

LINUX CONTAINERS AND KUBERNETES

01



Table of contents

1. Why containers?	7
Managing infrastructure at scale	12
Underutilised compute resources	17
Poor resource allocation	24
Proliferation of languages and frameworks	33
Containers in the shipping industry	37
Linux containers	38
What are containers made of?	42
 2. Containers and Docker	 46
Running containers	53
Building Docker images	55
Managing containers	58

Sharing the file system	60
COPYing files	61
Forwarding ports	63
Managing configuration	69
Attaching to running containers	70

3. Installing Docker **72**

Windows 10	74
macOS	81
Ubuntu	84

4. Getting started with Docker **89**

Running Docker containers	91
Docker registries	92
Running commands inside containers	93

Start, stop, list, kill	96
-------------------------	----

Terminology	101
-------------	-----

5. Creating images **102**

Your first Dockerfile	104
-----------------------	-----

Build the image	106
-----------------	-----

6. Managing state **109**

Sharing the file system	111
-------------------------	-----

Bidirectional sync	113
--------------------	-----

Baking files in the image	114
---------------------------	-----

7. Networking **118**

Connecting to the container	120
-----------------------------	-----

Bind container ports to the host	122
----------------------------------	-----

8. Packaging an application **124**

Hello World, Node.js	126
----------------------	-----

Creating an image	131
-------------------	-----

Configurations	134
----------------	-----

9. Debugging **136**

Attaching to a running container	138
----------------------------------	-----

Common problems with attaching to containers	141
--	-----

10. Labs **142**

11. Inspecting a container **144**

12. Attention to details	146
13. Building walls	148
14. Encrypted message	150
15. Debugging from scratch	153

Chapter 1

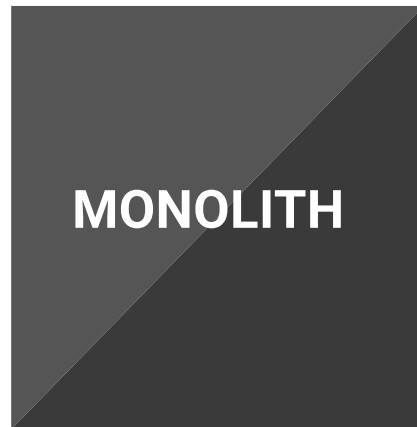
Why containers?

During the last few years, the industry has experienced a shift toward developing smaller and more focused applications. It doesn't come as a surprise that more and more companies are breaking their massive and static monoliths into a set of decoupled and independent components. *And rightly so.* Services that are tiny in size are:

- **quicker to deploy** because you create and release them in smaller chunks
- **easier to iterate on**, since adding features happens independently
- **resilient** — the overall service can still function despite one of the components not being available

Please bear in mind that developing smaller components have disadvantages too. As an example, managing dependencies and versioning is complicated when you have several components interacting with each other.

But how does that cultural shift impact the infrastructure?



1/5

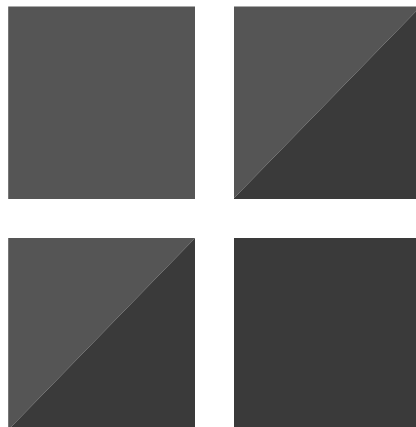
Applications packaged as a single unit as usually referred to as monoliths.

WHY CONTAINERS?



2/5

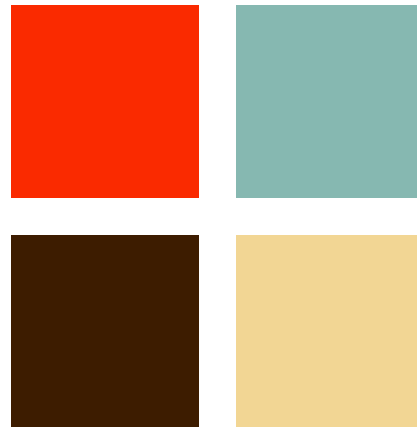
You might have started developing applications as a single units and then decided to break them down into smaller components.



