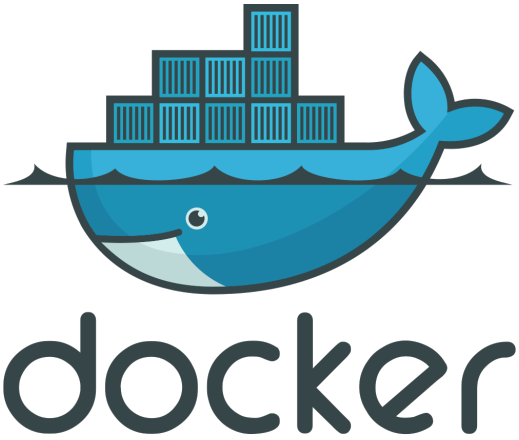
# Docker for Web Developers

Craig Buckler, @craigbuckler




DockerWebDev.com, v1.2.0

# Contents


0.1 Version history . . . . . 1
0.2 Preface . . . . . 2
0.3 Prerequisites . . . . . 3
0.4 Course website . . . . . 4
0.5 Book and/or videos? . . . . . 4
0.6 Example code . . . . . 4
0.7 Chat room . . . . . 5
0.8 Code conventions . . . . . 5
0.9 Further tips . . . . . 5
0.10 About me . . . . . 6
0.11 Copyright and distribution . . . . . 7

**1 Introduction** **9**
1.1 *"It works on my machine, buddy"* . . . . . 9
1.2 Virtual machining . . . . . 10
1.3 Docker delivers . . . . . 11
1.4 Nah, I'm still not convinced . . . . . 11
1.5 Isn't {insert-technology-here} where it's at? . . . . . 13
1.6 Key points . . . . . 15

**2 What is Docker?** **17**
2.1 Containers . . . . . 18
2.2 Images . . . . . 23
2.3 Volumes . . . . . 25
2.4 Networks . . . . . 26

2.5 Docker Compose . . . 27
2.6 Orchestration . . . 27
2.7 Docker client-server application . . . 28
2.8 Docker deployment strategies . . . 28
2.9 Simpler development and production . . . 30
2.10 When *not* to use Docker . . . 30
2.11 Docker alternatives . . . 32
2.12 Key points . . . 33

**3 How to install Docker 35**
3.1 Install Docker on Linux . . . 36
3.2 Install Docker on macOS . . . 38
3.3 Install Docker on Windows . . . 39
3.4 Test your Docker installation . . . 48
3.5 Key points . . . 50

**4 Launch a MySQL database with Docker 51**
4.1 Locate a suitable MySQL image on Docker Hub . . . 52
4.2 Launch a MySQL container . . . 54
4.3 Connect to the database using a MySQL client . . . 57
4.4 Connect to a container shell . . . 58
4.5 View, stop, and restart containers . . . 60
4.6 Define a Docker network . . . 61
4.7 Cleaning up . . . 63
4.8 Launch multiple containers with Docker Compose . . . 66
4.9 Key points . . . 70

**5 WordPress development with Docker 71**
5.1 WordPress requirements . . . 72
5.2 Docker configuration plan . . . 73
5.3 Docker Compose configuration . . . 75
5.4 Launch your WordPress environment . . . 79
5.5 Install WordPress . . . 80

5.6 Local WordPress Development . . . 83
5.7 Key points . . . 86

**6 Application development with Docker 87**
6.1 Container-based application development . . . 88
6.2 What is Node.js? . . . 89
6.3 *Hello World* application overview . . . 90
6.4 Docker configuration plan . . . 95
6.5 Dockerfiles . . . 96
6.6 Build an image . . . 102
6.7 Launch a production container from your image . . . 103
6.8 Launch a development environment with Docker Compose . . . 104
6.9 Live code editing . . . 106
6.10 Remote container debugging . . . 107
6.11 Create an image from a container . . . 117
6.12 Key points . . . 118

**7 Push your Docker image to a Repository 119**
7.1 Why push an image to Docker Hub? . . . 119
7.2 Docker Hub alternatives . . . 120
7.3 Image names and tags . . . 120
7.4 Create a Docker Hub repository . . . 121
7.5 Log in locally . . . 122
7.6 Build an application image . . . 122
7.7 Tag an image . . . 123
7.8 Push to Docker Hub . . . 124
7.9 Distribute your image . . . 125
7.10 Key points . . . 126

**8 Docker orchestration on production servers 127**
8.1 Dependency planning . . . 127
8.2 Application scaling . . . 128
8.3 Orchestration overview . . . 129

8.4 Docker Swarm . . . 130
8.5 Kubernetes . . . 133
8.6 Key points . . . 138

**9 Your Docker journey 139**
9.1 Docker's future . . . 139
9.2 Further Docker help . . . 140

**10 Appendix A: Docker command-line reference 141**
10.1 Log into Docker Hub . . . 141
10.2 Search Docker Hub . . . 141
10.3 Pull a Docker Hub image . . . 142
10.4 List Docker images . . . 142
10.5 Build an image from a Dockerfile . . . 142
10.6 Tag an image . . . 143
10.7 Push tagged images to Docker Hub . . . 143
10.8 Launch a container from an image . . . 143
10.9 List containers . . . 145
10.10 Run a command in a container . . . 145
10.11 Attach to a container shell . . . 145
10.12 Restart a container . . . 145
10.13 Pause a container . . . 146
10.14 Unpause (resume) a container . . . 146
10.15 View container metrics . . . 146
10.16 Increase container resources . . . 146
10.17 Stop a container . . . 147
10.18 Remove stopped containers . . . 147
10.19 View Docker volumes . . . 148
10.20 Delete a volume . . . 148
10.21 Bind mount a host directory . . . 148
10.22 Define a Docker network . . . 149
10.23 View networks . . . 149
10.24 Delete a network . . . 149

10.25 View system disk usage . . . 149
10.26 Full clean start . . . 150

**11 Appendix B: Dockerfile reference 151**
11.1 # comment . . . 151
11.2 ARG arguments . . . 151
11.3 ENV environment variables . . . 152
11.4 FROM <image> starting image . . . 152
11.5 WORKDIR working directory . . . 153
11.6 COPY files from the host to image . . . 153
11.7 ADD files . . . 153
11.8 Mount a VOLUME . . . 153
11.9 Set a USER . . . 154
11.10 RUN a command . . . 154
11.11 EXPOSE a port . . . 154
11.12 CMD execute container . . . 155
11.13 ENTRYPOINT execute container . . . 155
11.14 .dockerignore file patterns . . . 156

**12 Appendix C: Docker Compose reference 157**
12.1 Docker Compose CLI . . . 157
12.2 docker-compose.yml outline . . . 159
12.3 Starting image . . . 159
12.4 build an image from a Dockerfile . . . 160
12.5 Set the container_name . . . 160
12.6 Container depends_on another . . . 160
12.7 Set environment variables . . . 160
12.8 Set environment variables from a env_file . . . 161
12.9 Attach to Docker networks . . . 161
12.10 Attach persistent Docker volumes . . . 162
12.11 Set a custom dns server . . . 163
12.12 expose ports . . . 163
12.13 Define external_links to other containers . . . 164

12.14 Override the default `command` . . . 164
12.15 Override the default `entrypoint` . . . 164
12.16 Specify a `restart` policy . . . 164
12.17 Run a `healthcheck` . . . 165
12.18 Define a logging service . . . 165

**13 Appendix D: quiz project** **167**
13.1 Project overview . . . 168
13.2 Launch in development mode . . . 170
13.3 Launch in production mode . . . 171
13.4 Clean up . . . 171
13.5 Project file structure . . . 172
13.6 `nodejs` Docker image . . . 173
13.7 `nginx` Docker image . . . 177
13.8 `mongodb` Docker image . . . 180
13.9 Node.js build process . . . 181
13.10 Node.js Express.js application . . . 182
13.11 Client-side files . . . 189
13.12 Key points . . . 194

## 0.1 Version history

#### v1.2.0, January 2021 release

- revisions to MySQL (chapter 4) and WordPress (chapter 5)
- GitHub Container Registry link (chapter 7)
- example quiz project update (appendix D)
- minor updates throughout

#### v1.1.0, October 2020 release

- image description clarification (chapter 2)
- Windows WSL2 installation and settings (chapter 3)
- MySQL credentials (chapter 4)
- `docker compose down` usage (chapter 4)
- reddit.com link (chapter 9)
- Dockerfile command corrections (Appendix B)
- links to example code directories
- minor updates and clarifications throughout

#### v1.0.0, July 2020

- initial release

## 0.2 Preface

*Docker is the most useful web development tool you're not using.*

Using Docker, you can:

- **install and run dependencies in minutes**. This includes web servers, databases, language runtimes, applications such as WordPress, and more.
- **manage isolated applications**. Your PC is not *polluted*; you can run multiple editions of any software on the same device at the same time, e.g. MySQL 5 and 8.
- **use your favorite development tools**, editors, and workflows. Web development with Docker is no more difficult than developing code on your local system.
- **distribute your web application** to others on your team. It won't matter if they use another operating system or some dependencies are not available on their platform.
- **deploy your application** to live production servers. It's guaranteed to work and offers scaling opportunities.

Despite these benefits, Docker is often shunned by web developers. It's considered too technical, unnecessary, or something for DevOps experts. Terminology and resources can be impenetrable and tutorials rarely explain how to use Docker during development. I first tried Docker in 2016 and gave up. It took another three years before I realised what I'd been missing.

This course concisely illustrates how to setup good Docker development environments with examples you can adapt for your own web development projects. You'll be running a database, a WordPress environment, and a Node.js application on Windows, macOS, or Linux in minutes. You'll discover how to edit and debug live code using browser DevTools and VS Code. You'll find out how to share your application with others and push to production servers.

I considered naming this book *"Docker: the Good Parts"* or *"Docker Essentials"*.

Perhaps *"How to Use Docker Quickly and Easily for Web Development Projects Without Having to Wade Through Complex Documentation First"* is more apt, but a little long (unless you're into SEO).

## 0.3 Prerequisites

This book is technology-agnostic where possible. The examples refer to specific web development dependencies such as PHP, Node.js, MySQL, and WordPress but you do not require working knowledge of those technologies. All Docker commands and techniques can be used on Windows, macOS, or Linux and adapted to your own stack.

Ideally, you should know a little about web development concepts:

1. web servers and browsers
2. client-side HTML, CSS, and JavaScript
3. server-side languages or runtimes such as Node.js, PHP, Python, Ruby, .NET, etc.
4. databases such as MySQL, PostgreSQL, MongoDB, etc.
5. other dependencies used by your web application, such as build tools, queuing systems, caches, etc.

You don't need to be a full-stack developer, but it's practical to have some knowledge of how these technologies mesh together.

You will also require a terminal and a text editor. Some familiarity with the command-line and Git will be useful.

### 0.3.1 Docker Community Edition

You should be running a recent edition of Windows, macOS, or Linux which supports Docker. All commands shown are cross-platform unless stated otherwise.

The open source (free) Docker Community edition version 19 and Docker Compose 1.26 have been used to create examples and code snippets. These should be compatible with later versions, but consider upgrading if you have earlier installations.

The commercial Docker Enterprise Edition (EE) is primarily a support plan, so it *should* be compatible.

### 0.3.2 Docker Hub

Docker Hub is a service for finding and sharing container images. It's not necessary to create a DockerHub account, but you will need to sign-up at https://hub.docker.com/ if you want to push your own images.

## 0.4 Course website

Course resources, links, announcements, and breaking amendments can be found at dockerwebdev.com

## 0.5 Book and/or videos?

This course is provided as a book and a set of videos depending where you purchased it. The book contains in-depth information but the videos quickly demonstrate concepts, code, and results. They cover the same topics so use either as you prefer.

You can purchase either, both, or the other option on the dockerwebdev.com website.

## 0.6 Example code

You may have received the example code in a ZIP archive, but you can also access the GitHub repository:

https://github.com/craigbuckler/docker-web

Those comfortable with Git can fork the repository and clone their own version. Alternatively, click the **Clone or download** button and choose **Download ZIP**.

## 0.7 Chat room

A course chat room is available at discord.com for registered users to discuss Docker concepts and problems. Your registration invite link is available in your book/course receipt email.

## 0.8 Code conventions

Terminal commands are presented in a code block. A backslash denotes a line break for easier reading, e.g.

```
docker run -d --rm \
  --name mongodb \
  -p 27017:27017 \
  --mount "src=mongodata,target=/data/db" \
  mongo:4
```

These commands can be copied and pasted as-is on Linux, macOS, and Windows terminals using the Windows Subsystem for Linux (WSL).

Windows `cmd` and PowerShell terminal users must remove the `\` and line breaks before pasting.

Code examples such as JavaScript may have additional whitespace in the book. Please refer to the original source and avoid copying directly from the PDF.

## 0.9 Further tips

> Additional information and asides are shown in a breakout box.

These tips show useful options but are not part of the main tutorial.

## 0.10 About me

I'm Craig Buckler – a freelance UK web developer. I've been coding web sites and apps since the mid-1990s (using IE2, kids!) and have been fortunate to undertake projects and technologies which interest me.

You may have encountered my work at SitePoint.com where I've written more than 1,200 tutorials and authored several books including Jump Start Web Performance, Browser DevTool Secrets, and Your First Week With Node.js. I've also developed video courses for O'Reilly.

I mainly bang on about web standards, performance, and keeping things simple.

### 0.10.1 Hire me

I'm available for consultancy, coding, speaking, mentoring, or training. My specialisms include system design, resilient web development, Progressive Web Apps, performance, accessibility, and … *Docker*.

More information and contact details:

- craigbuckler.com personal site
- optimalworks.net business site

## 0.11 Copyright and distribution



Kindle editions of the book purchased from Amazon use Digital Rights Management (DRM): you will only be able to view it on a compatible device.

The book and video files purchased from the dockerwebdev.com website do not use DRM. You are free to copy and use the files on any of your devices without restriction.

Of course, that means you *could* distribute, re-brand, or sell this to others. *Please don't!* This course is the culmination of many months effort. It's self-published – I don't receive income or commission from a publishing company.

Benefits to those buying the course:

1. You'll receive updates and amendments as necessary.
2. You can access the chat room for further support.
3. You can become an affiliate and receive income from your sales.
4. You're enabling the production of further courses.
5. You'll receive my eternal gratitude and can sleep well at night.

Many thanks for buying this course. I hope you find it useful and it changes the way you approach web development. I look forward to receiving your comments and feedback on Twitter @craigbuckler or the course chat room.

# 1 Introduction

> Does our web development stack really need another technology?

Modern web development involves a deluge of files, systems, and components:

- HTML content and templates
- CSS stylesheets and preprocessors such as Sass
- client-side JavaScript including frameworks such as React, Vue.js, and Svelte
- build tools such as bundlers, minifiers, etc.
- web servers such as NGINX or Apache
- server-side runtimes and frameworks including Node.js, PHP, Python, Ruby, .NET etc.
- databases such as MySQL, MariaDB, SQL Server, or MongoDB
- other services for caching, message queues, email, process monitoring, etc.
- Git and Github for source control

Managing this stack can be a *challenge*.

> How many hours do you spend installing, configuring, updating, and managing software dependencies on your development PC?

## 1.1 *"It works on my machine, buddy"*

Imagine your latest application has become successful. You've had to hire another developer to give you more time to rake in money. They turn up at work on day one, clone your repository, launch the code, and – **BANG** – it fails with an obscure error message.

Debugging may help, but your environments are not the same…

- you use a Mac, they use Windows
- you developed the app using Node.js v10, they have v14 installed
- you used MongoDB v3.6, they're on v4.2

The differences mount up.

You may be able to solve these issues within a few hours, but…

- Can you keep every dependency synchronized?
- Is that practical as the team and number of devices grow?
- Are those dependencies available on all development OSes and the production servers?

Some companies would implement a locked-down device policy, where you're prevented from using the latest or most appropriate tools. *(Please don't be that boss!)*

## 1.2 Virtual machining

Rather than restricting devices and software, the application could be run within a Virtual Machine (VM). A VM allows an operating system to be installed in an emulated hardware environment; in essence, it's a PC running on your PC.

Cross-platform VM options include VMware and VirtualBox. You could create a Linux (or other) VM with your application and all its dependencies. The VM is just data: it can be copied and run on any *real* Windows, macOS, or Linux device. Every developer – and the live server – could run the same environment.

Unfortunately, VMs quickly become impractical:

- VM disk images are large and difficult to clone
- an individual VM could be updated automatically or by a single developer so it's out of sync with others
- a VM requires considerable computing resources: *it's a full OS running on emulated hardware within another OS*.

## 1.3 Docker delivers

Docker solves all these problems and more. Rather than installing dependencies on your PC, you run them in lightweight isolated VM-like environments known as *containers*.

In a single command, you can download, configure, and run whatever combination of services or platforms you require. Yes, a single command. *(Admittedly, it can be quite a complicated command, but that's where this book comes in!)*

Development benefits include:

- all developers can use the same Docker containers on macOS, Linux, and Windows
- installation, configuration, maintenance, and testing of applications becomes easier
- applications run in virtual environment isolated from your development PC
- multiple versions of the same application or runtime can be used on the same PC at the same time, e.g. PHP 5.6, 7.0, 7.4 etc.
- developers retain all the benefits of local development and can experiment without risk.

Similar Docker environments can also be deployed in production:

- continuous integration and delivery processes can be simplified for rapid deployment with zero downtime
- performance can be improved with horizontal scaling. It's possible to add more application containers to cope with increased traffic.
- services are more robust. If a container fails, it can be automatically restarted with zero downtime.
- applications can be secured. Containers can be configured to communicate only with each other and not the outside world. A MySQL database could be made available to a WordPress container without exposing itself to the host OS and beyond.

## 1.4 Nah, I'm still not convinced

*Neither was I.*

When I first encountered Docker, it seemed like an unnecessary and somewhat daunting hurdle. I had plenty of experience running VMs and configuring software dependencies – *surely I didn't need it?*

Docker documentation is comprehensive but it has a steep learning curve. Tutorials are often poor and:

1. presume the reader fully understands all the jargon,
2. fail to explain or over-explain esoteric points, and
3. rarely address how Docker can be used during development.

   When I started, I presumed Docker couldn't handle dynamic application restarts or debugging. Tutorials often claimed every code change required a slow and cumbersome application rebuild.

*I gave up.*

I was eventually shown the light by another developer *(thanks Glynne!)* That led to several months deep-diving into Docker and I realised what I'd been missing.

**Example:** I've created many WordPress-based websites.

I'd usually develop these directly on Windows or an Ubuntu VM, where it's necessary to install/update Apache, SSL, PHP, MySQL, and WordPress itself. All before commencing the real development work.

The equivalent Docker process takes minutes to initialize and can be cloned for every new project (see WordPress development with Docker). Each installation exists in its own isolated environment which can be source-controlled and distributed to other developers.

That said, I've never deployed WordPress to a production server using Docker. WordPress hosting is ubiquitous and inexpensive; I'm happy to let someone else manage those dependencies. However, potential problems are minimized because I replicated the production server environment on my development PC.

It is considerably easier to build applications with Docker. Without wanting to sound like a salesperson, *Docker will revolutionize your development!*

## 1.5 Isn't {insert-technology-here} where it's at?

Docker helps regardless of which web development approach and stack you're using. It provides a consistent environment at build time and/or closely matches the dependencies on your production server(s).

Your Docker environment:

1. works without an active/fast internet connection (useful when travelling, during demonstrations, etc.)
2. permits experimentation without risk. No one will mind if you accidentally wipe your local MySQL database.
3. is free from cost and usage restrictions.

### 1.5.1 Monolithic web applications

Monolithic applications contain a mix of front-end and back-end code. Typically, the application uses a web server, server language runtime, data stores, and client-side HTML, CSS, JavaScript and frameworks to render pages and provide APIs. WordPress is a typical example.

Docker can be used to replicate that environment so all dependencies are available on your development PC.

### 1.5.2 Serverless web applications

Serverless applications implement most functionality in the browser typically with a JavaScript framework to create a Single Page Application (SPA). The core site/application is downloaded once.

Additional data and services are provided by small APIs perhaps running as serverless functions. Despite the name, servers are still used – but you don't need to worry about managing them. You create a function which is launched on demand from a JavaScript Ajax request, e.g. code that emails form data to a sales team.

Docker can be used in development environments to:

1. run build processes such as JavaScript module bundling and Sass preprocessing
2. serve the web application, and
3. emulate infrastructures for serverless function testing.

### 1.5.3 Static sites

A static site is constructed using a build process which places content (markdown files, JSON data, database fields, etc.) into templates to create folders of static HTML, CSS, JavaScript, and media files. Those pre-rendered files can be deployed anywhere: no server-side runtime or database is required.

Static sites are often referred to as the *JAMstack* (JavaScript, APIs, and Markdown). All content is pre-rendered where possible, but dynamic services such as a site search can adopt server-based APIs.

Docker can be used to provide a reproducible build environment on any development PC.

## 1.6 Key points

What you've learned in this chapter:

1. Docker can launch all your application's dependencies in individual containers.

   This includes servers, databases, language runtimes, etc. In most cases, these will require little or no configuration.

2. Docker is cross-platform.

   It runs on Windows, macOS, and Linux. Your application will work on any PC.

3. Docker can – *and should* – be used in your development environment.

   You can also use it in production systems if it's practical to do so.

The next chapter describes Docker concepts in more detail.

# 2 What is Docker?

Most tutorials attempt to explain Docker concepts first. That can be daunting so here's the TL;DR alternative…

- Docker runs an application such as MySQL in a single **container**.

  It's a lightweight virtual machine-like package containing an OS, the application files, and all dependencies.

- Your web application will probably require several containers; your code (and language runtime), a database, a web server, etc.

- A container is launched from an **image**.

  In essence, it's a container template which defines the OS, installation processes, settings, etc. in a **Dockerfile** configuration. Any number of containers can be started from the same image.

- Containers start in clean (image) state and data is not permanently stored.

  You can mount Docker **volumes** or bind host folders to retain state between restarts.

- Containers are isolated from the host and other containers.

  You can define a **network** and open TCP/IP ports to permit communication.

- Each container is started with a single Docker command.

  **Docker Compose** is a utility which can launch multiple containers in one step using a `docker-compose.yml` configuration file.

- Optionally, **orchestration** tools such as Docker Swarm and Kubernetes can be used for container management and replication on production systems.

You're welcome to skip the rest of this chapter and jump straight into the Docker examples. It's worth coming back later: the concepts discussed below may change how you approach web development.

## 2.1 Containers

Recall how you could use a Virtual Machine (VM) to install a web application and its dependencies. VM software such as VMware and VirtualBox are known as *hypervisors*. They allow you to create a new virtual machine, then install an appropriate operating system with the required application stack (web server, runtimes, databases, etc.):

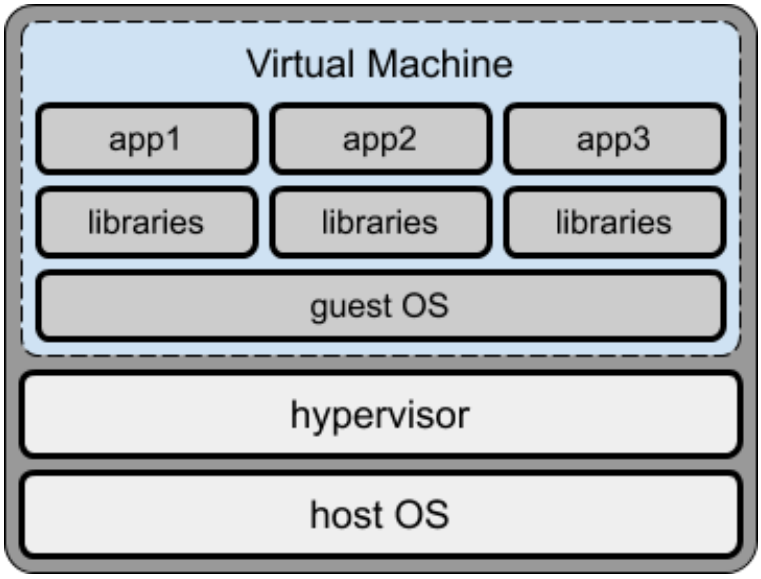



**Figure 2.1:** single Virtual Machine

In some cases, it may **not** be possible to install all applications in a single VM so multiple VMs become necessary:

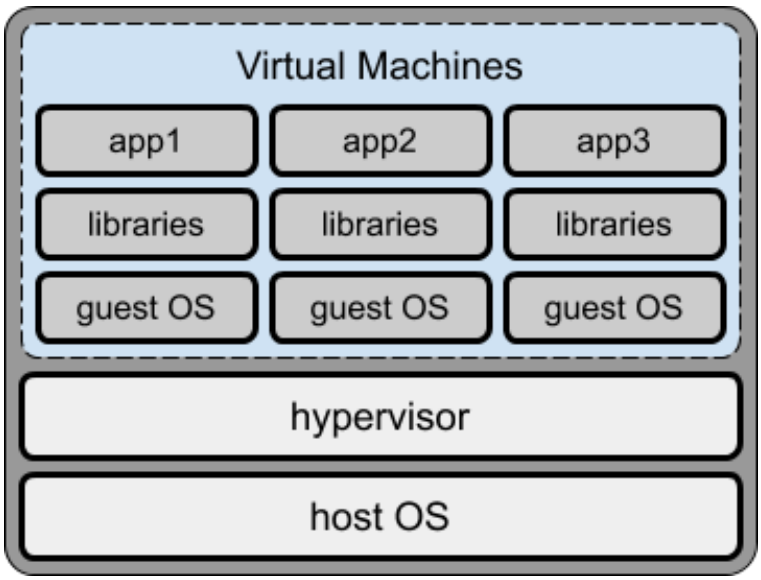



**Figure 2.2:** multiple Virtual Machines

Each VM is a full OS running on emulated hardware in a host OS with access to resources such as networks via the hypervisor. This is a considerable overhead, especially when a dependency could be tiny.

Docker launches each dependency in a separate *container*. It helps to think of a container as a mini VM with its own operating system, libraries, and application files.

In reality:

- a virtual machine hypervisor emulates hardware so you can run a full Operating System
- Docker emulates an Operating System so you can run isolated applications within their own file system.

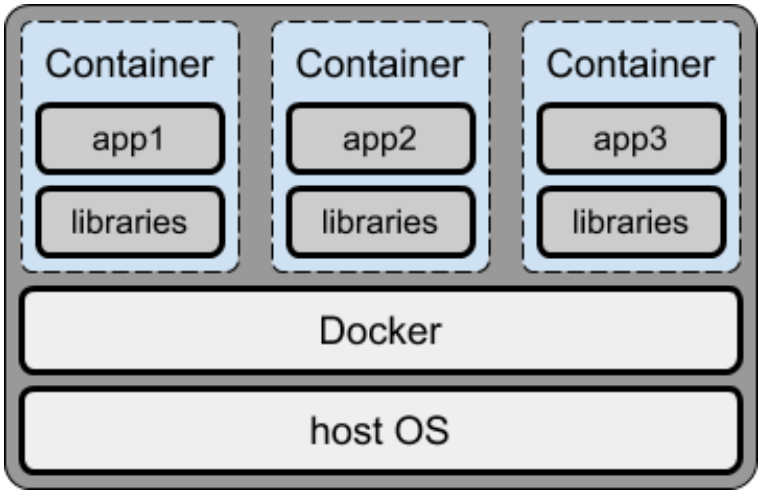



**Figure 2.3:** multiple Docker containers

A container is effectively an isolated wrapper around an executable so Docker requires far fewer host OS resources than a VM.

> It's technically possible to run all your application's dependencies in a single container, but there are no practical benefits for doing so and management becomes more difficult.

**Always use separate containers for your application, the database, and any other dependencies you require.**

### 2.1.1 Containers are isolated

Each container is available at `localhost` or `127.0.0.1`, but a TCP port must be exposed to communicate with the application it runs, e.g.

- port `80` or `443` for a HTTP or HTTPS web servers
- `3306` for MySQL
- `27017` for MongoDB

Docker also allows you to access the container shell to enter terminal commands and expose further ports to attach debuggers and investigate problems.

### 2.1.2 Containers are stateless and disposable

**Data written to the container's file system is lost the moment it is shuts down!**

Any number of containers can be launched from the same base image (see below). This makes scaling easy because every container instance is identical and disposable.

This may change the way you approach application development if you want to use Docker on production servers. Presume your application has a variable which counts the number of logged-in users. If it's running in two containers, either could handle a login so each would have a different user count.

Dockerized web applications should therefore avoid retaining state data in variables and local files. Your application can store data in a database such as redis, MySQL, or MongoDB so state persists between container instances.

> It may be impractical to deploy an existing application using Docker containers if it was developed in a non-stateless way from the start. However, you can still run the application in Docker containers during development.

