build is done, you can check the “Tags” tab, and you’ll see that your image was created.

Now if you update the GitHub repository with some changes, it will automatically trigger a build on the Docker Hub. This sounds cool, let’s check this out!

7. Automatically triggering a build from GitHub

We'll apply a simple change by adding a readme file to our project. GitHub repositories need a readme file anyway so that the community knows how to use your open source project. Let's create a file called `README.md` in our repository and add the below contents:

```
## Overview

This repository contains a Docker image with PHP and the Composer

The Dockerfile has ONBUILD triggers to copy `composer.json` and the

Once `composer.json` is copied into the image composer install is
```

Let's add the new file to our repo, commit the changes and push the changes to GitHub:

```
~/Development/docker-php-composer$ git add README.md

~/Development/docker-php-composer$ git commit -am "readme added"

[master f0990de] readme added
 1 file changed, 7 insertions(+)
 create mode 100644 README.md

~/Development/docker-php-composer$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
```

```
Writing objects: 100% (3/3), 494 bytes | 494.00 KiB/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To github.com:takacsmark/docker-php-composer.git  
 07a10bc..f0990de master -> master
```

Now if you go to the Docker Hub, you'll see that a build has been triggered after the change automatically.

Once the image is built, you'll see a nice little extra on the "General" tab; your readme file is displayed as the full description of your repository. I think this starts looking good.

Also if you check out your GitHub repository in a browser, you'll see your readme file appear in a nice, user-friendly way.

Make the image more generic

Our PHP and Composer image is ready to use, and it looks pretty useful, I have showcased all the important points that I wanted to teach you in this exercise about Docker.

There is, however, one thing that I'd like to fix.

We have used the `ONBUILD` instruction in the Dockerfile a couple of times. This way we assumed that certain files would be present on the host machine build time.

A better practice is to provide a sample Dockerfile in the documentation of our repository that describes how to use our image as a parent image for other projects.

Please remove the lines that start with `ONBUILD` instructions in the base Dockerfile.

Now that we removed the `ONBUILD` part, we need to improve our documentation with a sample `Dockerfile` so that people will know how to use our image.

The easiest way is to create an example. We add the parts that so far existed in the `ONBUILD` instructions of the parent to the Dockerfile to our documentation. Let's update `README.md` in our php-composer project directory to reflect the changes:

Overview

This repository contains a Docker image with PHP and the Composer

**The image is meant to be used as the parent image for your Docker

This image uses the latest version of Composer. You can find detai

Usage

This image has Composer installed and available as a global command

Dockerfile example

You can create a Dockerfile based on this image and use it in your

```
```
FROM takacsmark/php-composer:7.2.2-apache

COPY ./composer.json /var/www/html/
RUN composer install && \
 rm -rf /var/www/html/composer-cache

COPY ./public /var/www/html/public
```

### `composer.json` example

Please make sure to specify a composer cache location. The above Dockerfile uses ONBUILD to do this. This is a good idea because it makes your Dockerfile portable, and it's a good idea to keep your code clean and simple.

```
{
 "config": {
 "cache-dir": "/var/www/html/composer-cache"
 },
 "require": {
 "slim/slim": "^3.0"
 }
}
```

```

I'm more happy with this version of the file, because it does not have version specific information hardcoded, and we are not making superfluous assumptions with `ONBUILD`.

Create your own Docker Registry

We used the Docker Hub, a public registry, in the examples so far. Docker has a component called a Docker Registry that you can use to set up your registry to store your organization's Docker images.

The Docker Registry is available as a Docker Image, so it is very convenient to set up and use your custom registry.

We'll set up a registry on your local machine. You can set up an externally available Docker Registry with a TLS certificate for your domain. You can read more about this on the Docker site here:

<https://docs.docker.com/registry/deploying/#run-an-externally-accessible-registry>.

The Docker Registry has its own page on the Docker Hub: https://hub.docker.com/_/registry/. Let's pull the registry to your machine; I'll use the latest version, that is 2.6.2 right now.

```
$ docker pull registry:2.6.2
```

The `docker inspect registry:2.6.2` command will reveal that the image exposes port 5000. So we can start our registry like this:

```
$ docker run --rm -d -p 5000:5000 registry:2.6.2
```

When you push your images to the Docker Hub, you use the naming convention of *userame/repository_name* when you are naming your images.

When you use a private registry, the naming convention changes to *registry_url/repository_name*, which means that you have to re-tag your images to a different name to be able to push them to your registry.

This may lead to confusion in projects and may require you to create additional scripts and environment settings to manage your assets.

Let's pick an image on your machine and push it to your local registry. I'll pick alpine:3.7 and first rename it to be used with my registry.

```
$ docker image tag alpine:3.7 localhost:5000/myalpine:3.7
```

We are ready to push:

```
$ docker push localhost:5000/myalpine:3.7  
The push refers to repository [localhost:5000/myalpine]  
cd7100a72410: Pushed  
3.7: digest: sha256:8c03bb07a531c53ad7d0f6e7041b64d81f99c6e493cb39
```

Let's remove both the original Alpine image and our own Alpine image from the host machine so that we'll not have

the image cached on the system and let's make a clean pull:

```
$ docker image rm alpine:3.7  
$ docker image rm localhost:5000/myalpine:3.7
```

Now let's pull our image from the local registry:

```
$ docker pull localhost:5000/myalpine:3.7  
3.7: Pulling from myalpine  
Digest: sha256:8c03bb07a531c53ad7d0f6e7041b64d81f99c6e493cb39abba:  
Status: Downloaded newer image for localhost:5000/myalpine:3.7
```

The registry stores your image data in a volume. You can specify a named volume or bind mount a host directory to store your images.

Integration with CI tools

CI/CD integration is discussed in the Chapter called “Build, ship and run a Flask web app end to end”.

Ship Summary

The great thing about Docker is that during the shipment of your project, you can use the same artifacts that you used during the build stage. Depending on your shipment workflow and automation tools, this may happen in various ways.

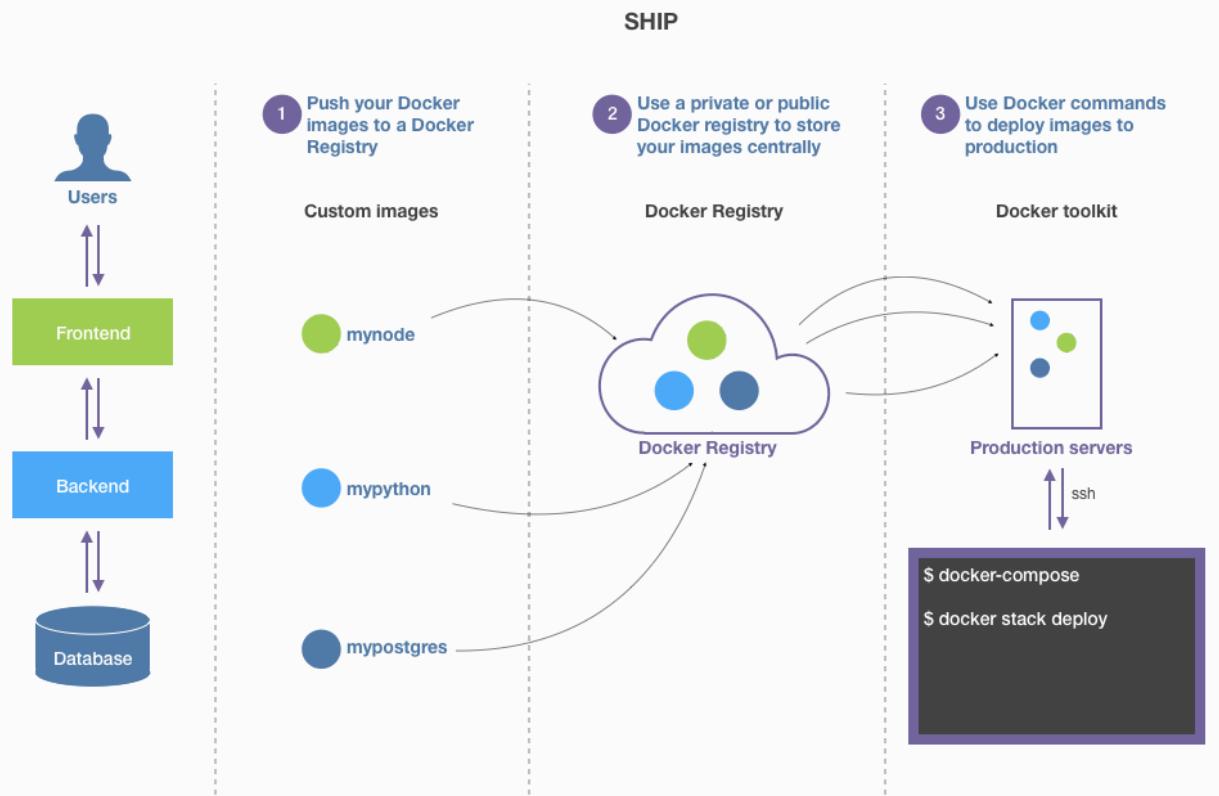
When working with Docker, the best practice is that you ship your Docker Image(s) to a public or private registry and you'll pull and start them on your production infrastructure with Docker tools.

People new to Docker often think that you ship your code to your production server independently and then you use Docker to start up a container separately and add your code on the server. This is technically feasible, but this is not the Docker way of shipping applications.

When you use Docker, your Docker images contain both your application and the environment it needs to run. Your code lives in the image, not outside. Your image is your application ready to run in its own environment.

The idea is to ship your image to your production environment and start up as many containers using your

Docker image as needed by your application. This is especially beneficial if you run multiple replicas of your application on different servers. All you need to do is to distribute the right Docker image to all servers in your cluster via a Docker Registry.



Run dockerized applications

In this chapter, we'll build examples to run our Docker based applications in a production-like scenario.

You can use the basic methods like `docker container run` and `docker-compose up` in smaller and simpler projects to run applications in production mode. You can even specify policies for restarting containers in case of a failure, and you can specify resource constraints with these tools. Feel free to explore the command line reference to learn about these methods.

In this chapter, I would like to introduce you to more Docker tools that are specifically geared to the needs of production operations. In this chapter, we'll run our Docker based applications in computer clusters.

Run applications in a cluster

Docker was designed with multi-host, horizontally scaled production operations in mind. This means that you can use Docker to build applications that run on multiple servers and run multiple instances of your containers so that you can serve a high load with your applications.

You can use the same Docker artifacts that we've already learned to run your application in a cluster. The Dockerfile defines your images, and the Docker Compose file defines how your containers should be run and how they should interact.

We used the `docker-compose` command to start up multi-container applications on your machine. Docker Compose works on a single host, but it does not work in a computer cluster, where multiple computers work together.

To run your multi-container application in a computer cluster, you need a solution that can manage your application across several computers. Such a solution is called an orchestrator.

The orchestrator's key roles and responsibilities include:

- Abstract away the underlying infrastructure details
- Maintain the desired state of your cluster described in a descriptor file like the Compose file
- Scale your services based on your priorities
- Secure cluster network communications
- Allow service discovery in the cluster
- Load balance your services

- Rolling update of services

Docker provides Docker Swarm as its built-in container orchestration solution.

There are three main products considered by most projects for container orchestration: Docker Swarm, Kubernetes by Google and Apache Mesos. HashiCorp's Nomad project is often mentioned as the fourth option that most teams consider.

We'll work with Swarm and Kubernetes in this book.

Kubernetes was a Google project, open-sourced in 2014. It's maintained by the Cloud Native Computing Foundation. It's open source, and all of its services are free. Kubernetes is the successor of Google's large-scale cluster management solution called Borg. Kubernetes carries long years of Google's cloud experience.

Kubernetes has the largest community amongst the options I listed above, and it has the highest adoption in container orchestration today. Kubernetes is a clear market leader, and the fact that it's free and open source makes it a strong player in the long run.

Docker Swarm was first announced in 2014 and evolved quickly. Both Kubernetes and Swarm were successfully tested with thousands of nodes and tens of thousands of containers.

Swarm is easy to set up; it's supported by Docker natively, no special installation is required. Kubernetes is more complex to set up, especially if you are installing on bare metal servers. Due to Kubernetes' wide adoption, Docker's support for Kubernetes grows with every release.

Your first Swarm cluster

Docker Swarm comes bundled with the Docker Engine, so you don't need to install a new component to create a Swarm.

A Swarm cluster consists of multiple computers. The computers can be physical or virtual machines. You can run a cluster on your laptop for development and testing, or you can set up a cluster in your corporate infrastructure or the cloud.

Computers in a Swarm cluster are called nodes. Nodes can play two roles in a Swarm:

- Manager nodes are used to manage the cluster; you can execute Docker's Swarm management commands on manager nodes.
- Worker nodes are used to run your containers. Swarm schedules containers on worker nodes.

A Swarm node acts as worker and manager at the same time unless you explicitly limit its role.

Let's create a Docker Swarm on your machine with the `docker swarm init` command:

```
$ docker swarm init
```

```
Swarm initialized: current node (q007k1uz4ys4fj2stn55e1wsy) is now
```

```
To add a worker to this swarm, run the following command:
```

```
    docker swarm join --token SWMTKN-1-0h9uzb2rrqu5g9a1ts08ul04kks-
```

```
To add a manager to this swarm, run 'docker swarm join-token manager' and
```

You created a Swarm on your machine, meaning you created a computer cluster with a single computer in the cluster, that is your local machine.

You can add further machines to the cluster as described in the command output above (`docker swarm join --token SWMTKN-1-0h9uzb2rrqu5g9a1ts08ul04kksa8j40ua5nugjosky8sg23s-80sg1z7esixsx5zd0eqola066 192.168.65.3:2377`). Other nodes can join your Swarm by IP address with a join token in either master or worker role. The output contains the command to join your Swarm as a worker. We'll do this later. Please note that one node can be part of one swarm only.

After `docker swarm init`, your host machine has the manager role in the cluster, and your host is also the single worker

node.

Managing your Swarm

The `docker swarm` command gives you tools to manage your Swarm cluster, and you can conduct the following key tasks:

- initialize a Swarm
- get Swarm join tokens
- join the Swarm
- update the Swarm
- leave the Swarm

Initialize a Swarm

We already used the `docker swarm init` command. You can use `docker swarm init --advertise-addr <ip|interface>[:port]` to specify the address to join your Swarm.

Get Swarm join tokens

Your Swarm has two tokens, one for joining the Swarm as a manager and another one for joining as a worker. You can

acquire these tokens by using `docker swarm join-token`
(worker|manager) .

```
$ docker swarm join-token manager
```

To add a manager to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-0h9uzb2rrqu5g9a1ts08ul04kks
```

```
$ docker swarm join-token worker
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-0h9uzb2rrqu5g9a1ts08ul04kks
```

Join the Swarm

You can use the `docker swarm join` command with either a manager or worker token as shown in the previous point.

Update the Swarm

You can use the `docker swarm update` command to update parameters of your Swarm.

Leave the Swarm

You can use `docker swarm leave` on a Swarm node to make it leave the Swarm.

If you want to remove the Swarm entirely, you have to force leave the Swarm on the last manager node with `docker swarm leave -f`. You use this command to leave the single node Swarm on your machine that you just started.

Your first Swarm deployment

Deploying an application to a Swarm cluster is convenient because you can use a Docker Compose file just like the ones we used before.

We used a Compose file with the `docker-compose` command before, we started up containers on a single machine, plus we were able to create a user-defined network for the application and defined named volumes to store data.

The `docker-compose` command is limited to a single host, and it has no load balancing features.

To deploy your application described in your Compose file(s), you use the `docker stack deploy` command.

The key difference between Docker Compose and Docker Stack is that you use Docker Compose on a single computer,

while you use Docker Stack in a Swarm cluster. Both commands use the Compose file to describe your stack.

You'll find that certain sections of the Compose file will only work with Docker Compose, and some other options will work with Docker Stack. For example, the `build` option in your service description in the Compose file will work with Docker Compose only and will be ignored by Docker Stack.

You can define deployment preferences under your services in the Compose file with the `deploy` property. This will be used by Docker Stack, and it will be ignored by Docker Compose.

The [Docker Compose file reference](#) has clear information about Docker Compose and Stack compatibility.

The reason behind the double role of the Compose file is historical. The Docker Compose command existed first. It had the stack description in the Compose file, but it was not capable of Swarm features.

Then Swarm and Stack were developed, today we can use our existing Compose files rather than moving to a new format. The benefits are great, but we have to keep in mind that certain parts of the Compose file will work with Compose or Stack only.

Let's deploy your first application to our Swarm. We'll create a Flask application similar to the one in `example5`.

I created a directory called `example12` with the below empty files:

```
~/Development/example12$ tree
.
├── Dockerfile
├── app.py
└── docker-compose.yml
```

My Dockerfile is almost like the one in example 5:

```
FROM python:3.6.4-alpine3.7

RUN pip install Flask
WORKDIR /usr/src/flaskapp
COPY ./app.py .

CMD python app.py
```

My `app.py` is like the one in example 5:

```
import socket

from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World! - " + socket.gethostname()

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

I have included `socket`, to have access to `socket.gethostname()`, this will give me the name of the host machine. In our case, the name of the host machine will be the container id. We'll use this to see how our application scales.

Let's define our `docker-compose.yml`:

```
version: '3'

services:
  python:
    build: .
    image: takacsmark/example12:1.0
    ports:
      - 5000:5000
```

We have to build our image with `docker-compose build` before we can deploy to the Swarm.

```
~/Development/example12$ docker-compose build
```

Let's deploy our stack with `docker stack deploy`. We have to specify the name of the stack in the command, let's call it `flaskapp`, and we also have to define the location of the compose file with `-c` or `--compose-file` flag. Our command is as follows:

```
~/Development/example12$ docker stack deploy flaskapp \
-c docker-compose.yml

Creating network flaskapp_default
Creating service flaskapp_python
```

Two things happened:

1. Docker automatically created a network for our stack, called `flaskapp_default`, the type of this network is `overlay`, we can use it to connect multiple Docker hosts. You can use `docker network` commands to list and inspect your new networks.
2. Docker started up the `python` service and called it `flaskapp_python`.

As you can see both network and service name got prefixed with the name of the stack, `flaskapp`.

You can access the Flask app on <http://localhost:5000>:



Managing your Stack

These are the typical tasks you'll do with Docker Stacks:

- deploy or update a stack
- list stacks
- list services in a stack
- list tasks in a stack
- remove a stack

Deploy or update a stack

We have used `docker stack deploy` to deploy our stack. You can use the same command to update your stack when your configuration or source changes.