3/5

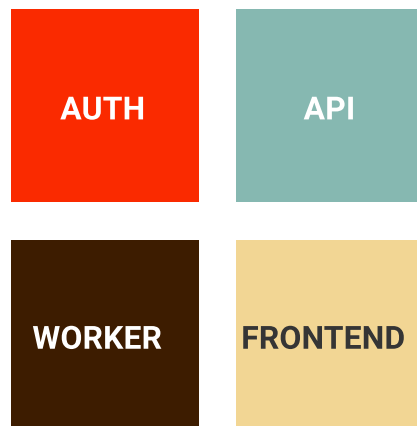
Or you may have inherited a large application from someone else and decided to separate several concerns into different modules.

WHY CONTAINERS?



4/5

At the end of the process you end up with a collection of smaller, more focussed apps.



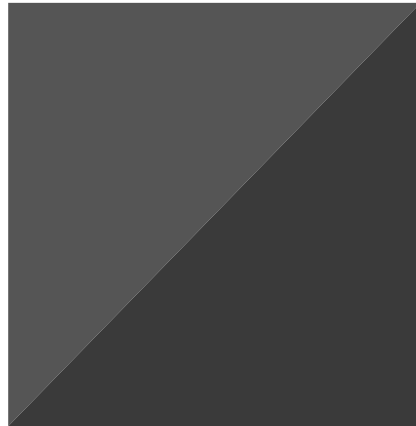
5/5

From a single deployable unit to four. Is the change impacting how you deploy the applications?

1. Managing infrastructure at scale

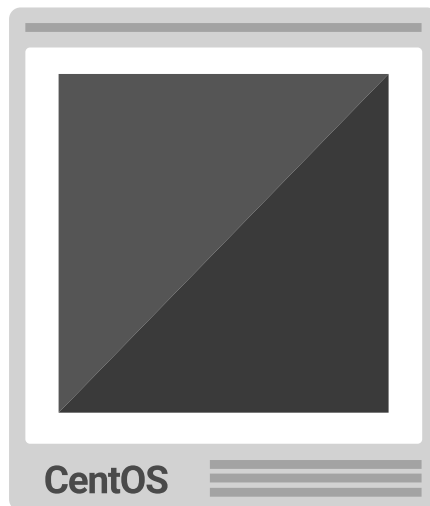
Developing services out of smaller components, in fact, introduces a different challenge. In the past, you might have deployed applications in virtual machines. When for every application, you can refactor the same app in a collection of four components, **you have at least four times more apps to develop, package and release.** You also have three more virtual machines to manage. It's not uncommon for a small service to be made out of a dozen components such as a front-end app, a backend API, an authorisation server, or an admin portal.

WHY CONTAINERS?



1/6

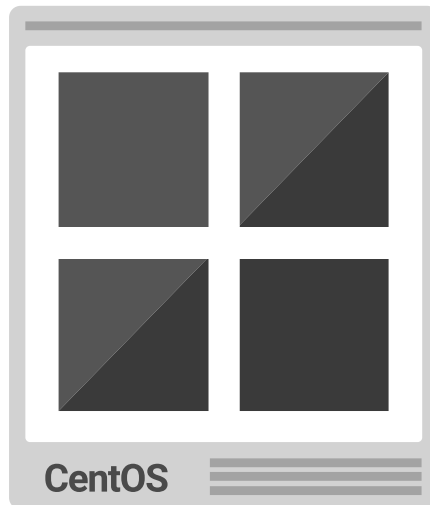
When you have a single large application,
you can deploy it inside a virtual machine.



2/6

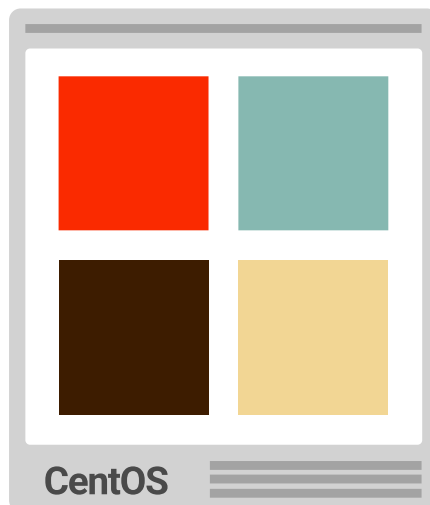
When you have a single large application,
you can deploy it inside a virtual machine.

WHY CONTAINERS?