Which begs the question: *what if your database is running in a container?*

It will also lose data when it restarts, so Docker offers volumes and host folder bind mounts.

> You may be thinking, "ahh, I can get around the state issue by never stopping a container!" That's true. Presuming your application is 100% bug-free. And your runtime is 100% reliable. And the OS never crashes. And you never need update the host OS or the container itself.

### 2.1.3 Containers run on Linux

It doesn't matter what host OS you're using: *Docker containers run natively on Linux*. Even Windows and macOS run Docker containers inside Linux…

The macOS edition of Docker requires VirtualBox.

The Windows edition of Docker allows you to switch between either:

1. the Windows Subsystem for Linux (WSL) 2: a highly-integrated seamless VM which is available on all editions of Windows, or
2. Hyper-V: the Microsoft hypervisor provided with Windows 10 Professional and Enterprise.

It is therefore more efficient to run Docker on Linux but this rarely matters on a development PC. *Use whatever OS and tools you prefer.*

However, if you are using Docker to deploy your application, Linux is the best choice for your live server.

## 2.2 Images

A Docker image is a snapshot of a file and operating system with libraries and application executables. In essence, an image is a *recipe* or *template* for creating a container. *(In a similar way that some computer languages let you define a reusable* `class` *template for instantiating objects of the same type.)*

Any number of containers can be started from a single image. This permits scaling on production servers, although you're unlikely to launch multiple containers from the same image during development.

The Docker Hub provides a repository of commonly-used images for:

- dependencies such as NGINX, MySQL, MongoDB, Elasticsearch, redis etc.
- language runtimes or frameworks such as Node.js, PHP, Python, Ruby, Rust, and any other language you've heard of.
- applications such as WordPress, Drupal, Joomla, Nextcloud etc. *(These often require additional containers such as databases.)*

Reminder: sign-up for Docker Hub account if you'd like to publish your own images.

### 2.2.1 Dockerfile

An image is configured using a Dockerfile. It typically defines:

1. a starting base image – usually an operating system
2. work directories and user permissions
3. all necessary installation steps, such as defining environment variables, copying files from the host, running install processes, etc.
4. whether the container should attach one or more volumes for data storage
5. whether the container should join a network to communicate with others
6. which ports (if any) are exposed to `localhost` on the host
7. the application launch command.

In some cases, you will use an image as-is from Docker Hub, e.g. MySQL. However, your application will require it's own custom Dockerfile.

### 2.2.2 Development and production Dockerfiles

It is possible to create two Dockerfile configurations for your application:

1. one for development.

   It would typically activate logging, debugging, and remote access. For example, during Node.js development, you might want to launch your application using Nodemon to automatically restart it when files are changed.

2. one for production.

   This would run in a more efficient and secure mode. For Node.js deployment, it's likely to use the standard `node` runtime command.

However, a simpler process is described throughout this book.

### 2.2.3 Image tags

Docker Hub is to Docker images what Github is to Git repositories.

Any image you create can be pushed to Docker Hub. Few developers do this, but it may be practical for deployment purposes or when you want to share your application with others.

Images are name-spaced with your Docker Hub ID to ensure no one can use the same name. They also have a tag so you can create multiple versions of the same image, e.g. `1.0`, `1.1`, `2.0`, `latest` etc.

```
<Your-Docker-ID>/<Your-Docker-Hub-Repository>:<tag>
```

Examples: `yourname/yourapp:latest`, `craigbuckler/myapp:1.0`.

Official images on Docker Hub don't require a Docker ID, e.g. `mysql` (which presumes `mysql:latest`), `mysql:5`, `mysql:8.0.20`, etc.

## 2.3 Volumes

Containers do not retain state between restarts. This is generally a *good thing*; any number of containers can be started from the same base image and each can handle incoming requests regardless of how or when they were launched (see Orchestration).

However, some containers – such as databases – absolutely must retain data so Docker provides two storage mechanism types:

1. *Volumes*: a Docker-managed file system, and
2. *Bind mounts*: a file or directory on the host machine.

Either can map to a directory on the container, such as `/data/db` for MongoDB storage.

Volumes are the recommended way to persist data. In some cases, it's the only option – for example, MongoDB does not currently support bind mounts on Windows or macOS file systems.

However, bind mounts are practical during development. An application folder on the host OS can be mounted within the container so any file changes trigger an application restart, browser refresh, etc.

> It is possible to mount the same volume or bind mount on two or more containers. Read-only access should be fine, but you could encounter issues if more than one container attempted to write to the same file at the same time!

## 2.4 Networks

Any TCP/IP port can be exposed on a container, such as `3306` for MySQL. This allows the applications on the host to communicate with the database system at `localhost:3306`.

An application running in another container could **not** communicate with MySQL because `localhost` resolves to itself. For this reason, Docker creates a virtual network and assigns each running container a unique IP address. It's then becomes possible for one container to communicate with another using its address.

Unfortunately, Docker IP addresses can change every time a container is launched. An easier option is to create your own Docker virtual network. Any container added to that network can communicate with another using its name, i.e. `mysql:3306` resolves to the correct address.

Container TCP/IP ports can be exposed:

1. within the virtual network only, or
2. within the virtual network and to the host.

Presume you are running two containers on the same Docker network:

1. a container named `phpapp` which exposes a web application on port `80`
2. a container named `mysql` which exposes a database on port `3306`.

During development, you would want both ports exposed to the host. The application can be launched in a web browser at `http://localhost/` (port 80 is the default) and MySQL clients can connect to `http://localhost:3306/`.

In production environments, the `mysql` port need not be exposed to the host. The `phpapp` container can still communicate with `mysql:3306`, but unscrupulous crackers would not be able to probe port `3306` on the host.

With careful planning, it's possible to create complex Docker networks which heighten security, e.g. `mysql` and `redis` containers can be accessed by `phpapp` but they cannot access each other.

## 2.5 Docker Compose

A single container is launched with a single `docker` command. An application requiring several containers – say Node.js, NGINX, and MongoDB – must be started with three commands. You could launch each in three terminals in the correct order (probably MongoDB, then the Node.js application, then NGINX).

Docker Compose is a tool for managing multiple containers with associated volumes and networks. A single configuration file, normally named `docker-compose.yml`, defines the containers and can override Dockerfile settings where necessary.

It's practical to create a Docker Compose configuration for development. You could also create one for production, but there are better options…

## 2.6 Orchestration

Containers are portable and reproducible. A single application can be scaled by launching identical containers on the same server, another server, or even a different data center on the other side of the world.

The process of managing, scaling, and maintaining containers is known as *orchestration*. Docker Compose can be used for rudimentary orchestration, but it's better to use specialist tools such as:

- Docker Swarm or
- Kubernetes

Cloud hosts offer their own orchestration solutions, such as AWS Fargate, Microsoft Azure, and Google Cloud. These are often based on Kubernetes but may have custom options or tools.

## 2.7 Docker client-server application

Docker is a client-server application. The server is responsible for container management and is controlled via a REST API. The command-line interface (CLI) communicates with this API, so it's possible to run a server daemon anywhere and connect from another device.

This rarely matters during development: the Docker server and CLI is installed on the same PC.

> You can communicate with the API using any HTTP client such as cURL. This is beyond the scope of this book, but it allows you to programmatically run any Docker process.

## 2.8 Docker deployment strategies

You can use Docker and containers in any way that is practical for your project.

This book suggests you **always** use Docker during development. It allows you to create robust and portable environments where your application and each dependency run in separate containers. Chapters 4, 5, 6, and Appendix D provide recipes you can adapt to your projects.

However, deploying your application to a live server raises further options to consider…

### 2.8.1 Use Docker for development only

Docker is used to emulate your live server's production environment on your development PC. The live server itself does not use containers.

This may be practical when you're using infrastructures, platforms, or software as a service (IaaS, PaaS, SaaS) where a pre-built environment is provisioned for you. Possible examples include serverless and WordPress hosts.

### 2.8.2 Use Docker on production servers where practical

Your live production server uses Docker containers for some – *but not all* – dependencies. Your application is likely to be a good candidate, but a database could be provided by a cloud service, and a load balancer could be supplied by the hosting company.

Your development PC can still emulate this environment using Docker containers. That said, a test database could be provided by the same cloud service to eliminate compatibility issues.

### 2.8.3 Use Docker for both development and production

You use *mostly* identical Docker containers in both development and production. It may be necessary to create slightly different live server configurations or consider orchestration options.

### 2.8.4 Concurrent processing considerations

Runtimes such as Node.js and Python run scripts on a single processing thread. A server with 16 CPU cores executing a single instance of an application will have fifteen cores sitting idle!

> Note: some stacks alleviate this situation with a web server. PHP is single-threaded, but Apache launches a thread for each user request so multiple PHP processes run in parallel. This method has its own resourcing problems, though.

Multiple instances of Node.js applications can be launched on the same server using clustering or process managers such as PM2. However, it is generally more practical to use Docker to launch and manage multiple application containers as resources permit. Each container is isolated so, if an individual instance crashes, it will not affect others and can be restarted.

## 2.9 Simpler development and production

This book uses the following approach where practical:

1. An application Dockerfile configures the **production** environment only.
2. Docker Compose is used to override this base configuration for **development** purposes.

An image can therefore be used as-is on production servers regardless of whichever orchestration or deployment process is adopted.

Don't worry about this for now – the process will become clearer in the following chapters.

## 2.10 When *not* to use Docker

Using Docker during development has no downsides. It enables you to install dependencies on any OS and emulate a live system. You can easily share that isolated environment with others while retaining your favorite editor and tools.

However, Docker is not a magical solution which solves all your production woes! There are situations when Docker may not be appropriate…

1. **Your application is not stateless**

   Dockerizing an existing monolithic application can be difficult if it was not originally designed for a container-based deployment. Programs which store state in variables or files will need to be adapted to use other data stores.

2. **You're using a Windows Server**

   Docker is native on Linux but Windows runs containers in a Hyper-V virtual machine or WSL2 (effectively another VM). It's an additional overhead and, although Docker lets you run Linux dependencies, it may be more practical to provision a Linux server.

3. **Performance is critical**

   Docker containers have imposed CPU and RAM limits. These are configurable, but an application running on the host OS will always be faster.

   That said, Docker *can* implement parallel processing by scaling horizontally if your application generally runs on a single CPU core.

4. **Stability is important**

   Docker is mature, but it's another dependency to install, update, and manage. *Do you have in-house container management expertise?*

   Your application may seem more robust since containers can be scaled and automatically restarted. That doesn't mean it's crashing less often than before!

5. **To store mission-critical data**

   Volumes and bind mounts can store persistent data, but these are more difficult to manage and back-up than standard file system options.

6. **To improve security**

   Containers are isolated but, unlike a real VM, they are not fully sandboxed from the host OS. Docker provides options for hiding dependencies, but it's not a substitute for robust security.

7. **To create GUI applications**

   Someone, somewhere will have created a cross-platform graphical interface application using containers. That doesn't make Docker the ideal solution!

8. **Because Docker is cool**

   Jumping on a technology bandwagon without proper investigation and justification is doomed to fail.

## 2.11 Docker alternatives

Docker is the most-used container solution but it's not the only option. Alternatives include:

- Apache Mesos
- containerd
- Linux containers
- RedHat OpenShift and its Docker-compatible podman manager
- OpenVz

## 2.12 Key points

What you've learned in this chapter:

1. The Docker server manages *containers*.

   It's an isolated wrapper around an application, which seems similar to a virtual machine but is more lightweight.

2. Containers are launched from a single *image* template configured by a *Dockerfile*.

   Images for hundreds of applications are available on Docker Hub.

3. Containers are stateless, but can attach to Docker disk *volumes* or *bind-mounted* folders on the host OS.

4. Containers can expose application ports and communicate over internal Docker *networks*.

   Ports can also be exposed to the host OS.

5. *Docker Compose* can be used to launch multiple containers at once.

6. *Orchestration* tools such as Docker Swarm and Kubernetes can be used to launch and scale containers across multiple systems in production environments.

7. Docker is practical during development.

   However, it's not necessarily essential or practical to use it for every application component on production systems.

Enough theory. It's time to install Docker…

# 3 How to install Docker

Docker can be installed on Linux, mac OS, or Windows 10.

Requirements and installation instructions can be found on the Docker Docs help pages.

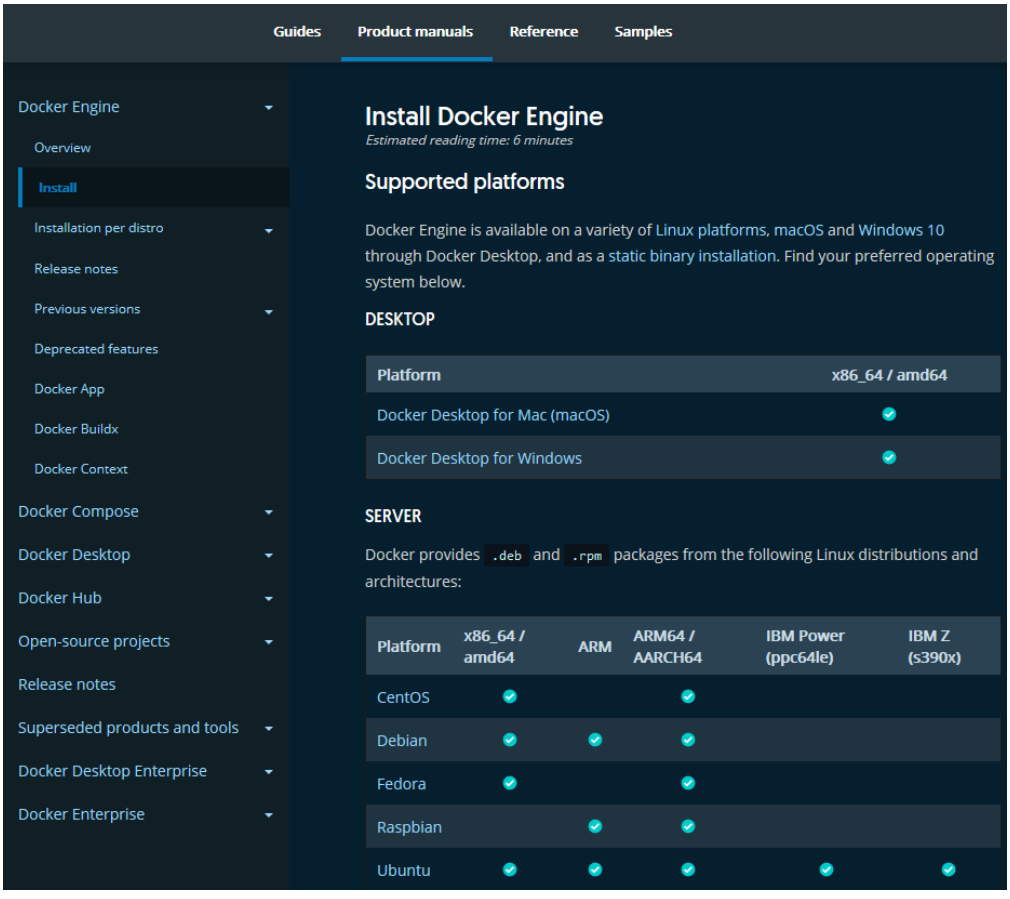



**Figure 3.1:** Docker Docs installation

## 3.1 Install Docker on Linux

Docker is often available in official Linux repositories, although these usually offer older editions. The latest edition is supported on recent 64-bit editions of popular Linux distros:

- Ubuntu (and derivatives such as Mint)
- CentOS
- Debian
- Fedora

Static binaries are available for other distros, although Googling *"install Docker on [your OS]"* may provide easier instructions, e.g. *"install Docker on a Raspberry Pi"*.

Follow the Docker documentation for your distro. For example, Docker for Ubuntu is installed with the following commands:

```
sudo apt-get remove docker docker-engine docker.io containerd runc

sudo apt-get update

sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo apt-key add -

sudo apt-key fingerprint 0EBFCD88

sudo add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) \
  stable"

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io
```

> Convenience scripts are also available to run these commands for you, but the Docker documentation warns they are a security risk and should not be used in production environments:
>
> ```
> curl -fsSL https://get.docker.com -o get-docker.sh
> sudo sh get-docker.sh
> ```

To run Docker commands as a non-root user (without `sudo`), create and add yourself to a `docker` group:

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Then reboot to apply all changes.

### 3.1.1 Install Docker Compose on Linux

Docker Compose is installed separately using the command:

```
sudo curl \
  -L "https://github.com/docker/compose/releases/download/<VERSION>/ \
      docker-compose-$(uname -s)-$(uname -m)" \
  -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
```

where `<VERSION>` is the latest release number, e.g. `1.27.4`.

## 3.2 Install Docker on macOS

Docker Desktop for macOS Sierra 10.13 and above can be downloaded from Docker Hub. The package includes the Docker server, CLI, Docker Compose, Docker Swarm, and Kubernetes.

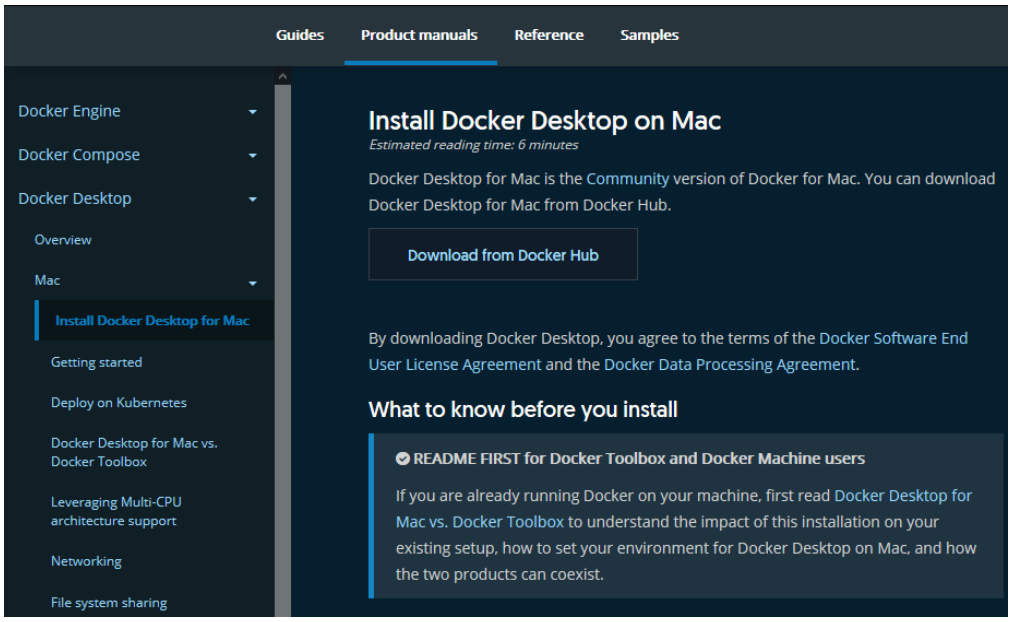



**Figure 3.2:** Docker Desktop for macOS

Two editions are available: **stable** and **edge** with experimental features. The **stable** version is best for most developers.

Double-click `Docker.dmg` to open the installer, then drag the Docker icon to the **Applications** folder. Double-click **Docker.app** in that folder to launch Docker.

After completion, the whale icon in the status bar indicates Docker is running and commands can be entered in the terminal.


**Figure 3.3:** Docker icon on macOS status bar

## 3.3 Install Docker on Windows

Docker Desktop for Windows requires either WSL2 or Hyper-V.

### 3.3.1 Windows Subsystem for Linux (WSL) 2

WSL allows you to run full Linux environments directly on Windows 10.

> IMPORTANT! You can **not** install the Linux edition of Docker within a WSL-powered Linux distro. You must install Docker Desktop for Windows which allows Docker commands to be run in all Windows and Linux terminals.

WSL2 is the recommended default option for Docker on Windows. It is faster than Hyper-V and available in all editions of Windows from the May 2020 update (version 2004, OS build 19041).

> Windows 10 S is not supported but you can normally upgrade to Home in the Settings.
>
> You may be able to trigger the 2004 update: click **Check for updates** in the **Update & Security** panel of **Settings**. If your PC reports that 2004 is not yet available, you must either wait until Microsoft releases a fix for your device or use Hyper-V and switch to WSL2 later.

To install WSL2:

1. Enable hardware virtualization support in your BIOS.

   This will be active on most devices, but check by rebooting and accessing your PC's BIOS panels —- typically by hitting `DEL`, `F2`, or `F10` as your system starts. Look for **Virtualization Technology**, **VTx** or similar options. Ensure they are enabled, save, and reboot.

   > WARNING! Be careful when changing BIOS settings – one wrong move could trash your PC.

2. Enable the **Virtual Machine Platform** and **Windows Subsystem for Linux** options in the **Turn Windows features on or off** panel:

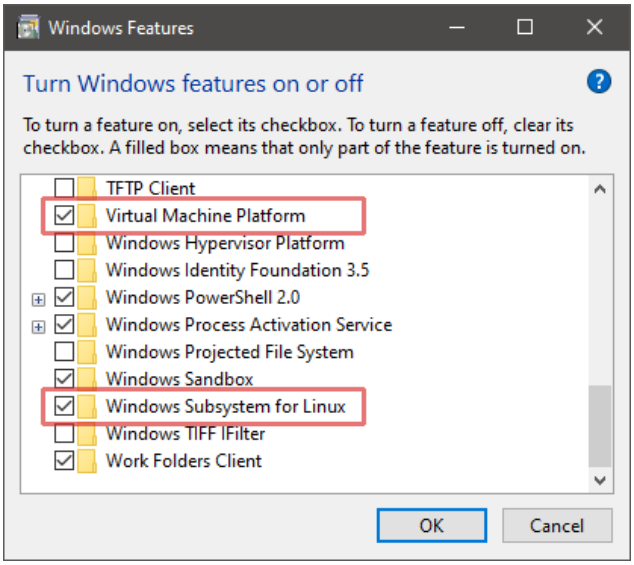



**Figure 3.4:** Enable WSL in Windows

This can be accessed by hitting the Start button and typing the panel name or from **Programs and Features** in the classic Control Panel.

3. Reboot, then enter the following command in a Windows Powershell or `cmd` prompt to set WSL2 as the default:

```
wsl --set-default-version 2
```

4. Download and install your preferred distro by searching for "Linux" in the **Microsoft Store** app. **Ubuntu** is a good choice.

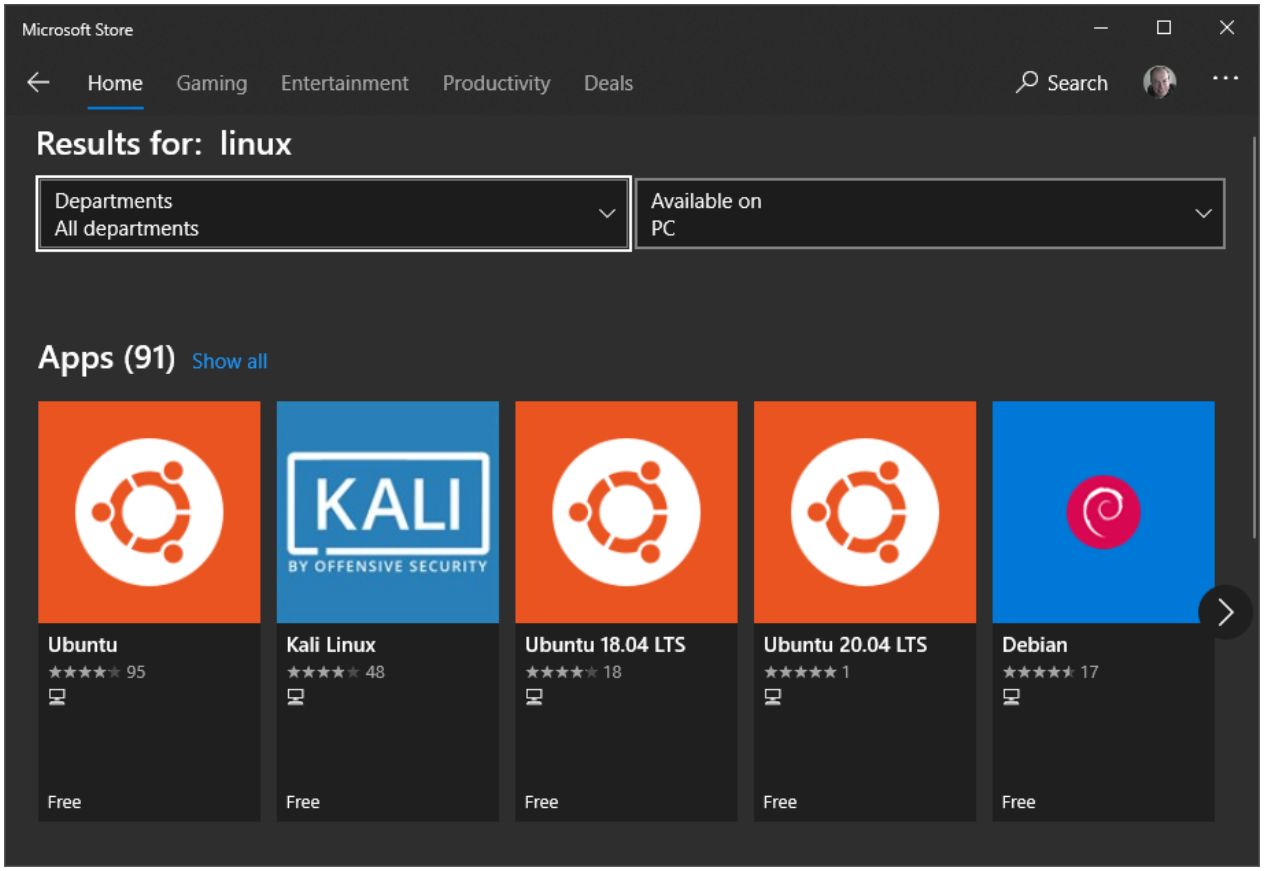



**Figure 3.5:** Windows Store

5. To complete the installation, launch your distro by clicking its Store's **Launch** button or choosing its icon from the Start menu.

> You *may* be prompted to install a kernel update – follow the instructions and launch the distro again.

6. Enter a Linux username and password. These are separate from your Windows credentials although choosing the same ones can be practical.

7. Ensure your distro is up-to-date. For example, on an Ubuntu bash prompt enter:

```
sudo apt update && sudo apt upgrade
```

You can now install Docker Desktop (see below). For the best performance and stability, store development files in your Linux file system and run Docker from your Linux terminal.

> More information about installing and using WSL2:
>
> - Windows Subsystem for Linux 2: The Complete Guide, and
> - optionally, Windows Terminal: The Complete Guide.

### 3.3.2 Hyper-V

The Microsoft Hyper-V hypervisor is provided free with Windows 10 Professional and Enterprise. *(Windows Home users must use WSL2.)*

To install Hyper-V:

1. Enable hardware virtualization support in your BIOS.

   This will be active on most devices, but check by rebooting and accessing your PC's BIOS panels —- typically by hitting `DEL`, `F2`, or `F10` as your system starts. Look for **Virtualization Technology**, **VTx** or similar options. Ensure they are enabled, save, and reboot.

   > WARNING! Be careful when changing BIOS settings – one wrong move could trash your PC.

2. Enable the **Hyper-V** option in the **Turn Windows features on or off** panel then reboot.

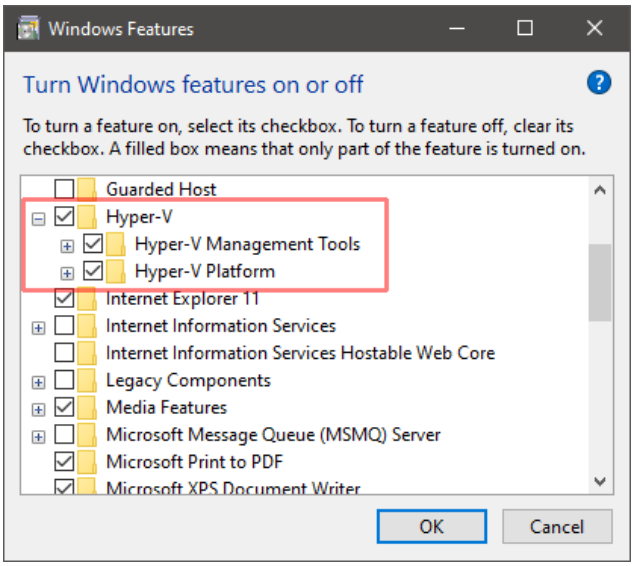



**Figure 3.6:** Enable Hyper-V in Windows

   This can be accessed by hitting the Start button and typing the panel name or from **Programs and Features** in the classic Control Panel.

You can now install Docker Desktop (see below).

### 3.3.3 Install Docker Desktop for Windows

Docker Desktop for Windows 10 can be downloaded from Docker Hub. The installer includes the Docker server, CLI, Docker Compose, Docker Swarm, and Kubernetes.