You can deploy more than one stacks to a Swarm. This is useful when you want to structure your application better.

List stacks

You can list the stacks that are deployed to your Swarm:

```
~/Development/example12$ docker stack ls
```

NAME	SERVICES
flaskapp	1

List services in a stack

The services that you define in your Compose file will be represented as services in the Swarm. There is a dedicated Docker management command called `docker service` to manage services in a Swarm. (Please note that services started with Docker Compose are not represented as services managed with `docker service`. Docker Services only apply to Swarms.)

This is useful to manage services in your stack individually. Let's list the services in our stack:

```
~/Development/example12$ docker stack services flaskapp
```

ID	NAME	MODE	REPLICAS
x5jqktr2jkix	flaskapp_python	replicated	1/1

Since a Swarm is a computer cluster, the service information shows the number of replicas running on all nodes. We'll scale up this app in a minute to see more of this.

List tasks in a stack

Tasks are the individual instances of a service. When you create service in the Swarm, the Swarm will schedule tasks

according to the deployment preferences you describe in the Compose file.

Let's list the tasks in our stack:

```
~/Development/example12$ docker stack ps flaskapp
```

ID	NAME	IMAGE
wqsjvltzsq0a	flaskapp_python.1	takacsmark/example12:1.0

The list contains all tasks scheduled across all nodes. A task is a container running on one of the nodes in the Swarm. The notion of a task is not completely equal to the container, though. The difference is that the task is created and managed by the Swarm, you do not manage tasks directly.

When you look at a task in the Swarm, you don't know where it is running, and you do not have a management command like `docker task` to manage tasks. You configure services in your Compose file or with the `docker service` command, and Docker will take care of tasks for you automatically.

This also means that you should not need to operate with `docker container` commands in the Swarm.

Remove a stack

You can remove a stack with `docker stack rm` if you need to.

Your first scaled service

We have one service running in our stack now, it's called `flaskapp_python`. We can either use the `docker service` command or the Compose file to scale this service. Let's see how we can use `docker service` first:

```
~/Development/example12$ docker service scale flaskapp_python=2  
  
flaskapp_python scaled to 2  
overall progress: 2 out of 2 tasks  
1/2: running [=====]  
2/2: running [=====]  
verify: Service converged
```

Now we have 2 replicas running:

```
~/Development/example12$ docker stack services flaskapp  
  
ID          NAME      MODE      REPLICAS  
x5jqktr2jkix  flaskapp_python  replicated  2/2
```

Let's check the status of individual tasks:

```
~/Development/example12$ docker stack ps flaskapp
```

ID	NAME	IMAGE
mb834gsoo716	flaskapp_python.1	takacsmark/example12:1.0
ofdawds5z0uj	flaskapp_python.2	takacsmark/example12:1.0

The command output shows that we are running two tasks in the `flaskapp` stack, these are the two replicas of the `flaskapp_python` service.

Now you have altering output in your Flask app if you refresh the page multiple times depending on which replica has been invoked to provide a response:



Managing Services

The Compose file is used in projects to define services and their desired state in the Swarm. You can use the `docker service` command to execute regular management tasks on your Swarm services. These are the typical activities:

- create a service
- list services
- list tasks

- scale a service
- show service logs
- inspect a service
- update a service
- rollback a service
- remove a service

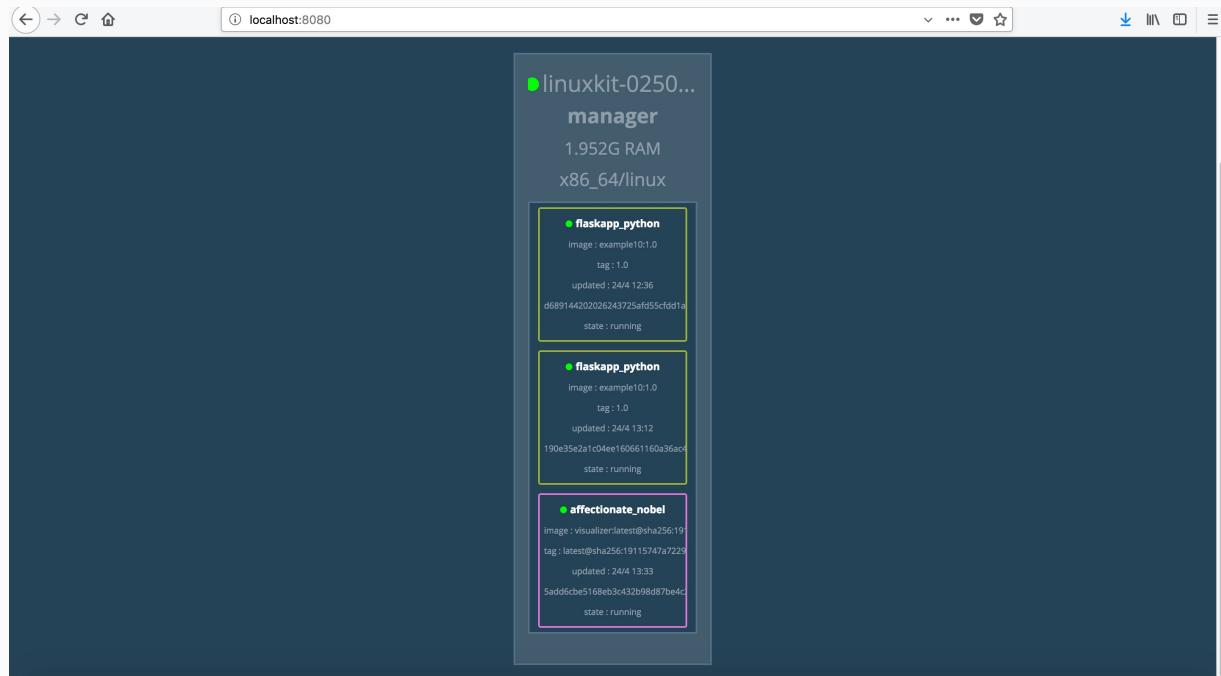
Create a service

The `docker service create` command is similar to `docker container run` in the sense that you specify an image name as an argument and Docker will start up a service based on that image and its options. `docker service create` is more than `docker container run`, because it'll start your service in a Swarm cluster, and you can specify deployment-related constraints, like the number of replicas to be run.

Let's create a new service on our Swarm cluster that displays visual information about the Swarm. This service uses the visualizer image:

```
~/Development/example12$ docker service create \
-p 8080:8080 \
--constraint=node.role==manager \
--mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
dockersamples/visualizer
```

If you visit <http://localhost:8080>, you'll see the visual representation of two `flaskapp_python` replicas and the visualizer running in the Swarm:



Update a service

You can change the configuration of a service with `docker service update`. The command has a rich option list that you can view with `docker service update --help`.

List services

You can use the `docker service ls` command to list the services in the Swarm.

List tasks

The `docker service ps` command lists the tasks for one or more services in the Swarm.

Scale a service

We have already used the `docker service scale` command to scale our `flaskapp_python` service.

Show service logs

You can use the `docker service logs` command to watch logs of either a service or a task. The command has options similar to `docker container logs`.

Inspect a service

Inspecting a service with `docker service inspect` will give you detailed information. Run `docker service inspect flaskapp_python` and note that the current state and previous state of the service are both displayed.

Rollback a service

You can rollback a service to its previous state with `docker service rollback`. Let's try it with our service:

```
~/Development/example12$ docker service rollback flaskapp_python
```

The service was rolled back to the previous state when we run only one replica. You can check the visualizer and see that one `flaskapp_python` box has disappeared.

Remove a service

You can use the `docker service rm` command to remove a service.

Run your stack on multiple nodes

We'll add multiple nodes to our swarm simulating a multi-machine cluster and run our stack on it. In this chapter, we start two nodes locally, and later on, we'll move to multiple nodes in the cloud.

First, let's clean up our current example by removing the stack and leave the swarm with the following commands:

```
~/Development/example12$ docker stack rm flaskapp  
~/Development/example12$ docker swarm leave -f  
Node left the swarm.
```

We create a cluster of two virtual machines on your computer. You need a virtualization solution for this task, let's start by installing VirtualBox, a free virtualization software.

Install VirtualBox

Get your copy of VirtualBox from <https://www.virtualbox.org/wiki/Downloads>. If you have Windows 10, you don't need to install VirtualBox; please follow the steps below.

Install docker-machine

Docker for Mac and Windows comes bundled with `docker-machine`.

Please try running `docker-machine` in Terminal. If your system does not have `docker-machine`, please install it as described on the Docker site here:

<https://docs.docker.com/machine/install-machine/>.

Start virtual nodes

Let's start two virtual machines that will be our nodes in our cluster:

```
~/Development/example12$ docker-machine create \
--driver virtualbox myvm1

~/Development/example12$ docker-machine create \
--driver virtualbox myvm2
```

If you are on Windows 10 you can use `--driver hyperv`, please read more about this here:

<https://docs.docker.com/machine/drivers/hyper-v/>.

The `docker-machine create ...` command starts virtual machines that already have Docker installed. We have two machines with the Docker daemon running. We can list them with this command:

```
~/Development/example12$ docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL
myvm1	-	virtualbox	Running	tcp://192.168.99.100:2376
myvm2	-	virtualbox	Running	tcp://192.168.99.101:2376

You can see the Docker version that's running in the machine in the command output. **You can also see the ip**

address of the machines; this will be useful in the example.

Start the Swarm on the manager virtual machine

Let's pick one virtual machine as the manager. I chose `myvm1`; it's easier to think of number 1 as the manager.

We use the `docker-machine ssh` command to open a shell to the virtual machine.

```
~/Development/example12$ docker-machine ssh myvm1
```

We have a prompt inside `myvm1`, and we can execute commands, so let's start a Swarm. We'll do this a bit differently than before because we'll advertise the IP address of the master so that clients can connect to it.

```
docker@myvm1:~$ docker swarm init \  
--advertise-addr 192.168.99.100
```

```
Swarm initialized: current node (in9kssw5tfir3rr1r2xtfu681) is now
```

```
To add a worker to this swarm, run the following command:
```

```
docker swarm join --token SWMTKN-1-1xtqbz5db41l5w7hk0p1r8r4lw1
```

```
To add a manager to this swarm, run 'docker swarm join-token manager'
```

Note how I used the IP address of `myvm1` that we got from the previous `docker-machine ls` command output.

As you can see, the output of the command contains the join command that we can use in `myvm2` to join the swarm. Please leave `myvm1` by issuing the `exit` command in Terminal.

```
docker@myvm1:~$ exit
```

SSH into `myvm2` and join the swarm like this:

```
~/Development/example12$ docker-machine ssh myvm2  
  
docker@myvm2:~$ docker swarm join --token SWMTKN-1-1xtqbz5db41l5w7  
bgfty9y3uef-cmghx7gv05gkzqat00povt99g 192.168.99.100:2377  
This node joined a swarm as a worker.
```

Please use the command that you got in the output of `docker swarm init` on the manager node. You can leave `myvm2`; we have the swarm set up.

Push your example image to the registry

Your swarm cluster will fetch Docker images from the Docker Hub or any public or private registry that you configure.

Build your image:

```
~/Development/example12$ docker-compose build
```

Push your image to the Docker Hub:

```
~/Development/example12$ docker image push \
takacsmark/example12:1.0
```

Your manager node(s) schedule application service tasks in your cluster. Your nodes will dynamically pull the images that are needed to conduct work from a central registry.

Deploy your stack to the Swarm

Let's deploy the `example12` stack using the manager node. You can run Swarm management Docker commands on the manager node(s) only.

If you SSH into `myvm1` and try to deploy the stack, you'll realize that the local `docker-compose.yml` file is not accessible in the virtual machines. Docker provides a neat solution to this situation. We can configure our shell to talk to the

Docker daemon in the virtual machine with `docker-machine env` :

```
~/Development/example12$ docker-machine env myvm1

export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/home/takacsmark/.docker/machine/machines/myvm1/certs"
export DOCKER_MACHINE_NAME="myvm1"
# Run this command to configure your shell:
# eval $(docker-machine env myvm1)
```

Focus on the last two lines. They say to run the command `eval $(docker-machine env myvm1)` to configure the shell. So let's do it:

```
~/Development/example12$ eval $(docker-machine env myvm1)
```

We get no output, but there is a way to verify the effects of the command. Configuring the shell will result in `myvm1` becoming active in `docker-machine ls` :

NAME	ACTIVE	DRIVER	STATE	URL
myvm1	*	virtualbox	Running	tcp://192.168.99.100:2376
myvm2	-	virtualbox	Running	tcp://192.168.99.101:2376

The little `*` next to `myvm1` denotes that it is the active node now. Another way to check this is this command:

```
~/Development/example12$ docker-machine active  
myvm1
```

Now you can deploy your stack as you did before:

```
~/Development/example12$ docker stack deploy flaskapp \  
-c docker-compose.yml  
  
Creating network flaskapp_default  
Creating service flaskapp_python
```

The only difference is that you'll access the services using the IP address of your nodes, not `localhost` anymore. You can access the Flask app using the IP address of both `myvm1` and `myvm2`. You can find out the IP address of the virtual machines with `docker-machine ls`, as we did before.

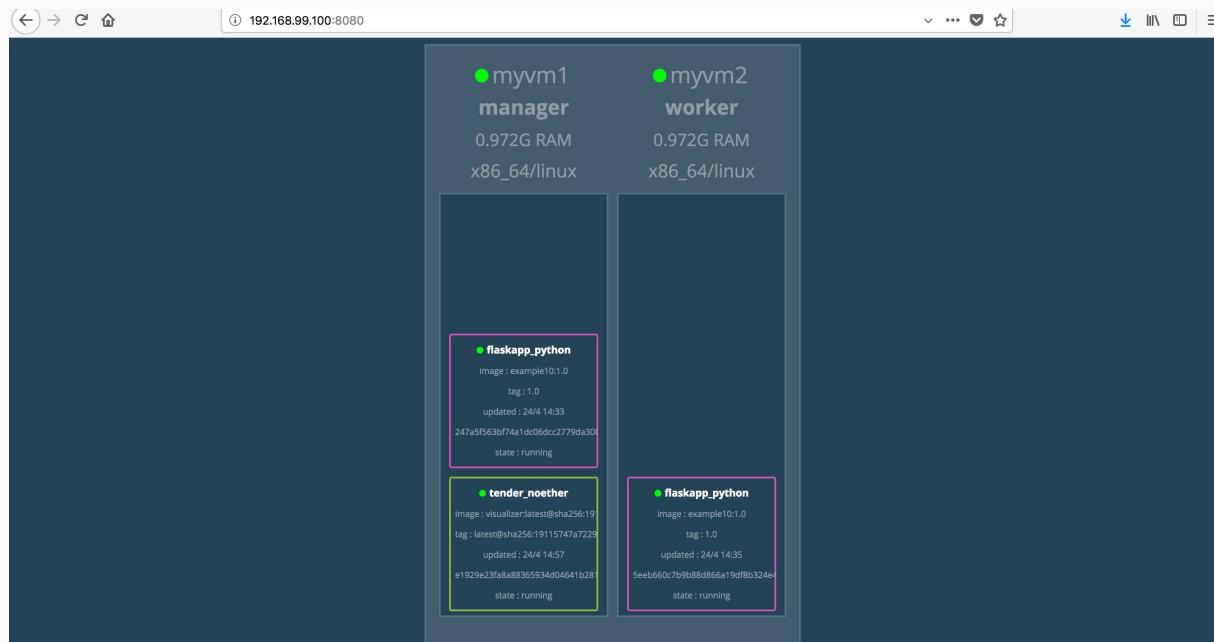
Let's check the following addresses in our browser. They'll give the same output. Please use your IP addresses; I'm using mine here: manager node - <http://192.168.99.100:5000/>, worker node - <http://192.168.99.101:5000/>. Deployment takes a while, so be patient and refresh your browser.

Add the visualizer service

Let's add the visualizer service again:

```
~/Development/example12$ docker service create \
-p 8080:8080 \
--constraint=node.role==manager \
--mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
dockersamples/visualizer
```

You can access the visualizer on both nodes on port 8080. The image shows both nodes and the distribution of the tasks.

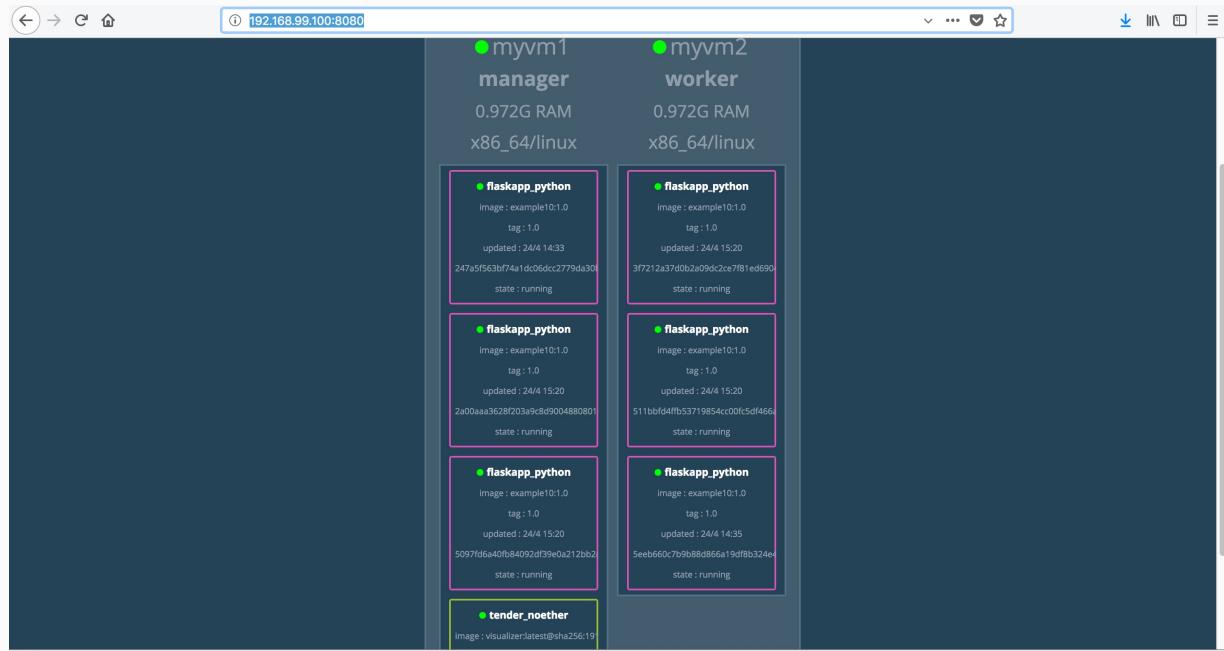


Scaling your service

Let's scale our Flask service to have 6 replicas in the Swarm:

```
~/Development/example12$ docker service scale flaskapp_python=6
```

Docker distributes the new replicas automatically across the nodes.



Invoking our service several times in the browser on <http://192.168.99.100:5000/> will produce responses from 6 different tasks now.

Managing nodes

You can use the `docker node` command to manage the nodes in the Swarm. The typical tasks are the following:

- list nodes

- list running tasks
- inspect nodes
- promote, demote nodes
- update nodes
- remove nodes

List nodes

You can list nodes in the Swarm with `docker node ls`:

```
~/Development/example12$ docker node ls
```

ID	HOSTNAME	STATUS
i3h271agh7kw7set82rqfq3h *	myvm1	Ready
4s6lom8bx9cxto4pyg0n0pizw	myvm2	Ready

List running tasks

You can list running tasks on a node with `docker node ps [NODE]`. It will list the current node if you omit the NODE argument.

Inspect nodes

The `docker node inspect` command will give you detailed information about a Swarm node.

Promote, demote nodes

You can use the `docker node promote`, and `docker node demote` commands to promote a node to manager or demote a node to worker, respectively.

Update nodes

The `docker node update` command lets you change the properties of a node. You can use this command to add or update node labels, which you can use later to define your service placement constraints.

Remove nodes

You can remove one or more nodes from the Swarm with `docker node rm`.

Add deployment configuration to Compose file

We can define deployment configuration in our `docker-compose.yml` file, so let's define 2 replicas of the python service:

```
version: '3'

services:
  python:
    build: .
    image: takacsmark/example12:1.0
    ports:
      - 5000:5000
    deploy:
      replicas: 2

  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
```

The `deploy` keyword gives us the opportunity to define deployment options in the `docker-compose` file. You can read the specs here <https://docs.docker.com/compose/compose-file/#deploy>.

We specified a deployment constraint for the visualizer. The constraint `[node.role == manager]` means that we deploy the visualizer on a manager node, not a worker node. This

image was designed to run on a manager; this is the way it can discover Swarm services.

Please remove the running visualizer service that we started separately with

```
~/Development/example12$ docker service rm NAME
```

We can redeploy the stack with the same command we used to start it, Docker will initiate an update of the stack automatically:

```
~/Development/example12$ docker stack deploy flaskapp \
-c docker-compose.yml
```

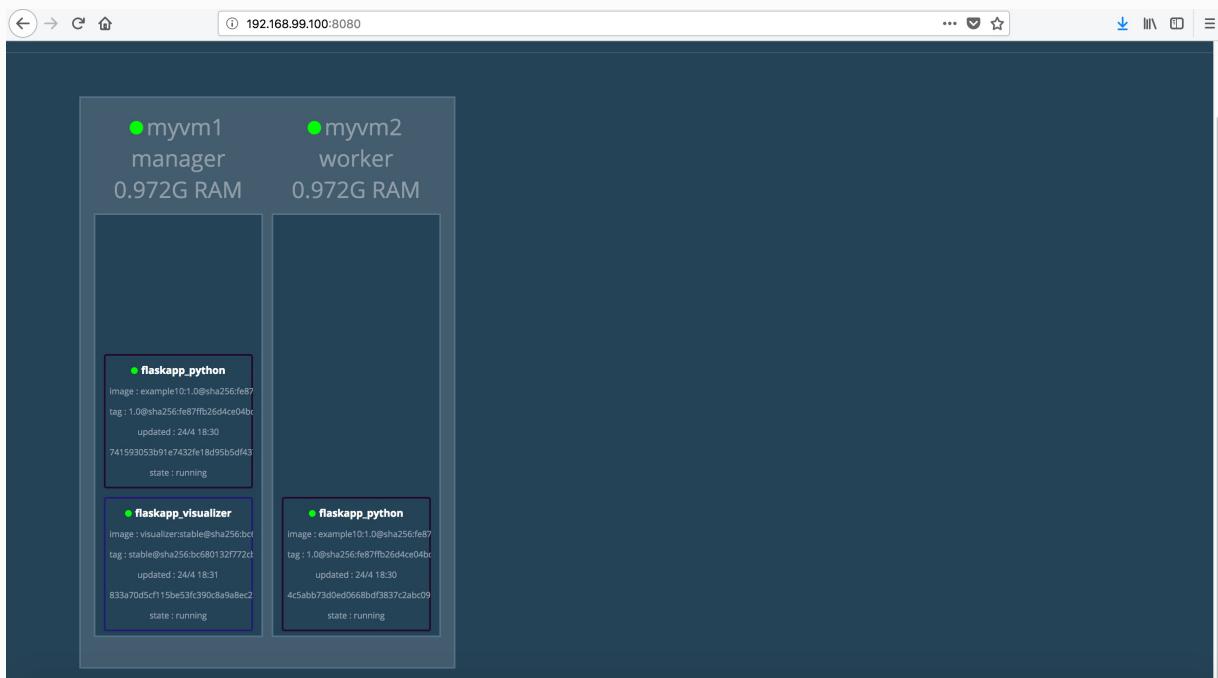
Additional to the default networks, `bridge`, `host` and `none`, Docker has created 3 new networks:

- `docker_gwbridge` connects Docker daemons on the nodes in the Swarm. This network was created when we initiated the swarm with `docker swarm init`
- `ingress` is the default overlay network where Docker services are connected. The services are connected in a routing mesh, which means that you can reach any service on any node (on the service's port) even if the given service is not

running on that node. The ingress network was also created when we run `docker swarm init`.

- `flaskapp_default` is the overlay network that was defined automatically for our stack when it was deployed.

You may have to wait for a short while until Docker pulls the visualizer image and visit <http://192.168.99.100:8080> and check the visualization of our swarm:



Shut down the cluster

Let me show you how you can disconnect your shell from the virtual machine. First, let's remove the stack with:

```
~/Development/example12$ docker stack rm flaskapp
```

```
Removing service flaskapp_python
Removing service flaskapp_visualizer
Removing network flaskapp_default
```

Then disconnect the shell:

```
~/Development/example12$ eval $(docker-machine env -u)
```

You can stop and start your virtual machines with `docker-machine stop` and `docker-machine start`.

Run Summary

You can run your Docker stack in the cloud or on-premise with Docker tools. For small stacks, you can use the Docker Compose tool.

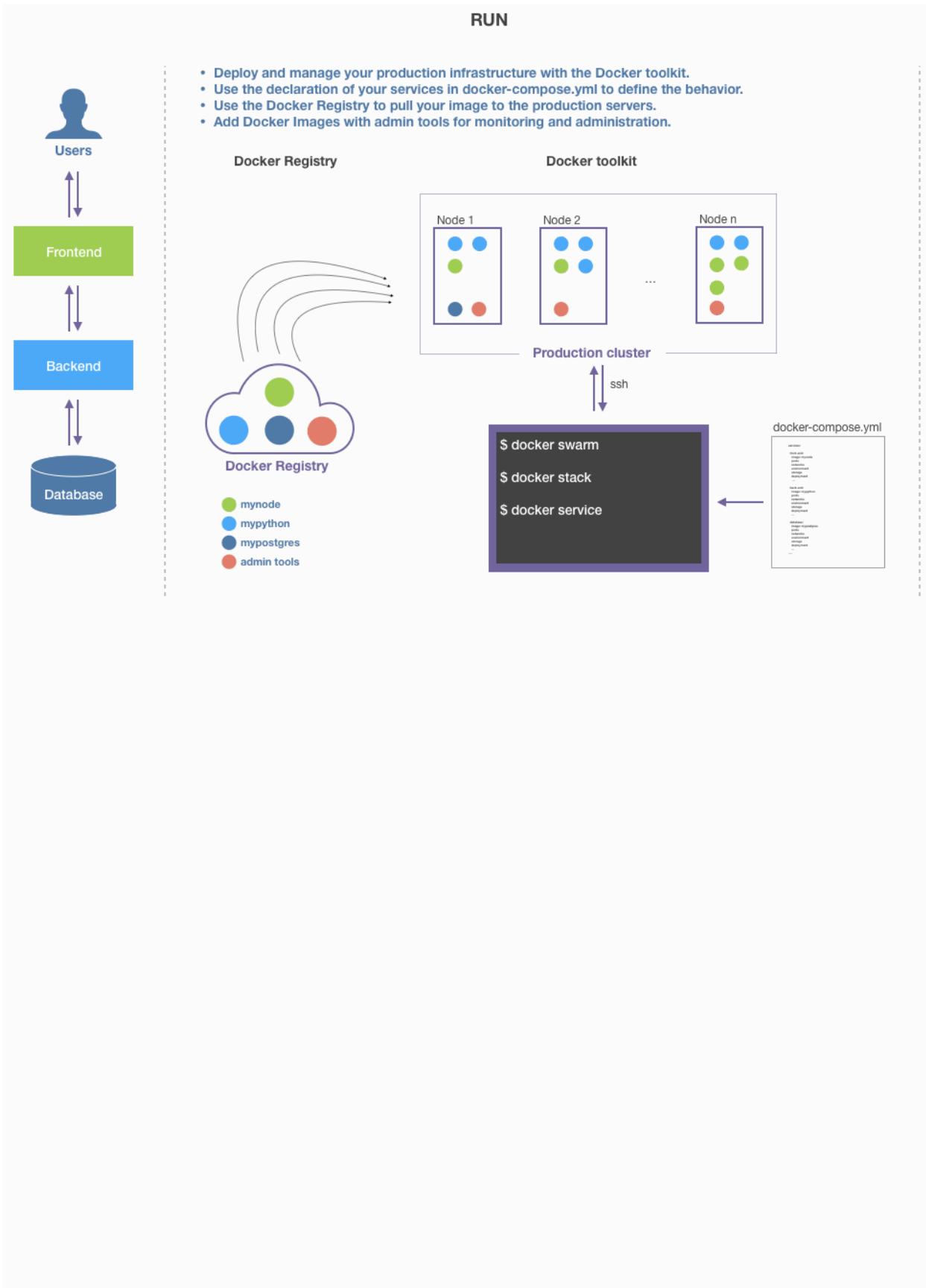
For high availability, fault tolerant clusters there are dedicated solutions to manage your cluster, called orchestrators.

Docker Swarm is Docker's built-in orchestrator solution. There are also other solutions to manage containerized clusters, the market leading product today is Kubernetes by Google.

You use a public or private registry to pull your project's Docker images to your production cluster. You can use a Compose file to define deployment constraints and run your application in a cluster.

You can use Docker commands to manage various aspects of your cluster. The key components are the Swarm, stacks, services, nodes, and machines.

The figure below depicts the orchestration scenario with Docker Swarm.



Build, ship and run a Flask web app end to end

Congratulations! You have learned the basics of building, shipping and running dockerized applications.

Let's develop your skills further building an application end to end together with the tools you already know and let's learn some new tricks along the way.

In this example we'll take an existing Flask application boilerplate from GitHub and dockerize it. We'll build with our project team in mind, and we'll create the development environment, CI integration, and production management.

We'll deploy our application to AWS with Docker Swarm and with we'll deploy with Kubernetes, too.

I've picked a robust initial codebase for our project. In this example, we'll work with the Flask boilerplate repository on <https://github.com/sloria/cookiecutter-flask>. All credit for the repo codebase goes to the contributors of that repo. We'll take the application code and extend the solution so that you can build, ship and run your projects with Docker based on this template.

Initialize the project

If you have a look at the GitHub repo at <https://github.com/sloria/cookiecutter-flask>, you can see that this project is a template and uses the cookiecutter project templating tool (<https://github.com/audreyr/cookiecutter>).

This means that this repository is a template for other projects and you can use the command `cookiecutter https://github.com/sloria/cookiecutter-flask.git` to generate the initial codebase for your project.

The cookiecutter tool lets you configure the template to reflect your specific situation. The configuration includes your name, which appears in the project license, the project name, your GitHub username, your project title, and description. All this information will be reflected in the project source code that we'll generate from the cookiecutter template.

We'll also set the name of the project directory as a config parameter.

According to the information on GitHub, we can generate the initial code in two steps: `$ pip install cookiecutter $ cookiecutter https://github.com/sloria/cookiecutter-flask.git`

As the first step, let's create a Docker image that generates the initial source code. We'll call this

process initialization.

The cookiecutter tool can be installed as a Python package with `pip`, so if we want to automate the project initialization, we should start from a Python Docker image, install cookiecutter and generate our project with our custom configuration.

I've built and tested the solution in the book with the Dec 14, 2018 version of the repo, with commit id

`bbb17404c6ac053161a2436b5f195cf44f0002cd` at

<https://github.com/sloria/cookiecutter-flask/commit/bbb17404c6ac053161a2436b5f195cf44f0002cd>.

We'll build our solution with this commit, because the repo is maintained actively and we need a stable version for our exercise.

Let's create our project directory as the first step. I call mine `example-flask-sql-ci` because we'll have a Flask app with an SQL database and continuous integration (CI) implemented in this example. My project is located in my home directory under `~/Development/example-flask-sql-ci`.

The plan is to use Docker images for initialization, development, testing, and production. So it's a good idea to create a dedicated `docker` directory in the project, we'll manage our Docker files in there.

I have created a directory called `init` inside `docker`, and my project looks like this now:

```
~/Development/example-flask-sql-ci$
```

```
tree . |— docker |  └ init
```

We can provide our custom configuration settings in a YAML file to cookiecutter. Let's create it under the `init` directory with the name `cookiecutter.yml` and the following contents:

```
default_context: full_name: "Mark Takacs" email:  
"mark@takacsmark.com" github_username: "takacsmark" app_name: "app"  
project_name: "Flask SQL CI Docker" project_short_description:  
"Docker example"
```

Please feel free to customize the information to yourself, note that the value of `app_name` will be used as the directory name for your generated application, so please avoid spaces. The best thing is to keep the value `app` so that it will be easier to follow the book.

Next, we create our `Dockerfile` under `docker/init/`. We'll start from a Python Alpine image. Copy the `cookiecutter.yml` configuration file into the image and run the project generation. I'll pass the template repository URL as an argument to the Dockerfile to make it more generic. This is the code:

```
ARG python_image_version FROM  
python:$python_image_version ARG repo ARG repo_release_version  
WORKDIR /root/ COPY cookiecutter.yml cookiecutter.yml RUN pip  
install cookiecutter && \ apk --update add git RUN git clone $repo  
&& \ cd cookiecutter-flask && \ git reset --hard  
$repo_release_version RUN cookiecutter --no-input --config-file  
cookiecutter.yml cookiecutter-flask
```

You should have great Dockerfile skills by now. I would explain some simple considerations. I chose `/root/` as the working directory, this is the home directory of the root user.

You can see that I defined three arguments with `ARG` and called them `python_image_version` , `repo` and `repo_release_version` .

We use `python_image_version` to externally specify the Python image version to be used. This will be useful during future version upgrades.

The argument `repo` represents the URL of the cookiecutter repo. The value of `repo` is used with `git clone` in the `$repo` form.

After cloning the repo with `git` , we reset the repository the the specific commit we are working with with the use of `repo_release_version` .

The last line in the Dockerfile is the command that will generate our source code from the template. With the `--no-input` flag we turn off user inputs for cookiecutter, so it'll run in non-interactive mode and will take inputs from the config file automatically.

When we build this image later, it will include the generated source code in the `/root/` directory under `/root/app` .

The next step is to build our image using the Dockerfile. Since we are creating a teamwork scenario, we'll declare our build flow in a `docker-compose.yml` file and build our image automatically, so that other team members can quickly reproduce the steps.

Let's add `docker-compose.yml` to our `docker/init/` directory with the following contents:

```
version: '3'
services:
  cookiecutter:
    build:
      context: .
      args:
        repo: https://github.com/sloria/cookiecutter-flask.git
        repo_release_version: bbb17404c6ac053161a2436b5f195cf44f0002cd
        python_image_version: 3.7.1-alpine3.8
    image: takacsmark/cookiecutter:1.0-initiator
    container_name: flask-sql-ci-initiator
    command: tail -f /dev/null
```

In our `docker-compose.yml` file, we defined one service called `cookiecutter`, and we defined the `build` configuration. Build context is the directory context of the Docker build, in our case, it's the current directory (`.`, i.e., `docker/init/`).

In the `build` section, we defined the `build` argument called `repo` under `args:`. This is the `repo` name argument that we are using in the Dockerfile.

We also defined the `repo_release_version` and `python_image_version` arguments, we'll centralize these in an environment file later on, but let's build gradually now.

The line `image: takacsmark/cookiecutter:1.0-initiator` defines the name of the image, this will be the result of the build.

The line `container_name: flask-sql-ci-initiator` specifies the name of the container to be used when we are starting up a container with `docker-compose up`. We defined the name explicitly so that we can automate `docker cp` with the container name and build a script to copy the generated files from the container to the host machine as you'll see soon.

With `command: tail -f /dev/null` we are overriding the default command of our image. The reason for this is that the default command in the Python image would exit immediately when we start the container, but we need a command that runs longer so that we have time to copy the files from the running container to the host. The command `tail -f /dev/null` is widely used in this case in the community (<https://github.com/docker/compose/issues/1926>). You'll see this in more detail in a minute.

Now we have the tools to automatically build an image with our generated project code and we can start up a container from it with one single command; `docker-compose up`. What we need now, is some way to copy the generated project code from the container to the host machine.

We'll write a shell script to do this, let's place it under `docker/init/` and call it `init.sh`:

```
#!/bin/sh
scriptdir=$(dirname "$0")
docker-compose -f $scriptdir/docker-compose.yml up -d --build
&& docker cp flask-sql-ci-initiator:/root/app ./app
docker-compose -f $scriptdir/docker-compose.yml down
```

Make sure to make the file executable with:

```
~/Development/example-flask-sql-ci/docker/init$ chmod +x init.sh
```

We want to make this script generic enough to be able to run from the root directory of our project. Therefore on the second line, we acquire the directory name of `init.sh` so that we can specify file paths relative to the location of our `init.sh` file.

We bring up the `cookiecutter` service with `docker-compose up`. We have to explicitly specify the path of the `docker-compose.yml` file because we plan to run the script from the root directory of our project and not in `docker/init/`. The `docker-compose` command is looking for the `docker-compose.yml` file in the current directory.

When you run a Linux or Unix shell script, the current directory is the directory where you are calling the script from. I plan to call the script from the root of my project

`~/Development/example-flask-sql-ci`. Therefore this will be the current directory. This means that the `docker-compose up` command will be looking for `docker-compose.yml` in `~/Development/example-flask-sql-ci`.

Our `docker-compose.yml`, however, is located in the `docker/init/` subdirectory, so `docker-compose` won't find it unless we specify its location with the `-f` option.

We copy the generated code files with `docker cp flask-sql-ci-initiator:/root/app ./app` from the image to the host machine. Here `./app` is relative to the project root directory.

On the last line, we stop the Docker container with `docker-compose down`, again specifying the location of `docker-compose.yml`.

Now you are ready to invoke all this automation from your project directory. We'll use a `Makefile` to do so. The program `make` is a build tool on Unix like systems (like Linux and MacOS). A `Makefile` is a file that describes the build steps for the `make` tool. You can find a gentle introduction to Makefile with Docker here: <https://serversforhackers.com/c/dckr-dev-workflow>.

Let's put our `Makefile` to the root of the project:

```
~/Development/example-flask-sql-ci$ tree .
├── docker
│   └── init
├── cookiecutter.yml
├── docker-compose.yml
└── Dockerfile
└── init.sh
└── Makefile
```

Let's add the first target to initialize our project in the

```
Makefile : .PHONY: init init: docker/init/init.sh
```

All I'm doing here is to define that the `init` target should execute the script `docker/init/init.sh`. We can run this script with `make init`.

`Makefile` target names represent files that are generated by make. For the reason that our `init` target does not generate

a file called `init`, we marked this target as `.PHONY`.

Now we can initialize our project from the root directory of the project like this:

```
~/Development/example-flask-sql-ci$ make  
init docker/init/init.sh Creating network "init_default" with the  
default driver Creating flask-sql-ci-initiator ... done Stopping  
flask-sql-ci-initiator ... done Removing flask-sql-ci-initiator ...  
done Removing network init_default
```

As a result, we generated our project files from the cookiecutter template project on GitHub into our container and copied the results from the container to our host machine's project directory.

You can find the generated project code files under the `app` directory in your project folder. This is the result of the initialization.

You need to do this only once when you start your project. You'll be working on the code files afterwards, so it'll become part of your codebase, and we'll check the files into source control, but before we do that, we'll build the development and production docker tools around the project template, and we'll enhance automation to be a bit more elegant.

Refine initialization

We can make our initialization more generic by defining our project properties in environment variables and give one environment file to our users to specify these variables.

First of all, let's bring our environment variables into one file in the root of our project. Let's create a file called `.env_vars` in the root of the project. (We'll change this name later, but that's another learning step.)