3/6

As soon as you decide to break the application into smaller components, you should find a way to isolate those components so that failures as segregated and scaling happens independently.



4/6

As soon as you decide to break the application into smaller components, you should find a way to isolate those components so that failures as segregated and scaling happens independently.

WHY CONTAINERS?



5/6

You might be tempted to wrap each app into a separate virtual machine.



6/6

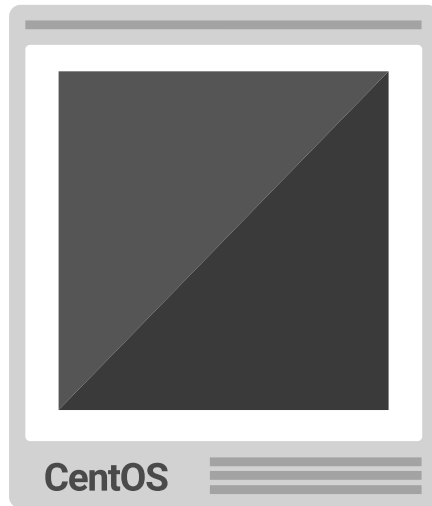
When you have several components, you end up using more virtual machines than you did previously with a single unit.

You might wonder what's the issue with having several virtual machines.

2. Underutilised compute resources

Each virtual machine comes with an operating system that consumes part of the memory and CPU resources allocated to it. When you create a t2.micro Elastic Compute Cloud instance in Amazon with 1GB of memory and 1 vCPU, you end up using 700MB in memory, and 80% of the CPU after you remove the overhead of the operating system. Similarly, when you request for four larger instances such as a t2.xlarge with 4 vCPU and 16 GiB of memory each, you will end up wasting about 3 vCPU and 1.2GiB of memory in total. The overhead adds up to about 2 to 3%. *Not a lot.* When you refactor the four applications in a collection of 4 smaller components, you have 16 virtual machines deployed in your infrastructure. Each virtual machine has an operating system that wastes 300MiB of memory and 0.2 vCPU. In total, you waste 4.8GiB of memory (300MiB times 16 apps) and 3.2 vCPU (0.2 vCPU times 16 apps). **That's a 6 to 10% of resources you pay for, but can't use.** *If your cloud bill is \$1000 per month, you are wasting \$100 per month in running operating systems.* Let's have a look at an example.

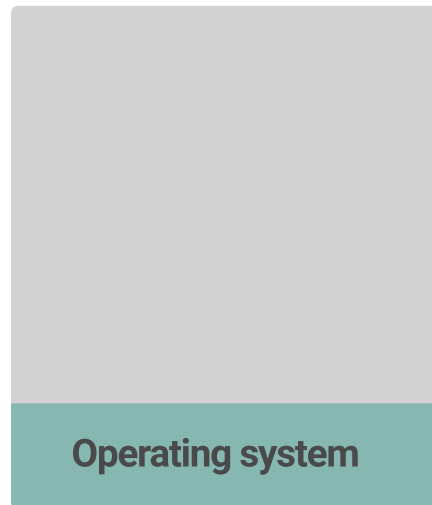
WHY CONTAINERS?



1/9

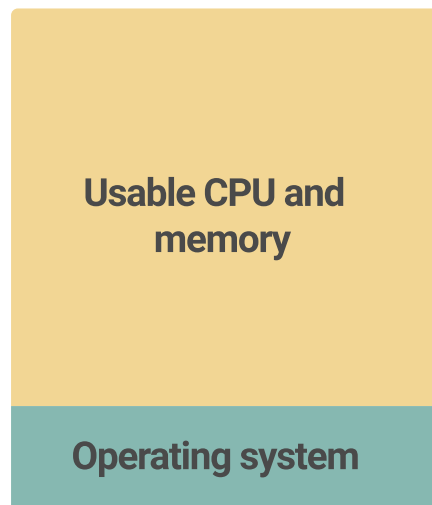
Consider the following virtual machine.

WHY CONTAINERS?



2/9

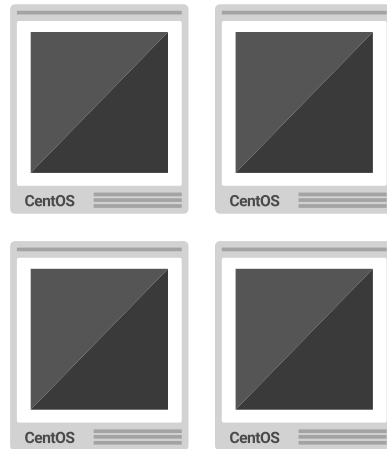
Part of the resources of that virtual machine are used for the operating system.