Two editions are available: **stable** and **edge** with experimental features. The **stable** version is best for most developers.

Double-click `Docker Desktop Installer.exe` to start the installation process. After completion and launch, the whale icon in the notification area of the task bar indicates Docker is running and ready to accept commands in the Windows Powershell/`cmd` terminal (and Linux if using WSL2).

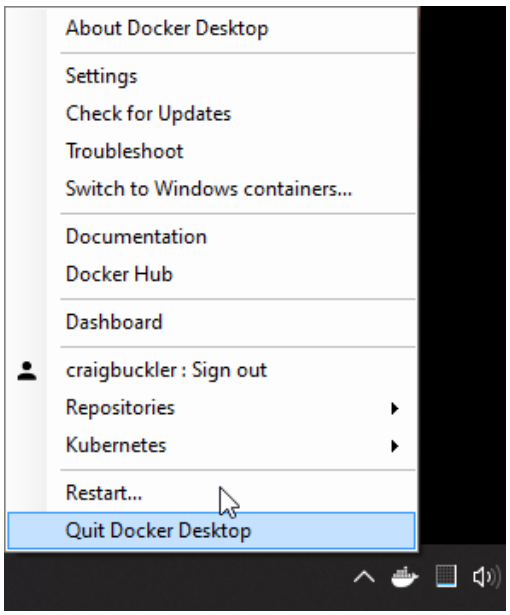



**Figure 3.7:** Docker icon on Windows task bar

### 3.3.4 Docker Engine Settings

Docker uses WSL2 as the default engine when available. You will be prompted to confirm this choice during installation and after WSL2 is installed.

Alternatively, WSL2 can be enabled by checking **Use the WSL 2 based engine** in the **General** tab of **Settings** accessed from the Docker task bar icon. Unchecking the option reverts to Hyper-V.

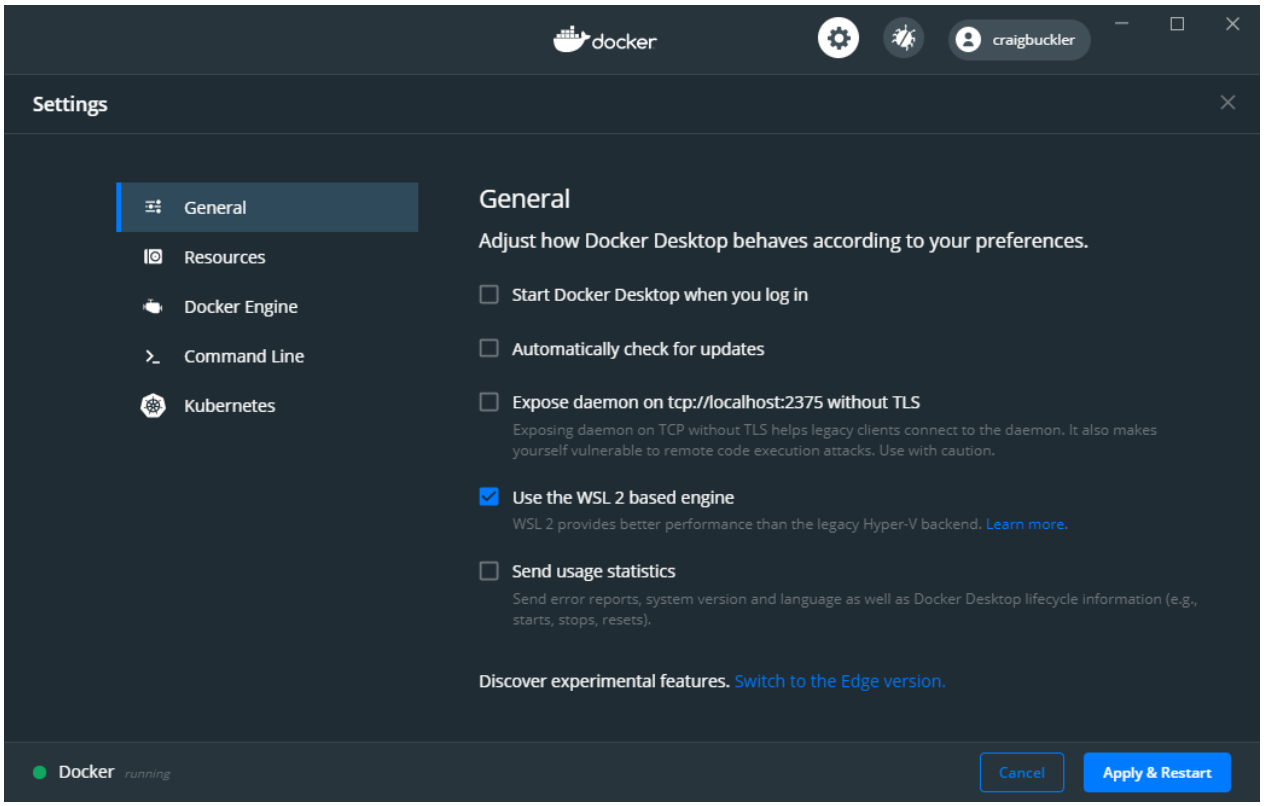



**Figure 3.8:** Docker Windows engine

**When using WSL2**, at least one Linux distro must be enabled – the default is chosen. You can also permit Docker commands in other distros by accessing the **WSL integration** panel in the **Resources** section of the Docker **Settings**:

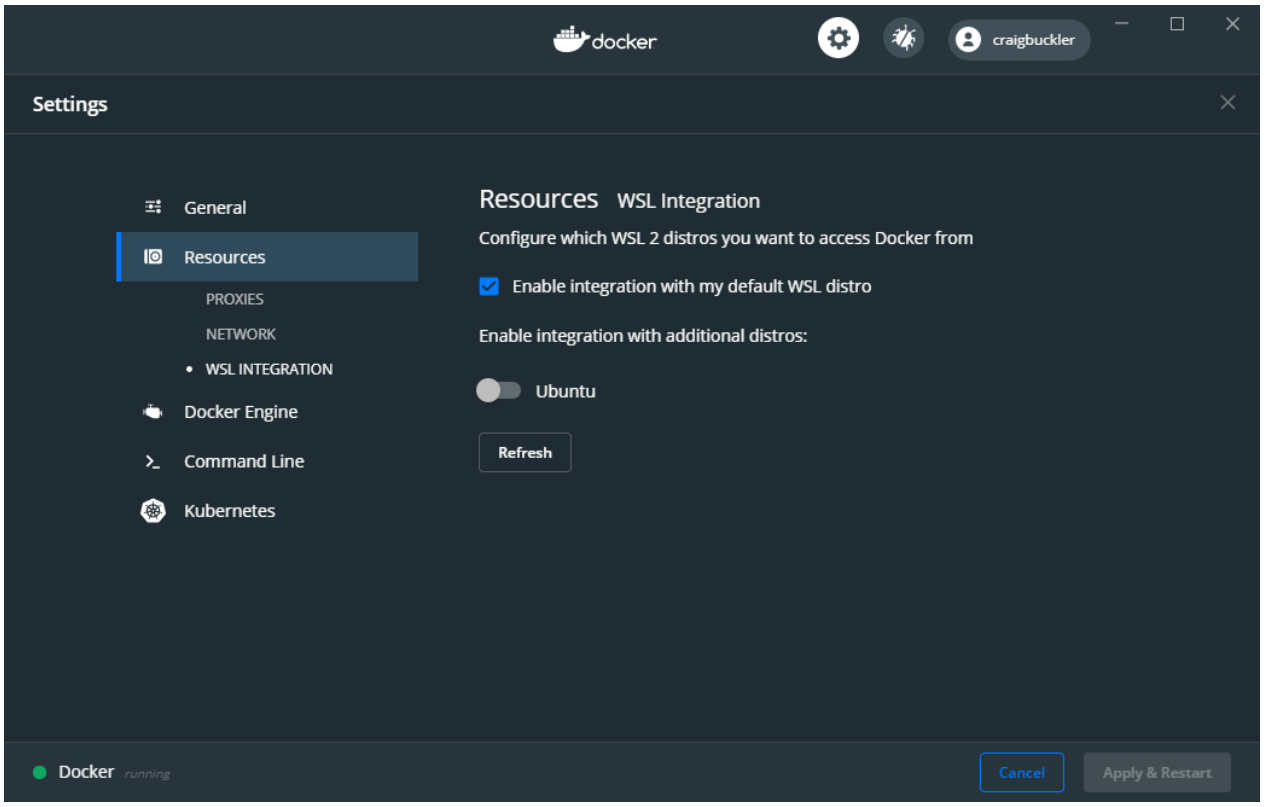



**Figure 3.9:** Docker Windows WSL2 selection

**When using Hyper-V**, Docker must be granted access to the Windows file system. Select the drives it is permitted to use by accessing the **File Sharing** panel in the **Resources** section of the Docker **Settings**:

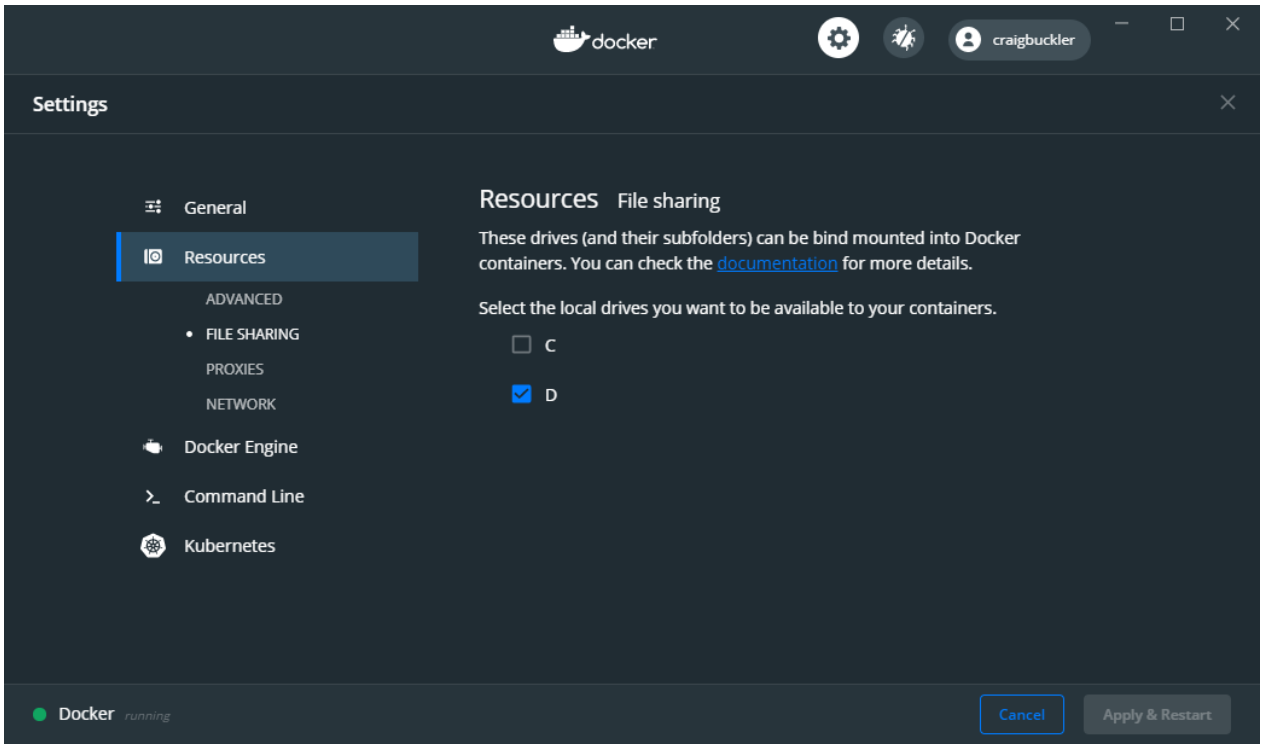



**Figure 3.10:** Docker file sharing in Windows

*(This option was named **Shared Drives** in previous editions of Docker Desktop.)*

## 3.4 Test your Docker installation

Check Docker has successfully installed by entering the following command in your terminal:

```
docker version
```

A response similar to the following is displayed:

```
Client: Docker Engine - Community
 Version:           19.03.12
 API version:       1.40
 Go version:        go1.13.10
 Git commit:        abcdef0
 Built:             Mon Jun 22 15:45:36 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.12
  API version:      1.40 (minimum version 1.12)
  ...etc...
```

Ensure Docker Compose is working by entering:

```
docker-compose version
```

To receive something like:

```
docker-compose version 1.27.2, build 8d51620a
docker-py version: 4.3.1
CPython version: 3.7.7
OpenSSL version: OpenSSL 1.1.1c  10 Sep 2019
```

Optionally, try entering:

```
docker run hello-world
```

to verify Docker can pull an image from Docker Hub and start containers as expected…

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows your installation appears to be working correctly.
```

## 3.5 Key points

What you've learned in this chapter:

1. How to install and configure Docker on your Linux, macOS, or Windows system.
2. How to install Docker Compose.
3. How to test the Docker installation.

The following chapters demonstrate how to use Docker during development. The first shows how to run the MySQL database, but it could be *any* software dependency you could need.

# 4 Launch a MySQL database with Docker

In this chapter you will discover how to launch a MySQL container using both the Docker CLI and Docker compose.

MySQL is a popular SQL database if you're developing an application which requires data storage. It doesn't matter if you've never used MySQL or a database before – the tutorial will help you become familiar with launching Docker containers. The concepts can be applied to any dependencies your application requires.

> The files created in this chapter are contained in the `mysql` directory of the example code repository provided at https://github.com/craigbuckler/docker-web

## 4.1 Locate a suitable MySQL image on Docker Hub

Access Docker Hub, and enter "`MySQL`" in the search box to locate the page with the official images:

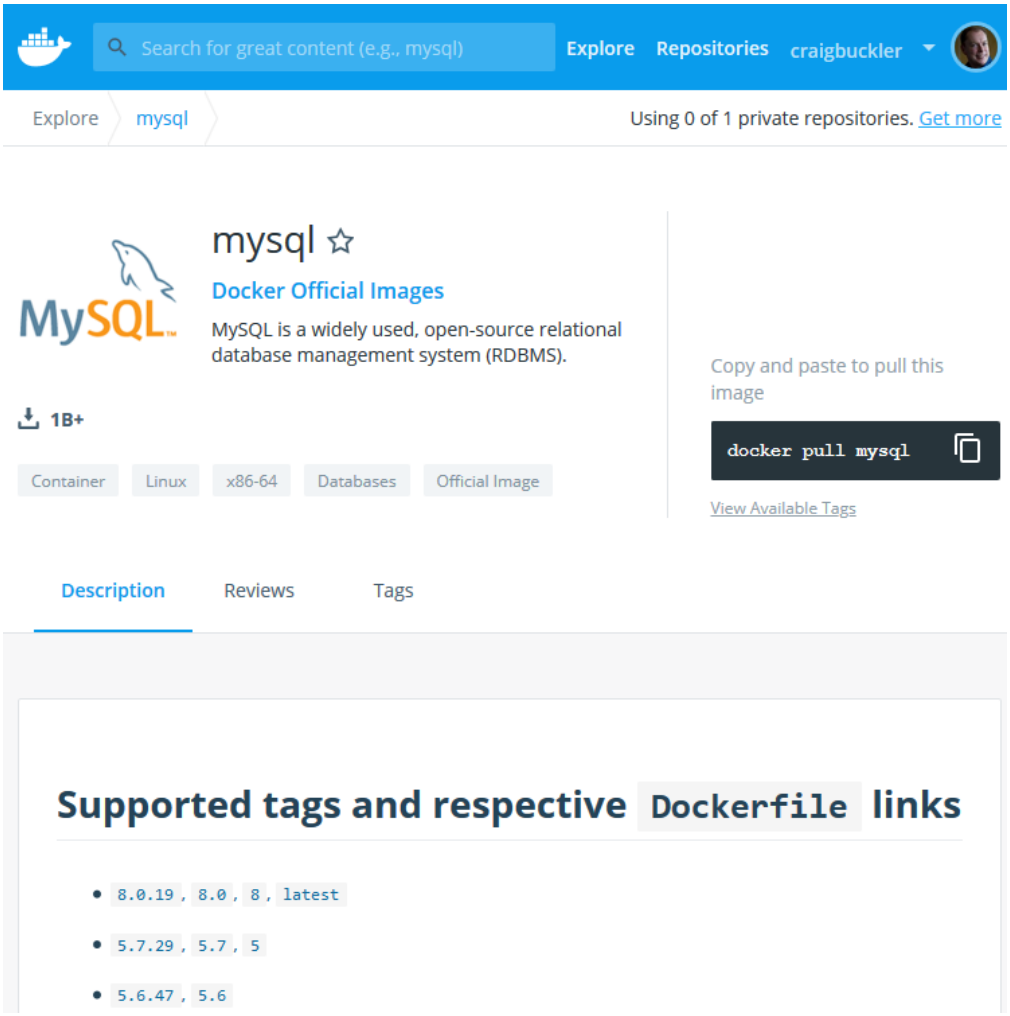



**Figure 4.1:** Docker Hub MySQL images

> Despite the prompts, you can use Docker Hub without signing in. You only require an account to push your own application images.

Docker repositories offer one or more variations of an image each with its own *tag*. Tags often match the application's MAJOR.MINOR.PATCH number if releases follow semantic versioning concepts.

For MySQL, you can choose from various releases such as 5.6, 5.7, or 8.0:

- If you select an exact version, such as `8.0.19`, that edition of MySQL will always be installed.
- Selecting the major and minor version, such as `8.0`, will install the latest release of that minor edition. That could be `8.0.19` now, but become `8.0.20` tomorrow.
- Selecting just the major version, such as `8`, will install the latest major release. That could be `8.0.19` now, but could become `8.1.0` next week.
- Selecting `latest` (or not specifying a tag) will install the latest release regardless of version. That could be `8.0.19` now, but could be version `9.0.0` or higher next month.

For this example, `latest` is a good choice but you would normally use an exact edition so all developers and the production server are using the same dependency.

> Docker makes it easy to install and test database upgrades at any time.

## 4.2 Launch a MySQL container

Launch a MySQL container by entering the following command in your terminal (**Windows users:** please remove the back-slashes and line breaks):

```
docker run \
  -it --rm --name mysql \
  -p 3306:3306 \
  --mount "src=mysqldata,target=/var/lib/mysql" \
  -e MYSQL_ROOT_PASSWORD=mysecret \
  mysql
```

All Docker CLI commands start with `docker` and an instruction such as `run` followed by options.

`docker run` creates a container from a specified image (`mysql` on the last line) and starts it. That image is downloaded if it's not already available on the host.

It can take several minutes to download the image, launch a container, and initialize MySQL the first time the command is run. Subsequent launches will be almost instantaneous. The database will be ready to use when you see:

```
[System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections.
```

`docker run` offers numerous options (refer to Appendix A) but the main ones you will use are:

| option | description |
|---|---|
| `-d` | run a container as a background process (which exits when the application ends) |
| `-it` | keep a container running in the foreground (even after the application ends), and show an activity log |
| `--rm` | remove the container after it stops |
| `--name` | name a container (a random GUID is used otherwise) |

| option | description |
|---|---|
| `-p` | map a host port to a container port |
| `--mount` | create a persistent Docker-managed volume to retain data. The string specifies a `src` volume name and a `target` where it is mounted in the container's file system |
| `-v` | mount a host folder using the notation `<hostdir>:<containerdir>` |
| `-e` | define an environment variable |
| `--env-file` | read environment variables from a file where each line defines a `VAR=value` |
| `--net` | connect to specific Docker network |
| `--entrypoint` | override the default starting application |

> If you do not specify `--rm`, the container will remain available even once it has stopped. It's possible to restart it, but there's rarely any benefit – it's simpler to execute the same `docker run` command again.

### 4.2.1 MySQL credentials

MySQL stores user access credentials such as the root user's password in an internal database. This is created when MySQL initializes storage space on its first launch.

The `docker run` command above stores all database data in a Docker volume named `mysqldata`. This retains data between restarts – *including the **first** root password you set*.

If you alter the `run` command to use a different password on a subsequent launch (e.g. `-e MYSQL_ROOT_PASSWORD=NewSecret`), it will be ignored. To change the root password, you can either issue a MySQL ALTER USER command using a MySQL client or – *more drastically* – delete the Docker volume and initialize the database again.

It's good practice to create a MySQL user which is granted limited rights to a specific database, i.e. it can read data but not alter table structures or examine other databases. Application configuration often achieved by setting environment variables (see the MySQL image documentation).

Your application *could* launch its own MySQL container, so using a `root` user with all privileges seems less dangerous. However, a locked-down user will always be more secure and prevent malicious or accidental damage.

The examples shown in this chapter use `root` for brevity, but you should use better credentials during development and deployment.

## 4.3 Connect to the database using a MySQL client

Once the database container has started, you can use any MySQL client application installed on your host PC to connect to `localhost:3306` with the user ID `root` and password `mysecret`.

If you don't have a MySQL client to hand, Adminer is a lightweight PHP-based option. It is also available as a Docker image and can be launched in another terminal with:

```
docker run \
  -it --rm --name adminer \
  -p 8080:8080 \
  adminer
```

After it's started, open http://localhost:8080/ in your browser and enter your MySQL login credentials:

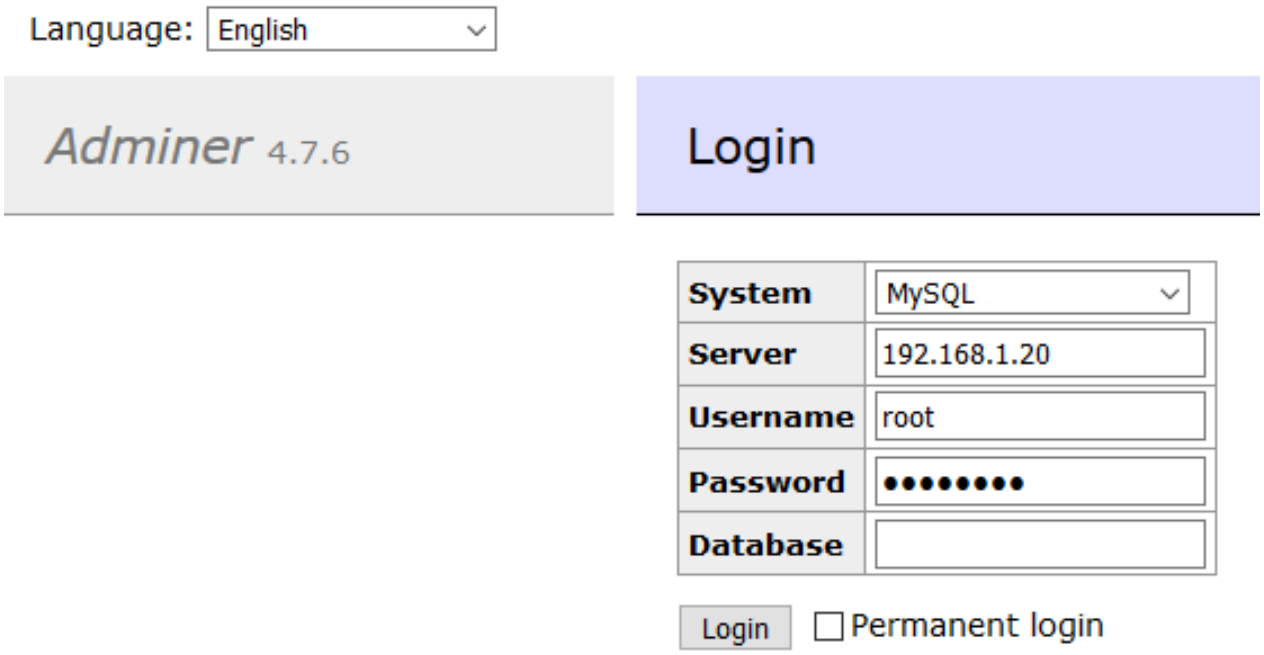



**Figure 4.2:** Adminer login screen

Note that you **cannot** use `localhost` as the server name since Adminer will resolve that to its own container! Instead, you can:

1. Enter `host.docker.internal`.

   Docker Desktop routes this domain to your PC's network IP address, but it may not be available on all systems.

2. Use your actual network IP address (`192.168.1.20` is shown in the screen above)

   This can be obtained with the `ifconfig` command on macOS and Linux or `ipconfig` on Windows.

3. Or use the container's IP address assigned by Docker.

   Docker creates it's own virtual network. `docker inspect mysql` returns information about the container in JSON format. You can locate the `"IPAddress"` value using the `-f` format option:

```
docker inspect \
  -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
  mysql
```

> Alternatively, you can define a Docker network (see below). Any container attached to that network can reference another container by its defined `--name` and have it resolve correctly, i.e. you can enter `mysql` as the server name on the Adminer login screen.

## 4.4 Connect to a container shell

Every Docker container runs an isolated Linux environment. You can connect to its shell and run commands, examine logs, or perform any other activities.

> Remember containers are stateless! Any changes you make will be lost whenever the container is restarted.

Presuming your MySQL container is still running, open another terminal and enter:

```
docker exec -it mysql bash
```

> Some lightweight images using Alpine Linux do not offer the `bash` shell. If the command fails, try using `docker exec -it mysql sh` instead.

For example, you could access the MySQL command line and list databases:

```
> mysql -u root -pmysecret

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.
All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help.
Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

Enter an `exit` command to quit the shell.

## 4.5 View, stop, and restart containers

Containers running in interactive mode (started with the `-it` switch) can normally be stopped by pressing `Ctrl` | `Cmd` + `C`.

> By *"normally"*, I mean *"it won't work some of the time"*. Applications will not always shut down gracefully or signal they have terminated. If an interactive container fails to stop, you'll need to execute the `stop` command described below.

A list of running containers can be viewed by entering:

```
docker container ls
```

or the shorter:

```
docker ps
```

*(Run this in another terminal tab/window if your containers are running in interactive mode.)*

A list of active containers is displayed. This can be quite wide – a truncated version is shown here:

```
CONTAINER ID  IMAGE    STATUS      PORTS           NAMES
ef3bab04fc8f  adminer  Up 16 mins  8080->8080/tcp  adminer
793003e459e6  mysql    Up 17 mins  3306->3306/tcp  mysql
```

Each container is assigned a hexadecimal ID which can be used as a reference in Docker commands. However, specifying a container `--name` makes management considerably easier.

Containers can be restarted by providing a list of one or more names to `docker container restart`. This could be useful if you want to erase browser sessions or temporary data:

```
docker container restart adminer mysql
```

You can also `pause` and `unpause` running containers if, for example, you wanted to test how an application reacted to a database failure:

```
docker container pause mysql
docker container unpause mysql
```

Similarly, containers can be stopped with `docker container stop`:

```
docker container stop adminer mysql
```

Containers started with the `-rm` option are removed when they are stopped. If you do not use that option (or experience Docker issues), you can list all containers with:

```
docker ps -a
```

Individual containers can be removed using their name or ID, e.g.

```
docker container rm mysql
```

Alternatively, remove all stopped containers with:

```
docker container prune
```

## 4.6 Define a Docker network

When a container is started, it's assigned an IP address on the default Docker network. However, this can change on subsequent restarts so it is difficult to communicate between containers using an IP address.

`docker run` has `--ip` and `--ip6` options so you can define a fixed IP, but it is generally easier to refer to another container using it's `--name`. That can be achieved by creating your own Docker network.

Stop any running containers then create a new network, e.g. named `mysqlnet` here:

```
docker network create --driver bridge mysqlnet
```

Any container can connect to this network using the `--net` option when it is launched. MySQL example:

```
docker run \
  -d --rm --name mysql \
  -p 3306:3306 \
  --mount "src=mysqldata,target=/var/lib/mysql" \
  -e MYSQL_ROOT_PASSWORD=mysecret \
  --net mysqlnet \
  mysql
```

and Adminer:

```
docker run \
  -d --rm --name adminer \
  -p 8080:8080 \
  --net mysqlnet \
  adminer
```

Each container's name now resolves on the Docker `mysqlnet` network. You can therefore enter `mysql` as the server name on the Adminer login screen.

> You could hard-code the `mysql` name and access credentials into your application. However, it's more practical to define these using environment variables.

## 4.7 Cleaning up

Docker can use significant quantities of disk space! Scary usage statistics are returned when entering:

```
docker system df
```

To view all containers, both running and stopped, enter:

```
docker container ls -a
```

Note that containers are usually small because they are stateless and launch from a specific image.

To view all images, both active and dangling (those not associated with a container), enter:

```
docker image ls -a
```

To view all Docker-managed disk volumes, enter:

```
docker volume ls
```

To view all Docker networks, enter:

```
docker network ls
```

With just MySQL and Adminer, you'll have used almost 1GB of space. This will rise as you start using further dependencies and creating your own images.