```
#Version of the Dockerization
solution #This version number is used in the custom image tags
APP_VERSION=2.0.0 PYTHON_IMAGE_VERSION=3.7.1-alpine3.8 #URL of the
cookiecutter template repo, do not chage it
TEMPLATE_REPO_URL=https://github.com/sloria/cookiecutter-flask.git
# We use the current working commit on github as of Dec 14, 2018 #
Wanted to use Release 18.0.0, commit
e25a27b33971432e710dda4cf16d4493515b2ed # But that release has
webpack dev server version conflict with webpack # and got npm
error when running. This has been fixed in the sloria repo in the
version we use here
REPO_RELEASE_VERSION=bbb17404c6ac053161a2436b5f195cf44f0002cd
#####
# Initialization
variables # # These variables are used in cookiecutter # code
generation and will added to the # source code
#####
# Customize these
variables before initialization #Docker username used for image
tagging DOCKER_USERNAME=takacsMark #Personal data FULL_NAME=Mark
Takacs EMAIL=mark@takacsMark.com GITHUB_USERNAME=takacsMark
#Project data PROJECT_NAME=Flask MySQL CI Docker
PROJECT_SHORT_DESCRIPTION=Docker example #App name defines the name
of the app directory APP_NAME=myflaskapp
```

The idea is to drive our `docker-compose.yml` files with environment variables to make our project more configurable. We'll use `APP_VERSION` and `DOCKER_USERNAME` in our Docker image names, and we put `TEMPLATE_REPO_URL` and `REPO_RELEASE_VERSION` into the central environment file, too.

I moved the cookiecutter variables into the `.env_vars` file, too and will configure `cookiecutter.yml` using these variables. Let's start with this.

Let's create a file called `cookiecutter.template.yml` under the `docker/init/` directory and add the following lines:

```
default_context: full_name: "${FULL_NAME}" email: "${EMAIL}"
github_username: "${GITHUB_USERNAME}" app_name: "${APP_NAME}"
project_name: "${PROJECT_NAME}" project_short_description:
"${PROJECT_SHORT_DESCRIPTION}"
```

This file is similar to our `cookiecutter.yml`. The difference is that we have replaced the variable values with environment variable values. We'll use a Linux tool called `envsubst` to substitute the values in the file with their environment variable equivalent values. We can now delete `docker/init/cookiecutter.yml` because we'll create that file automatically with `envsubst`.

```
~/Development/example-flask-sql-ci$ rm \
docker/init/cookiecutter.yml
```

We need to make two changes in our `Makefile` in the root directory of our project:

1. Include the `.env_vars` file and export the environment variables for the `Makefile` to be used. (When you run `make`, the command will run everything in a subshell, so what we'll do is to export the environment variables into this subshell.)
2. Use `envsubst` to generate the `docker/init/cookiecutter.yml` file from the `docker/init/cookiecutter.template.yml` template in the `Makefile`.

I prefer to automate as much as possible in one place, so let's move the steps from `docker/init/init.sh` into the `Makefile` updated with our environment variables. The result is the following

```
Makefile : #Import and expose environment variables
variables cnf ?= .env_vars include $(cnf) export $(shell sed
's/=.*//' $(cnf)) .PHONY: init #Generate project codebase form
GitHub using cookiecutter init: envsubst
<docker/init/cookiecutter.template.yml
>docker/init/cookiecutter.yml docker-compose -f docker/init/docker-
compose.yml up -d --build docker cp flask-sql-ci-
initiator:/root/$(APP_NAME) ./$(APP_NAME) docker-compose -f
docker/init/docker-compose.yml down rm
docker/init/cookiecutter.yml
```

The steps are the same that we used in the `init.sh` script we use the `APP_NAME` variable to specify the directory of the

generated code.

Note that I added `--build` to `docker-compose -f docker/init/docker-compose.yml up -d`, this way we'll always rebuild the image with every initialization run.

You can remove `init.sh` now:

```
~/Development/example-flask-sql-ci$ rm docker/init/init.sh
```

Instruct Compose to pass the environment variables to the build:

```
version: '3' services: cookiecutter: build: context: . args: repo: ${TEMPLATE_REPO_URL} repo_release_version: ${REPO_RELEASE_VERSION} python_image_version: ${PYTHON_IMAGE_VERSION} image: ${DOCKER_USERNAME}/cookiecutter:${APP_VERSION}-initiator container_name: flask-sql-ci-initiator command: tail -f /dev/null
```

To test the initialization again, we need to clean up after the generation process. Let's issue the command in terminal:

```
~/Development/example-flask-sql-ci$ rm -rf ./app/
```

Note that I have changed the name of the `app` directory to `myflaskapp` in `.env_vars` file so that we can make sure we are generating the new solution.

Let's also remove the initialization image as part of the cleanup. Let's issue the command:

```
~/Development/example-flask-sql-ci$ docker image \ rm takacsmark/cookiecutter:initiator
```

We'll automate these steps in the `Makefile` so that we can run the cleanup many times if we need to. Let's create another target in the `Makefile` with the name `init-purge` and add code to remove the generated `app` directory from the project. We'll add the `APP_NAME` variable from the environment variables file as the name of the directory so that there is only a single point of definition of the application name.

I have added comments a neat `help` target to display usage information for users who type `make` on the command line.

The `Makefile` looks like this now:

```
#Import and expose environment variables
cnf ?= .env_vars
include $(cnf)
export $(shell sed 's/=.*//' $(cnf))

.PHONY: init

help:
    @echo
    @echo "Usage: make TARGET"
    @echo
    @echo "Dockerization of the sloria/cookiecutter-flask temp"
    @echo
    @echo "Targets:"
    @echo " init           initialize your app code base from GitHub"
    @echo " init-purge     clean up generated code"

#Generate project codebase form GitHub using cookiecutter
init:
    envsubst <docker/init/cookiecutter.template.yml >docker/i
```

```
docker-compose -f docker/init/docker-compose.yml up -d --build
docker cp flask-sql-ci-initiator:/root/${APP_NAME} ./$(API_NAME)
docker-compose -f docker/init/docker-compose.yml down
rm docker/init/cookiecutter.yml

#Remove the generated code, use this before re-running the `init` target
init-purge:
    sudo rm -rf ./$(APP_NAME)
```

The `help` target is the first one on the list on the `Makefile` so it'll become the default target and will be run when users type `make` and hit `Enter`. Please note that targets starting with `.` on the `Makefile` are not considered as default targets and are not listed in bash completion. This way we can create targets that are a bit less visible and use them for subtasks. We'll do this later.

Running `make` will produce the following output:

```
~/Development/example-flask-sql-ci$ make Usage: make TARGET Flask
MySQL CI Docker project automation helper Targets: init generate
source codebase from GitHub repo init-purge clean up generated
code
```

We are ready to re-run our `init` process:

```
~/Development/example-flask-sql-ci$ make init
```

As a result, you should see a new directory in your project directory with the name `myflaskapp`.

The structure of the project is as follows, listed down to the 3rd level:

```
~/Development/example-flask-sql-ci$ tree -L 3 . |  
Makefile |-- docker |-- init |-- Dockerfile |  
cookiecutter.template.yml |-- docker-compose.yml |-- myflaskapp  
|-- LICENSE |-- Procfile |-- README.rst |-- assets |-- css |  
|-- img |-- js |-- autoapp.py |-- myflaskapp |-- __init__.py |  
|-- app.py |-- commands.py |-- compat.py |-- database.py |  
|-- extensions.py |-- public |-- settings.py |-- static |  
|-- templates |-- user |-- utils.py |-- webpack |-- package.json  
|-- requirements |-- dev.txt |-- prod.txt |-- requirements.txt  
|-- setup.cfg |-- tests |-- __init__.py |-- conftest.py |  
|-- factories.py |-- settings.py |-- test_forms.py |  
|-- test_functional.py |-- test_models.py |-- webpack.config.js  
15  
directories, 29 files
```

Set up the development environment

We'll use a Docker image to run our development server, and we'll specify our stack in `docker-compose.yml`. We'll create this configuration under the `docker/dev` directory.

Please create the directory `docker/dev` and create a `Dockerfile` inside.

The challenge we are facing now is to find out how the example project works so that we can start building the development environment. We can learn a lot by reading

the documentation in the generated code under `myflaskapp/README.rst`, the Quickstart part is especially relevant for the development environment: Run the following commands to bootstrap your environment

```
git clone https://github.com/takacsmark/myflaskapp cd myflaskapp pip install -r requirements/dev.txt cp .env.example .env npm install npm start # run the webpack dev server and flask server using concurrently You will see a pretty welcome screen. Once you have installed your DBMS, run the following to create your app's database tables and perform the initial migration :: flask db init flask db migrate flask db upgrade npm start
```

According to the above, we need the following steps to run the application in development mode:

- Clone the `myflaskapp` codebase - we don't need to do this, because we just generated it in our computers in the initialization steps, this is done, let's move on.
- Install Python dependencies from `requirements/dev.txt`; we'll do this in the `dev Dockerfile`.
- Use the environment variables from `.env.example`, we'll set these in the `dev docker-compose.yml` file. If you check this file, you'll see that we are using an SQLite database for development.
- Install `npm` (`node.js`) dependencies (`npm install`); we'll do this in the Dockerfile.

- Run `npm start` to start the servers, and this will be the `CMD` in our Dockerfile.
- We'll also need to run the database initialization steps once the server is started:
 - `flask db init`
 - `flask db migrate`
 - `flask db upgrade`

The documentation says: “# run the Webpack dev server and flask server using concurrently”. This means that we’ll run a Flask server and a Webpack development server concurrently.

The project is built on Flask and written in Python, so this was expected that we’d run a Flask server.

We’ll also run a Webpack (<https://webpack.js.org/>) dev server. Webpack is a Javascript based tool to bundle project resources for web projects. If you never met Webpack, this is what it does: most websites use Javascript code to make the website dynamic. You can use Webpack to pack your many Javascript source files into a neat bundle that you can distribute effectively. You can of course sophisticate this, and Webpack can do a lot more than this, but this is the core of what it does.

In the boilerplate code, the Webpack configuration is at <https://github.com/sloria/cookiecutter->

[`flask/blob/master/%7B%7Bcookiecutter.app_name%7D%7D/webpack.config.js`](#), or at `myflaskapp/webpack.config.js` in the generated code. Reading this code will uncover that we use Webpack to bundle our Javascript and CSS code in the `myflaskapp/assets` directory and the `font-awesome.css` and `bootstrap.css` files from the installed node modules (under `node_modules` that we'll install with `npm install`).

For our purposes, it's enough to know now that we use Webpack to serve the Javascript and CSS of the web application.

Webpack is written in Javascript, and its development dependencies are npm packages. We use npm packages described in `myflaskapp/package.json` for development.

We use `npm` to install the Bootstrap front-end framework(<https://getbootstrap.com/>), Font awesome icons (<https://fontawesome.com/>) and the JQuery Javascript library (<https://jquery.com/>). These packages are described as production dependencies in our `package.json` file in the project. Webpack will pack these components into our project bundle.

If you look at the `package.json` file, you'll see that in the `scripts` section we have scripts to start the project. We'll use the script under `npm start` to start our project in development mode.

There is one more component that we have not discussed.

It's Concurrently in the `npm start` script

(<https://github.com/kimmobrunfeldt/concurrently>).

Concurrently is a Javascript tool to run processes concurrently. We use it to run the Webpack dev server and Flask in parallel for development.

I know there is a lot you have to keep in mind right now. If you are inexperienced with the above tools and frameworks, I'd suggest spending some time experimenting with them.

Let's return to our `Dockerfile` and put it all together. We'd better start from a Python image because the core of our web application is written in Python. We need to add Node.js, which means we'll have to install Node.js into the image.

We'll need to copy the generated project code (under `myflaskapp`) into the image and install the dependencies.

Let's put all this into `docker/dev/Dockerfile`:

```
ARG python_image_version
FROM python:$python_image_version
ARG app_name
RUN apk add --update --no-cache nodejs nodejs-npm postgresql-dev
gcc python3-dev musl-dev libffi-dev
WORKDIR /root/
COPY $app_name $app_name
WORKDIR /root/$app_name
RUN pip install -r requirements/dev.txt
RUN npm install
CMD npx concurrently -n "WEBPACK,FLASK" -c "bgBlue.bold,bgMagenta.bold" "NODE_ENV=debug webpack-dev-server --port 2992 --hot --inline --host 0.0.0.0"
"flask run --host=0.0.0.0"
```

Let me highlight two important lines:

1. RUN apk add --update --no-cache nodejs nodejs-npm postgresql-dev gcc python3-dev musl-dev libffi-dev - we are installing more than nodejs and nodejs-npm here. The other packages are needed for the Psycopg install, which is the most popular Postgres SQL adapter for Python and you can find it in `myflaskapp/requirements/prod.txt`. The installation fails without these packages, I fixed it for your up-front, so that we don't run into this problem
2. CMD npx concurrently -n "WEBPACK,FLASK" -c "bgBlue.bold,bgMagenta.bold" "NODE_ENV=debug webpack-dev-server --port 2992 --hot --inline --host 0.0.0.0" "flask run --host=0.0.0.0" - note that we are not just executing `npm start` as suggested by the repository documentation. We added `--host 0.0.0.0` to both Webpack dev server and the Flask executable. This setting makes the Webpack dev server and Flask servers available on all network interfaces of our development container. It's a common mistake in Docker to run your server on `localhost` only inside the container, and it works perfectly if you `curl` the server from the container, but you cannot access the server from the outside. This is how to fix this issue.