Stop containers by specifying their name in a `stop` command:

```
docker container stop adminer mysql
```

All stopped containers, unused networks, and dangling images can be removed with:

```
docker system prune
```

This is safe and probably a *good idea* to run every so often.

The following command will do the same and also wipe any image not associated with a running container:

```
docker system prune -a
```

The latest `mysql` and `adminer` images will therefore have to be downloaded again if you require them.

### 4.7.1 Deleting disk volumes

Removing a Docker volume will wipe it's data forever! *There is no going back.*

If you're developing a database-driven application, it's usually practical to retain one or more data dumps which can be used to re-create the database in various states. Most MySQL client tools provide a dump or export facility, such as the **Export** link in Adminer.

Alternatively, the `mysqldump` utility provided with MySQL can be run by attaching to the container shell or using the `docker exec` command.

To back up a database named `mydb` to a file named `backup.sql` using the `root` MySQL credentials on Linux or macOS, enter:

```
docker exec mysql /usr/bin/mysqldump -u root -pmysecret mydb \
  > backup.sql
```

or on Windows PowerShell:

```
docker exec mysql /usr/bin/mysqldump -u root -pmysecret -r mydb | \
  Set-Content backup.sql
```

Assuming you're happy to proceed, you can view Docker volumes with:

```
docker volume ls
```

then delete any by ID or name:

```
docker volume rm <name>
```

Unused Docker volumes – *those not currently attached to a running container* – can be removed with:

```
docker volume prune
```

Alternatively, use `docker volume prune -a` will delete them all. *You only have yourself to blame!…*

### 4.7.2 Full clean start

Every unused container, image, volume, and network can be wiped with:

```
docker system prune -a --volumes
```

Add `-f` if you want to force the wipe without a confirmation prompt.

## 4.8 Launch multiple containers with Docker Compose

Starting Docker containers individually is not fun – especially when commands are long and difficult to remember. Fortunately, Docker Compose provides a way to build and launch containers, networks, and volumes from a single configuration file named `docker-compose.yml`.

YAML is a cheeky mnemonic for *YAML Ain't Markup Language*: a common, compact data format often used for configuration purposes. It uses new lines and tab stops rather than the quotes and brackets favored by JSON.

Docker Compose offers many configuration options (refer to Appendix C), but an example is the best way to illustrate common settings. Create the following `docker-compose.yml` anywhere on your system:

```
version: '3'
services:

  mysql:
    image: mysql
    container_name: mysql
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
    volumes:
      - mysqldata:/var/lib/mysql
    ports:
      - "3306:3306"
    networks:
      - mysqlnet
    restart: on-failure

  adminer:
    image: adminer
    container_name: adminer
    depends_on:
      - mysql
    ports:
      - "8080:8080"
    networks:
      - mysqlnet
    restart: on-failure

volumes:
  mysqldata:

networks:
  mysqlnet:
```

This creates identical containers to the `docker run` commands you used above with the `volumes` and `networks` listed at the bottom. It also defines two new settings:

- `depends_on` start-up dependencies: `adminer` will start after `mysql` has launched
- `restart on-failure`: the container is automatically restarted if the application stops with an exit code.

If you've not stopped your existing containers, do so now with:

```
docker container stop adminer mysql
docker system prune
```

Now launch Docker Compose from the same directory as your `docker-compose.yml` file using:

```
docker-compose up
```

The `mysql` and `adminer` images will be downloaded if necessary and the two containers are started. By default, `docker-compose` runs as a foreground task and shows a log of all container activity. MySQL is ready to use when you see:

```
mysql | ... [Server] X Plugin ready for connections.
```

You can now start Adminer at http://localhost:8080/ and connect to the `mysql` server using the user ID `root` and password `mysecret`.

The `mysql` and `adminer` network names resolve within the Docker network. If you want to connect to MySQL using a client installed on your host OS, enter `localhost:3306` as the server/port.

If you specify a single port value in `docker-compose.yml`, it is only exposed within the Docker network, e.g.

```
    ports:
      - "3306"
```

In this case, MySQL would *not* be accessible to the host, although Adminer could still communicate with `mysql:3306` on the internal network.

You can use the services and connect to their shells in the same way as before. The containers will continue to run until:

1. you press `Ctrl`|`Cmd + C` in the terminal where you launched `docker-compose`, or
2. you enter `docker-compose down` in another terminal in the same directory as your `docker-compose.yml` file.

From this point forward, `docker-compose` will be used where possible. It's rarely necessary to enter individual `docker` commands, but it's useful to know they're available since Docker Compose may not be available on every system.

### 4.8.1 Other Docker Compose options

Docker Compose configuration files specify a version at the top:

```
version: '3'
```

This specifies a level of compatibility. At the time of writing, `3.8` is the latest version, but you would only need to specify that if you used options unavailable in previous releases.

The `-f` option allows you to specify an alternative configuration filename, e.g.

```
docker-compose -f ./my-config.yml up
```

Docker Compose can be run as a background service using `-d`. This may be practical on a live server where foreground logging is not required:

```
docker-compose up -d
```

The active containers can then be viewed using `docker ps` or:

```
docker-compose ps
```

and stopped using:

```
docker-compose down
```

> Note that `docker-compose stop` stops containers without removing them. They can then be restarted with `docker-compose start`.

Refer to Appendix C for further options.

## 4.9 Key points

What you've learned in this chapter:

1. Finding application images on Docker Hub.
2. Launching MySQL and Adminer containers.
3. Mounting Docker volumes to store persistent data.
4. Connecting to a container's shell to issue commands.
5. Stopping running containers.
6. Defining a Docker network for easier name resolution.
7. Cleaning Docker files.
8. Launching multiple containers with Docker Compose.

For bonus points, try launching a PostgreSQL or MongoDB database as a container.

Running MySQL as a containerized service is a simple example of the possibilities offered by Docker. The first launch may have taken a few minutes, but how long would it have taken you to download, install, and configure MySQL on your host OS? Could you have retained that version, upgraded, or even run multiple editions on the same PC at the same time?

In the next chapter, you'll use Docker to install and develop a WordPress-powered website. It introduces the concept of Docker *bind mounts* – a way to update source files on your host PC which are executed in a running container.

# 5 WordPress development with Docker

WordPress is a PHP and MySQL-based open-source Content Management System first launched in 2003. It powers 40% of all websites, is supported by many web hosts, is easy to set-up, and offers thousands of free themes and plugins.

If you're a web developer, you've probably encountered WordPress at some point in your career. Even modern static sites can adopt WordPress as a headless CMS where it is used to edit content that is fed into a site generator.

Even if you don't use *(or don't want to use)* WordPress, the concepts demonstrated in this chapter show how to execute an application in a Docker container but continue to edit code on your local PC. This is essential for development but the technique is rarely discussed in Docker tutorials.

> The files created in this chapter are contained in the `wordpress` directory of the example code repository provided at https://github.com/craigbuckler/docker-web

## 5.1 WordPress requirements

To develop a WordPress-powered site, you must install:

1. a web server, typically Apache
2. the PHP runtime and extensions, then configure the web server accordingly
3. MySQL or MariaDB then define a new database for WordPress use
4. WordPress itself, plus any required themes and plugins.

Tools such as XAMPP can ease some of that effort but, with Docker, you'll be running WordPress and developing code within minutes.

> Windows users will also discover WordPress runs significantly faster on a Docker-powered Linux-based file system than a native NTFS drive.

## 5.2 Docker configuration plan

You will be pulling two Docker images:

1. wordpress

   The latest image is a good choice. It provides the stable versions of Debian Linux, Apache, PHP, and WordPress. There are more lightweight `alpine` images, although these do not match production hosting environments and may not work as expected.

2. mysql:5

   At the time of writing, WordPress is not compatible with the new authentication methods the latest releases of MySQL. You can resolve those issue, but it's easier to use version 5.

The containers launched from these images will be added to a `wpnet` Docker network.

### 5.2.1 Docker volumes

Two Docker volumes will be created:

1. `wpdata` for the MySQL database (mounted to `/var/lib/mysql` in the MySQL container), and
2. `wpfiles` for WordPress application files (mounted to the Apache server root directory at `/var/www/html` in the WordPress container).

> Some developers will claim that mounting the `wpfiles` volume is an anti-pattern because the WordPress container is no longer stateless. This is true, but WordPress is not a stateless application. Docker is being used to emulate a live server environment for development purposes. You are unlikely to use the same configuration in production.

Mounting the `wpfiles` volume has a number of benefits:

1. Docker start-up is faster.

   There's no need to copy the core WordPress files every time the container is launched.

2. The WordPress application can be updated automatically.

   This would happen on live installations, so replicating it during development is useful.

> If you choose not to mount a `wpfiles` volume, you can run `docker pull wordpress` every so often to download the latest application image.

It should not be necessary to inspect or modify files stored on either volume. However, the MySQL data can be accessed at `localhost:3306` from any MySQL client installed on your PC.

### 5.2.2 Development directory bind mount

A `wp-content` sub-directory will be created in the project directory on your host PC. This contains all plugins, themes, and uploaded assets and mounts to `/var/www/html/wp-content` in the WordPress container.

WordPress developers should only ever create and modify files in the `wp-content` directory.

> There's nothing to stop you editing core WordPress files, but the changes would disappear the moment the application is updated. Don't bother trying!

As a bonus, this makes it easier to commit `wp-content` files to Git/other repositories – it is the only WordPress folder on your host PC.

### 5.2.3 `localhost` domain alternative

WordPress will be launched from the `localhost` domain on port `8001`. However, that can be impractical when you're developing several sites:

- your browser may cache files from one site and show them on another.
- WordPress stores the domain in its database, so you might want to use something similar to the production name.

Optionally, you can configure other domains for development use in your `hosts` file. This is located at:

- `/etc/hosts` on Linux and macOS, and
- `C:\Windows\System32\drivers\etc\hosts` on Windows (administrator permissions are required to view and edit it).

Add a line to the bottom of the `hosts` file, e.g.

```
127.0.0.1    dev.wordpress
```

and save it. The changes are applied immediately on Linux and macOS, but Windows users must run `nbtstat -R` or reboot.

From then on, you can use http://dev.wordpress rather than http://localhost – both resolve to the loopback address `127.0.0.1`.

## 5.3 Docker Compose configuration

Create a new WordPress project directory – `wordpress` is a good choice:

```
mkdir wordpress
cd wordpress
```

You'll be using Docker Compose, so create a `docker-compose.yml` file in the root of the directory and add:

```
version: '3'
services:

  mysql:
    image: mysql:5
    container_name: mysql
    environment:
      - MYSQL_DATABASE=wpdb
      - MYSQL_USER=wpuser
      - MYSQL_PASSWORD=wpsecret
      - MYSQL_ROOT_PASSWORD=mysecret
    volumes:
      - wpdata:/var/lib/mysql
    ports:
      - "3306:3306"
    networks:
      - wpnet
    restart: on-failure

  wordpress:
    image: wordpress
    container_name: wordpress
    depends_on:
      - mysql
    environment:
      - WORDPRESS_DB_HOST=mysql
      - WORDPRESS_DB_NAME=wpdb
      - WORDPRESS_DB_USER=wpuser
      - WORDPRESS_DB_PASSWORD=wpsecret
    volumes:
      - wpfiles:/var/www/html
      - ./wp-content:/var/www/html/wp-content
    ports:
      - "8001:80"
    networks:
      - wpnet
    restart: on-failure

volumes:
  wpdata:
  wpfiles:
networks:
  wpnet:
```

The following sections describe this configuration in detail.

### 5.3.1 Environment variables

The MySQL environment variables `MYSQL_DATABASE`, `MYSQL_USER`, and `MYSQL_PASSWORD` define a new database named `wpdb` which can be accessed by the user `wpuser` using the password `wpsecret`.

These database access values are then set in the WordPress environment variables `WORDPRESS_DB_NAME`, `WORDPRESS_DB_USER`, and `WORDPRESS_DB_PASSWORD`.

A password is also defined for the MySQL `root` user in `MYSQL_ROOT_PASSWORD`. You're unlikely to need the `root` user, but it could be useful to back-up data or perform other database maintenance.

### 5.3.2 Exposed ports

MySQL's port `3306` is exposed to the host PC so it's possible to connect and inspect a database using any MySQL client.

Alternatively, you could set:

```
    ports:
      - "3306"
```

This would make the port available to other containers within the Docker network but **not** expose it to the host. The WordPress container would be unaffected, but MySQL clients on the host would be unable to connect to the database.

Finally, the Apache's port `80` is exposed as `8001` on the host.

> Linux and macOS systems do not normally allow applications to use ports under `1000` unless they are running with root user privileges. Windows users may also find port 80 is hogged by the Skype app.

### 5.3.3 WordPress volume and bind mount

The `wordpress` container defines:

1. a `wpfiles` Docker volume mounted to the Apache server root directory at `/var/www/html`, and
2. a `wp-content` sub-directory in the host's project directory mounted to `/var/www/html/wp-content`.

> If you were using `docker run` rather than Docker Compose, the equivalent options are:
>
> ```
> --mount "src=wpfiles,target=/var/www/html"
> -v $PWD/wp-content:/var/www/html/wp-content
> ```
>
> `$PWD` references the current directory on Linux and macOS. It is not available on Windows so the full path must be specified using forward-slash notation, e.g.
>
> ```
> -v /c/projects/wordpress/wp-content:/var/www/html/wp-content
> ```
>
> This is not necessary in Docker Compose configurations which support relative file references.

## 5.4 Launch your WordPress environment

Ready? Open a terminal, access your project directory, and enter:

```
docker-compose up
```

The process can take several minutes on the first run since Docker must download the images, initialize the database, and copy application files.

Various errors may be logged, such as:

```
wordpress | MySQL Connection Error: (2002) Connection refused
```

but don't worry about them. Eventually, MySQL will be ready:

```
mysql     | [Note] mysqld: ready for connections.
```

Check that a `wp-content` directory has been created in your project directory. It should contain `plugins` and `themes` sub-directories as well as an `index.php` file.

## 5.5 Install WordPress

Open http://localhost:8001/ (or the `hosts` domain you created) in your browser. The first time you do this, the WordPress installation screen appears:

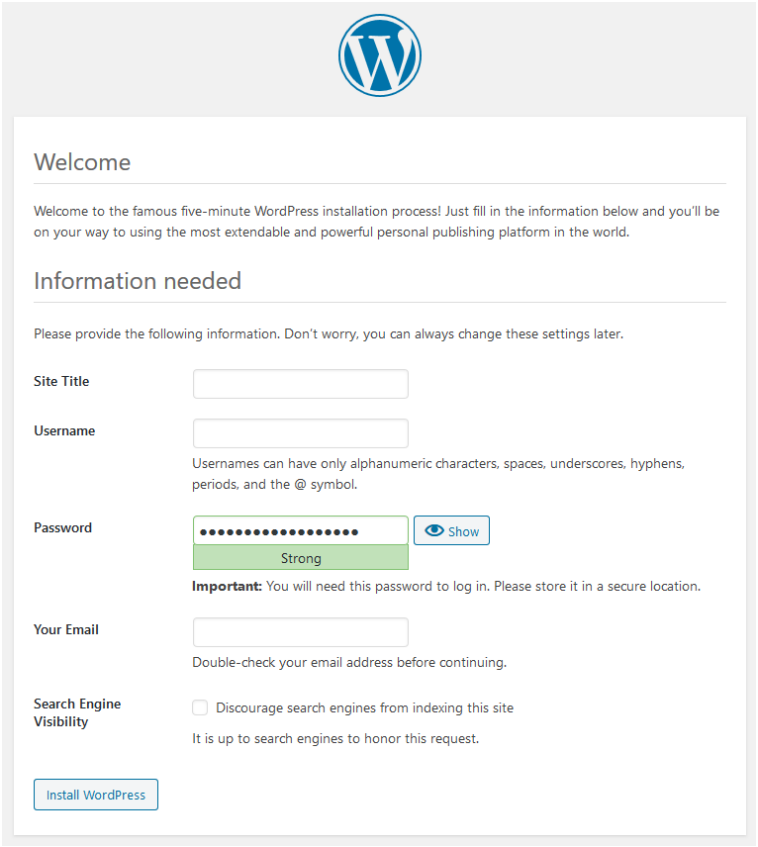



**Figure 5.1:** WordPress installation screen

Complete the fields and click **Install WordPress** to configure the database.

> Check *Discourage search engines from indexing the site*. Search engines won't be able to access your local installation, so there's little point pinging them every time a page or post is added.

You will now be prompted to log on (at http://localhost:8001/wp-admin). Enter the username and password you specified to access the administration dashboard:

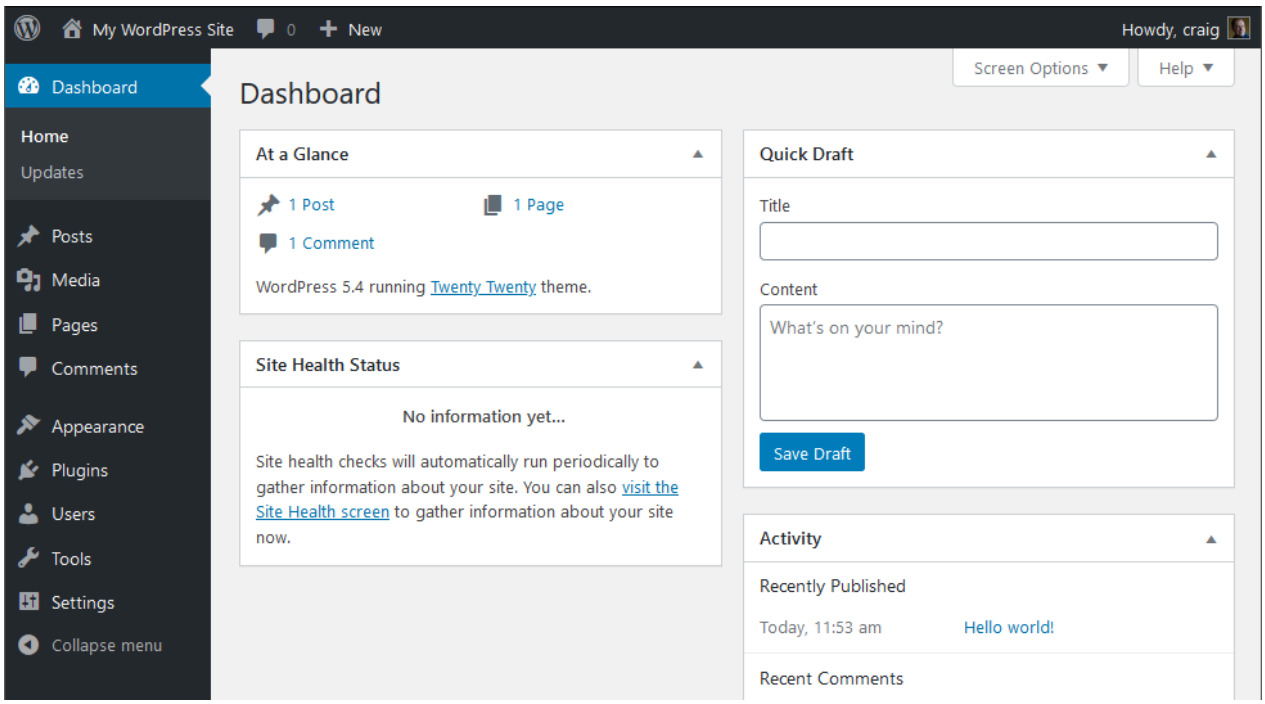



**Figure 5.2:** WordPress dashboard

The WordPress-controlled website can be viewed at http://localhost:8001/ *(your site may look different depending on the WordPress version, theme, and settings)*:

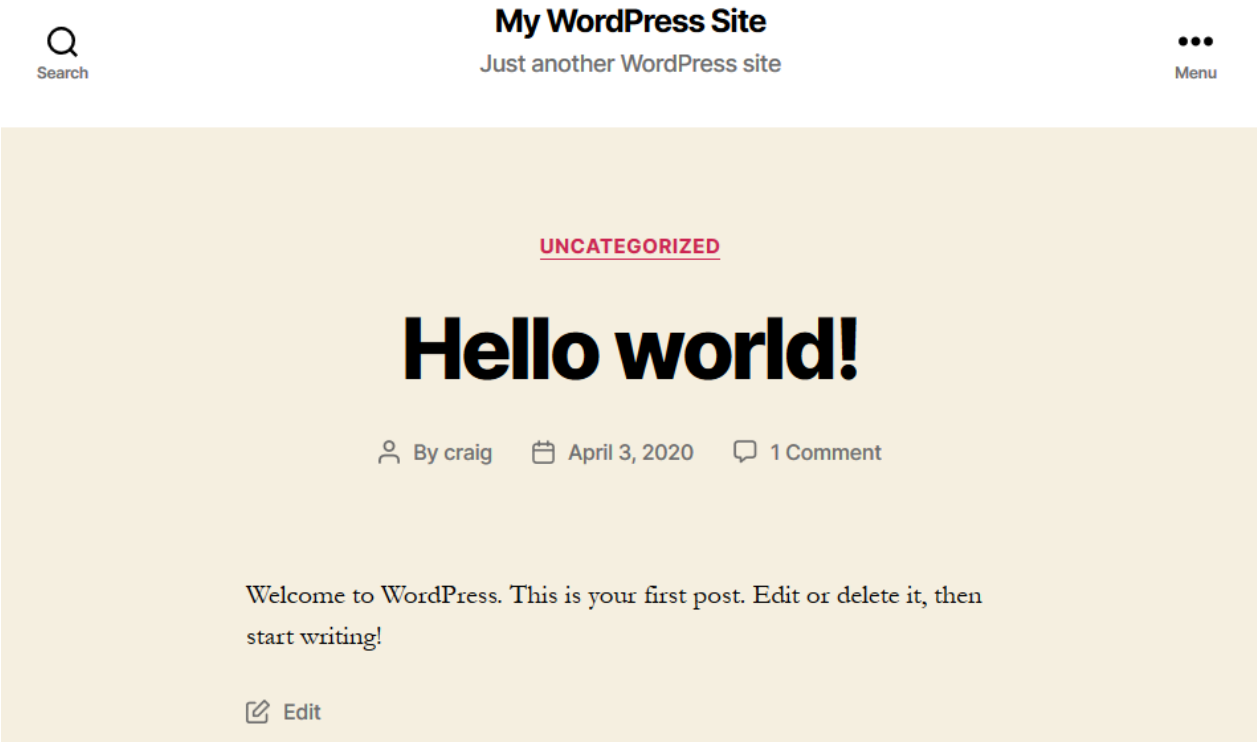



**Figure 5.3:** WordPress website

## 5.6 Local WordPress Development

You can now add, edit, or remove themes and plugins in the `wp-content` directory on your host PC.

### 5.6.1 Administration panel permissions

WordPress allows you to install, edit, and delete theme and plugin files directly from its administration panels. This is permitted on Windows, but Linux and mac OS users must grant access rights to the project's `wp-content` folder by entering the following command on the host:

```
chmod 777 ./wp-content -R
```

If you do **not** grant access rights:

- WordPress and users testing your site will not be able to change code in the `wp-content` directory (that could be a *good thing*).
- You must manually download plugins and themes from WordPress.org or elsewhere. Extract the ZIP file to an appropriate `wp-content` sub-directory before enabling it in the administration panels.
- Some plugins, such as caching systems, will not operate.

### 5.6.2 Develop a new theme

To start developing a new WordPress theme, create a directory in `wp-content/themes` on your PC, e.g. `docker-basic`. Add a `style.css` file with the following code:

```
/*
Theme Name: Docker Basic Theme
Version: 1.0.0
License: MIT
*/
```

Now add an `index.php` file with the following code:

```
<!DOCTYPE html>
<html <?php language_attributes(); ?>>
<head>
<title><?php bloginfo('name'); ?></title>
<link rel="stylesheet" href="<?php bloginfo('stylesheet_url'); ?>">
<?php wp_head(); ?>
</head>
<body>

  <header>
    <h1><?php bloginfo('name'); ?></h1>
    <p><?php bloginfo('description'); ?></p>
  </header>

  <main>
    <?php
      if ( have_posts() ) : while ( have_posts() ): the_post(); ?>

      <article id="post-<?php the_ID(); ?>">
        <h2><?php the_title(); ?></h2>
        <?php the_excerpt(); ?>
      </article>

    <?php
      endwhile;
    endif;
    ?>
  </main>

  <?php wp_footer(); ?>
</body>
</html>
```

Activate this theme in the **Appearance** > **Themes** screen of the WordPress administration panels.

Open or refresh the http://localhost:8001/ home page to see your new theme in its award-winning glory:


**Figure 5.4:** new WordPress theme

This is a minimal example, but you can add content and edit the WordPress theme using any editors and tools installed on your host PC.

To finish for the day, stop Docker by running `docker-compose down` in the project directory or pressing `Ctrl` | `Cmd + C` in the terminal. Subsequent restarts of `docker-compose up` will take a few seconds.

## 5.7 Key points

What you've learned in this chapter:

1. Running WordPress and its MySQL database in Docker containers.
2. Mounting host OS directories into a container.
3. Local development with changes instantly applied to running containers.

For bonus points, try adding Adminer to your environment. Perhaps attempt to create Dockerized versions of the Drupal CMS or PrestaShop ecommerce system.

You've now used several pre-built Docker images. In the next chapter, you'll create an image containing a simple Node.js application. This introduces the concept of Dockerfiles and the Docker image building process.

# 6 Application development with Docker

Until now, you've used pre-built Docker images such as MySQL, Adminer, and WordPress. They're useful but you'll eventually want to run your own programs in a container.

In this chapter, you'll be creating a Node.js *"Hello World"* application which is built into a Docker image and launched as a container. By default, the image will be ready for deployment on a production server, but Docker Compose will be used to override some settings to create a development and debugging environment. You'll be able to edit source code on your host PC but the files will be executed within a continually-running container. This has several benefits:

- Docker will manage dependencies for you – there's no need to install and maintain language runtimes
- the process is little different to developing locally – you can use whatever editor and tools you prefer
- the container is isolated – your application cannot do anything nasty such as crash the host PC or wipe files
- you can distribute your application to other developers or testers – it will run identically on any other device with zero configuration.

> The files created in this chapter are contained in the `nodejs` directory of the example code repository provided at https://github.com/craigbuckler/docker-web

## 6.1 Container-based application development

Docker simplifies web development: **any** web application you create can be run in a single container.

**BUT...** if you want deploy similar containers to live production servers, the application should be stateless. Any number of instances can be started and any can react to requests. In practical terms, your application should *not* store essential state data in local files or memory.

> Example: an application stores login credentials in memory when a user logs in. A single container is used during development so everything works as expected.
>
> The application is then deployed to a production server and run in two containers which receives requests via a load balancer. A user accesses the system and has their login processed by container1. Their next request is served by container2: it does not have the user's state and redirects to the login page.

This may change the way you approach application development. Isolated containers should retain data in central repositories such as a database.

My advice: stateless web applications are a *good thing*. They permit horizontal scaling – you can add more machines/containers as usage demands increase. It's easy to make that decision at the start of a project, but converting a legacy stateful application may not be viable.

> Applications written in PHP can be easier to containerize because HTTP requests are stateless by default. It may be more difficult to convert a monolithic Node.js or Python application which was designed to run on a single production server.

None of this matters during development because you'll usually run your application in a single container. You don't have to use containers in production if it's not practical.

## 6.2 What is Node.js?

Node.js is a popular, high-performance, JavaScript runtime built with the Chrome browser's V8 JavaScript engine. It's typically used for server-side web development, but has also been adopted for client-side build tools, desktop applications, embedded systems, and more.

After installing Node.js, you can execute a JavaScript file using:

```
node ./index.js
```

where `index.js` is a single entry script. It can be named anything but many projects use that name.

Until now, you've been using Docker images provided at Docker Hub. This chapter illustrates how to build your own Docker image which installs and executes your application in both development and production environments.

You may have no interest in Node.js, but it's similar to other runtimes such as PHP, Python, Ruby, Go, and Rust. The same Docker concepts apply regardless of the runtime language you're using.

## 6.3 *Hello World* application overview

This project creates a *"Hello World"* application using the Express.js framework for Node.js.

> Express.js is overkill for this example, but it makes the application less verbose and easier to extend with your own code.

The http://localhost:3000/ root URL returns "`Hello World!`" as plain text:

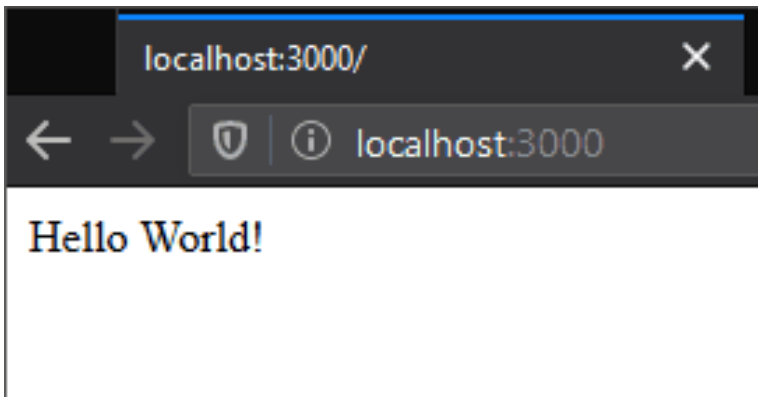



**Figure 6.1:** Node.js Hello World

Calling the same URL from a client-side Ajax request returns the JSON-encoded object:

```
{ "message": "Hello World!" }
```

> Ajax calls can be identified when an incoming request has a `X-Requested-With` HTTP header set to `XMLHttpRequest`. This is added by most Ajax libraries.

You can also add a string to the URL path, e.g. http://localhost:3000/Craig returns "`Hello Craig!`" as text or `{ "message": "Hello Craig!"}` as appropriate.

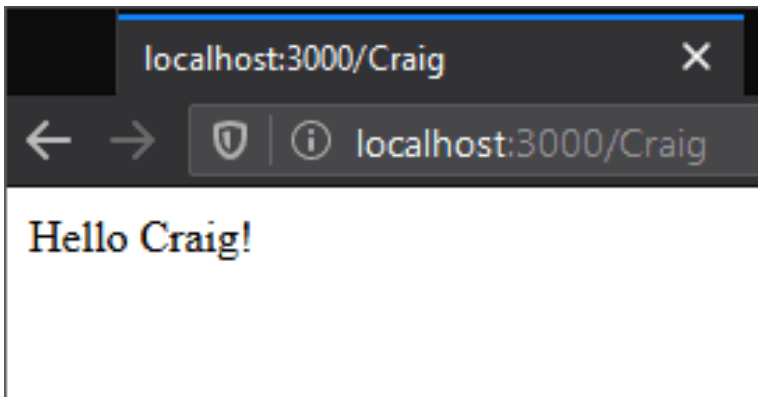



**Figure 6.2:** Node.js Hello custom string

### 6.3.1 `package.json` settings

Node.js applications require a **`package.json`** file in the project's root directory. This defines settings and required modules which are pulled using `npm` – the Node.js package manager:

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "A basic hello-world application.",
  "main": "index.js",
  "scripts": {
    "debug":
      "nodemon --trace-warnings --inspect=0.0.0.0:9229 ./index.js",
    "start": "node ./index.js"
  },
  "author": "Craig Buckler",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.2"
  }
}
```

Two launch commands are defined in the `scripts` section:

1. `start` (run with `npm start`) for production systems, and
2. `debug` (run with `npm run debug`) for development systems.

During development:

- Nodemon is used to restart the application when an application file is changed. It is installed as a `"devDependency"` when the `NODE_ENV` environment variable is set to `development`.
- `--trace-warnings` outputs stack traces when JavaScript Promises fail, and
- `--inspect` starts the V8 inspector so debuggers can be attached.

Nodemon use a `nodemon.json` configuration file which specifies which directories to watch and ignore:

```
{
  "script": "./index.js",
  "ext": "js json",
  "ignore": [
    "node_modules/"
  ],
  "legacyWatch": true,
  "delay": 200,
  "verbose": true
}
```

> Note: `legacyWatch` is best for Alpine Docker images (see below).

### 6.3.2 `index.js` application script

A single `index.js` script handles all functionality to start Express.js and define a `/` root route *(the application route taken when the / root URL path is entered!)*. It is not necessary to read or understand this code:

```
// main application
'use strict';

const
  // HTTP port from NODE_PORT environment variable
  port = process.env.NODE_PORT || 3000,

  // Express.js module
  express = require('express'),
  app = express();


// root route with optional name
app.get('/:name?', (req, res) => {

  // returned message
  const message = `Hello ${ req.params.name  || 'World' }!`;

  if (req.xhr) {

    // JSON response (HTTP header "X-Requested-With: XMLHttpRequest")
    res
      .set('Access-Control-Allow-Origin', '*')
      .json({ message });

  }
  else {

    // text response for all other requests
    res.send( message );

  }

});

// start HTTP server
app.listen(port, () =>
  console.log(`server running on port ${port}`)
);
```

### 6.3.3 Local Node.js development

To run this application on your local PC, you would need to:

1. Install Node.js.
2. Navigate (`cd`) to the project directory in your terminal.
3. Run `npm install` to download the module `"dependencies"` (and `"devDependencies"` when `NODE_ENV` is set to `development`).
4. Launch the application with `npm start` or `npm run debug`.

None of this is necessary because it will be handled by Docker. You do not need to install Node.js locally, although you may require it to run debugging tools on your host PC.

## 6.4 Docker configuration plan

A `Dockerfile` configuration file specifies the steps required to build and run your bespoke application in an image which can be launched as a Docker container.

Some developers create two Dockerfiles: one for development that is optimized for debugging and one for production that is optimized for speed and security. However, in this example, you will create:

1. a single `Dockerfile` for production use, and
2. a single `docker-compose.yml` which uses the production image but overrides the settings for development purposes.

This should require less effort and fewer system resources.

> You *could* use Docker Compose on your production server and define a specific configuration file, i.e. `docker-compose-production.yml`. However, the orchestration chapter suggests better alternatives.

## 6.5 Dockerfiles

A `Dockerfile` defines the build steps required to install and execute an application in order to create a ready-to-run image.

It's common to start with a base image from Docker Hub. This application requires a Node.js image:

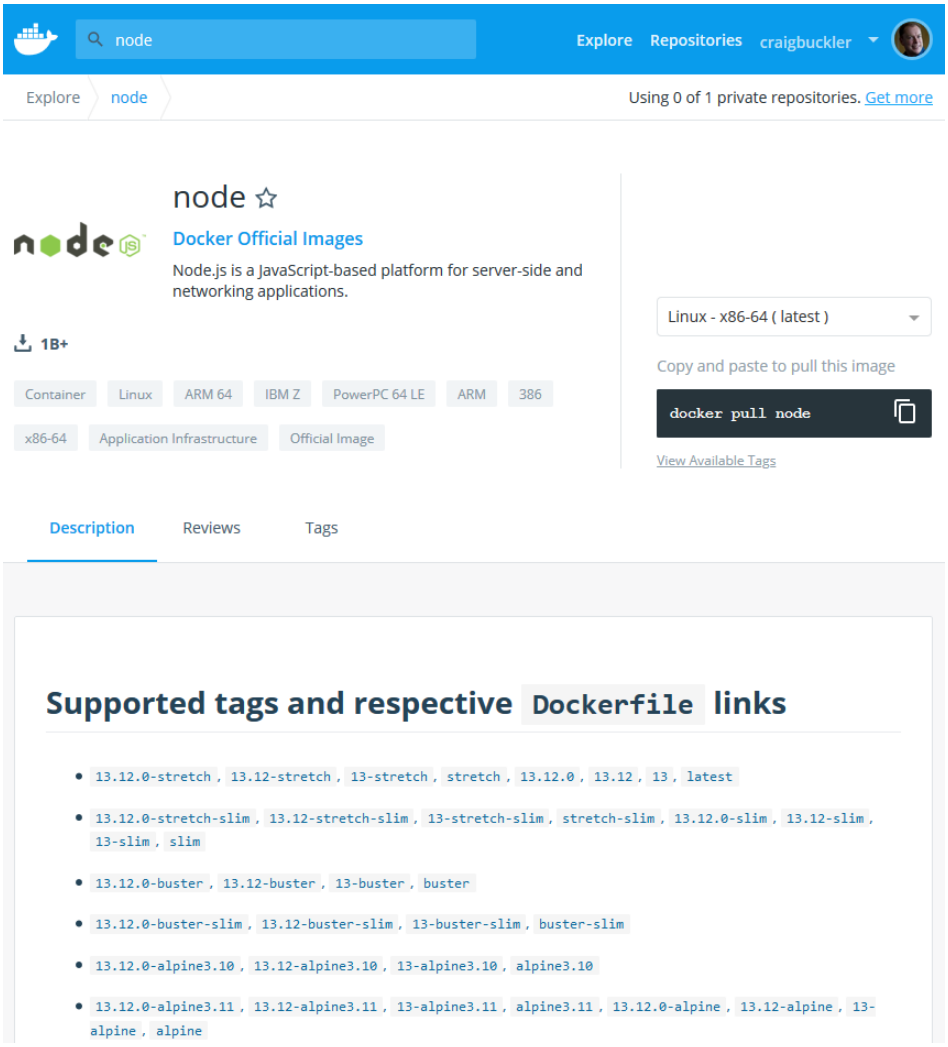



**Figure 6.3:** Docker Hub Node.js images

Each tag references a separate image (created with its own `Dockerfile`). For Node.js:

- many images are large and require a 100MB+ download because they contain a full Linux OS. These are recommended for most applications.
- `slim` images are are cut-down versions of the main Linux images with a minimum set of packages required to run Node.js. These may be useful if you're deploying Node.js containers to an environment with space constraints.
- `alpine` images are based on Alpine Linux and are typically around 5MB. These are useful if you need the smallest possible image and have limited reliance on OS libraries.

`lts-alpine` is adequate for this example application – it provides a tiny image with the most recent Long-Term Support version of Node.js.

Create a `Dockerfile` in the application's root directory with the content:

```
# base Node.js LTS image
FROM node:lts-alpine

# define environment variables
ENV HOME=/home/node/app
ENV NODE_ENV=production
ENV NODE_PORT=3000

# create application folder and assign rights to the node user
RUN mkdir -p $HOME && chown -R node:node $HOME

# set the working directory
WORKDIR $HOME

# set the active user
USER node

# copy package.json from the host
COPY --chown=node:node package.json $HOME/

# install application modules
RUN npm install && npm cache clean --force

# copy remaining files
COPY --chown=node:node . .

# expose port on the host
EXPOSE $NODE_PORT

# application launch command
CMD [ "node", "./index.js" ]
```

Starting from the `FROM node:lts-alpine` base image, each line defines a step to install and run the Node.js application in production mode.

Various Dockerfile commands can be used (refer to Appendix B), but the most important are:

| command | description |
|---|---|
| # | a comment |
| `FROM` | build the image from a Docker Hub starting image |
| `ARG` | define a variable that can be passed by the build command |
| `ENV` | set an environment variable |
| `WORKDIR` | set a working directory |
| `USER` | set the active user |
| `VOLUME` | create a volume mount point so that directory can be accessed from other containers in the Docker network |
| `RUN` | execute a command, such as `npm install` to download modules |
| `COPY` | copy files from the host machine |
| `EXPOSE` | expose a port to the host |
| `CMD` | default launch command (which can be overridden) |
| `ENTRYPOINT` | default launch command for executable images |

Refer to Appendix B for more information.

### 6.5.1 User security

Dockerfile commands run as a root (super) user when the image is created. This is generally safe; you could restart a container if something catastrophic occurred.

That said, it is safer to run your application as a more restricted user. A `node` user is created to launch the application in the `Dockerfile` above. It does not have rights to wipe the file system even if the application misbehaved or was maliciously controlled by a nefarious criminal!

### 6.5.2 Native launch command

It's best to launch applications by directly calling their executable:

```
CMD [ "node", "./index.js" ]
```

This ensures system messages such as `STDERR` are returned to Docker so it can react accordingly, e.g. restart a container when an application crashes.

Failure messages may not be detected if you use a third-party launch system, e.g. `npm start`. The `npm` application would receive errors, but it may not pass them through to Docker. This matters less when running Docker in development mode since you'll normally see logged errors.

### 6.5.3 Image layers

Every line of the `Dockerfile` creates a separate (hidden) image.

> After building, you can see every layer by entering `docker image ls -a`

These layers allow changes to be made more efficiently. For example, modifying the `EXPOSE` port is instantaneous because none of the previous build steps would change. However, using a different `FROM` image would require a complete re-build.

Ideally, Dockerfile commands should be ordered from least-likely to most-likely to change. This is the reason **`package`**`.json` is copied and modules are installed (`npm install`) *before* copying other application files. Imported modules are less likely to change than your own source files.

### 6.5.4 .dockerignore

The `COPY . .` command copies all application files from the host directory to the Docker image. It's unlikely you'll need everything so a `.dockerignore` file can be defined to omit files or directories which match name patterns. It will be familiar to anyone who has used Git's `.gitignore` file:

```
Dockerfile
docker*.yml

.git
.gitignore
.config

.npm
.vscode
node_modules

README.md
```

## 6.6 Build an image

To build an image named `nodehello` from your `Dockerfile`, run the following command in the project's root directory:

```
docker image build -t nodehello .
```

> The period at the end of the command is essential – it references the application path. You can also use `-f <file>` if you didn't name your build file "`Dockerfile`".

The build process could take several minutes as all steps are executed. Once it has completed, run `docker image ls` to view the new `nodehello` image (as well as the base `lts-apline` image it was created `FROM`):

```
REPOSITORY  TAG          IMAGE ID      CREATED          SIZE
nodehello   latest       f5bf8030cd5a  20 seconds ago   89.9MB
node        lts-alpine   f77abbe89ac1  12 days ago      88.1MB
```

Enter `docker image ls -a` to reveal the hidden images created at each step of your `Dockerfile`:

```
REPOSITORY  TAG          IMAGE ID      CREATED          SIZE
nodehello   latest       f5bf8030cd5a  2 minutes ago    89.9MB
<none>      <none>       9ceef3937b4a  2 minutes ago    89.9MB
<none>      <none>       e5bb2afb74b5  2 minutes ago    89.9MB
<none>      <none>       89b2f587c265  2 minutes ago    89.8MB
<none>      <none>       17e464183259  2 minutes ago    88.1MB
<none>      <none>       481fc173212a  2 minutes ago    88.1MB
<none>      <none>       8e078964714e  2 minutes ago    88.1MB
<none>      <none>       3bfa8ac75748  2 minutes ago    88.1MB
<none>      <none>       21439fb6ba1e  2 minutes ago    88.1MB
<none>      <none>       22a986df6cdd  2 minutes ago    88.1MB
<none>      <none>       79c4ae3fc5b5  2 minutes ago    88.1MB
node        lts-alpine   f77abbe89ac1  12 days ago      88.1MB
```

Each image is an incremental difference from the previous one. Therefore, `docker system df` reports that 89.9MB has been used in total (rather than 88MB for each hidden image):

```
TYPE            TOTAL   ACTIVE  SIZE      RECLAIMABLE
Images          2       0       89.89MB   89.89MB (100%)
Containers      0       0       0B        0B
Local Volumes   0       0       0B        0B
Build Cache     0       0       0B        0B
```

## 6.7 Launch a production container from your image

A container named `nodehello` (or whatever you prefer) can be launched from your image in interactive mode:

```
docker run -it --rm --name nodehello -p 3000:3000 nodehello
```

Open http://localhost:3000/ in your browser to see *"Hello World!"*

The container runs in its default production mode. If you make any changes to the application, you would need to rebuild the image again. This is clearly impractical, so press `Ctrl`|`Cmd + C` to stop and remove the container.

## 6.8 Launch a development environment with Docker Compose

Docker Compose can override the image's default production settings to launch a container in development mode.

Create the following `docker-compose.yml` in your project's root directory:

```
version: '3'

services:

  nodehello:
    environment:
      - NODE_ENV=development
    build:
      context: ./
      dockerfile: Dockerfile
    container_name: nodehello
    volumes:
      - .:/home/node/app
    ports:
      - "3000:3000"
      - "9229:9229"
    command: /bin/sh -c 'npm install && npm run debug'
```

In previous chapters, you referenced a specific `image:` from Docker Hub. In this file, you're using a `build:` option to create an image from a `Dockerfile`:

- `context:` is the relative path to the location of your `Dockerfile`, and
- `dockerfile:` is the name of your `Dockerfile`.

Additionally:

1. the `NODE_ENV` environment variable is set to `development`
2. port `9229` is exposed for remote Node.js debugging
3. no `restart` option is defined – if the application crashes, you want to know about it!

4. the application directory on your host PC is mounted to `/home/node/app` in the container's file system, and
5. the `Dockerfile` `CMD` application launch command is replaced with `/bin/sh` to start a new shell and execute `npm install` followed by `npm run debug`.

When the host directory is mounted into the container, none of the module `"dependencies"` required in **`package`**`.json` will be found. There are also `"devDependencies"` which were never installed. The `npm install` installs all modules to the host's application directory when the container is launched for the first time.

> Directories named `.config`, `.npm`, and `node_modules` will be created in the project's root directory on the host. Note:
>
> - If you encounter any `npm` failures during installation, try deleting those directories before trying again.
> - Add those names to `.gitignore` to ensure the directories are not committed to your Git repository.
> - Do not attempt to run the Node.js application from your host OS! The modules may be configured for Linux.
>
> Creating these directories can be slow on Windows hosts. The quiz project in Appendix D shows how to mount them in Docker volumes which makes building considerably faster.

Ensure any existing containers have been stopped then launch Docker Compose:

```
docker-compose up
```

The `nodehello` container will be started in development mode.

Docker Compose will use the `nodehello` image built earlier or initiate a build when that does not exist. You can also instruct Docker Compose to rebuild the image – perhaps after making changes to your `Dockerfile`:

```
docker-compose up --build
```

Docker Compose can be stopped with `Ctrl`|`Cmd` + `C` or entering `docker-compose down` in another terminal. Try live code editing and debugging before you do that.

### 6.8.1 Launch a development environment with `docker run`

Below the surface, Docker Compose issues a `docker run` command similar to:

```
docker run -it --rm \
  --name nodehello \
  -p 3000:3000 \
  -p 9229:9229 \
  -e NODE_ENV=development \
  -v $PWD:/home/node/app \
  --entrypoint '/bin/sh' \
  nodehello \
  -c 'npm install && npm run debug'
```

> Windows users: `$PWD` references the current directory on Linux and macOS. The full path must be specified using forward-slash notation on Windows e.g. `-v /c/projects/nodejs:/home/node/a`

The command launches the container in development mode and mounts the project directory on the host to `/home/node/app`. This is more complex than using Docker Compose, but may want to use the command in shell scripts or similar processes.

## 6.9 Live code editing

When the `nodehello` container is running in development mode, you can edit `index.js` and Nodemon will restart the application when you save the file.

For example, examine line 17:

```
const message = `Hello ${ req.params.name  || 'World' }!`;
```

and change `Hello` to `Hey there`:

```
const message = `Hey there, ${ req.params.name  || 'World' }!`;
```

Save the file and the container log will indicate that Nodemon has restarted the application. Refresh http://localhost:3000/ in your browser to see the updated message.

## 6.10 Remote container debugging

This section describes how to debug a Node.js application running in a Docker Container. It can be skipped if you have no interest in Node.js, although other runtimes offer similar facilities.

### 6.10.1 Node.js debugging overview

To debug a Node.js application, you can:

1. Set the `NODE_ENV` environment variable to `development`.

   Many modules use this to show more verbose errors or record logs. It has already been set in `docker-compose.yml`.

2. Launch `node` with the `--inspect=0.0.0.0:9229` parameter.

   This starts the debugger listening on port 9229 and allows connections from any device. The `npm run debug` command defined in **`package.json`** has this set.

3. Optionally, set breakpoints within your JavaScript code with `debugger;` statements.

You can attach to the Node.js debugger using the Chrome browser, VS Code, or any other compatible application.

> The default `--inpect` option (without an IP:port) opens port 9229 but only permits connections from the same device.
>
> A specific IP:port can be passed to `node --inspect` to limit connections to known devices.
>
> The `--inspect-brk` option (with or without an IP:port) sets a breakpoint on the first statement so the application is paused immediately on execution.

### 6.10.2 Node.js debugging with Chrome

Google Chrome, and Chromium-based browsers such as Brave, Edge, Opera, and Vivaldi, have a built-in Node.js debugger. Ensure your application is running in a development mode container, then launch the browser and enter chrome://inspect in the address bar:

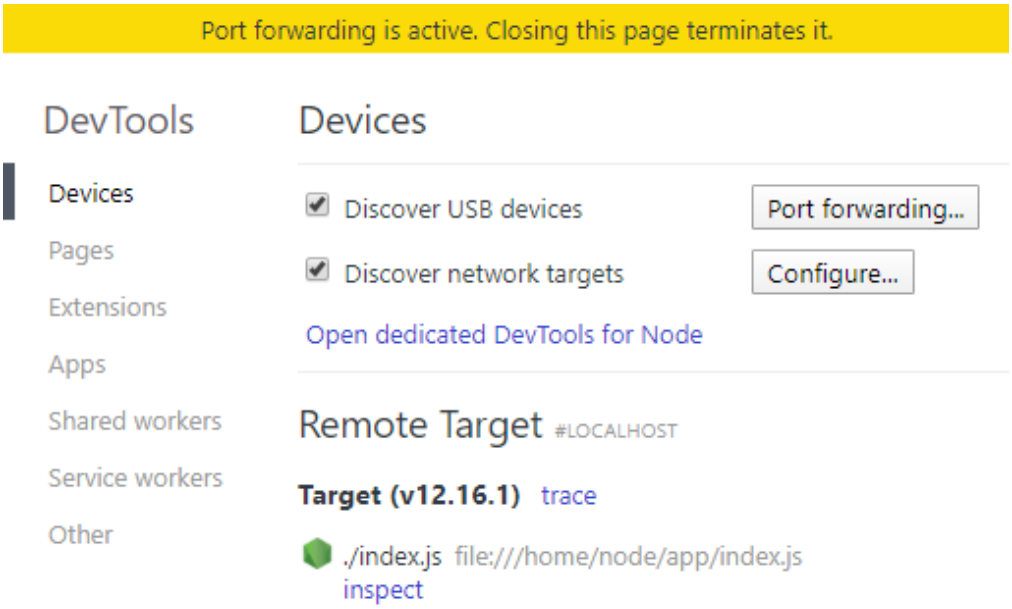



**Figure 6.4:** Chrome inspect page

> If you don't see a **Remote Target**, ensure **Discover network targets** is checked, click **Configure...** and add a connection for (or a network address if you're running the container on another device).

Click the **inspect** link below the **Target** to launch DevTools. This will be familiar if you've tried client-side debugging. There are application and memory profilers, but the **Sources** panel is most useful for debugging. You can also press `Esc` to show the **Console** in the lower pane:

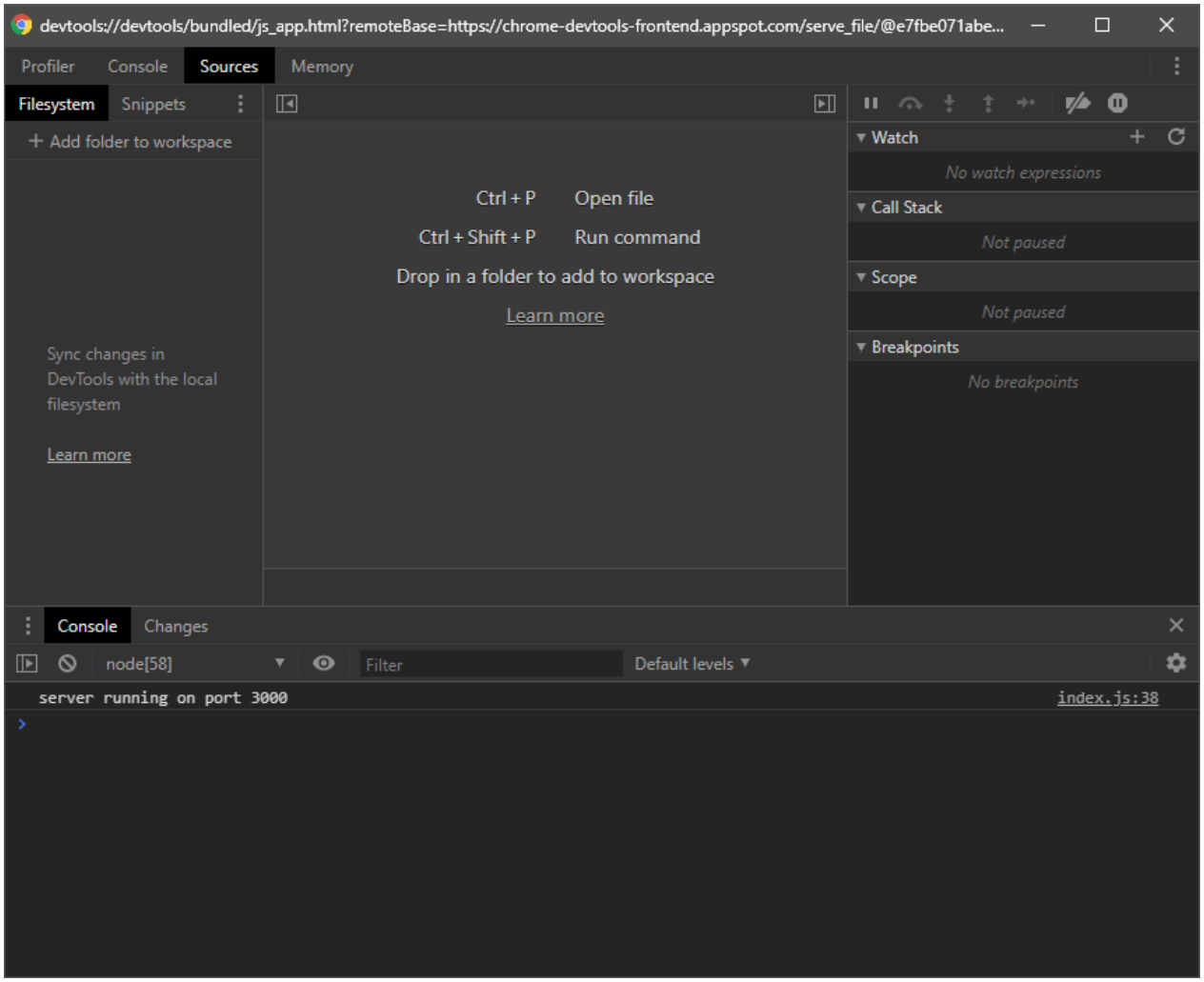



**Figure 6.5:** DevTools Sources panel

Press `Ctrl`|`Cmd + P`, enter `index.js`, and select the file at `home/node/app/index.js`. Click any line number to set a breakpoint (line 19 is set here):

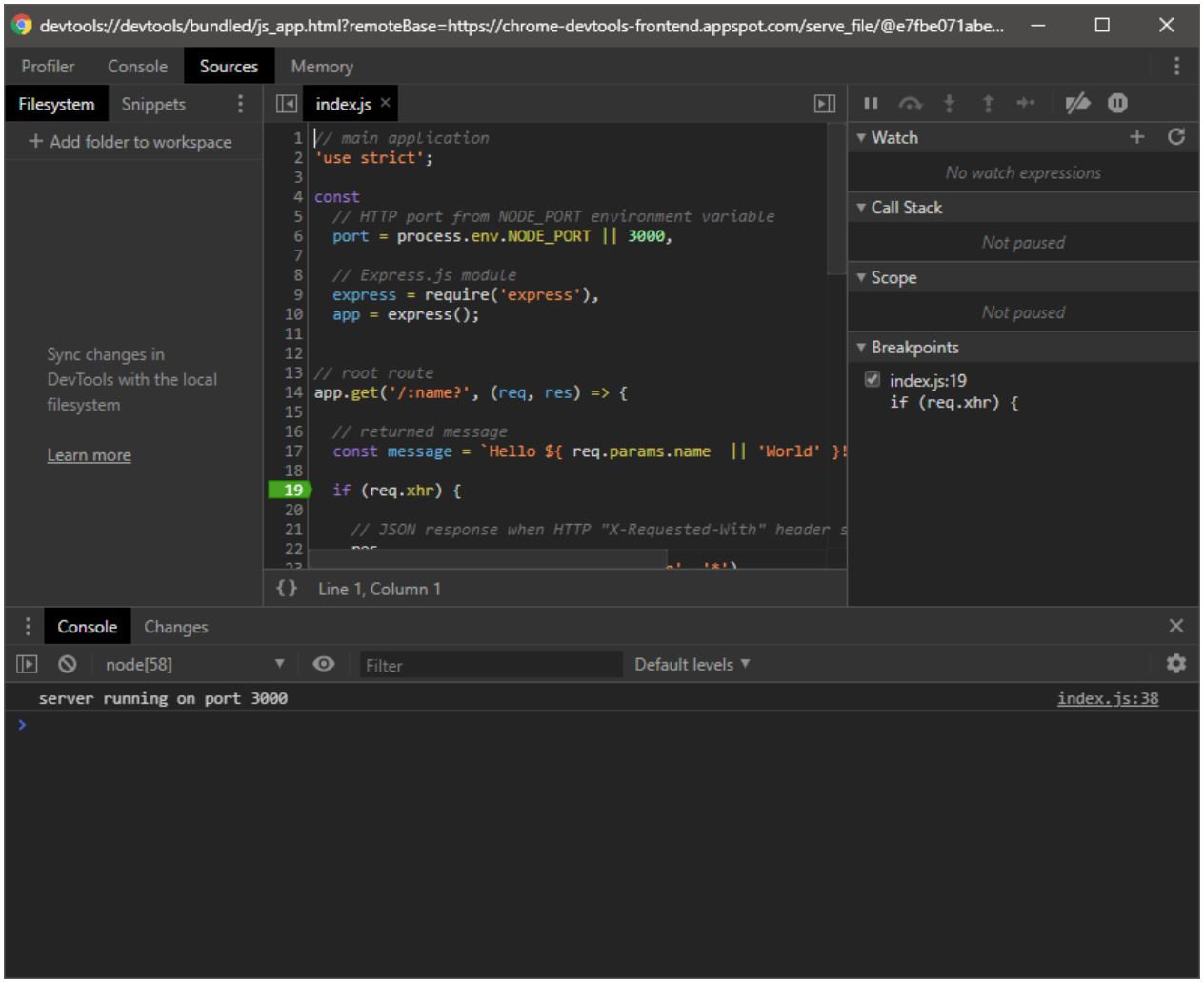



**Figure 6.6:** DevTools set breakpoint

Load or refresh http://localhost:3000/ in any browser and DevTools will pause on that line. You can now inspect the call stack and variable state. In this case, the `message` variable has been set to `"Hello World!"` in the right-hand **Scope** pane:

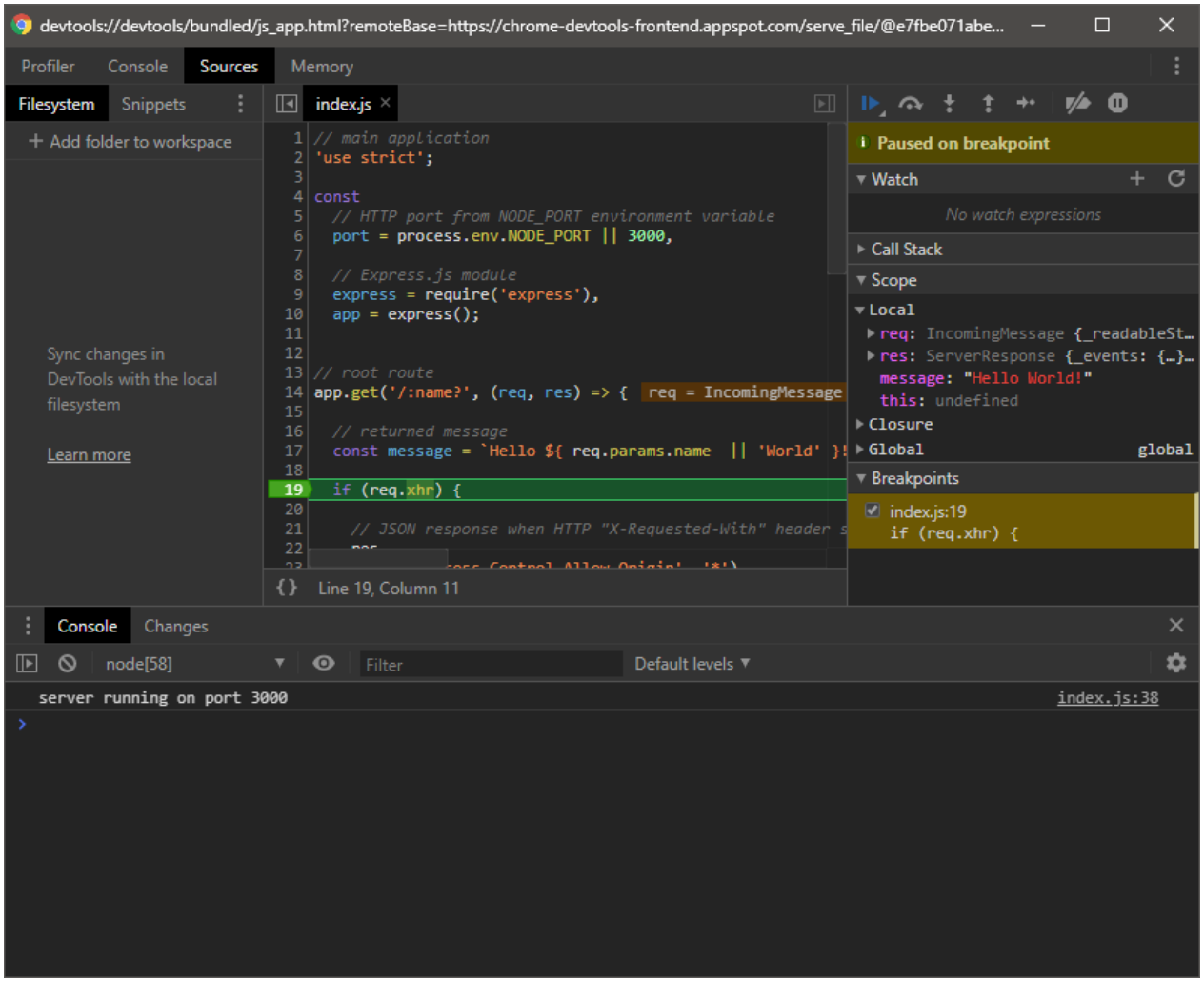



**Figure 6.7:** DevTools execution paused

The icons along the top of that panel can be clicked to:

| icon | description |
|---|---|
| resume | execute code until the next breakpoint (if any) |
| step over | run the next statement but do not pause on any functions it calls |
| step into | run the next statement and follow calls into other functions (including asynchronous functions such as `setTimeout`) |
| step out | finish the function and return to the calling statement |
| step | similar to step into, but asynchronous functions will not be called immediately |
| deactivate | disable or enable all breakpoints |
| exceptions | pause when an error is raised |

Try reloading http://localhost:3000/ using `Ctrl + F5` (or `Shift + F5` in some browsers) to bypass the browser cache. You will notice the debugger pauses on the breakpoint line twice:

- `message` is set to `"Hello World!"` in the first pass
- `message` is set to `"Hello favicon.ico!"` in the second.

You have a bug! You're welcome to attempt a fix (solution below).

> **Chrome Filesystem workspaces**
>
> The *+ Add directory to workspace* option allows you to select a source directory which Chrome can map to remote files. Unfortunately, it won't work in this case because, although your files are local, they're mounted within a container at `/home/node/app/`.

### 6.10.3 Node.js debugging with VS Code

The free Visual Studio Code editor has built-in Node.js debugging facilities as well as debugger extensions for all popular platforms.

> VS Code also has extensions to help with Docker management and file syntaxes as well as remote development to open directories inside containers. These are not used or required here because the debugger running in your container permits remote access. However, they offer some interesting development options.

Ensure your application is running in development mode, then launch VS Code, open the project's root directory, and load `index.js`. A breakpoint can be added by clicking to the left of any line:

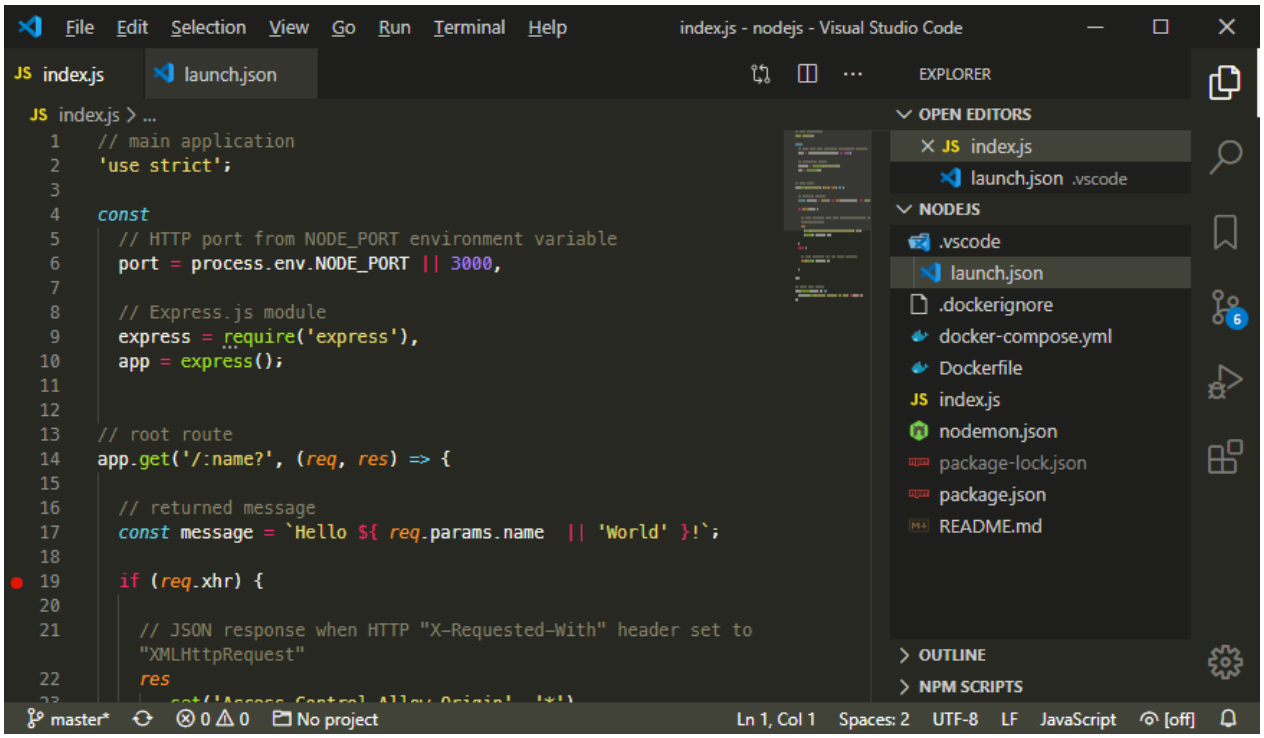



**Figure 6.8:** VS Code set breakpoint

*(An "unbound" warning message may appear the first time you do this.)*

Click the **Debugger** icon in the activity bar followed by **create a launch.json file**. Choose **Node.js** as the environment then add a process to **Attach** to a remote program. The following configuration is recommended – it maps the container's `/home/node/app` directory to the application's local directory:

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit:
  // https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "attach",
      "name": "Attach",
      "port": 9229,
      "protocol": "inspector",
      "localRoot": "${workspaceFolder}",
      "remoteRoot": "/home/node/app",
      "skipFiles": [
        "<node_internals>/**"
      ]
    }
  ]
}
```

Save `launch.json` (which is added to a new `.vscode` directory in your project) and a **Attach** option will appear in the drop-down at the top of the debugger pane:

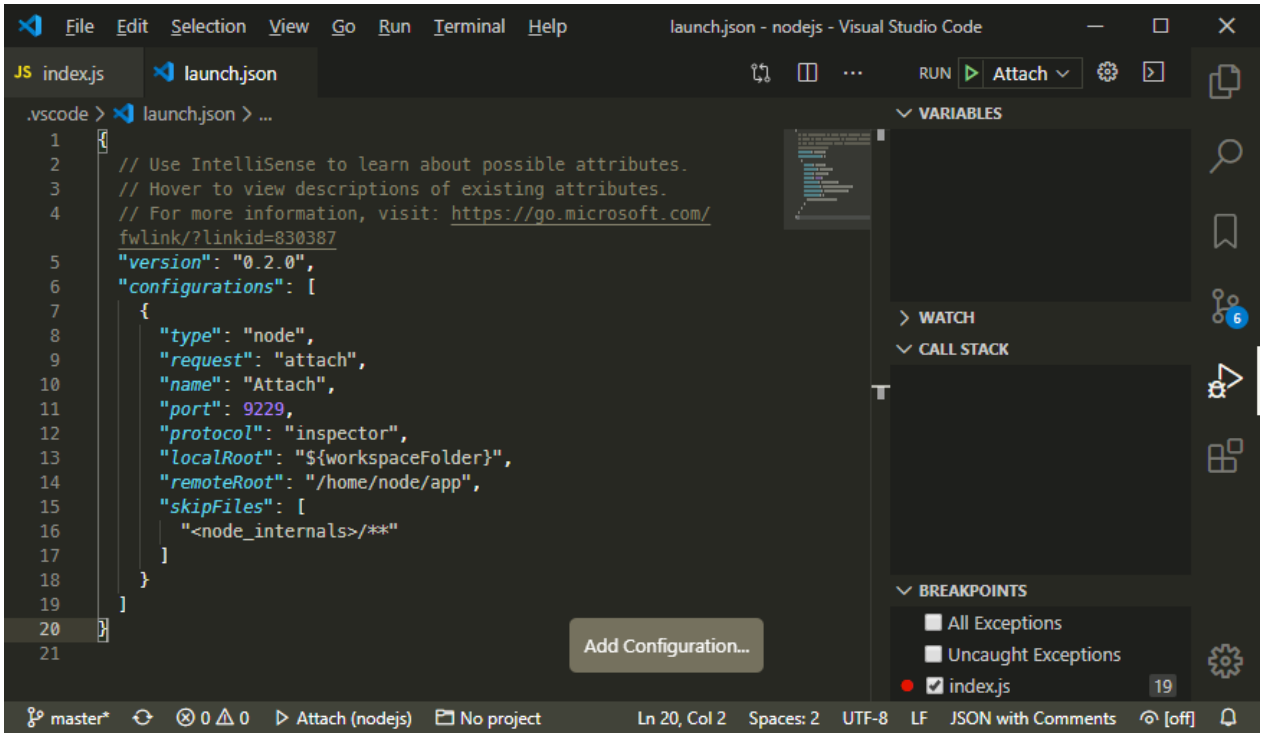



**Figure 6.9:** VS Code attach

Click the green **RUN** icon to start the debugger, then reload http://localhost:3000/ in a browser. VS Code will pause the script at the breakpoint where you can inspect variables and the call stack:

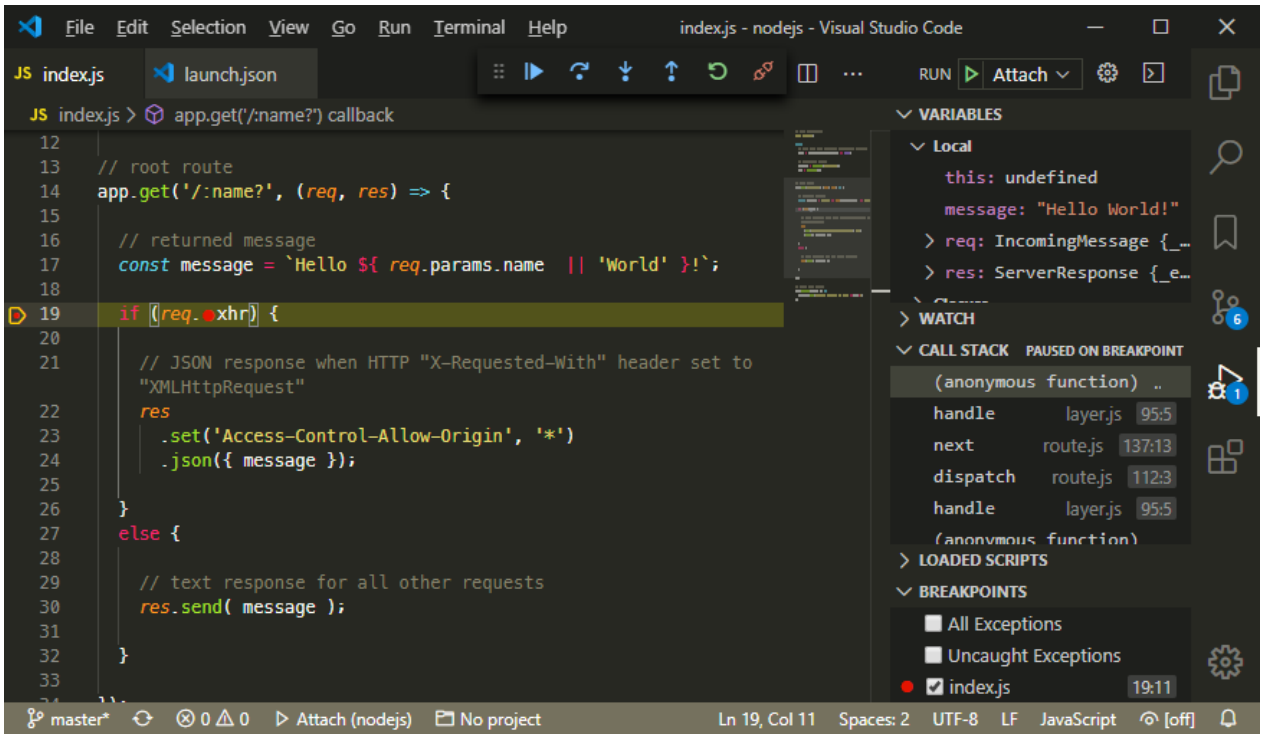



**Figure 6.10:** VS Code execution paused

The icon bar at the top offers similar resume, step over, step into, and step out icons as Chrome. There is also a **restart** icon and a **detach** icon to quit debugging.

## 6.11 Create an image from a container

Finally, it's possible to create a new image from a running container using `docker container commit` or the shorthand:

```
docker commit [CONTAINER_NAME] [IMAGE_NAME]
```

This option is more fragile than using a `Dockerfile`, but it could be useful if you need to retain the current state of your application for development or testing purposes. For example, perhaps you've experienced a race condition or similar bug which is difficult to replicate.

## 6.12 Key points

What you've learned in this chapter:

1. Building your own production-level Docker image with a `Dockerfile`.
2. Launching a container from that image in both production and development environments.
3. Using Docker Compose for easier development.
4. Auto-restarting a Node.js application when mounted files change.
5. Debugging an application running in a container.

For a more complex multi-container example, refer to the quiz project in Appendix D. Or perhaps create a simple Dockerized application in your language of choice.

In the next chapter, you will use push the Docker image you've just created to Docker Hub … *and discover why that could be useful*.

### 6.12.1 Fixing the favicon.ico bug

Did you work out why the example application made two HTTP requests during debugging?

When a browser makes its first request for a web page it also requests a `favicon.ico` image. This is the icon shown to the left of the title in the browser tab. Web servers normally send an appropriate image or return an HTTP `404 Not found`.

The application currently treats this request in the same way as any other and returns a `Hello favicon.ico` message which the browser cannot use. It's hardly catastrophic, but both the browser and server are doing unnecessary work.

The easiest solution is to return a `404` error by adding a further route function after line 10:

```
app.get('/favicon.ico', (req, res) => res.sendStatus(404));
```

Alternatively, you could serve a real favicon image from an Express.js static directory or by using a module such as serve-favicon.

# 7 Push your Docker image to a Repository

Docker Hub is to Docker what Github is to Git. *Kind of.*

You will primarily use Docker Hub to locate and use application images – such as Apache, PHP, Node.js, Python, MySQL, or MongoDB. However, it's also possible to push your own application images to Docker Hub.

## 7.1 Why push an image to Docker Hub?

Why indeed. Many developers never will. Your `Dockerfile` can be added to any project repository to build an application image during development or as part of a production deployment process.

However, pushing an image to Docker Hub offers several benefits:

1. It is easier to distribute a pre-built and tested image with your team.
2. The image can be pulled on production servers.

   Deployment is simpler and faster because the image has already been built.
3. A published image can be used by anyone.

   Your application can be shared with users, clients, and other developers throughout the world.

Docker Hub allows you to create any number of public repositories. These appear in search results and can be used by anyone – *but remember you will be distributing your application's source code*.

You can also create one private repository and purchase more if necessary. Access to private repositories can be granted to named organizations and individual Docker Hub users.

## 7.2 Docker Hub alternatives

As well as Docker Hub, there are plenty of alternative hosted and self-hosted container registries. All provide private repositories with varying price points:

- Amazon ECR
- Azure
- Canister
- Cloudsmith
- GitHub
- GitLab
- Google
- Harbor
- JFrog Artifactory
- Portus
- Quay.io
- Sonatype Nexus

This chapter uses Docker Hub, but the process is similar for other repositories. Typically, you will log into a different system and/or specify a URL when pushing the image.

## 7.3 Image names and tags

Every image on Docker Hub is assigned a unique name:

```
[your_user_name]/[image_name]:[tag_name]
```

The following sections refer to the `nodehello` image created in the last chapter which is uploaded to my `craigbuckler` account.

Tags are different variations of an image. You first saw these when running the MySQL container – the official MySQL image offers various editions of `5.6`, `5.7`, and `8.0`.

A default tag of `latest` is applied to any new image. That version is downloaded when no tag is specified when pulling an image.

A single image can have any number of tags applied. The onus is on you – the image developer – to tag images appropriately and ensure the most recent stable release is tagged with `latest`.

## 7.4 Create a Docker Hub repository

The first step is to create a repository at Docker Hub. Log in at https://hub.docker.com/ and click **Repositories** then **Create Repository** (or open https://hub.docker.com/repository/create):

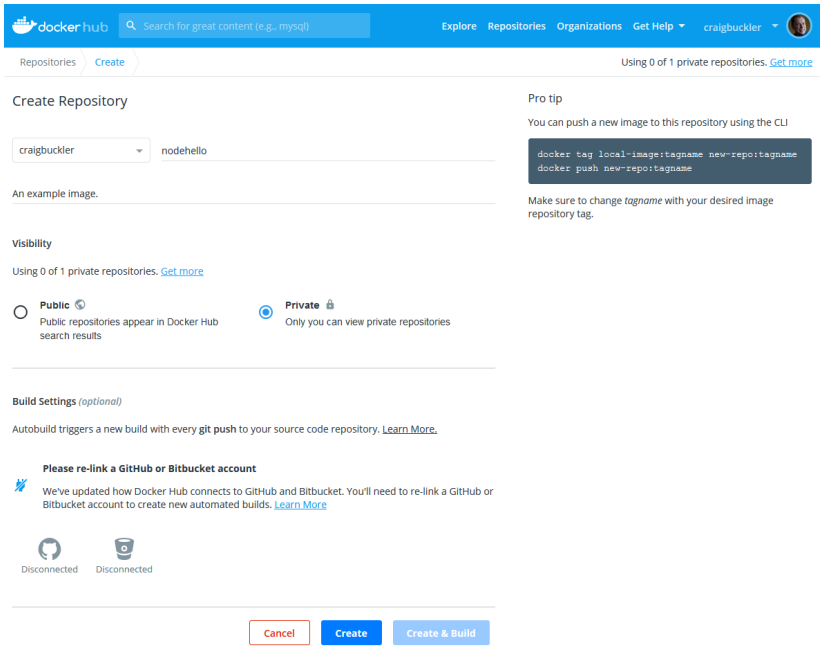



**Figure 7.1:** create repository on DockerHub

Enter the repository name, choose whether it's public or private, and click **Create**.

> Docker Hub can also automatically build images when you push `Dockerfile` configurations to Github or BitBucket Git project repositories.

## 7.5 Log in locally

Log on your development PC with the same Docker Hub credentials using:

```
docker login
```

> Alternatively, you can choose *log in* from the Docker Desktop menu on Windows and macOS.

To log into an alternative image repository (not Docker Hub), use:

```
docker login <url>
```

## 7.6 Build an application image

Build an image from your application's Dockerfile. The Node.js example from the previous chapter:

```
docker image build -t nodehello .
```

Test the image by launching a container, e.g.

```
docker run -it --rm --name nodehello -p 3000:3000 nodehello
```

Presuming everything works as expected, press `Ctrl`|`Cmd + C` to stop and remove the container. List the available images using:

```
docker images ls
```

In this example, `nodehello` has been created and the `latest` tag is automatically applied:

```
REPOSITORY  TAG          IMAGE ID      CREATED          SIZE
nodehello   latest       7076917cd3fd  15 minutes ago   89.9MB
node        lts-alpine   f77abbe89ac1  2 weeks ago      88.1MB
```

## 7.7 Tag an image

You can now tag the image with your user name, repository name, and tag name so it's ready to push to Docker Hub, e.g.

```
docker tag nodehello craigbuckler/nodehello:firsttry
```

Running `docker image ls` again returns:

```
craigbuckler/nodehello  firsttry    7076917cd3fd  18 mins ago  89.9MB
nodehello               latest      7076917cd3fd  18 mins ago  89.9MB
node                    lts-alpine  f77abbe89ac1  2 weeks ago  88.1MB
```

The tagged version *points* to the `nodehello` original. You can create as many tags as you like for the same image, e.g.

```
docker tag nodehello craigbuckler/nodehello:latest
```

> The `docker image build -t` option can specify a fully-qualified name so it's possible to build and tag an image in a single command:
>
> ```
> docker image build -t craigbuckler/nodehello:firsttry .
> ```
>
> However, building and testing before tagging is safer – it's more difficult to accidentally push a failing image!

## 7.8 Push to Docker Hub

Push your tagged images to Docker Hub using:

```
docker push craigbuckler/nodehello
```

This could take some time for larger images, but results will eventually be displayed:

```
The push refers to repository [docker.io/craigbuckler/nodehello]
6766d962cd48: Pushed
91341a5db1c1: Pushed
ecb74f18e87d: Pushed
181dbdd006ce: Pushed
7c9ecf609394: Mounted from library/node
d570627dd098: Mounted from library/node
1d3a976388c0: Mounted from library/node
beee9f30bc1f: Mounted from library/node
firsttry: digest: sha256:a010ec80a540f618c4801c5e63bd7c4bb8d size: 1991
```

Click the **Repositories** link in Docker Hub again to view your images. The uploaded image will be available at the URL:

```
https://hub.docker.com/repository/docker/[your_user_name]/[image_name]/
```

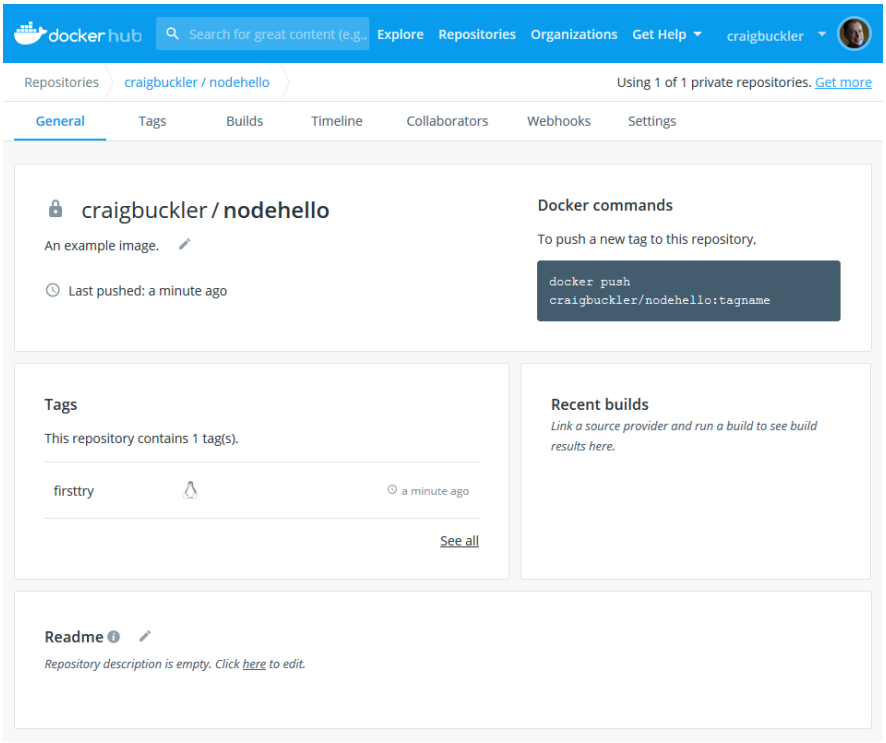



**Figure 7.2:** image pushed to Docker Hub

## 7.9 Distribute your image

You can now wipe all images from your system *(if you're absolutely certain, of course…)*

```
docker system prune -af
```

A container can then be launched directly from your Docker Hub image like you've done for other applications, e.g.

```
docker run -it --rm --name nodehello -p 3000:3000 \
  craigbuckler/nodehello:firsttry
```

## 7.10 Key points

What you've learned in this chapter:

1. Image repository options.
2. Building and tagging an image.
3. Pushing an image to a repository for easier distribution.

Bonus points: try automatically building Docker Hub images from your project's Github or BitBucket repository.

In the next chapter, you will take your first steps with orchestration and using Docker containers in production environments.

# 8 Docker orchestration on production servers

This book primarily explains how to use Docker during development in order to emulate a production environment. Your live server may not use Docker and that's fine if, for example, you're running a WordPress site hosted by a specialist company.

However, deploying application containers to live production servers offers several benefits. Containers can be:

- monitored for availability or speed
- restarted on failure
- scaled according to demand
- updated without downtime (presuming at least one application container remains active while others update).