```
Let's set up our docker/dev/docker-compose.yml to build the
image and start a container: version: '3.5' services: web:
build: context: ../../ dockerfile: docker/dev/Dockerfile args:
app_name: ${APP_NAME} python_image_version: ${PYTHON_IMAGE_VERSION}
repo_release_version: ${REPO_RELEASE_VERSION} image:
${DOCKER_USERNAME}/flask-sql-ci-web:${APP_VERSION}-dev environment:
FLASK_APP: autoapp.py FLASK_DEBUG: 1 FLASK_ENV: development
DATABASE_URL: sqlite:///tmp/dev.db SECRET_KEY: not-so-secret
ports: - 5000:5000 - 2992:2992 volumes: - type: bind source:
../../${APP_NAME} target: /root/${APP_NAME} -
/root/${APP_NAME}/node_modules
```

Let's note a couple of things here:

- We are using `version: '3.5'` of the docker-compose file specification. The reason for this is that we are using the long version of the bind mount under `volumes:` and this requires version 3.5.
- We have one service that we call `web`. We are using SQLite, so we don't need a dedicated database server for development.
- **The build part is the key learning here.** The context is `../../`, this means that the context of the build is two directory levels higher from the location of the `docker-compose.yml` file which we are looking at. This means that the root of our project directory is our build context. The build context is the directory from where Docker will

take files for the build. Since we are copying our application directory into the image our context needs to be the project root directory.

- **The location of the Dockerfile in the build is relative to the build context.** So that's why we have specified docker/dev/Dockerfile under dockerfile.
- The APP_NAME, PYTHON_IMAGE_VERSION and REPO_RELEASE_VERSION variables are passed to the build, check the Dockerfile to see how we use them.
- We have given a name to the image to be built, we use the DOCKER_USERNAME and APP_VERSION variables.
- Under environment we are specifying all the variables that are described in myflaskapp/env.example. We use a dummy secret key here, we'll see really secret secret keys in our production config.
- Ports are coming from two sources:
 - package.json says, that Webpack dev server is listening on port 2992.
 - port 5000 is default for Flask and we are not changing it in package.json or in the code settings, so we use 5000.

- In the `volumes:` section we bind mount the source code into the image so that we can check our development changes. The bind mount will override the `/root/myflaskapp` directory entirely in the Docker container and the installed `node_modules` will not be available in the container, which would give an error. Therefore we map the underlying `/root/${APP_NAME}/node_modules` directory from the image into the container to make it work. (Same trick we have used in the Node.js example before.)

Let's add a new target to our `Makefile` that builds the development image:

```
#Build the development image dev-build:  
docker-compose -f docker/dev/docker-compose.yml build
```

The build should succeed, we are good to go, let's start up the servers:

```
~/Development/example-flask-sql-ci$ make dev-up
```

Initialize the database

The documentation gives us clear instructions how to set up the database.

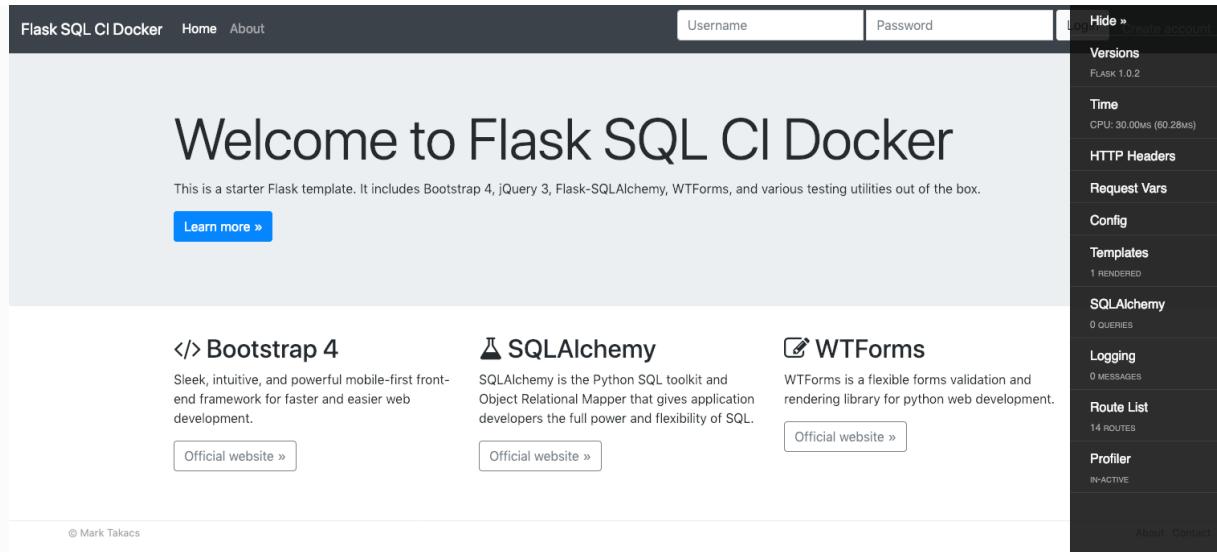
During a development project, we usually use development databases with a sanitized and obfuscated subset of production data or a dataset prepared for development specifically.

We'll not simulate this scenario here. We'll manually prepare the database for development.

The documentation says that we can accomplish this with Flask commands. So let's go into our running container and issue the commands as described in the documentation:

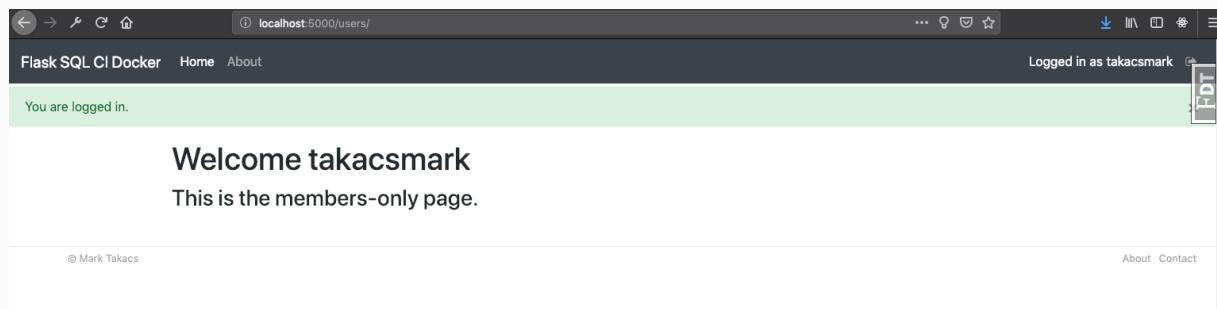
```
~/Development/example-flask-sql-ci$ docker exec \ -ti dev_web_1  
/bin/sh ~/myflaskapp # flask db init && \ flask db migrate && \  
flask db upgrade Creating directory /root/myflaskapp/migrations ...  
done Creating directory /root/myflaskapp/migrations/versions ...  
done Generating /root/myflaskapp/migrations/script.py.mako ... done  
Generating /root/myflaskapp/migrations/env.py ... done Generating  
/root/myflaskapp/migrations/README ... done Generating  
/root/myflaskapp/migrations/alembic.ini ... done Please edit  
configuration/connection/logging settings in  
'/root/myflaskapp/migrations/alembic.ini' before proceeding. INFO  
[alembic.runtime.migration] Context impl SQLiteImpl. INFO  
[alembic.runtime.migration] Will assume non-transactional DDL. INFO  
[alembic.autogenerate.compare] Detected added table 'users' INFO  
[alembic.autogenerate.compare] Detected added table 'roles'  
Generating /root/myflaskapp/migrations/versions/dacc2a4ba927_.py  
... done INFO [alembic.runtime.migration] Context impl SQLiteImpl.  
INFO [alembic.runtime.migration] Will assume non-transactional DDL.  
INFO [alembic.runtime.migration] Running upgrade -> dacc2a4ba927,  
empty message
```

Visiting <http://localhost:5000>, you should see this website and you can create an account. (You may see error messages while the application is starting, be patient and refresh the page a few times.):



Create a user in the application

Now if you create a user on the website, your data will be saved, and you can log in.



Finishing touches

Your SQLite development database files are located under `/tmp/dev.db` in the development Docker container. I decided not to create a volume for this, but if you want to store

these files outside the container and reuse them throughout multiple runs, it's a good exercise to add a volume to the solution.

There is one side effect of the data migration we executed; you'll see a directory in the project directory, called `migrations`.

Let's make sure that we don't check this directory into source control. I added a `.gitignore` file to the root of the project directory: `*/migrations/**`

It would be best to add a few more targets to the `Makefile` that we can use to shut down and monitor the development environment.

```
#Bring down development environment
dev-down:
    docker-compose -f docker/dev/docker-compose.yml down

#List development containers
dev-ps:
    docker-compose -f docker/dev/docker-compose.yml ps

#Show development logs
dev-logs:
    docker-compose -f docker/dev/docker-compose.yml logs -f
```

I've put these lines to the end of the `Makefile`.

Fix docker-compose commands

We added a couple of commands to the `Makefile`, but the chances are that we'll run `docker-compose` commands outside the `Makefile`, too.

If we run a simple command we'll notice some warnings:

```
docker-compose \ -f docker/dev/docker-compose.yml ps
WARNING: The APP_NAME variable is not set. Defaulting to a blank string.
WARNING: The PYTHON_IMAGE_VERSION variable is not set. Defaulting to a blank string.
WARNING: The REPO_RELEASE_VERSION variable is not set. Defaulting to a blank string.
WARNING: The DOCKER_USERNAME variable is not set. Defaulting to a blank string.
WARNING: The APP_VERSION variable is not set. Defaulting to a blank string.
Name  Command State Ports -----
----- dev_web_1
/bin/sh -c npx concurrentl ... Up 0.0.0.0:2992->2992/tcp,
0.0.0.0:5000->5000/tcp
```

The results are there, but the output says that the variables are not set. When we run `docker-compose` commands from the `Makefile`, this is not an issue, because we export all variables from `.env_vars` at the beginning of the `Makefile`.

We would need some way to export the variables on the host machine so that they will be available on the command line.

Docker gives a convenient solution to this issue. If you have a file called `.env` in the directory where you issue the `docker-compose` commands on the command line, `docker-compose` will automatically pick up the values from `.env` and use them when processing the `docker-compose.yml` file.

If we rename `.env_vars` to `.env`, then the variables will be available when we issue `docker-compose` commands in the root directory of our project. So let's rename the file:

```
~/Development/example-flask-sql-ci$ mv .env_vars .env
~/Development/example-flask-sql-ci$ docker-compose \ -f
docker/dev/docker-compose.yml ps Name Command State Ports -----
-----
----- dev_web_1 /bin/sh -c npm start Up 0.0.0.0:2992->2992/tcp,
0.0.0.0:5000->5000/tcp
```

As you see the warning is gone.

We also need to update our `Makefile` to reflect the new filename. The entire `Makefile` looks like this at the end of the development stage:

```
#Import and expose environment variables cnf
?= .env include $(cnf) export $(shell sed 's/=.*//' $(cnf)) .PHONY:
init help: @echo @echo "Usage: make TARGET" @echo @echo
"Dockerization of the sloria/cookiecutter-flask template repo"
@echo @echo "Targets:" @echo " init initialize your app code base
from the sloria/cookiecutter-flask repo" @echo " init-purge clean
up generated code" @echo " dev-build build the Docker image for
development" @echo " dev-up start the app in development mode with
Docker Compose" @echo " dev-down stop the app in development mode
with Docker Compose" @echo " dev-ps list development containers"
```

```
@echo " dev-logs follow development logs" #Generate project  
codebase form GitHub using cookiecutter init: envsubst  
<docker/init/cookiecutter.template.yml  
>docker/init/cookiecutter.yml docker-compose -f docker/init/docker-  
compose.yml up -d --build docker cp flask-sql-ci-  
initiator:/root/${APP_NAME} ./${APP_NAME} docker-compose -f  
docker/init/docker-compose.yml down rm docker/init/cookiecutter.yml  
#Remove the generated code, use this before re-running the `init`  
target init-purge: sudo rm -rf ./${APP_NAME} #Build the development  
image dev-build: docker-compose -f docker/dev/docker-compose.yml  
build #Start up development environment dev-up: docker-compose -f  
docker/dev/docker-compose.yml up -d #Bring down development  
environment dev-down: docker-compose -f docker/dev/docker-  
compose.yml down #List development conatiners dev-ps: docker-  
compose -f docker/dev/docker-compose.yml ps #Show development logs  
dev-logs: docker-compose -f docker/dev/docker-compose.yml logs -f
```

Now that development is up and running you can take some time and enjoy the results. Once you're done, stop the development stack with: `~/Development/example-flask-sql-ci$ make dev-down`

Run tests with Travis CI

The boilerplate code already has automated test cases included. According to the documentation (https://github.com/sloria/cookiecutter-flask/blob/master/%7B%7Bcookiecutter.app_name%7D%7D/

[README.rst](#)), we can use the command `flask test` to run the tests.

In our exercise, we'll use a continuous integration testing solution called Travis CI (<https://travis-ci.org/>). The good thing about Travis CI is that it's free for open source projects, it runs in the cloud, no complex setup required.

We'll do the following steps to set up our automated testing:

- Add our project to GitHub as an open source project
 - Sign up to Travis CI and integrate it with your GitHub account
 - Create the Travis CI configuration
 - Run the tests
 - Bonus: Display test results in your GitHub repo

Add our project to GitHub as an open source project

Create a repository on GitHub

As a first step, let's create a GitHub repository on <https://github.com/>. Please go to the navigation bar and

click the “+” icon next to your account picture in the upper right corner and click “New repository”. Fill in the data as follows:

- give your repo a name; mine will be called `flask-sql-ci-docker`
- keep the “public” radio button selected
- do not change anything else (Don’t add the README, .gitignore and License, we’ll do that later.)

Click “Create repository”.

You’ll get clear instructions to set up your project and push it to GitHub; we’ll continue with that.

Prepare the local project

The remote side on GitHub is ready. The next step is to configure our project on your machine to use that repository for source control. (You need git on your machine, please install it, if you don’t have it, as described here: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.) Please open up your Terminal and in your project’s root directory issue the commands that are listed on your GitHub repo. Note that our commands are a little different because we have to add all files in the project to the repo, so we have a command `git add .` also incorporated. My flow is this:

```
~/Development/example-flask-sql-ci$ echo \ "# flask-sql-ci-docker"
```

```
>> README.md ~/Development/example-flask-sql-ci$ git init  
~/Development/example-flask-sql-ci$ git add .  
~/Development/example-flask-sql-ci$ git commit -m "first commit"  
~/Development/example-flask-sql-ci$ git remote add origin <copy  
your repo url here> ~/Development/example-flask-sql-ci$ git push -u  
origin master
```

Congratulations, now you have your repository ready on GitHub!

Sign up to Travis CI and integrate with your GitHub account

Please visit <https://travis-ci.org/> and sign up with your GitHub account.

Once your account is ready, click your account image in the upper right corner and you'll be presented with the list of your GitHub repositories. All you have to do is flip the switch next to your `flask-sql-ci-docker` project.

That's it. We are ready to move on.

Create the Travis CI configuration

We have come to the part where we set up Travis CI to test our repository automatically when we push changes to

GitHub. We'll do the configuration in a file called `.travis.yml` as per the Travis convention.

We'll complete two main steps:

1. Prepare the Travis configuration in `.travis.yml` to build a Docker container for testing.
2. Create a new target in our `Makefile` to run the tests, we'll use this target in `.travis.yml`

Prepare `.travis.yml`

If you are not familiar with Travis CI concepts, I'd suggest reading the getting started guide (<https://docs.travis-ci.com/user/getting-started/>) and the using Docker guide <https://docs.travis-ci.com/user/docker/>.

When Travis CI builds our environment, it'll bring up a virtual machine in the cloud and install our project every time you trigger a test (i.e. push changes or merge pull requests into the repo). The default environment is a “Container Based Ubuntu 14.04” as described in the Travis documentation (<https://docs.travis-ci.com/user/getting-started/>).

Let's create the file `.travis.yml` in the root of our project. As the guide suggests we prepare our environment and install Docker and docker-compose in the `before_install` section. Remember, we are in an Ubuntu VM on Travis.

```
sudo: required

language: python

services:
  - docker

env:
  - DOCKER_COMPOSE_VERSION=1.19.0-rc3

before_install:
  - sudo apt-get update
  - sudo apt-get -y -o Dpkg::Options::="--force-confnew" install docker-compose
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/1.19.0/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

script:
  - make test-run
```

Create the test target in the Makefile

We configured our tests to run `make test-run` as our script, so let's prepare this target in our `Makefile`. We have to take care of the following steps:

- Start up our project
- Run the test against the running application
- Clean up after the container

I decided to reuse the Docker image and docker-compose definition that we created for development.

When Travis builds your project, it has access to files in your GitHub repository, so you can use the `Makefile`, `docker-compose.yml` and other tools, just like you can use them on your machine.

Our `test-run` code is the following:

```
#Run tests
test-run:
    docker-compose -f docker/dev/docker-compose.yml up -d
    sleep 10
    docker-compose -f docker/dev/docker-compose.yml exec web
    docker-compose -f docker/dev/docker-compose.yml down
```

On the first line, we start up our project the same way we start it in development.

We need to wait until the container comes up and is ready to receive requests. Waiting is not a sophisticated way to do this, it would be better to set up a script that would periodically check the availability of the server, but for now, let's use `sleep 10` to wait 10 seconds.

Then we `exec` the command `flask test` against the running container in the `web` service.

On the last line we clean up after the test, and we are done.

You can test how this works on your local machine:

```
$ make test-run
docker-compose -f docker/dev/docker-compose.yml up -d
Creating network "dev_default" with the default driver
Creating dev_web_1 ... done
sleep 10
docker-compose -f docker/dev/docker-compose.yml exec web flask test
=====
platform linux -- Python 3.7.1, pytest-3.7.4, py-1.7.0, pluggy-0.8.0
cachedir: .pytest_cache
rootdir: /root/myflaskapp, inifile:
collected 21 items

tests/test_forms.py::TestRegisterForm::test_validate_user_already_
tests/test_forms.py::TestRegisterForm::test_validate_email_already_
tests/test_forms.py::TestRegisterForm::test_validate_success PASSED
tests/test_forms.py::TestLoginForm::test_validate_success PASSED
tests/test_forms.py::TestLoginForm::test_validate_unknown_username_
tests/test_forms.py::TestLoginForm::test_validate_invalid_password_
tests/test_forms.py::TestLoginForm::test_validate_inactive_user PA_
tests/test_functional.py::TestLoggingIn::test_can_log_in_returns_20_
tests/test_functional.py::TestLoggingIn::test_sees_alert_on_log_out_
tests/test_functional.py::TestLoggingIn::test_sees_error_message_
tests/test_functional.py::TestLoggingIn::test_sees_error_message_
tests/test_functional.py::TestRegistering::test_can_register PASSE_
tests/test_functional.py::TestRegistering::test_sees_error_message_
tests/test_functional.py::TestRegistering::test_sees_error_message_
tests/test_models.py::TestUser::test_get_by_id PASSED
tests/test_models.py::TestUser::test_created_at_defaults_to_dated_
tests/test_models.py::TestUser::test_password_is_nullable PASSED
tests/test_models.py::TestUser::test_factory PASSED
tests/test_models.py::TestUser::test_check_password PASSED
tests/test_models.py::TestUser::test_full_name PASSED
tests/test_models.py::TestUser::test_roles PASSED

=====
21 passed in 1.79 seconds ===
docker-compose -f docker/dev/docker-compose.yml down
```

```
Stopping dev_web_1 ... done
Removing dev_web_1 ... done
Removing network dev_default
```

All automated tests passed, we are on the right track.

Run the tests with Travis CI

The moment of truth is close. Let's add our `.travis.yml` to our repo and push the changes to GitHub:

```
~/Development/example-flask-sql-ci$ git add .travis.yml
~/Development/example-flask-sql-ci$ git commit -am"travis added"
~/Development/example-flask-sql-ci$ git push origin master
```

Going to the Travis CI website, you'll see the build starting up automatically, and you can follow the progress. Once the build is done you'll see a similar output:

The screenshot shows a Travis CI build page for the repository 'takacsmark / flask-sql-ci-docker'. The build status is 'unknown'. The master branch has a green checkmark and the commit 'test fix' is shown. A test named '#3 passed' ran for 2 minutes and 50 seconds, completed 11 minutes ago. There are links to 'Job log' and 'View config'. The job log shows the command-line output of the build process, including environment variable settings, git cloning, service starting, and system updates.