## 8.1 Dependency planning

There are many choices to make when planning a container-based production environment.

Some dependencies, such as a database, could be provided by cloud services. Specialist Software as a Service (SaaS) companies do the hard work for you: there's no need to worry about installation, maintenance, security, scaling, disk space, or back-ups.

Alternatively, you could choose to install and manage a database application yourself – perhaps for cost, security, or vendor lock-in reasons. It's possibly better to install it directly on the host OS rather than within a container with self-imposed CPU, RAM, and disk limits.

## 8.2 Application scaling

The example Node.js application runs on a single processing thread and other language runtimes use a similar model. Your server (or a container) could have many CPU cores but only one will actively execute the application.

Running the application in multiple containers permits more efficient parallel processing. This is preferable to solutions such as self-managed clustering or process managers like PM2 because containers are isolated and can be restarted automatically.

You *could* use Docker Compose or a script to launch multiple container instances of your application. If you were running them all on a single production server, each would require a different name and port exposed to the host, e.g. for three instances:

```
docker run -d --rm --name container1 -p 3001:3000 myimage
docker run -d --rm --name container2 -p 3002:3000 myimage
docker run -d --rm --name container3 -p 3003:3000 myimage
```

> `--restart=always` can be added to each command to ensure Docker restarts the application when the container exits or crashes.

A load balancer such as NGINX (running on the host OS or in another container) can forward incoming requests to one of the application ports. Three users accessing at the same time could be processed by different containers running in parallel. *In theory*.

If this sounds like hard work: *it is*. And that's before you consider sharing volumes and distributing containers between other real and/or virtual servers on the same network. Fortunately, there are easier solutions…

## 8.3 Orchestration overview

*Orchestration* is a process used to manage, scale, and maintain container-based applications across one or more devices.

Tools to manage container-based applications are named *orchestrators*. One of the first was Apache Mesos, which was soon followed by Kubernetes – an open source system developed by Google and often abbreviated to *K8S*. Docker fully embraces Kubernetes, but also provides the simpler Docker Swarm.

*Then things get complicated.*

There are dozens of implementations (the Kubernetes documentation lists at least fifteen) and cloud provider may offer their own Kubernetes-based orchestration services with different terminology and methods: AWS Fargate, Google Cloud, Microsoft Azure, Alibaba Cloud, Tencent etc.

These providers also supply their own load balancing, data, and file storage products which may be preferable to containerized services.

Just to add to the confusion, there are dozens of companies providing alternative/easier interfaces to the large cloud services.

### 8.3.1 Choosing an orchestration service

There are many options which change regularly and could fill a book in their own right.

For smaller or test applications, Docker Swarm running on inexpensive hosting such as DigitalOcean is a simple way to get started. An example configuration is shown below.

Larger applications with tens of thousands of users are likely to benefit from a more advanced Kubernetes-based service. An overview of Kubernetes is provided below but I recommend you seek expert advice for your platform of choice: *you're likely to be spending a lot of money!*

## 8.4 Docker Swarm

Docker Swarm is provided in your Docker installation so it's possible to run it on one or more development machines before deployment. The terminology in brief…

---

*A **swarm** is a set of nodes which are either physical or virtual servers. Each swarm must have at least one **manager** node which manages the **service**: a set of **tasks** (containers) is distributed to **worker** nodes.*

---

If you've not previously used Swarm, `docker system info` returns a report including:

```
 Swarm: inactive
```

Execute the following command to define your machine as the swarm manager:

```
docker swarm init
```

The result:

```
Swarm initialized: current node (yvhtgc5lypxnygtijxh) is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join --token
  SWMTKN-1-0p6w735dvmxq82dcmpcxnykj3obmpfm0wqwcdg4kc17irwmu9y
  192.168.65.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

Other physical or virtual devices on your network which have Docker installed can join the swarm by issuing the (long) `docker swarm join --token` command shown.

Enter `docker node ls` to view all nodes within your swarm. A single development machine will display something similar to:

```
ID            HOSTNAME        STATUS  AVAILABILITY  MANAGER STATUS
yvhtgc59 *    docker-desktop  Ready   Active        Leader
```

Pull any images you require before creating a service – unlike `docker run` and `docker compose`, they won't be pulled automatically, e.g.

```
docker pull craigbuckler/nodehello
```

Then create a service which launches a single container on the swarm:

```
docker service create --name nodehello -p 3000:3000 \
  craigbuckler/nodehello
```

Execute `docker service ls` to show the running tasks:

```
ID          NAME       MODE         REPLICAS  IMAGE      PORTS
h7xdw61f8   nodehello  replicated   1/1       nodehello  *:3000->3000/tcp
```

`docker container ls` will also show a single running container (presuming you're running the swarm on one machine).

### 8.4.1 Scale a swarm service

Your application can be scaled as it becomes increasing popular. To launch three container instances for the `nodehello` service, enter:

```
docker service scale nodehello=3
```

`docker service ls` now reports three replicas running on your single installation, any of which can handle requests to http://localhost:3000/:

```
ID          NAME       MODE         REPLICAS  IMAGE      PORTS
h7xdw61f8   nodehello  replicated   3/3       nodehello  *:3000->3000/tcp
```

`docker container ls` shows those containers presuming they're all running on your local machine:

```
CONTAINER ID  IMAGE       STATUS          PORTS     NAMES
4748e3dd2981  nodehello   Up 31 seconds   3000/tcp  nodehello.2.9c5283
9d9b5afc565b  nodehello   Up 32 seconds   3000/tcp  nodehello.3.norbua
0f083e40fc37  nodehello   Up 6 minutes    3000/tcp  nodehello.1.qjmaki
```

> Alternatively, the `docker service create` command has an optional `--replicas` which could have been used instead of scaling later, e.g.
>
> ```
> docker service create --name nodehello -p 3000:3000 --replicas=3 \
>   craigbuckler/nodehello
> ```

### 8.4.2 Stop and remove a swarm service

Services can be listed by executing `docker service ls`.

A service name can then be passed to the `rm` command to stop it, e.g.

```
docker service rm nodehello
```

All containers started on swarm nodes are automatically stopped and removed.

### 8.4.3 Remove a node from a swarm

Any worker node can leave the swarm by entering this command on the device:

```
docker swarm leave
```

The manager node can leave the swarm and disable swarm mode using:

```
docker swarm leave --force
```

## 8.5 Kubernetes

Kubernetes is considerably beyond the scope of this book, but it uses the following concepts and terminology…

---

*A **cluster** is a set of nodes which are either physical or virtual servers (like a Docker swarm). At least one **master** (like a swarm manager) controls all nodes through the Kubernetes API server. Communication with the master is handled by the **kubectl** CLI tool.*

*Each node runs an agent process known as a **kubelet**. It is responsible for receiving information from the master to start, stop, or modify groups of containers known as **pods** which have shared storage and network resources.*

---

Your host may have their own Kubernetes methods and recommendations, so it will be practical to provision and configure a test platform within the same environment.

Another option is the play with Kubernetes tool allows you configure a cluster without installing anything locally.

Finally, it's possible to install Kubernetes on your Windows, macOS, or Linux development PC. Kubernetes is provided in Docker Desktop for Windows and macOS: to enable it, choose **Settings** from the Docker icon menu, select the **Kubernetes** pane, check **Enable Kubernetes**, and click **Apply & Restart**. This process may take several minutes to complete.

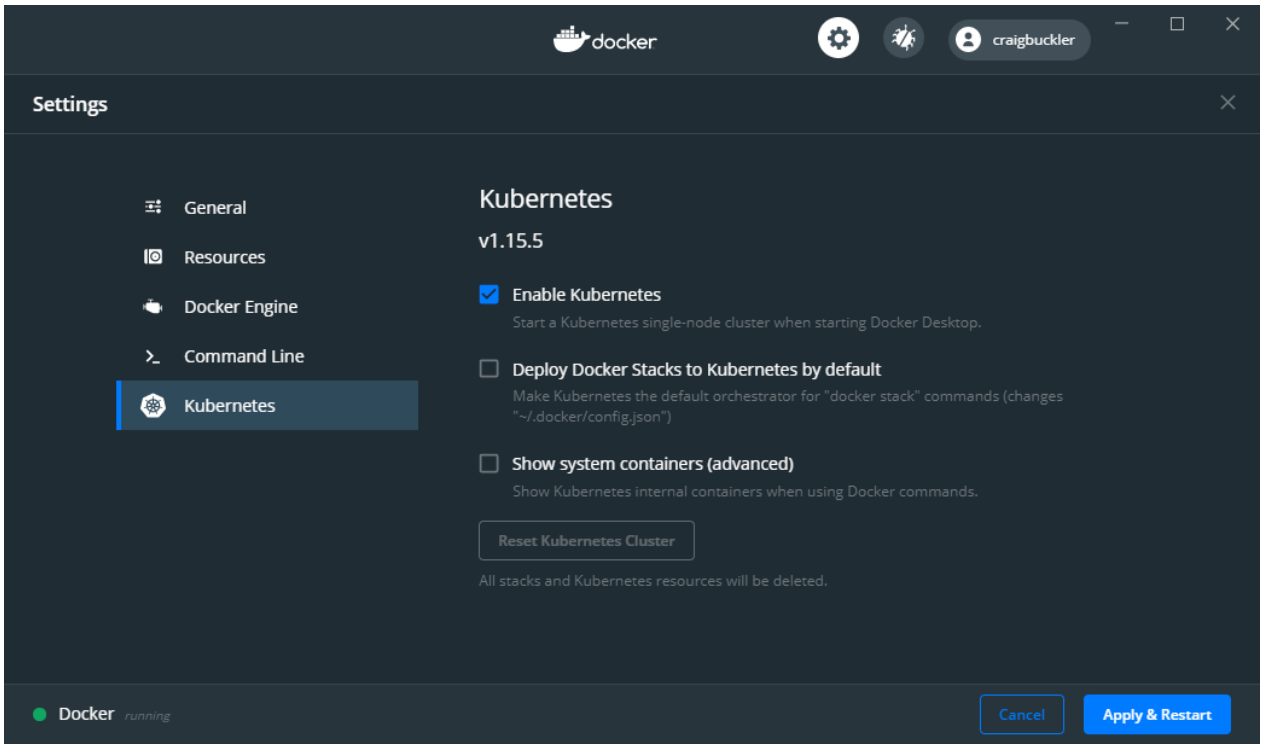



**Figure 8.1:** enable Kubernetes

### 8.5.1 Kubernetes deployment

Kubernetes deployments are defined YAML files. Unlike traditional programming steps, these describe a desired end state which Kubernetes aims to achieve.

The following example defines a `deployment.yaml` file to run the latest NGINX web server in two replica pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # 2 pods
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

To deploy:

```
kubectl apply -f ./deployment.yaml
```

> You can also host the deployment file on a web server and use its URL as the reference, e.g.
>
> ```
> kubectl apply -f https://myserver.com/deployment.yaml
> ```

Information about the `nginx-deployment` can be displayed with:

```
kubectl describe deployment nginx-deployment
```

The pods created during the deployment can be listed:

```
kubectl get pods -l app=nginx
```

to reveal pod names:

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-1771418926-7o5ns   1/1     Running   0          10h
nginx-deployment-1771418926-r18az   1/1     Running   0          10h
```

More information about any pod can then be viewed, e.g.

```
kubectl describe pod nginx-deployment-1771418926-7o5ns
```

Node IP addresses and ports are only available within the cluster. A load balancer can be added to expose them externally:

```
kubectl expose deployment nginx-deployment \
  --port=80 --target-port=80 \
  --name=nginxlb --type=LoadBalancer
```

### 8.5.2 Deployment updates

To update the deployment, change the YAML file and run the same `kubectl apply` again. This can be useful when you need to increase the number of replicas owing to increased demand.

Deployments can be stopped and deleted by passing its name to the `delete` command:

```
kubectl delete deployment nginx-deployment
```

### 8.5.3 Kubernetes resources

Several other tools may be useful:

- Kompose: a tool to convert Docker Compose to Kubernetes configuration files
- minikube: quickly define a Kubernetes cluster on a development machine
- Helm: manages Kubernetes resources

Further Kubernetes links:

- Kubernetes.io
- Kubernetes documentation
- Deploy to Kubernetes
- Docker and Kubernetes
- Play with Kubernetes

## 8.6 Key points

What you've learned in this chapter:

1. Orchestration on production servers.
2. Using Docker Swarm to scale an application.
3. The basics of Kubernetes deployment.
4. Hosts which offer container orchestration services.

# 9 Your Docker journey

Web developers often shun Docker and I was a skeptic for many years. It took several months wading through an impenetrable jungle of conceptual overviews, terminology, and misinformation to experience my first eureka moment.

I now find it unthinkable to approach a project without considering Docker. Installing and managing dependencies is no longer a hurdle and I'm not afraid to try new software. I can safely run an obscure Java application or beta Rust-driven database without risking stability, security, and sanity.

*Docker is liberating.*

I hope this book provides you with enough knowledge to understand Docker and be enthusiastic about using it during web development.

## 9.1 Docker's future

Many developers are yet to discover the benefits of Docker and containerization. It's a relatively new concept (2013) and remains a complex topic even for IT professionals.

My prediction: *Docker will become easier.*

Deployment and orchestration are especially difficult so huge advances from cloud hosts are likely in the next few years.

Apple, Google, and Microsoft are also considering Docker-like technologies within their main operating systems. Cross-platform software can already be developed using options such as Electron or Qt, but containers could allow more flexible development of secure applications which run anywhere without modification or recompilation.

Finally, Unikernels push containerization further by packaging an application into a fast, ultra-lightweight virtual machine with no operating system. They run directly on a hypervisor which handles hardware interoperability so no host OS is required. Unikernals are new and will take several years to become a practical option for development and deployment.

## 9.2 Further Docker help

The appendices in this book provide further information for:

1. using the `docker` command line
2. building your own images with a `Dockerfile`
3. managing multiple containers with `docker-compose`
4. creating a multi-container application

The Docker Documentation (docs.docker.com) is an excellent but somewhat daunting resource. It probably provides the information you want, but finding it is another matter!

The discord.com chat room is available as part of this course to discuss Docker concepts and problems with web developers who have probably encountered similar issues. Your registration invite link is available in your book/course receipt email.

The Docker Community Forums (forums.docker.com) is another great place to seek help and discuss topics or best practices.

There is also a thriving Docker community on reddit.com which can offer rapid assistance. You could also consider StackOverflow although questions tend to be highly specific and answered less frequently.

The official Docker Community (docker.com/docker-community) and Docker Twitter account provides news, events, meet-ups, and training resources from around the world.

Of course, there is Google. Enter a question or error message and you'll probably find information on Stack Overflow and similar sites. Always check the document date and feedback to ensure it relates to your installation.

Finally, you can send me a message on Twitter @craigbuckler or *(blatent plug alert)* … consider hiring me.

# 10 Appendix A: Docker command-line reference

The Docker command-line reference provides full documentation, but the commands below will be used most often.

## 10.1 Log into Docker Hub

```
docker login
```

Or login to another registry:

```
docker login <url>
docker login -u <id> -p <password> <url>
```

## 10.2 Search Docker Hub

Search the Docker Hub image repository:

```
docker search [options] <keyword>
```

Example: find up to five Docker Hub images that reference `php` ordered by stars (how many people liked the image):

```
docker search --limit 5 php
```

## 10.3 Pull a Docker Hub image

Download one or more images from Docker Hub:

```
docker pull [options] <image>
```

Example: pull the latest Node.js Long-Term-Support Alpine Linux image:

```
docker pull node:lts-alpine
```

## 10.4 List Docker images

```
docker image ls
docker images
```

View all images, both active and dangling (those not associated with a container):

```
docker image ls -a
```

## 10.5 Build an image from a Dockerfile

```
docker image build -t <image_name> .
```

Assuming a `Dockerfile` is in the current directory:

```
docker image build -t myimage .
```

If an alternative file name was used:

```
docker image build -f mybuildfile.txt -t myimage .
```

## 10.6 Tag an image

```
docker tag <image_name> [user/repository:tag]
```

Example: tag the `helloworld` image with `latest` in the Docker Hub `helloworld` repository owned by `myname`:

```
docker tag nodeworld myname/helloworld:latest
```

## 10.7 Push tagged images to Docker Hub

```
docker push [user/repository]
```

Example:

```
docker push myname/helloworld
```

## 10.8 Launch a container from an image

Start running an image as a container instance:

```
docker run [options] <image> [command] [args...]
docker container run [options] <image> [command] [args...]
```

Launch a container named `mysql` (`--name`) from the latest `mysql` image in interactive mode (`-it` shows a terminal log), that exposes port `3306` to the host (`-p`), stores data in a Docker volume named `mysqldata` (`--mount`), sets the root user password to `mysecret` (`-e` environment variable), and deletes the container once it has stopped (`-rm`):

```
docker run \
  -it --rm --name mysql \
  -p 3306:3306 \
  --mount "src=mysqldata,target=/var/lib/mysql" \
  -e MYSQL_ROOT_PASSWORD=mysecret \
  mysql
```

Useful options include:

| option | description |
|---|---|
| `-d` | run a container as a background process (which exits when the application ends) |
| `-it` | keep a container running in the foreground (even after the application ends), and show an activity log |
| `--rm` | remove the container after it stops |
| `--name` | name a container (a random GUID is used otherwise) |
| `-p` | map a host port to a container port |
| `--mount` | create a persistent Docker-managed volume to retain data. The string specifies a `src` volume name and a `target` where it is mounted in the container's file system |
| `-v` | mount a host folder using the notation `<hostdir>:<containerdir>` |
| `-e` | define an environment variable |
| `--env-file` | read environment variables from a file where each line defines a `VAR=value` |
| `--net` | connect to specific Docker network |
| `--entrypoint` | overrides the default starting application |

## 10.9 List containers

Active containers can be viewed with:

```
docker container ls
docker ps
```

Also view stopped containers which have not been removed:

```
docker container ls -a
```

## 10.10 Run a command in a container

```
docker exec -it <container_id_or_name> [command] [args...]
```

Example: list files in the root directory of the container named `mysql`:

```
docker exec -it mysql sh -c "ls /"
```

## 10.11 Attach to a container shell

To access a container shell to issue commands and examine files:

```
docker exec -it mysql bash
```

`sh` may be necessary in lightweight containers without `bash`.

Enter `exit` to quit the shell.

## 10.12 Restart a container

```
docker container restart <container_id_or_name>
```

## 10.13 Pause a container

```
docker container pause <container_id_or_name>
```

## 10.14 Unpause (resume) a container

```
docker container unpause <container_id_or_name>
```

## 10.15 View container metrics

View the CPU, memory usage, and network activity of all containers:

```
docker stats
```

Specific containers can be examined by adding one or more IDs/names to the command.

## 10.16 Increase container resources

Container resources are limited by Docker, typically to:

- 2 CPUs
- 2GB RAM
- 512MB swap space (virtual RAM)

Resources can be increased if necessary using `docker run` options such as:

| option | description |
|---|---|
| `--cpus` | CPUs, e.g. `--cpus="4"`. Zero sets no limit, and fractions can also be used, e.g. `"2.5"` |

| option | description |
| --- | --- |
| `--memory` | RAM, e.g `--memory="4g"` (4GB). A minimum of `4m` (4 MB) is permitted |
| `--memory-swap` | swap space, e.g. `--memory-swap="1g"` |

However, horizontally scaling your application using multiple containers could be a better option.

## 10.17 Stop a container

When a container is running in interactive mode (`-it`), press `Ctrl` | `Cmd + C` to stop it. If that fails, or the container is running in the background, use:

```
docker container stop <container_id_or_name>
```

To stop the `mysql` container:

```
docker container stop mysql
```

## 10.18 Remove stopped containers

```
docker container rm <container_id_or_name>
```

or remove all stopped containers:

```
docker container prune
```

## 10.19 View Docker volumes

Volumes are Docker-managed disks mounted into a container (using `--mount`) to provide persistent storage between restarts. List created volumes:

```
docker volume ls
```

## 10.20 Delete a volume

```
docker volume rm <volume_name>
```

Example:

```
docker volume rm mysqldata
```

Or remove all unused volumes not currently attached to a running container:

```
docker volume prune
```

## 10.21 Bind mount a host directory

A directory on the host can be mounted into a container using `-v hostdir:containerdir`. For example, mount the current directory's `code` sub-directory into the container's `/home/app` directory:

```
-v $PWD/code:/home/app
```

`$PWD` references the current directory on Linux and macOS. It is not available on Windows so the full path must be specified using forward-slash notation, e.g.

```
-v /c/project/code:/home/app
```

## 10.22 Define a Docker network

Using a Docker network allows containers to communicate with each other using their container name (`--name`) rather than an IP address.

Create a network:

```
docker network create --driver bridge <network_name>
```

Example named `mynet`:

```
docker network create --driver bridge mynet
```

Attach the container to that network in `docker run` with `--net mynet`.

## 10.23 View networks

```
docker network ls
```

## 10.24 Delete a network

```
docker network rm <network_id_or_name>
```

Remove all unused networks not currently being used by a running container:

```
docker volume prune
```

## 10.25 View system disk usage

```
docker system df
```

## 10.26 Full clean start

Delete every unused container, image, volume, and network:

```
docker system prune -a --volumes
```

Add `-f` to force the wipe without a confirmation prompt.

# 11 Appendix B: Dockerfile reference

A `Dockerfile` is a plain-text file describing the steps to build an image, typically for your own application. To build a Docker image from a `Dockerfile` in the current directory, enter:

```
docker image build -t <image_name> .
```

The period at the end of the command references the current directory. Append `-f <file>` to use an alternative path or file name.

The syntax is described in the `Dockerfile` reference, but the commands below will be used most often.

## 11.1 # comment

Lines beginning with a hash `#` denote a comment:

```
# my comment
```

## 11.2 ARG arguments

Variables can be passed at build time with the `--builg-arg <name>=<value>` option. The value is imported with an `ARG` statement:

```
# get myvar
ARG myvar
```

A default value can be defined when none is passed:

```
# get myvar, with default of "myimage"
ARG myvar=myimage
```

Use its value elsewhere by prepending a `$` symbol:

```
RUN echo $myvar
```

Where the value must be set in part of a string, a `${name}` substitution can be defined:

```
FROM ${myvar}:latest
```

## 11.3 ENV environment variables

Set an environment variable:

```
ENV HOME=/home/app
```

Use its value elsewhere by prepending a `$` symbol or `${}` container:

```
RUN echo $HOME
RUN echo ${HOME}
```

## 11.4 FROM <image> starting image

Create a new image using an existing image as a starting point. This will usually be the first command in a `Dockerfile`:

```
# latest Node LTS on Alpine Linux
FROM node:lts-alpine
```

## 11.5 WORKDIR working directory

Set the working directory for any following `COPY`, `ADD`, `RUN`, `CMD` and `ENTRYPOINT` instructions:

```
WORKDIR /home/myapp
```

## 11.6 COPY files from the host to image

Using file patterns:

```
# copy all files to current folder
COPY . .

# copy all txt files to /doc/ folder
COPY *.txt /doc/

# copy all files and assign ownership to a user and group
# (Linux containers only)
COPY --chown=myuser:mygroup . .
```

## 11.7 ADD files

`ADD` is similar to `COPY` but also supports using URLs and tar files as the source. This may be useful, although using a `RUN` with chained `curl` and `tar` commands will create a smaller Docker image with fewer layers.

## 11.8 Mount a VOLUME

Create a mount point which can be accessed by other containers:

```
RUN mkdir -p /data/myvol
VOLUME /data/myvol
```

Multiple volumes can be specified:

```
VOLUME /myvol1 /myvol2 /myvol3
```

Example use: client-side HTML, CSS, JavaScript, and media files could be built in a **`static`** folder on a Node.js container. The files can be served directly by an NGINX web server running in another container.

## 11.9 Set a USER

Define the user (and optionally a group) to use for any following `RUN`, `CMD` and `ENTRYPOINT` instructions:

```
USER myuser
```

## 11.10 RUN a command

`RUN` issues instructions during the build, such as installation or configuration commands. Any number are permitted in the `Dockerfile`.

Execute commands using shell form:

```
RUN npm install
```

or `exec` (array) form which runs an executable directly:

```
RUN ["npm", "install"]
```

## 11.11 EXPOSE a port

Informs Docker that the container will listen on a specified network port. Note that the exposed port must be *published* with `-p` when using `docker run`:

```
EXPOSE 3000
```

## 11.12 CMD execute container

`CMD` sets a default application start instruction if none is specified when launching the container with `docker run`. Only one is permitted in the `Dockerfile`, so only the last is processed.

Commands can be executed using shell form:

```
CMD node ./index.js
```

or `exec` (array) form – the recommended option – to directly run the executable:

```
CMD ["node", "/index.js"]
```

In general, use `CMD` to provide a default command that can be overridden on the command line when the container is launched.

## 11.13 ENTRYPOINT execute container

`ENTRYPOINT` sets a start instruction when building an executable Docker image. The application will run when the container is launched. Only one is permitted in the `Dockerfile` and it overrides any `CMD` instructions.

Commands can be executed using shell form:

```
ENTRYPOINT node ./index.js
```

or `exec` (array) form – the recommended option – to directly run the executable:

```
ENTRYPOINT ["node", "/index.js"]
```

## 11.14 `.dockerignore` file patterns

Specifies filename patterns to omit when using `COPY` and `ADD`. Example content:

```
Dockerfile
docker*.yml

.git
.gitignore
.config

.npm
.vscode
node_modules

README.md
```

# 12 Appendix C: Docker Compose reference

Docker Compose is a utility for launching and managing multiple containers. It is more powerful and easier than using a series of `docker run` commands.

The configuration is usually defined in a `docker-compose.yml` (YAML) file. The Docker Compose reference provides full documentation, but the commands below will be used most often.

## 12.1 Docker Compose CLI

Docker Compose commands can be used in the same directory as your `docker-compose.yml` configuration file. The most-used include…

Launch all containers:

```
docker-compose up
```

Images can be rebuilt with `docker-compose up --build` or using `docker-compose build` first.

Specify an alternative directory and/or filename:

```
docker-compose -f ./my-config.yml up
```

> `-f` is necessary in the commands below if you are running `docker-compose` from a directory other than where the configuration YML file is stored.

Start as a background service:

```
docker-compose up -d
```

View active containers:

```
docker-compose ps
```

View container logs:

```
docker-compose logs
```

Pause running containers with:

```
docker-compose pause
```

then resume with:

```
docker-compose unpause
```

Restart all stopped and running containers:

```
docker-compose restart
```

Stop containers without removing them:

```
docker-compose stop
```

Start stopped containers:

```
docker-compose start
```

Remove stopped containers:

```
docker-compose rm
```

Stop and remove running containers, images, volumes, and networks:

```
docker-compose down
```

## 12.2 `docker-compose.yml` outline

YML is a standard data format using new lines, tabs, colons, and dashes to indicate sections and data.

All configuration files must specify:

1. Docker Compose version compatibility
2. the services (containers) to launch
3. networks (if used), and
4. volumes (if used).

```
version: '3'

services:

  # container
  mycontainer:
    # ...definition...


networks:

volumes:
```

The following sections describe common settings to configure service containers for Docker Compose v3 and above.

## 12.3 Starting `image`

Specify a starting repository image, e.g. for the `latest` MySQL:

```
  mycontainer:
    image: mysql
```

## 12.4 `build` an image from a `Dockerfile`

A new image can be built by specifying the relative path `context`, the name of the `dockerfile` in that location, and any associated build `args` passed to `Dockerfile` `ARG` instructions:

```
  mycontainer:
    build:
      context: ./
      dockerfile: Dockerfile
      args:
        - arg1=val1
        - arg2=val2
        - arg3=val3
```

## 12.5 Set the `container_name`

Set the container name. This can be used for inter-container communications across the same Docker network:

```
  mycontainer:
    container_name: mycontainer
```

## 12.6 Container `depends_on` another

Express a dependency between services to ensure one or more other containers have started before launching this one:

```
  mycontainer:
    depends_on:
      - containerA
      - containerB
```

## 12.7 Set `environment` variables

Define any number of individual environment variables:

```
  mycontainer:
    environment:
      - MYVAR1=value1
      - MYVAR2=value2
      - MYVAR3=value3
```

## 12.8 Set environment variables from a `env_file`

Sets of environment variables can be defined in a `.env` file, e.g.

```
# example values
MYVAR1=value1
MYVAR2=value2
MYVAR3=value3
```

All values can be imported using `env_file`:

```
  mycontainer:
    env_file: .env
```

Multiple files can also be specified:

```
  mycontainer:
    env_file:
      - ./one.env
      - ./two.env
      - ./three.env
```

## 12.9 Attach to Docker `networks`

Join one or more Docker networks (created on first use):

```
  mycontainer:
    networks:
      - mynetwork
```

It is also possible to set aliases (alternative hostnames) and IP addresses for the container on the network:

```
  mycontainer:
    networks:
      mynetwork:
        aliases:
          - myname1
          - myname2
        ipv4_address: 172.16.238.20
        ipv6_address: 2001:3984:3989::20
```

The networks (and optional configurations) must be referenced after the `services:` definition at the bottom of `docker-compose.yml`:

```
networks:
  mynetwork:
```

## 12.10 Attach persistent Docker `volumes`

Mount a Docker volume (created on first use) or bind a host directory:

```
  mycontainer:
    volumes:

      # Docker volume
      - type: volume
        source: rootfiles
        target: /var/www/html

      # bind host directory
      - type: bind
        source: ./myfiles
        target: /var/www/html/myfiles
```

A shorter syntax can also be used which defines `<source>:<destination>`. The `<source>` is presumed to be Docker volume unless it starts with `.` or `..` relative file paths.

```
  mycontainer:

    # idential to above
    volumes:
      - rootfiles:/var/www/html
      - ./myfiles:/var/www/html/myfiles
```

Docker volumes (and optional configurations) must be referenced after the `services:` definition at the bottom of `docker-compose.yml`:

```
volumes:
  rootfiles:
```

## 12.11 Set a custom dns server

Define one or more DNS servers:

```
  mycontainer:
    dns:
      - 1.1.1.1
      - 8.8.8.8
```

## 12.12 expose ports

Ports can be exposed to the host using “host:container” notation:

```
  mycontainer:
    expose:
      - "3000:3000"
```

Using a single container value, such as `"3000"`, only exposes the port to other containers on the same Docker network.