```
1 Worker information
2 Network availability confirmed.
3
4 127.0.0.1 localhost nettuno travis vagrant
5 127.0.1.1 travis-job-858409ab-76bf-4f95-86c0-7aab71e39c50 travis-job-858409ab-76bf-4f95-86c0-7aab71e39c50 ip4-loopback trusty64
6
7 W: http://ppa.launchpad.net/couchdb/stable/ubuntu/dists/trusty/Release.gpg: Signature by key 15866BAFD9BCC4F3C1E0DFC7D69548E1C17EAB57 uses weak digest
8 algorithm (SHA1)
9
10
11 $ git clone --depth=50 --branch=master https://github.com/takacsmark/flask-sql-ci-docker.git takacsmark/flask-sql-ci-docker
12 git.checkout 0.08s
13 services 0.02s
14
15 $ sudo service docker start
16
17 Setting environment variables from .travis.yml
18 $ export DOCKER_COMPOSE_VERSION=1.19.0-rc3
19
20 $ source ~/virtualenv/python2.7/bin/activate
21
22 $ python --version
23 Python 2.7.14
24 $ pip --version
25 pip 9.0.1 from /home/travis/virtualenv/python2.7.14/lib/python2.7/site-packages (python 2.7)
26 before_install.1 3.17s
27
28 $ sudo apt-get update
```

From now on every time you push new changes to GitHub, the automated tests will run.

Bonus: Display test results in your GitHub repo

Displaying test status information on your GitHub repository is a great way to give transparency to your co-workers and users about the overall health of your application.

You can easily add the status icon to your GitHub repo from Travis CI. Click the status image next to the name of your repo as shown in the picture:

The screenshot shows a GitHub repository page for 'takacsmark / flask-sql-ci-docker'. At the top, there's a navigation bar with links for 'Log in', 'Status', and 'Help'. On the right, there's a user profile icon and a 'Mark' button. Below the navigation, the repository name 'takacsmark / flask-sql-ci-docker' is displayed, followed by a small circular status icon with a question mark and the text 'build unknown'. A large green arrow points from the text 'Status image' to this icon. Below the repository name, there are tabs for 'Current', 'Branches', 'Build History', and 'Pull Requests', with 'Current' being selected. Under the 'Current' tab, there's a list of branches: 'master' (with a green checkmark) and 'test fix'. The 'master' branch has a build status of '#3 passed' with a green checkmark, indicating it ran for 2 min 50 sec 11 minutes ago. There are also links to 'Compare' and 'Branch master'. Below this, a message says 'takacsmark authored and committed'. At the bottom of the page, there's a 'Job log' section with a 'View config' link and a 'Raw log' button. The job log shows a series of terminal commands run by the Travis CI worker, including network configuration, cloning the repository, starting Docker, setting environment variables, and updating packages.

In the popup window choose “markdown” in the second drop-down list and copy the code. Let’s open up our `README.md` file and paste it under the repo title. My file looks like this; I also added some more info for users:

```
# flask-sql-ci-docker
A containerized version of the
https://github.com/sloria/cookiecutter-flask template project. This
repo contains the configuration to initialize, develop, test and
```

```
run the Flask project based on the cookiecutter-flask template. [!
[Build Status](https://travis-ci.org/takacsmark/flask-sql-ci-docker.svg?branch=master)](https://travis-ci.org/takacsmark/flask-sql-ci-docker) **This repo is currently work in progress.**
Documentation will follow.
```

We'll commit this change to git in a way that it will not trigger a new Travis build. Changing file like `README.md` does not require a new build. We can use `[ci skip]` in the commit command to achieve this:

```
~/Development/example-flask-sql-ci$ git commit \ -am "readme update [ci skip]"
```

Please use `git push origin master` to push your changes to GitHub.

Visiting your repo on GitHub should give you a similar result with the Travis status displayed in your repo.

The screenshot shows a GitHub repository page for 'flask-sql-ci-docker' by takacsmark. The repository has 9 commits, 1 branch, 0 releases, and 1 contributor. The latest commit was 27 seconds ago. The README.md file contains the following content:

```
flask-sql-ci-docker
A containerized version of the https://github.com/sloria/cookiecutter-flask template project.

This repo contains the configuration to initialize, develop, test and run a flask project based on the cookiecutter-flask template.

build passing

This repo is currently work in progress. Documentation will follow.
```

Set up the production environment

In this chapter we'll set up the production environment for our project. First, we'll create the Docker image and docker-compose files. We'll run production in Swarm mode, so we'll prepare our environment accordingly.

We'll run production locally, plus we'll deploy our application to AWS Cloud servers. We'll use Docker's Cloud tools to

create a Swarm cluster on AWS and will deploy our components with Docker tools.

Prepare the Docker image and the Compose file

Our first job is to find out how to start the boilerplate project in production mode. The documentation at

https://github.com/sloria/cookiecutter-flask/tree/master/%7B%7Bcookiecutter.app_name%7D%7D

has clear instructions. To deploy:

```
export FLASK_ENV=production  
export FLASK_DEBUG=0 export DATABASE_URL=<YOUR DATABASE URL> npm  
run build # build assets with webpack flask run # start the flask  
server
```

What this means is, that we'll not run a Webpack server in production. We'll use Webpack to build our assets with

```
npm  
run build
```

 and then serve them with the Flask server.

We'll have to copy our code into the image, install npm dependencies with

```
npm install
```

 and then run

```
npm run build
```

, which will finally generate our assets into the

```
myflaskapp/static/build
```

 directory. After the Webpack build is done, we do not need Node and npm in the image anymore.

To accomplish the above tasks we'd better create a multi-stage build in our

```
Dockerfile
```

. First, we build the static assets with Webpack and copy the results into the production

image. This way we won't have the libraries and dependencies needed by Webpack in our final image. Which makes our image smaller, thus giving us faster deployment and startup times.

We'll use a PostgreSQL server (<https://www.postgresql.org/about/>) in production, so we'll have to add Postgres to our stack for production in our `docker-compose.yml`.

Let's create the directory `docker/prod` for our production files. Let's start with the Dockerfile:

```
ARG python_image_version
FROM python:$python_image_version as builder
RUN apk add --update --no-cache nodejs nodejs-npm
ARG app_name
WORKDIR /root/
COPY $app_name $app_name
WORKDIR /root/$app_name
RUN npm install && \
  npm run build
ARG python_image_version
FROM python:$python_image_version
RUN apk add --update --no-cache
postgresql-dev gcc python3-dev musl-dev libffi-dev
ARG app_name
WORKDIR /root/
COPY --from=builder /root/$app_name $app_name
WORKDIR /root/$app_name
RUN rm -rf migrations dev.db node_modules
&& \
  pip install -r requirements/prod.txt
COPY docker/prod/*.sh ./
RUN chmod +x ./*.sh
ENTRYPOINT [ "./entrypoint.sh" ]
```

I hope you understand most of the Dockerfile by now. Let me highlight some critical decisions that I made to build this example:

- In the builder stage, we focus on running `npm run build`, so this stage is about `npm` dependencies. `npm run build` is a script that is defined in our

`package.json` file. The code in `package.json` is the following: `NODE_ENV=production webpack --progress --colors -p`. This line runs webpack, in this mode, Webpack will generate the output bundle from our assets as described in `myflaskapp/webpack.config.js` in the entry and output sections.

The final results will be saved to `myflaskapp/static/build` as stated in `webpack.config.js`. This means that we can copy the code from the builder image over to the final image and we're good to go.

- In the second stage, we install the Linux packages needed by the project. We copy the entire codebase from the builder stage (this includes the generated static bundle).
- I have added an `entrypoint` script where we'll manage the secret keys of the project, we'll see how this is done in detail later.

Let's add the Compose file.

```
version: '3.5'

services:
  web:
    build:
      context: ../../
      dockerfile: docker/prod/Dockerfile
```

```
        args:
            app_name: ${APP_NAME}
            python_image_version: ${PYTHON_IMAGE_VERSION}
        image: ${DOCKER_USERNAME}/flask-sql-ci-web:${APP_NAME}
        ports:
            - 5000:5000
    environment:
        POSTGRES_PASSWORD: /run/secrets/db_password
        FLASK_APP: autoapp.py
        FLASK_DEBUG: 0
        FLASK_ENV: production
        APP_SECRET: /run/secrets/app_secret
    networks:
        - flaskappnet
    secrets:
        - app_secret
        - db_password
    deploy:
        replicas: 2
postgres:
    image: postgres:${POSTGRES_IMAGE_VERSION}
    environment:
        POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    volumes:
        - type: volume
          source: proddb
          target: /var/lib/postgresql/data
    networks:
        - flaskappnet
    secrets:
        - db_password
    deploy:
        placement:
            constraints: [node.role == manager]
visualizer:
    image: dockersamples/visualizer:stable
    ports:
        - "8080:8080"
    volumes:
```

```
        - "/var/run/docker.sock:/var/run/docker.sock"
deploy:
    placement:
        constraints: [node.role == manager]
secrets:
    db_password:
        file: ../../_db_password.txt
    app_secret:
        file: ../../_app_secret.txt
volumes:
    proddb:
networks:
    flaskappnet:
```

Several things are happening here:

- I have changed the exposed ports of the `web` service compared to the development version, I eliminated the port of Webpack dev server, 2992, because we don't have it in production.
- I added the environment variables from the documentation to the `environment` section of the `web` service. `FLASK_APP`, `FLASK_DEBUG` and `FLASK_ENV` are defined explicitly, and `POSTGRES_PASSWORD` and `APP_SECRET` are defined as Docker secrets, I'll explain secrets in more details in a second.
- I added a dedicated user defined network to the stack and named it `flaskappnet`. You can see the definition of the network at the bottom of the file, and not how I added them under both `web` and Postgres services under `networks`.

- I decided to deploy 2 replicas of the web services just to show you replication.
- I added the Postgres service that will serve as our database. We'll use an official Postgres image, I specified the Postgres image version in the `.env` file (`POSTGRES_IMAGE_VERSION=11.1-alpine`).
- I attached the `db_password` secret to the Postgres service, the official Postgres image can handle secrets out of the box, I just have to specify the environment variable `POSTGRES_PASSWORD_FILE` to set the database password from a secret.
- I put the database on the manager node (we'll run a cluster with one manager node), so that the data volume will be always available, since we are always starting the database on the same node.
- I added a visualizer service, so that we can visually follow what's going on.

About Docker secrets

Docker secrets

(<https://docs.docker.com/engine/swarm/secrets/>) let you pass sensitive information to your application in a secret way. Docker secrets are only available in Swarm mode.

The key advantage of Docker secrets is that you can manage your secret information with a central tool. Your secret values are transferred and passed to containers in an encrypted way.

The good thing about secrets is that the values are not stored in the containers' file system. They are stored in memory. They are not environment variables, so they are not visible when you do `docker inspect` on your containers.

The secrets are also not entered into the `Dockerfile` or `docker-compose.yml`. You can create a secret with a command line command or from a file.

The fact that we are using secrets is defined in the `docker-compose.yml`, as you've seen above. As a first step I defined `secrets:` on the top level of the `docker-compose.yml` hierarchy right above `volumes:` and `networks:`: `secrets: db_password: file: ../../_db_password.txt app_secret: file: ../../_app_secret.txt`

This part tells Docker that we'll have a secret called `db_password` and it'll get its value from the `_db_password.txt` file that is in the root of our project. We'll create this file and add our password to the file. The other secret is the `app_secret` that we define in the `_app_secret.txt` file.

Let's add these two files to the root of our project. This is my `_db_password.txt` file: `secretpassword1234`

And this is my `_app_secret.txt`:

```
mysupersecretappsecret1223
```

Please make sure that you do not check these files into source control, add these lines to your `.gitignore` file:

```
_db_password.txt _app_secret.txt
```

Then we tell Docker to grant access to the `db_password` secret to the containers that are started for both services (`web` and `postgres`). The `web` service has access to both secrets as shown below, while the Postgres service only needs the database password: `secrets: - app_secret - db_password`

Then we create an environment variable in both services to access the secret. Please note that it's not the secret's value that's in the environment variable. The value of the environment variable is a file path: `/run/secrets/db_password` and `/run/secrets/app_secret`. This means that Docker will mount the secret into the memory of the container.

We'll have to read the secret from memory as if we would read a file.

The Postgres image takes care of secrets automatically. If you append `_FILE` to the environment variable names, the container will automatically retrieve secrets from such variables. Thus we use the environment variable `POSTGRES_PASSWORD_FILE` to pass the secret to Postgres. You can

read more about this here:

https://hub.docker.com/_/postgres/.

While Postgres knows how to handle secrets, our Python app has no tools to retrieve secret values. We have to do the coding manually.

Accessing the secrets in the web service

Both the application `SECRET_KEY` and the `DATABASE_URL` are implemented as environment variables in the Flask project. So we'll have to turn the contents of the files

`/run/secrets/db_password` and `/run/secrets/app_secret` into environment variables.

In order to achieve this, I have added two shell script files to our production solution under the `docker/prod` directory. One file will export the secrets from memory as environment variables, let's call it `exportsecret.sh`:

```
#!/bin/sh
export
SECRET_KEY=$(cat $APP_SECRET)
export
DATABASE_URL=postgresql://postgres:$(cat
$POSTGRES_PASSWORD)@postgres/postgres
```

Please note how I'm constructing a formal database URL to access the Postgres database. The Postgres host is called `postgres`, because `postgres` is the name of the Postgres service in our Compose file. I'm using `postgres` as the

database user name and the database name, too, but you can add more environment variables to customize these values.

Let's source this file in our `entrypoint.sh` and start up the Flask server: `#!/bin/sh . ./exportsecret.sh flask run --host=0.0.0.0`

This will give us the secrets as environment variables.

Build the production stack

First, let's build the production image. We can do it with this command: `~/Development/example-flask-sql-ci$ docker-compose \ -f docker/prod/docker-compose.yml build`

You can also add this command to your `Makefile` like this:

```
#Build production prod-build: docker-compose -f  
docker/prod/docker-compose.yml build
```

Now you can use `make prod-build` to build the image. The build should succeed, if you experience problems, please investigate going through the steps again.

Run the production stack

We'll use `docker stack deploy` to deploy our application to the Swarm. Unlike `docker-compose`, `docker stack deploy` will not pick up our environment variables from the `.env` file, so we'll have to deploy using our `Makefile`. Let's add the target to deploy to our `Makefile`. Let's also add another target to remove the deployment:

```
#Deploy production stack prod-deploy:  
docker stack deploy myflaskapp -c docker/prod/docker-compose.yml  
#Remove production stack prod-rm: docker stack rm myflaskapp
```

It's time to run our stack. First, we initialize the Swarm:

```
~/Development/example-flask-sql-ci$ docker swarm init
```

Deploy the stack to the swarm:

```
~/Development/example-flask-sql-ci$ make prod-deploy  
  
docker stack deploy myflaskapp -c docker/prod/docker-compose.yml  
Ignoring unsupported options: build  
  
Creating network myflaskapp_flaskappnet  
Creating secret myflaskapp_db_password  
Creating service myflaskapp_web  
Creating service myflaskapp_postgres
```

Set up the database

We have to initialize the database the same way we initialized the development db. Let's find out the name of the container that runs our app:

```
~/Development/example-flask-sql-ci$ docker container ls CONTAINER ID IMAGE COMMAND CREATED  
STATUS PORTS NAMES 3982fe309de8 takacsmark/flask-sql-ci-web:2.0.0-
```

```
prod "./entrypoint.sh" 45 seconds ago Up 43 seconds  
myflaskapp_web.1.mepfdb6asiiv92owlwixp3bei e89dee9266eb  
takacsmark/flask-sql-ci-web:2.0.0-prod "./entrypoint.sh" 45  
seconds ago Up 43 seconds myflaskapp_web.2.hdnllx5kr3v1z09zmq2juhwz9  
86e01e66d44a postgres:11.1-alpine "docker-entrypoint.s..." About a  
minute ago Up About a minute 5432/tcp  
myflaskapp_postgres.1.pmp17uilfpq9yordwz5soebiv c4afa1401b13  
dockersamples/visualizer:stable "npm start" 2 minutes ago Up 2  
minutes 8080/tcp myflaskapp_visualizer.1.bvraw29uomzctywwymibq2j6f
```

Mine is called `myflaskapp_web.1.mepfdb6asiiv92owlwixp3bei`, I'll run a `docker exec` command against this container and initialize the database like this: `~/Development/example-flask-sql-ci$ docker exec \ -ti myflaskapp_web.1.mepfdb6asiiv92owlwixp3bei /bin/sh`

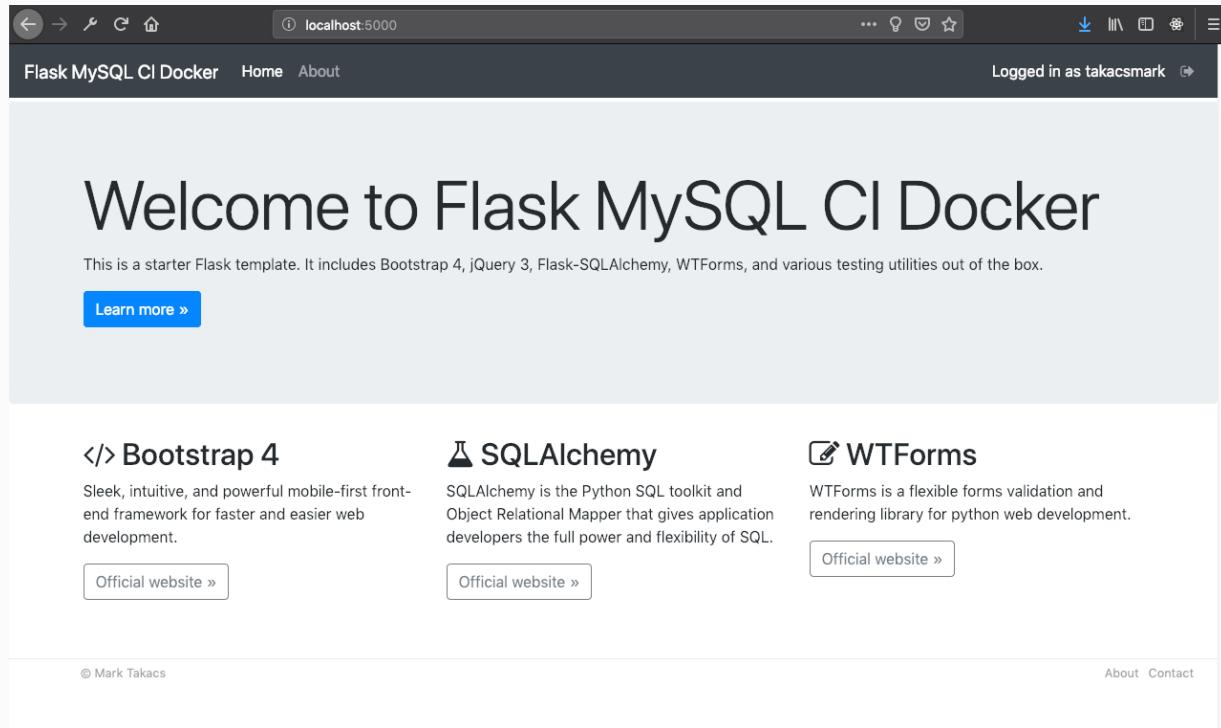
Then we'll have to source the environment variables manually by running the following command: `~/myflaskapp # . ./exportsecret.sh`

Now we can run the database migration commands:

```
~/myflaskapp # flask db init && flask db migrate && flask db upgrade
```

Visiting <http://localhost:5000/>, you'll see the familiar website with the difference, that the development toolbar has disappeared from the right side of the site.

Now you can create a user and log in to the application. My site looks like this:



Deploy to Amazon

In this exercise, we'll deploy our Flask application to Amazon Web Services (AWS). AWS has a one year free trial period, called AWS Free Tier that includes a good amount of cloud services for free. You can sign up for the free tier here: <https://aws.amazon.com/free/>, and read more about what's included here: <https://aws.amazon.com/free/faqs/>.

Please note that you may have to pay a small fee when you run the examples.

Set up an AWS cluster

Once you've signed up for the free tier, we'll start building. The easiest way to start up a Docker Swarm cluster on AWS is to use an Amazon CloudFormation template.

Docker provides a link to the latest stable Docker release CloudFormation template on this webpage:

<https://docs.docker.com/docker-for-aws/#docker-community-edition-ce-for-aws>.

There is one pre-requisite you'll need to complete. Please create an Amazon EC2 Key Pair as described here:

<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html#having-ec2-create-your-key-pair>.

Make sure to save the `pem` file to your computer and change its permissions to 400 with `chmod 400 key_name.pem`.

Let's start the stack set-up by launching the wizard here:<https://docs.docker.com/docker-for-aws/#docker-community-edition-ce-for-aws>.

Docker Community Edition (CE) for AWS

Quickstart

If your account has the proper permissions, you can use the blue button from the stable or edge channel to bootstrap Docker for AWS using CloudFormation. For more about stable and edge channels, see the [FAQs](#).

Stable channel	Edge channel
This deployment is fully baked and tested, and comes with the latest CE version of Docker.	This deployment offers cutting edge features of the CE version of Docker and comes with experimental features turned on, described in the Docker Experimental Features README on GitHub. (Adjust the branch or tag in the URL to match your version.)
This is the best channel to use if you want a reliable platform to work with.	This is the best channel to use if you want to experiment with features under development, and can weather some instability and bugs. Edge is for users wanting a drop of the latest and greatest features every month.
Stable is released quarterly and is for users that want an easier-to-maintain release pace.	We collect usage data on edges across the board.
Deploy Docker Community Edition (CE) for AWS (stable)	Deploy Docker Community Edition (CE) for AWS (edge)
Deploy Docker Community Edition (CE) for AWS (stable) uses your existing VPC	Deploy Docker Community Edition (CE) for AWS (edge) uses your existing VPC

Make sure the the S3 template is selected, and the template URL is <https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tpl>. Click next.

Create stack

Select Template

Select the template that describes the stack that you want to create. A stack is a group of related resources that you manage as a single unit.

Design a template Use AWS CloudFormation Designer to create or modify an existing template. [Learn more.](#)

Design template

Choose a template A template is a JSON/YAML-formatted text file that describes your stack's resources and their properties. [Learn more.](#)

Select a sample template

Upload a template to Amazon S3
 Browse... No file selected.

Specify an Amazon S3 template URL

<https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tpl> [View/Edit template in Designer](#)

[Cancel](#) [Next](#)

Adjust the number of master and worker nodes; we can use a smaller cluster than the default. Also, specify the SSH key to be used. Click through the rest of the wizard.

The screenshot shows the 'Specify Details' step of the CloudFormation wizard. The 'Stack name' is set to 'Docker'. In the 'Parameters' section, three green arrows point to specific fields: 'Number of Swarm managers?' (set to 1), 'Number of Swarm worker nodes?' (set to 3), and 'Which SSH key to use?' (set to 'docker-cloud').

Once your cluster is up and running, navigate to the outputs tab on the stack page and find the link to master node management (you may have to refresh the page to make the tabs visible):

The screenshot shows the AWS CloudFormation console interface. At the top, there's a navigation bar with the AWS logo, 'Services', 'Resource Groups', and user information ('takacsmark', 'Ohio', 'Support'). Below the navigation is a search bar with 'CloudFormation' and 'Stacks' dropdowns, and a 'Create Stack' button.

The main area displays a table of stacks. One stack is listed:

Stack Name	Created Time	Status	Description
Docker	2018-03-26 15:42:34 UTC+0200	CREATE_COMPLETE	Docker CE for AWS 18.03.0-ce (18.03.0-ce-aws1)

Below the stack list is a green box containing the text "Link to manager nodes".

The "Outputs" tab is selected, showing the following table:

Key	Value	Description	Export Name
DefaultDNSTarget	Docker-ExternalLoa-821VNWPUUQLE-1661028294.us-east-2.elb.amazonaws.com	Use this name to update your DNS records.	
ZoneAvailabilityComment	This region has at least 3 Availability Zones (AZ). This is ideal to ensure a fully functional Swarm in case you...	Availability Zones Comment	

A green arrow points from the text "Link to manager nodes" to the "Managers" row in the table, which contains a link: <https://us-east-2.console.aws.amazon.com/ec2/v2/home?region=us-east-2#instances>tag:aws:autoscaling:groupName=Docker-ManagerAsg-1BZASYG5KZFZ1;sort=desc:dnsName>. This row is also highlighted with a green rounded rectangle.

At the bottom of the table, there are columns for "NodeSecurityGroupID" (sg-657f370e), "SecurityGroup ID of NodeVpcSG", and a "Details" button.

Click the link and find the public address of your manager node as shown below.

The screenshot shows the AWS EC2 Instances page. On the left, there's a sidebar with various navigation links like EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, AMIs, Bundle Tasks, Volumes, Snapshots, Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces, Load Balancers, Target Groups, and Auto Scaling. The main area shows a table of instances. One instance is selected, and its details are shown in a modal window. The modal has tabs for Description, Status Checks, Monitoring, and Tags. The Description tab shows the instance ID (i-02f41d56b82b7efd4), instance state (running), instance type (t2.micro), availability zone (us-east-2a), security groups (Docker-ManagerVpcSG-1VRARWDDOLSCP, Docker-SwarmWideSG-12EP4P89D9GST), and scheduled events. It also lists network interfaces, IAM role (Docker-ProxyRole-49E0T2AMQA6Q), key pair name (docker-cloud), EBS-optimized (False), and launch time (March 26, 2018 at 3:45:11 PM UTC+2). A green arrow points from the text 'Public IP address' to the 'IPv4 Public IP' field in the modal, which is highlighted with a green oval and contains the value '18.218.82.101'.

We are ready to `ssh` into the master node. Let's issue the following command:

```
ssh -i ~/.ssh/docker-cloud.pem
```

Please use your master node's IP address, and also update your ssh private key location and name if you used a different name.

My command line with the successful login looks like this:

```
~/Development/example-flask-sql-ci$ ssh \ -i ~/.ssh/docker-
cloud.pem docker@18.218.82.101 Welcome to Docker! ~ $
```

Let's see a little more info about this environment:

```
~ $ whoami
docker

~ $ docker --version

Docker version 18.03.0-ce, build 0520e24

' ' ~ $ docker node ls

ID          HOSTNAME
alxwul8yly3epa2gvuzqz153c * ip-172-31-1-87.us-east-2.compute.int
jb3b0x1c9ma9jhii0o18a56oz   ip-172-31-15-4.us-east-2.compute.int
```

Let's leave the master node now and prepare for deployment.

```
~ $ exit

Connection to 18.218.82.101 closed.
```

Deploy to the AWS Swarm

We should be completing the following steps:

- Configure the production stack to run in a cluster.
- Build and push the production image to Docker Hub.

- Deploy the stack to AWS

Configure the production stack

Our `docker-compose.yml` file is already ready to support production operation right now, let me explain the important points.

Our Postgres Instance stores data in the host file system of the node where it is running. Therefore we need to make sure that it always starts up on the same node where the data is saved. We achieve this by running our database on the master node.

Please note that the topic of managing container data in the cloud is a broad topic. Storing our database files in the host machine's file system gives us the safety that the data will be available if the Postgres container gets removed and a new container is created. If the node goes down, however, our data will be lost.

We could put our database files on a networked file system. You can achieve such a set up with Docker volume plugins. We'll not go into details of this topic in this book. Please note that this topic is something you'd want to explore in the future if you're building a large scale, high availability application.

We added `deploy` instructions to our `docker-compose.yml`. We are instructing Docker to run 2 replicas of our web service

so that we can check out scaling, we instruct Docker to run our Postgres database on the master node, and we're adding a visualizer service so that we can visually follow what's going on in the cluster.

Please double check these settings in the production `docker-compose.yml`.

Build and push the production image to Docker Hub

We already have built the production image. Let's push the image to the Docker Hub (Use the `docker login` command if you see permission issues.):

```
~/Development/example-flask-sql-ci$ docker push \ takacsmark/flask-sql-ci-web:prod
```

Deploy the stack to AWS

SSH into your AWS master tunneling the Docker socket to the remote host:

```
~/Development/example-flask-sql-ci$ ssh \ -i ~/.ssh/docker-cloud.pem \ -NL localhost:2374:/var/run/docker.sock \ docker@18.218.82.101 &
```

This way you can send commands to the Docker daemon in the cloud, while you can use the files on your local machine at the same time. This means that you can use `docker/prod/docker-compose.yml` on your local machine to deploy your application to the AWS cluster.

Let's add a new target to the `Makefile` to simplify the command:

```
#AWS deploy aws-deploy: docker -H localhost:2374 stack
```

```
deploy myflaskapp -c docker/prod/docker-compose.yml
```

Now run the deploy like this:

```
~/Development/example-flask-sql-ci$ make aws-deploy  
  
docker -H localhost:2374 stack deploy myflaskapp -c docker/prod/docker-compose.yml  
Ignoring unsupported options: build  
  
Creating network myflaskapp_default  
Creating network myflaskapp_flaskappnet  
Creating secret myflaskapp_db_password  
Creating service myflaskapp_visualizer  
Creating service myflaskapp_web  
Creating service myflaskapp_postgres
```

Initialize the production database

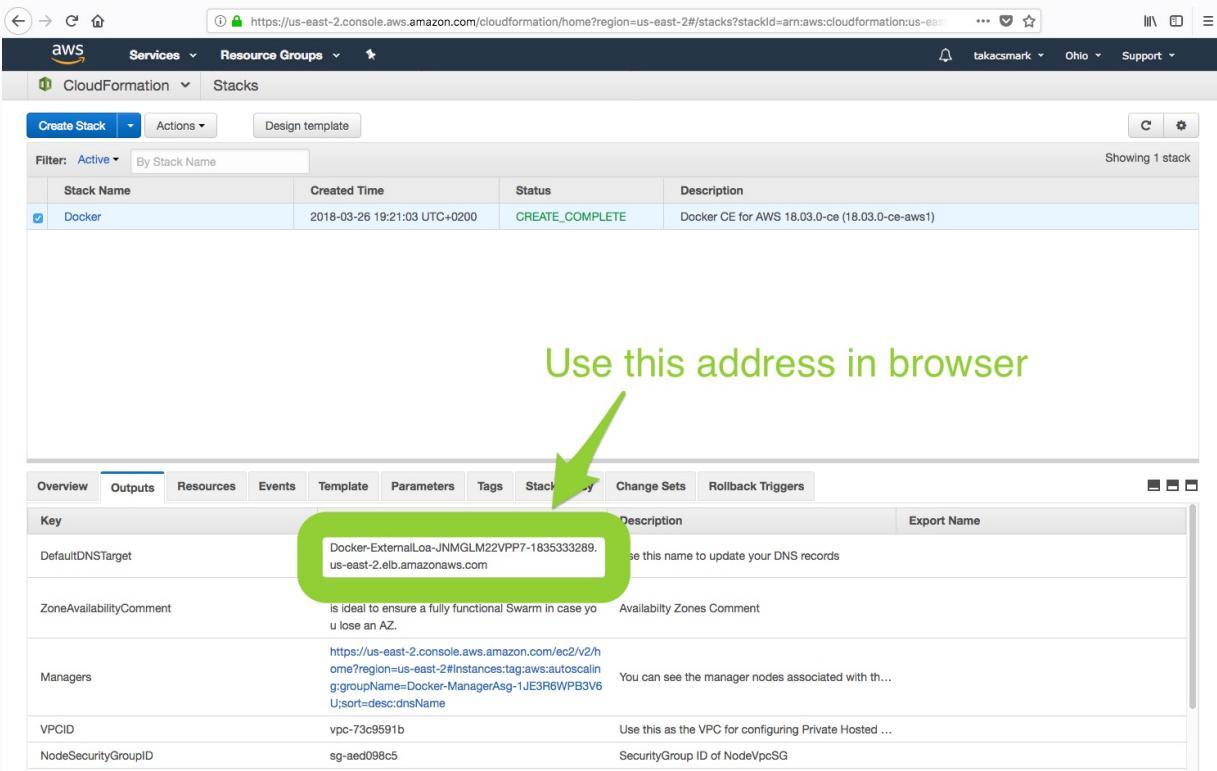
Let's open up an interactive shell and execute the database initialization steps in one of the web containers:

```
~/Development/example-flask-sql-ci$ docker -H localhost:2374 ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 6c6f052849a3  
takacsmark/flask-sql-ci-web:prod "/bin/sh -c 'flask run'" 11 minutes  
ago Up 11 minutes myflaskapp_web.1.slv1dfxwstt46nn5m58vutd3g  
8873c30b06e5 postgres:10.3-alpine "docker-entrypoint.s..." 11 minutes  
ago Up 11 minutes 5432/tcp  
myflaskapp_postgres.1.j5w5yo66tdxbesy7wnx0lochrh 228125850b38  
dockersamples/visualizer:stable "npm start" 11 minutes ago Up 11  
minutes 8080/tcp myflaskapp_visualizer.1.stz6xpuh7etlr1m8qko2x2k0t  
f4149e15fa7a docker4x/l4controller-aws:18.03.0-ce-aws1  
"loadbalancer run --" 26 minutes ago Up 26 minutes l4controller-aws1
```