## 12.13 Define `external_links` to other containers

Containers started outside the current `docker-compose.yml` file can be referenced assuming they are on the same Docker network:

```
  mycontainer:
    external_links:
      - mysql
```

## 12.14 Override the default `command`

Override the `Dockerfile` `CMD` command using shell or `exec` (array) forms:

```
  mycontainer:
    command: node ./test.js
    # or [ "node", "./test.js" ]
```

## 12.15 Override the default `entrypoint`

Override the `Dockerfile` `ENTRYPOINT` command using shell or `exec` (array) forms:

```
  mycontainer:
    entrypoint: node ./test.js
    # or [ "node", "./test.js" ]
```

## 12.16 Specify a `restart` policy

Restart policies include:

- `no`: never restart the container (the default)
- `always`: always restart the container when it stops
- `on-failure`: restart the container when it fails with an exit code

```
  mycontainer:
    restart: on-failure
```

## 12.17 Run a `healthcheck`

Configure a check that is run periodically to check a container is alive and responding:

```
  mycontainer:
    healthcheck:
      test: [ "CMD", "curl", "-f", "http://localhost:3000/test" ]
      interval: 1m00s
      timeout: 10s
      retries: 3
```

## 12.18 Define a logging service

A log can be output using the `driver` of `"json-file"`, `"syslog"`, or `"none"`:

```
  mycontainer:
    logging:
      driver: syslog
      options:
        syslog-address: "tcp://192.168.1.100:1234"
```

# 13 Appendix D: quiz project

The example `quiz` project creates a browser-based general knowledge application using Docker containers:

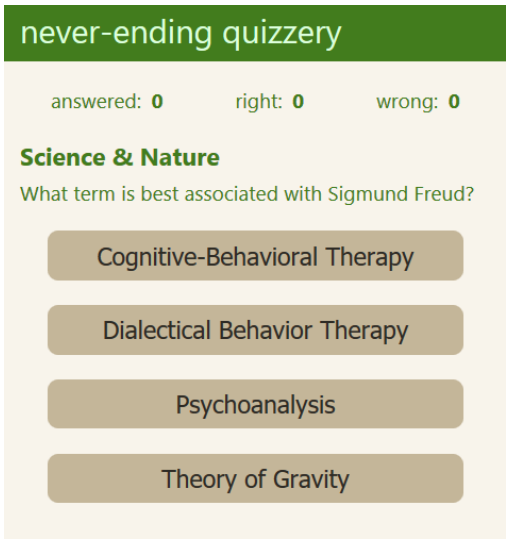



**Figure 13.1:** quiz application screen

It's intentionally simple and it's easy to cheat if you know a little about browser DevTools! However, similar Docker concepts will apply to most web applications.

> The files created in this chapter are contained in the `quiz` directory of the example code repository provided at https://github.com/craigbuckler/docker-web. Refer to README.md for quick start instructions.

## 13.1 Project overview

Three containers are connected to a `quiznet` Docker network:

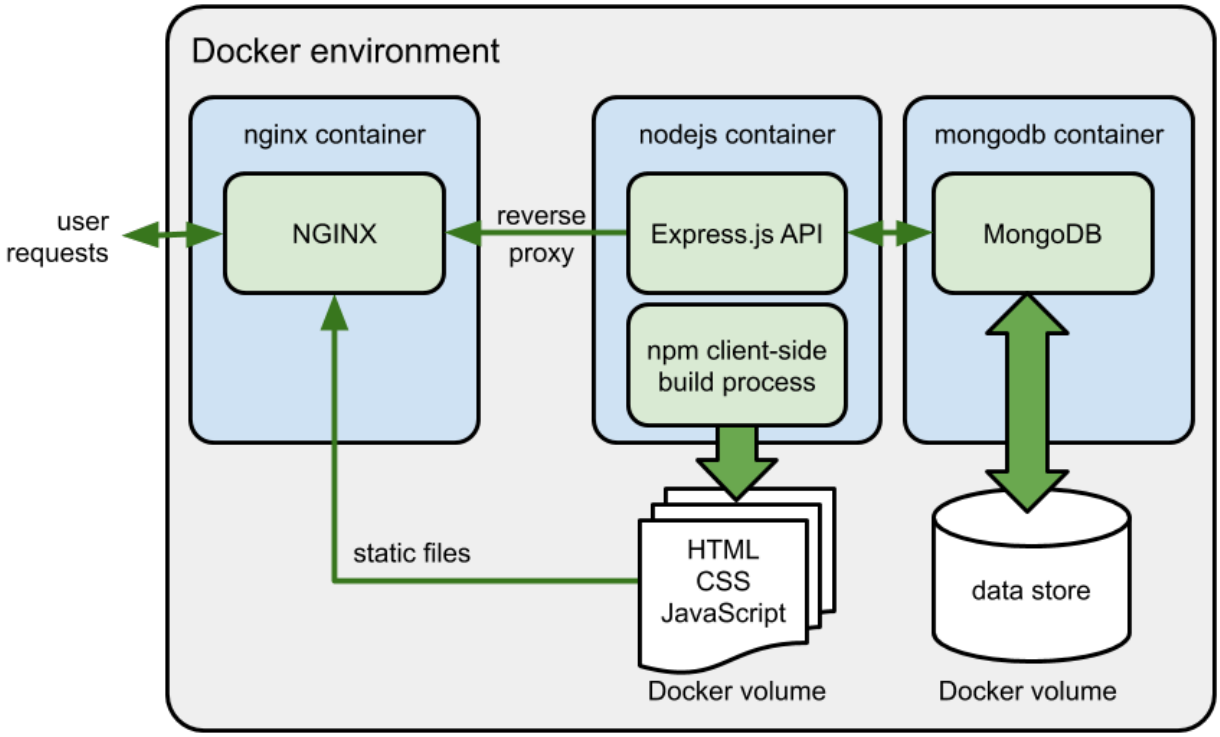



**Figure 13.2:** quiz application screen

### 13.1.1 MongoDB database (mongodb container)

This uses a Docker volume (`quizdata`) for question storage and exposes the database on port `27017`.

A collection named `quiz` is created in the `quiz` database which is accessed with the user ID `quizuser` and password `quizpass`.

### 13.1.2 Node.js application (`nodejs` container)

The Node.js container has two functions:

1. `index.js` launches an Express.js application exposed on port `8000` which:
   - fetches questions and answers from the Open Trivia Database API on start-up. These are formatted and stored in the MongoDB database.
   - provides a single `/question` endpoint. This queries the database and returns details of the next question as JSON-encoded data.

   During development, any changes to the Node.js files trigger an application restart using Nodemon.
2. Client-side HTML, CSS, and JavaScript implement the application logic to fetch questions and keep score. Source code contained in the `src` directory is built using npm scripts which run pug, PostCSS, and Rollup.js. The resulting files are output to a **`static`** directory (a shared Docker volume).

   In production, this processing occurs when the Docker image is built. During development, changes to `src` files in the host's project directory trigger a **`static`** file re-build and source maps are appended.

### 13.1.3 NGINX reverse proxy (`nginx` container)

An NGINX web server container serves files from the shared **`static`** volume. This is more efficient than serving via Express.js.

Other HTTP requests (such as the `/question` endpoint) are forwarded to the `nodejs` container. NGINX acts as a *reverse proxy*.

## 13.2 Launch in development mode

Launch the quiz in development mode with auto-building, source maps, and application restarts using the `docker-compose.yml` configuration in the `./quiz` project root:

```
cd quiz
docker-compose up
```

Access the application via NGINX at http://localhost:8080/

Additionally, you can access:

- the Node.js application directly at http://localhost:8000/
- the Node.js debugger at http://localhost:9229/
- the `quiz` database using a MongoDB client by connecting to http://localhost:27017/ with the user ID `quizuser` and password `quizpass`.

> Free MongoDB client applications include Compass, Robo 3T, and Mongoku (also available as a Docker image).
>
> Alternatively, access the container shell: `docker exec -it mongodb sh`
>
> and launch the MongoDB CLI: `mongo -u quizuser -p quizpass`
>
> Then issue queries and commands, e.g.
>
> ```
> use quiz;
> show collections;
> db.quiz.find({}, { _id:0, question:1 });
> ```

## 13.3 Launch in production mode

Each `Dockerfile` builds a production image. These can be launched without modification using the `docker-compose-production.yml` configuration in the `./quiz` project root:

```
cd quiz
docker-compose -f ./docker-compose-production.yml up
```

Append `-d` to run the quiz as a background process.

Access the application via NGINX at http://localhost:8080/

Direct access to other containers is not permitted.

> The default HTTP port `80` can be set at line 54 of `docker-compose-production.yml`. However, the container will fail to start if you have other applications using that port, such as another web server or Skype.

## 13.4 Clean up

To stop the quiz application, press `Ctrl` | `Cmd + C` or enter `docker-compose down` in another terminal (`cd` to the same directory).

The application's Docker containers, images, volumes, and networks can be removed with:

```
docker-compose rm
docker volume prune -f
docker image rm quiz_nodejs quiz_nginx
```

Alternatively, you can wipe all Docker data including base images:

```
docker system prune -af --volumes
```

## 13.5 Project file structure

The project root directory contains:

| file | description |
|---|---|
| `README.md` | technical information |
| `docker-compose.yml` | Docker Compose development mode configuration |
| `docker-compose-production.yml` | Docker Compose production mode configuration |

The `nginx` sub-directory contains:

| file | description |
|---|---|
| `nginx.Dockerfile` | the Dockerfile used to build the `nginx` Docker image |
| `nginx.conf` | the NGINX configuration file copied to the Docker image |

The `nodejs` sub-directory contains:

| file | description |
|---|---|
| `nodejs.Dockerfile` | the Dockerfile used to build the `nodejs` Docker image |
| `.dockerignore` | paths to omit when copying files to the `nodejs` Docker image |
| **`package`**`.json` | application dependencies and npm build scripts |

| file | description |
|---|---|
| `.env` | environment variables for database access |
| `index.js` | the main Express.js application |
| `lib/*` | custom modules used by the Express.js application |
| `src/*` | client-side HTML, CSS, and JavaScript source files |
| `nodemon.json` | Nodemon configuration for Express.js restarts |
| `postcss.config.js` | PostCSS configuration for processing CSS `src` files |
| `rollup.config.js` | Rollup.js configuration for processing JavaScript `src` files |

## 13.6 `nodejs` Docker image

The `nodejs.Dockerfile` creates a production Docker image which:

1. uses the tiny Node 14 Alpine Docker Hub image as a base
2. sets environment variables, including `NODE_ENV=production`
3. creates a working directory (`/home/node/app`) and grants access to the `node` user
4. copies **`package`**`.json` and installs modules
5. copies the remaining application files
6. runs the build script. This creates a **`static`** directory for client-side files which is mounted as a Docker volume and shared with other containers
7. exposes port `8000`, and
8. runs the Express.js application (`index.js`).

```
# base Node.js v14 image
FROM node:14-alpine

# environment variables
ENV NODE_ENV=production
ENV NODE_PORT=8000
ENV HOME=/home/node/app
ENV PATH=${PATH}:${HOME}/node_modules/.bin

# create application folder and assign rights to the node user
RUN mkdir -p $HOME && chown -R node:node $HOME

# set the working directory
WORKDIR $HOME

# set the active user
USER node

# copy package.json from the host
COPY --chown=node:node package*.json $HOME/

# install application modules
RUN npm install

# copy remaining files and build
COPY --chown=node:node . .
RUN npm run build

# share volume
VOLUME ${HOME}/static

# expose port on the host
EXPOSE $NODE_PORT

# application launch command
CMD [ "node", "./index.js" ]
```

`docker-compose.yml` overrides some `nodejs.Dockerfile` settings when running in development mode:

1. `NODE_ENV` is set to `development`
2. the project's `nodejs` sub-directory on the host is mounted to the container's `/home/node/app` directory so file changes can be monitored
3. port `9229` is also exposed so a Node.js debugger can be attached.
4. the application is launched by installing dependencies and running `npm run debug`.

```
  nodejs:
    environment:
      - NODE_ENV=development
    build:
      context: ./nodejs
      dockerfile: nodejs.Dockerfile
    container_name: nodejs
    depends_on:
      - mongodb
    volumes:
      - ./nodejs:/home/node/app
      - nodejsfiles:/home/node/app/static
      - nodejsfiles:/home/node/app/.config
      - nodejsfiles:/home/node/app/.npm
      - nodejsfiles:/home/node/app/node_modules
    networks:
      - quiznet
    ports:
      - "8000:8000"
      - "9229:9229"
    command: /bin/sh -c 'npm i && npm run debug'
```

> Node.js modules and npm data would normally be installed in the `nodejs` project directory on the host. Unfortunately, this can be slow on Windows file systems and you may encounter permission issues on macOS or Linux when attempting to write data to the **`static`** directory.
>
> A `nodejsfile` Docker volume is mounted to persistently store generated files. This improves reliability and the initial start-up speed.

`docker-compose-production.yml` retains the production `nodejs.Dockerfile` configuration. Port `8000` is only exposed within the Docker network and the container will restarts when it fails:

```
  nodejs:
    build:
      context: ./nodejs
      dockerfile: nodejs.Dockerfile
    container_name: nodejs
    depends_on:
      - mongodb
    networks:
      - quiznet
    ports:
      - "8000"
    restart: on-failure
```

## 13.7 `nginx` Docker image

The `nginx.Dockerfile` creates a production Docker image which:

1. uses the tiny stable Alpine Docker Hub image as a base
2. copies the `nginx.conf` configuration file, and
3. exposes the default HTTP port `80`.

```
FROM nginx:stable-alpine

COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 80
```

`nginx.conf` sets NGINX to listen for requests on port 80. When a URL is received, it attempts to locate a suitable file in the `/home/node/app/static/` shared Docker volume defined by the `nodejs` container. If nothing suitable is found, the request is forwarded to `http://nodejs:8000` to be processed by the Express.js application:

```
  # HTTP requests
  server {

    listen 80;
    listen [::]:80;

    server_name localhost; # domain

    # is a static file?
    location / {
      root /home/node/app/static/;
      index index.html;
      try_files $uri $uri/ @nodejs;
    }

    # reverse-proxy to Node.js app
    location @nodejs {
      proxy_pass http://nodejs:8000;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection 'upgrade';
      proxy_set_header Host $host;
      proxy_cache_bypass $http_upgrade;
    }

  }
```

The Docker Compose configuration is similar in both production and development modes (`restart` is only set in production):

```
  nginx:
    build:
      context: ./nginx
      dockerfile: nginx.Dockerfile
    container_name: nginx
    depends_on:
      - nodejs
    networks:
      - quiznet
    ports:
      - "8080:80"
    restart: on-failure
    logging:
      driver: "none"
```

> Setting the `logging` driver to `"none"` disables terminal logging. NGINX access logs are less useful during development.

## 13.8 mongodb Docker image

The latest MongoDB image is used directly from Docker Hub. The Docker Compose configuration is similar in both production and development modes:

```
  mongodb:
    environment:
      - MONGO_INITDB_ROOT_USERNAME=quizuser
      - MONGO_INITDB_ROOT_PASSWORD=quizpass
    image: mongo:latest
    container_name: mongodb
    volumes:
      - quizdata:/data/db
    networks:
      - quiznet
    ports:
      - "27017:27017"
    logging:
      driver: "none"
```

In production, port `"27017"` is only exposed within the Docker network and a `restart: on-failure` setting is added.

> Ideally, the user ID and password should be stronger! The `./nodejs/.env` file would need changing accordingly.

## 13.9 Node.js build process

Client-side files in the `src` directory are processed using npm scripts and output to a **`static`** directory. This is shared as a Docker volume so NGINX can serve HTML, CSS, and JavaScript requests without Express.js processing.

### 13.9.1 Production mode build

Production environment `build` scripts are defined in the **`package`**`.json "scripts"` section:

```
"build:htm":
  "pug -O ./src/html/data.json ./src/html/ -o ./static/",

"build:css":
  "postcss src/css/main.css -o static/css/main.css --no-map",

"build:es6":
  "rollup --config",

"build":
  "npm run build:htm && npm run build:css && npm run build:es6",

"start":
  "npm run build && node ./index.js"
```

`npm run build` is executed when the Docker image is built. This processes `src` files using pug, PostCSS, and Rollup.js to create optimised HTML, CSS, and JavaScript files in the **`static`** directory.

### 13.9.2 Development mode watch and build

Development mode `watch` scripts are also defined in the **`package`**`.json "scripts"` section:

```
"watch:htm":
  "pug -O ./src/html/data.json ./src/html/ -o ./static/ --pretty -w",

"watch:css":
  "postcss ./src/css/main.css -o ./static/css/main.css
    --map --watch --poll --verbose",

"watch:es6":
  "rollup --config --sourcemap inline --watch --no-watch.clearScreen",

"watch:app":
  "nodemon --trace-warnings --inspect=0.0.0.0:9229 ./index.js",

"debug":
  "concurrently 'npm:watch:*'",
```

pug, PostCSS, and Rollup.js *watch* options are used to monitor files and re-build when necessary. Similarly, Nodemon watches for Node.js application file changes and restarts accordingly.

`npm run debug` is executed by Docker Compose. This launches Concurrently which executes all npm `watch:*` scripts in parallel.

## 13.10 Node.js Express.js application

The primary `index.js` application uses `dotenv` to parse database settings in the `.env` file and define environment variables:

```
// main application

'use strict';

// load environment
require('dotenv').config();
```

The `./lib/db.js` module then initializes the database and exports a single `getQuestion()` function:

```
const
  // initialize database
  quizDB = require('./lib/db'),
```

Express.js is initialized with middleware functions to set the **`static`** directory, permit access from other domains using CORS, and ensure requests are never cached:

```
  // default HTTP port
  port = process.env.NODE_PORT || 8000,

  // express
  express = require('express'),
  app = express();


// static files
app.use(express.static('./static'));

// header middleware
app.use((req, res, next) => {

  res.set({
    'Access-Control-Allow-Origin': '*',
    'Cache-Control': 'must-revalidate, max-age=0'
  });
  next();

});
```

> It should not be necessary to set the `express.static` directory because NGINX will serve the files directly. However, you may need to connect to the Node.js application directly at http://localhost:8000/ or http://localhost:9229/ when debugging.
>
> You could also consider running the command in development mode by adding the condition:
>
> ```
> if (process.env.NODE_ENV !== 'production')
> ```

A single GET `/question` endpoint is defined which returns the next question in JSON format by calling `getQuestion()` in the `./lib/db.js` module:

```
// route: fetch a question
app.get('/question', async (req, res) => {

  const q = await quizDB.getQuestion();

  if (q) res.json(q);
  else res.status(500).send('service unavailable');

});
```

Finally, Express.js is launched on port `8000`:

```
// start HTTP server
app.listen(port, () =>
  console.log(`page hit web service running on port ${port}`)
);
```

### 13.10.1 `/lib/db.js` module

The `db.js` module defines several constants to control how many questions are required and fetched from the Open Trivia Database API:

```
'use strict';

const

  maxQuestions = 300,
  maxApiFetch = 50,
  maxApiCalls = 10,
```

A connection to the MongoDB database is made using the credentials supplied in the `.env` file:

```
  // MongoDB connect
  mongo = require('mongodb'),

  client = new mongo.MongoClient(
    `mongodb://${ process.env.MONGO_USERNAME }` +
    `:${ process.env.MONGO_PASSWORD }` +
    `@${ process.env.MONGO_HOST }:${ process.env.MONGO_PORT }/`,
    { useNewUrlParser: true, useUnifiedTopology: true }
  );
```

The `quiz` collection in the `quiz` database is referenced:

```
// database connection objects
let db, quiz;

// connect to MongoDB database
(async () => {

  try {

    await client.connect();
    db = client.db( process.env.MONGO_DB );

    // quiz collection
    quiz = db.collection('quiz');
```

The `init()` function is called which returns the number of questions available in the database:

```
    // initialize
    if (!await init()) {
      throw 'no questions in database';
    }

  }
  catch (err) {
    console.log('database error', err);
  }

})();
```

### 13.10.2 `./lib/db.js init()` function

`init()` fetches the current number of questions in the `quiz` collection and returns immediately when that is equal or greater to the `maxQuestions` constant (300).

```
// initialize database
async function init() {

  // question count
  let qCount = await quiz.countDocuments();
  if (qCount >= maxQuestions) return qCount;
```

Indexes are added to the `quiz` collection when it is empty:

```
  console.log('initializing quiz database...');

  // create indexes
  if (!qCount) {

    await quiz.createIndexes([
      { key: { category: 1 }},
      { key: { question: 1 } },
      { key: { used: 1 } }
    ]);

  }
```

Random questions from the Open Trivia Database API are retrieved using one or more concurrent HTTP `node_fetch` calls.

The question and answer data is converted from JSON, cleaned, and appended to a new object (`newQ`):

```
  const
    fetch = require('node-fetch'),
    lib = require('./lib'),
    batch = quiz.initializeUnorderedBulkOp(),

    maxReq = Math.min(maxApiFetch, maxQuestions - qCount),
    quizApi =
      `https://opentdb.com/api.php?type=multiple&amount=${ maxReq }`;

  (await Promise.allSettled(
    // make multiple API calls
    Array( Math.min(
        maxApiCalls,
        Math.ceil((maxQuestions - qCount) / maxReq)
      ) )
      .fill(quizApi)
      .map((u, i) => fetch(`${u}#${i}`))
  )
    .then(
      // parse JSON
      response => Promise.allSettled(
        response.map(res => res.value && res.value.json())
      )
    )
    .then(
      // extract questions
      json => json.map(j => j && j.value && j.value.results || [])
    ))
    .flat().forEach(q => {
      // format each question
      let
        correct = lib.cleanString(q.correct_answer),
        newQ = {
          category: lib.cleanString(q.category),
          question: lib.cleanString(q.question),
          answers:  q.incorrect_answers
                      .map(i => lib.cleanString(i))
                      .concat(correct).sort()
        };

      newQ.correct = newQ.answers.indexOf(correct);
```

> Note that `lib.cleanString()` is defined in `./lib/lib.js`. It removes unnecessary whitespace and converts `&` characters to HTML `&` entities.

`newQ` is inserted into the database using a bulk operation. A new document will be created if the question has not been added before:

```
      // database insert
      batch
        .find({ question: q.question })
        .upsert()
        .update({ $set: newQ });

    });
```

The database is updated and the total of new and existing questions is returned:

```
  // update database
  const
    dbUpdate = await batch.execute(),
    qAdded = dbUpdate.result.nUpserted;

  qCount += qAdded;

  console.log(`${ qAdded } questions added`);
  console.log(`${ qCount } questions available`);

  return qCount;

}
```

### 13.10.3 `./lib/db.js getQuestion()` function

`getQuestion()` is exported for use by calling modules.

The MongoDB `quiz` collection is ordered by each document's `used` value in ascending order so the first least-used question can be returned. That document has its `used` value incremented:

```
// get next question
module.exports.getQuestion = async () => {

  const
    nextQ = await quiz.findOneAndUpdate(
      {},
      { $inc: { used: 1 }},
      {
        sort: { used: 1 },
        projection:
          { _id: 0, category: 1, question: 1, answers: 1, correct: 1 }
      }
    );

  return (nextQ && nextQ.ok && nextQ.value) || null;

};
```

## 13.11 Client-side files

The quiz functionality is implemented in the client-side HTML, CSS, and JavaScript. Files contained in the `src` directory are built to the **`static`** directory.

### 13.11.1 HTML page

`./static/index.html` is generated from the `./src/html/index.pug` template and data defined in `./src/html/data.json`. The score, category, question, and four answer buttons are output. The file is minified in production mode.

### 13.11.2 CSS styles

`./static/css/main.css` is generated from `./src/css/main.css`. All `@import` references are merged into a single minified file. An inline source map is added in development mode.

### 13.11.3 JavaScript functionality

`./static/js/main.js` is generated from `./src/js/main.js`. This contains vanilla ES6/2015 code which is minified and used directly as a `<script type="module">` (this will work in all modern browsers but not Internet Explorer). An inline source map is added in development mode.

The code defines the question API URL:

```
// question API URL
const questionAPI = '/question';
```

and a `state` object which fetches the associated DOM node and sets an initial zero value:

```
// initialize state
const state = {};

'answered,right,wrong,category,question,a0,a1,a2,a3'
  .split(',')
  .forEach(prop => {
    state[prop] = {
      node: document.getElementById(prop),
      value: 0
    };
  });
```

An event handler listens for clicks on the `#answers` element which contains all the answer buttons:

```
// answer event handler
document
  .getElementById('answers')
  .addEventListener('click', answerHandler, false);
```

The first question is then fetched:

```
// start first question
(async () => await newQuestion())();
```

The `newQuestion()` function fetches the next question, updates the state object, renders those items to DOM content, and stores the ID of the correct answer:

```
// new question
async function newQuestion() {

  const question = await fetchQuestion();

  // populate state
  for (const prop in question) {
    if (state[prop] && state[prop].node) {
      state[prop].value = question[prop];
    }
  }

  // render question
  for (const prop in state) {
    if (state[prop].node) {
      state[prop].node.innerHTML = state[prop].value;
    }
  }

  // correct answer
  state.correct = 'a' + question.correct;

}
```

The `fetchQuestion()` function uses the Fetch API to retrieve the next question from the Node.js application. The 4-element `answers` array is converted to properties `a0`, `a1`, `a2`, and `a3` to match the `state` object.

```
// next question
async function fetchQuestion() {

  const
    call = await fetch(questionAPI),
    q = await call.json();

  q.answers.forEach((a, i) => { q['a'+i] = a; });
  delete q.answers;

  return q;

}
```

The `answerHandler()` function activates when an element within the `#answers` element is clicked. The function exits if an answer has already been received (`state.done` is **`true`**) or a button was not clicked:

```
// answer clicked
function answerHandler(e) {

  const clicked = e.target;
  if (state.done || clicked.nodeName !== 'BUTTON') return;

  state.done = true;
```

The ID of the correct answer button is determined, the number answered is incremented, and the clicked button has its focus unset:

```
  const correct = state[state.correct].node;

  state.answered.value++;
  clicked.blur();
```

CSS classes are added to indicate whether the answer was correct, highlight the right answer, and update the `state` counters:

```
  if (clicked === correct) {
    state.right.value++;
    clicked.classList.add('right');
  }
  else {
    state.wrong.value++;
    clicked.classList.add('wrong');
    correct.classList.add('right', 'reveal');
  }
```

After a three-second timeout, all the classes are reset and a new question is triggered. The game continues forever until the user becomes bored!

```
  setTimeout(async () => {

    clicked.classList.remove('right', 'wrong');
    correct.classList.remove('right', 'reveal');

    await newQuestion();
    state.done = false;

  }, 3000);

}
```

## 13.12 Key points

This quiz illustrates how to develop a complex multi-container web applications with build systems. Consider creating an application using your favorite language, database, and associated dependencies. For example:

1. the obligatory to-do app
2. a web-based notes system (bonus points for markdown support)
3. a recipe database containing ingredients, steps, and search – ideally based on ingredients you have!
4. an online poll system to create and host simple surveys
5. a real-time chat app.

   Consider how multiple Docker containers would permit horizontal scaling. People chatting in the same *room* could be connected to different containerized applications running on different servers. A backend redis, database, or queue system could broadcast messages to other containers.

*Best of luck!*