```
aws af23d5298eed docker4x/meta-aws:18.03.0-ce-aws1 "metaserver -  
iaas_pr..." 26 minutes ago Up 26 minutes 172.31.1.143:9024->8080/tcp  
meta-aws 300b3b1b3a28 docker4x/guide-aws:18.03.0-ce-aws1  
"/entry.sh" 26 minutes ago Up 26 minutes guide-aws 536b7dadea60  
docker4x/shell-aws:18.03.0-ce-aws1 "/entry.sh /usr/sbin..." 28  
minutes ago Up 27 minutes 0.0.0.0:22->22/tcp shell-aws  
~/Development/example-flask-sql-ci$ docker \ -H localhost:2374 exec  
\ -ti 6c6f052849a3 /bin/sh ~/myflaskapp # . ./exportsecret.sh  
~/myflaskapp # flask db init && \ flask db migrate \ && flask db  
upgrade Creating directory /root/myflaskapp/migrations ... done  
Creating directory /root/myflaskapp/migrations/versions ... done  
Generating /root/myflaskapp/migrations/env.py ... done Generating  
/root/myflaskapp/migrations/README ... done Generating  
/root/myflaskapp/migrations/alembic.ini ... done Generating  
/root/myflaskapp/migrations/script.py.mako ... done Please edit  
configuration/connection/logging settings in  
'/root/myflaskapp/migrations/alembic.ini' before proceeding. INFO  
[alembic.runtime.migration] Context impl PostgresqlImpl. INFO  
[alembic.runtime.migration] Will assume transactional DDL. INFO  
[alembic.autogenerate.compare] Detected added table 'users' INFO  
[alembic.autogenerate.compare] Detected added table 'roles'  
Generating /root/myflaskapp/migrations/versions/1bcf9954cd0a_.py  
... done INFO [alembic.runtime.migration] Context impl  
PostgresqlImpl. INFO [alembic.runtime.migration] Will assume  
transactional DDL. INFO [alembic.runtime.migration] Running upgrade  
-> 1bcf9954cd0a, empty message
```

Accessing the application

You can access the application using the DefaultDNSTarget field on the output tab, make sure to add port 5000 (:5000) after the URL.



Use this address in browser

Key	Description	Export Name
DefaultDNSTarget	Docker-ExternalLs-JNMGLM22VPP7-1835333289. us-east-2.elb.amazonaws.com	Use this name to update your DNS records
ZoneAvailabilityComment	Is ideal to ensure a fully functional Swarm in case you lose an AZ.	Availability Zones Comment
Managers	https://us-east-2.console.aws.amazon.com/ec2/v2/home?region=us-east-2#instances;tag:aws:autoscaling:groupName=Docker-ManagerAsg-1JE3R6WPB3V6	You can see the manager nodes associated with them.
VPCID	vpc-73c9591b	Use this as the VPC for configuring Private Hosted ...
NodeSecurityGroupID	sg-aed098c5	SecurityGroup ID of NodeVpcSG

Use the “create account” link to create an account on your production website.

Welcome to Flask MySQL CI Docker

This is a starter Flask template. It includes Bootstrap 4, jQuery 3, Flask-SQLAlchemy, WTForms, and various testing utilities out of the box.

[Learn more »](#)

</> Bootstrap 4

Sleek, intuitive, and powerful mobile-first front-end framework for faster and easier web development.

[Official website »](#)

SQLAlchemy

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

[Official website »](#)

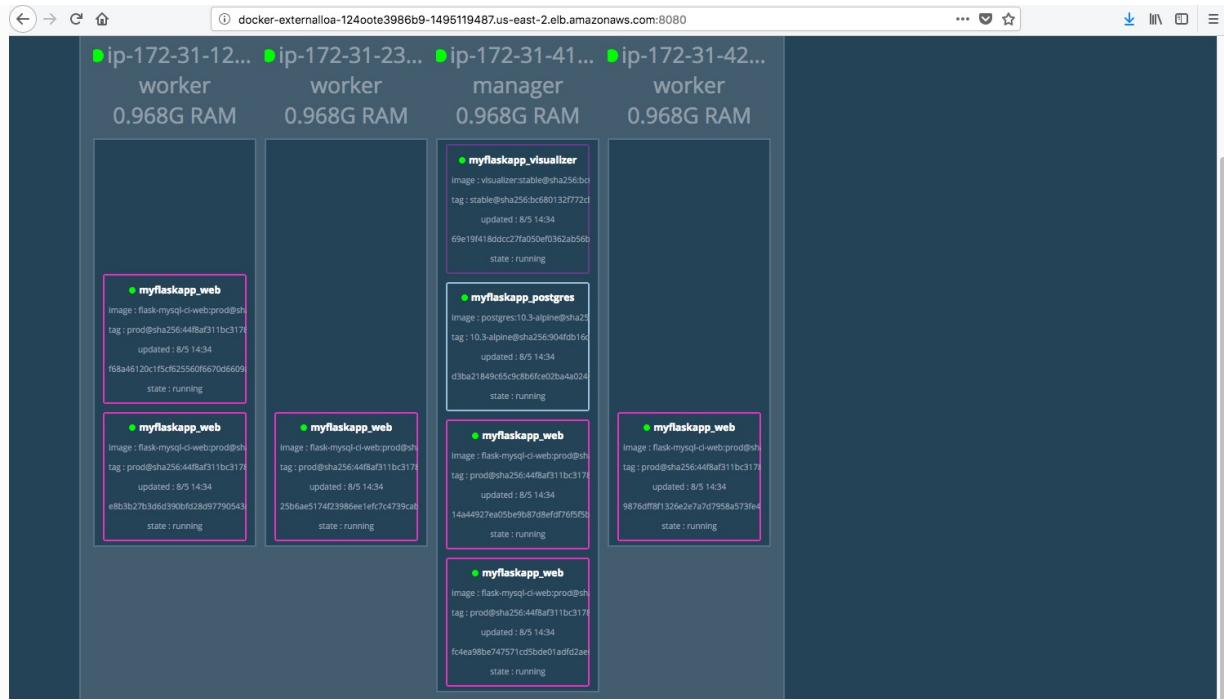
WTForms

WTForms is a flexible forms validation and rendering library for python web development.

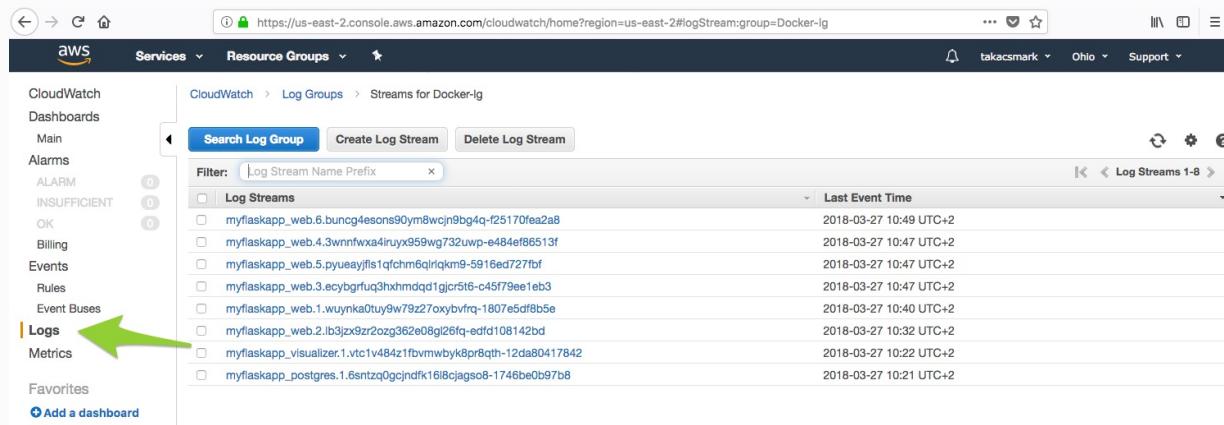
[Official website »](#)

Experiment with your cluster, try scaling up and down with commands like this one: `~/Development/example-flask-sql-ci$ docker \ -H localhost:2374 service scale myflaskapp_web=6`

You can check the results in the visualizer.



You can use Amazon's CloudWatch service to monitor your application, check out the logs:



To control your AWS expenses, remove the stack after you are finished:

Delete stack when finished

Key	Value	Description	Export Name
DefaultDNSTarget	Docker-ExternalLoa-21E0PZAFX311-1291106327.us-east-2.elb.amazonaws.com	Use this name to update your DNS records	
ZoneAvailabilityComment	This region has at least 3 Availability Zones (AZ). This is ideal to ensure a fully functional Swarm in case you lose an AZ.	Availability Zones Comment	
Managers	https://us-east-2.console.aws.amazon.com/ec2/v2/home?region=us-east-2#instances:tag:aws:autoscaling:groupName=Docker-ManagerAeg-LU213GWO1758;sort=desc:dn	You can see the manager nodes associated with this cluster.	
VPCID	vpc-5a7aea32	Use this as the VPC for configuring Private Hosted Zones	
NodeSecurityGroupID	sg-22347e49	SecurityGroup ID of NodeVpcSG	

Deploy to a Kubernetes cluster

Setting up Swarm was a breeze, both locally and in the cloud in the AWS example. The reason for this is that the details of the Swarm infrastructure are bundled with Docker, and all the complexity is hidden behind the `docker swarm` command.

Kubernetes, also called k8s or Kube, is not integrated so tightly into Docker, but Docker is getting closer and closer to Kubernetes, so this situation is likely to improve for the benefit of users.

Kubernetes is an open source product to automate the deployment, scaling, and management of containerized applications. The purpose of Kubernetes is the same as the purpose of Swarm, but there are differences in the terminology and tools.

We've already learned that you can initialize a Docker Swarm with the `docker swarm init` command, the Swarm features come built-in with the Docker Engine. In the Swarm world, our computers (or Docker Engines) in the cluster are called nodes, and we use services to scale and manage the components of our applications.

In the Swarm world, you can have manager nodes and worker nodes in your cluster as we've seen in the previous chapters. When you schedule services in your cluster, Swarm will schedule tasks on the cluster nodes.

In the Kubernetes world, we have master nodes and worker nodes. The role of the master node is the same as the role of the manager node in the Swarm.

Kubernetes schedules containers in "Pods". A Pod is the unit of deployment in Kubernetes; you can deploy one or more containers into a Pod. Containers deployed to the same Pod run on the same node and share resources.

From the Kubernetes documentation: "A pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled —

in a pre-container world, they would have executed on the same physical or virtual machine. Pods are a model of the pattern of multiple cooperating processes which form a cohesive unit of service. They simplify application deployment and management by providing a higher-level abstraction than the set of their constituent applications. Pods serve as the unit of deployment, horizontal scaling, and replication. Colocation (co-scheduling), shared fate (e.g., termination), coordinated replication, resource sharing, and dependency management are handled automatically for containers in a pod.”

Pods are created and managed with Deployments. Deployments are described in Yaml files, similar to the Compose file, but with a proprietary Kubernetes format. You describe the desired state of your application in Deployment objects, and the Kubernetes Deployment controller will align your application state to reflect the desired state.

Once your Pods are deployed based on their Deployment descriptors, you'll need a way to access your pods. Kubernetes Services are the way to address your Pods. Since Pods can be destroyed and re-created, you need a logical level that you can use to address the services provided by your Pods. Services are described in Yaml files, too.

We'll see an example for all of the above in this chapter.

There are many ways to set up a Kubernetes cluster. Today you can find 40+ options on the official Kubernetes setup site: <https://kubernetes.io/docs/setup/pick-right-solution/>.

We'll pick the simplest way and use a local Minikube cluster to deploy our application.

Please install a hypervisor, kubectl and Minikube as described [here](#).

Start up Minikube on your machine.

```
~ minikube start
```

Minikube has a nice visual dashboard let's start it up.

```
~ minikube dashboard &
```

Create Kubernetes deployment configuration

Our application uses the secret information that we store in our `_app_secret.txt` and `_db_password.txt` files. We used these files to create secrets in Swarm, and now we'll learn how to use the files to create secrets in Kubernetes (please make sure that there is no trailing newline character in the

```
password file): ~/Development/example-flask-sql-ci$ kubectl create  
secret generic app-secret --from-file ./_app_secret.txt secret  
"app-secret" created ~/Development/example-flask-sql-ci$ kubectl  
create secret generic db-password --from-file ./_db_password.txt  
secret "db-password" created
```

Double check the results:

```
~/Development/example-flask-sql-ci$ kubectl get secrets  
NAME          TYPE        DATA  
app-secret    Opaque      1  
db-password   Opaque      1  
default-token-2llsn  kubernetes.io/service-account-token 3
```

See the details of a secret:

```
~/Development/example-flask-sql-ci$ kubectl describe \  
secrets/db-password
```

```
Name:           db-password  
Namespace:     default  
Labels:         <none>  
Annotations:    <none>
```

```
Type:  Opaque
```

```
Data
```

```
====
```

```
_db_password.txt: 18 bytes
```

Let's see how to use these secrets and create the deployment configuration!

The Kubernetes deployment descriptor files are similar to the Compose file but reflect the Kubernetes concepts. We'll add two files under `docker/prod` :

- Deployment file for the web service - `web-deployment.yaml`.
- Deployment file for the database - `postgres-deployment.yaml`.

In both files, we create an object of the kind "Service" which represents our services in the application. In each file, we'll create another object of the kind "Deployment" which will be the deployment descriptor of our Pods.

Note that our Service in `web-deployment.yaml` is of type "LoadBalancer" and has an external port mapping of port 5000. This makes our web service accessible from the external world.

Our Service in `postgres-deployment.yaml` is of type "ClusterIP" which means, it is accessible from within the cluster.

Our web deployment definition in `web-deployment.yaml` is the following:

```
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    app: myflaskapp
spec:
  ports:
  - name: "5000"
    port: 5000
    targetPort: 5000
  selector:
    app: myflaskapp
  tier: web
  type: LoadBalancer
```

```
LoadBalancer --- apiVersion: apps/v1 kind: Deployment metadata:  
name: web labels: app: myflaskapp tier: web spec: selector:  
matchLabels: app: myflaskapp tier: web replicas: 2 strategy: type:  
Recreate template: metadata: labels: app: myflaskapp tier: web  
spec: containers: - image: takacsmark/flask-sql-ci-web:2.0.0-prod  
name: web env: - name: FLASK_APP value: autoapp.py - name:  
FLASK_DEBUG value: "0" - name: FLASK_ENV value: production - name:  
POSTGRES_PASSWORD value: /run/secrets/db-password - name:  
APP_SECRET value: /run/secrets/app-secret ports: - containerPort:  
5000 name: web volumeMounts: - name: db-password mountPath:  
"/run/secrets/db-password" subPath: "db-password" readOnly: true -  
name: app-secret mountPath: "/run/secrets/app-secret" subPath:  
"app-secret" readOnly: true restartPolicy: Always volumes: - name:  
db-password secret: secretName: db-password items: - key:  
_db_password.txt path: db-password - name: app-secret secret:  
secretName: app-secret items: - key: _app_secret.txt path: app-  
secret
```

Note how we specify the same environment variables as we specified in our Compose file. Please also note how I'm mapping the secrets as volumes to `/run/secrets`, so that they'll be accessible the exact same way that we are using them with Swarm in our `exportsecret.sh`.

The deployment definition of the database service in `postgres-deployment.yaml` is below.

```
apiVersion: v1  
kind: Service  
metadata:  
  name: postgres
```

```
labels:
  app: myflaskapp
spec:
  ports:
    - name: headless
      port: 55555
      targetPort: 0
  selector:
    app: myflaskapp
    tier: postgres
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: proddb
  labels:
    app: myflaskapp
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: myflaskapp
spec:
  selector:
    matchLabels:
      app: myflaskapp
      tier: postgres
  replicas: 1
  strategy:
    type: Recreate
```

```
template:
  metadata:
    labels:
      app: myflaskapp
      tier: postgres
  spec:
    containers:
      - image: postgres:11.1-alpine
        name: postgres
        env:
          - name: POSTGRES_PASSWORD
            valueFrom:
              secretKeyRef:
                name: db-password
                key: _db_password.txt
        volumeMounts:
          - name: proddb
            mountPath: /var/lib/postgresql/data
    volumes:
      - name: proddb
        persistentVolumeClaim:
          claimName: proddb
```

Deploy the application

With the deployment configuration and dashboard in place, we are ready to deploy our Flask application to a Kubernetes cluster in AWS.

Deploy the database with `kubectl create`:

```
~/Development/example-flask-sql-ci$ kubectl create \ -f
docker/prod/postgres-deployment.yaml service "postgres" created
```

```
persistentvolumeclaim "proddb" created deployment.apps "postgres"  
created
```

Deploy the web application:

```
~/Development/example-flask-sql-ci$ kubectl create \  
-f docker/prod/web-deployment.yaml  
  
service "web" created  
deployment.apps "web" created
```

Please go to your dashboard and see how your deployment, service, and pod objects are created.

The screenshot shows the Kubernetes Dashboard interface running on a local host at port 151621. The top navigation bar includes a back arrow, forward arrow, refresh icon, and a search bar with placeholder text "Search". On the right side of the header are icons for download, copy, and other dashboard functions.

The main content area is divided into several sections:

- Cluster Overview:** A sidebar on the left lists cluster components: Namespaces, Nodes, Persistent Volumes, Roles, and Storage Classes. Below this is a dropdown for the active namespace, currently set to "default".
- CPU usage:** A line chart showing CPU usage over time (from 13:46 to 13:55). The Y-axis represents CPU cores, ranging from 0.0003 to 0.001. The X-axis shows the time period. The chart shows a single data series starting near 0.0003 and rising slightly to about 0.0004 by 13:55.
- Memory usage:** A line chart showing memory usage over time (from 13:45 to 13:55). The Y-axis represents Memory (bytes), ranging from 31.0 Mi to 139 Mi. The X-axis shows the time period. The chart shows a single data series starting at 139 Mi and decreasing to approximately 124 Mi by 13:55.
- Workloads:** A section titled "Workloads Statuses" displays three green circular progress indicators, each labeled "100.00%". Below this are tables for "Deployments", "Pods", and "Replica Sets".
- Deployments:** A table listing two deployments:

Name	Labels	Pods	Age	Images
web	app: myflaskapp tier: web	2 / 2	19 days	takacsmark/flask-sql-ci-we
postgres	app: myflaskapp	1 / 1	19 days	postgres:11.1-alpine
- Pods:** A table listing three pods:

Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
web-87cb85cc8-9fgjv	minikube	Running	1	19 days	0	41.504 Mi
web-87cb85cc8-l55s7	minikube	Running	1	19 days	0	40.887 Mi
postgres-d977df768-c	minikube	Running	1	19 days	0	22.270 Mi
- Replica Sets:** A table showing replica sets, which is partially visible at the bottom of the screen.

Let's initialize the database manually. We'll start a shell in one of the pods to achieve this. Let's get the pods in the cluster:

```
~/Development/example-flask-sql-ci$ kubectl get pods NAME
READY STATUS RESTARTS AGE
postgres-7f966b8444-8smsc 1/1 Running 0
14m
web-75cbff64b5-lgv7n 1/1 Running 0 13m
web-75cbff64b5-w8zdm 1/1
Running 0 13m
```

Use `kubectl exec -it` to start an interactive shell against one of the running pods. Please copy the pod name from the output of the previous command:

```
~/Development/example-flask-  
sql-ci$ kubectl exec \ -it web-75cbff64b5-w8zdm -- /bin/sh  
~/myflaskapp # . ./exportsecret.sh ~/myflaskapp # flask db init  
Creating directory /root/myflaskapp/migrations ... done Creating  
directory /root/myflaskapp/migrations/versions ... done Generating  
/root/myflaskapp/migrations/env.py ... done Generating  
/root/myflaskapp/migrations/script.py.mako ... done Generating  
/root/myflaskapp/migrations/README ... done Generating  
/root/myflaskapp/migrations/alembic.ini ... done Please edit  
configuration/connection/logging settings in  
'/root/myflaskapp/migrations/alembic.ini' before proceeding.  
~/myflaskapp # flask db migrate INFO [alembic.runtime.migration]  
Context impl PostgresqlImpl. INFO [alembic.runtime.migration] Will  
assume transactional DDL. INFO [alembic.autogenerate.compare]  
Detected added table 'users' INFO [alembic.autogenerate.compare]  
Detected added table 'roles' Generating  
/root/myflaskapp/migrations/versions/97f2b66356e9_.py ... done  
~/myflaskapp # flask db upgrade INFO [alembic.runtime.migration]  
Context impl PostgresqlImpl. INFO [alembic.runtime.migration] Will  
assume transactional DDL. INFO [alembic.runtime.migration] Running  
upgrade -> 97f2b66356e9, empty message
```

Now you can access the application, and create an account and log-in. To access the web service run by Minikube use this command:

```
$ minikube service web
```

My application looks like this after creating my user and logging in.

The screenshot shows a web browser window with the URL `192.168.99.100:30769` in the address bar. The page title is "Flask MySQL CI Docker". A user is logged in as "takacsmark". The main heading is "Welcome to Flask MySQL CI Docker". Below it, a subtext states: "This is a starter Flask template. It includes Bootstrap 4, jQuery 3, Flask-SQLAlchemy, WTForms, and various testing utilities out of the box." A blue button labeled "Learn more »" is visible. The page features three sections: "Bootstrap 4" (with a link to the official website), "SQLAlchemy" (with a link to the official website), and "WTForms" (with a link to the official website). At the bottom, there are copyright and contact links.

Welcome to Flask MySQL CI Docker

This is a starter Flask template. It includes Bootstrap 4, jQuery 3, Flask-SQLAlchemy, WTForms, and various testing utilities out of the box.

[Learn more »](#)

Bootstrap 4

Sleek, intuitive, and powerful mobile-first front-end framework for faster and easier web development.

[Official website »](#)

SQLAlchemy

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

[Official website »](#)

WTForms

WTForms is a flexible forms validation and rendering library for python web development.

[Official website »](#)

© Mark Takacs

About Contact

You can monitor and manage your cluster from the dashboard, have fun!

Once you are done, stop the cluster with the below command:

```
~/Development/example-flask-sql-ci$ minikube stop
```