3/9

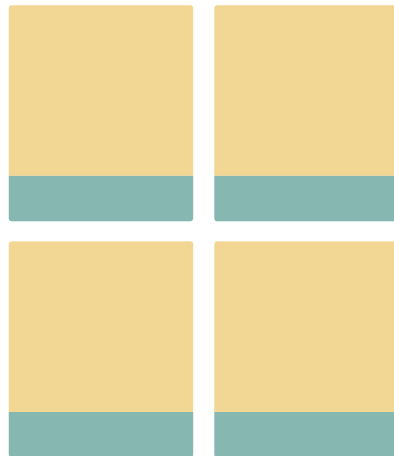
The application can use the rest of the CPU and memory.

WHY CONTAINERS?



4/9

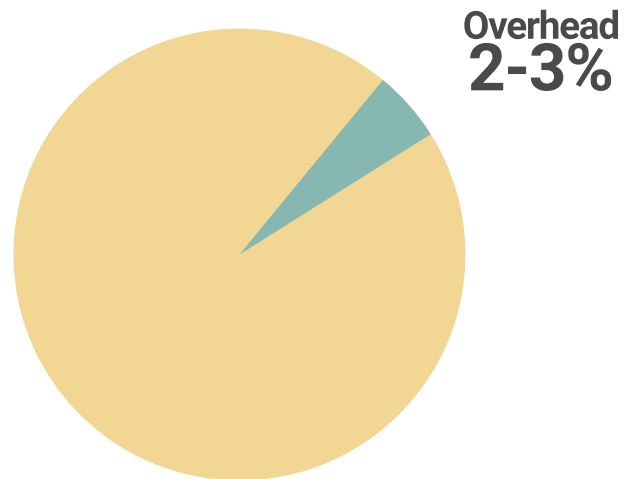
When you develop fewer and larger apps, you have a limited number of virtual machines deployed in your infrastructure. They also tend to use compute resources with higher specs.



5/9

Few large virtual machines with operating systems don't have a significant impact on the resource utilisation. The overhead from running the operating system is between 2 and 3%.

WHY CONTAINERS?



6/9

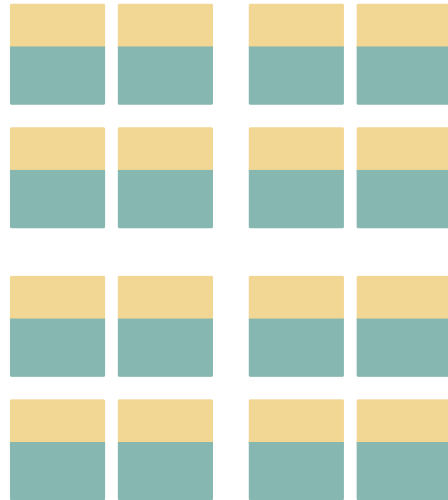
Now imagine having the four large units broken down into a collection of smaller components. Each component has its virtual machine.



7/9

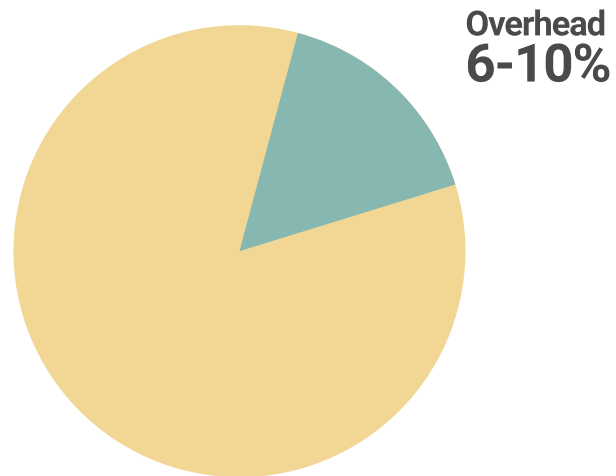
Each virtual machine has an operating system. Where previously you had four, now you have 16.

WHY CONTAINERS?



8/9

Having several virtual machines with generally more modest specs means that the operating system uses (in proportion) more resources.



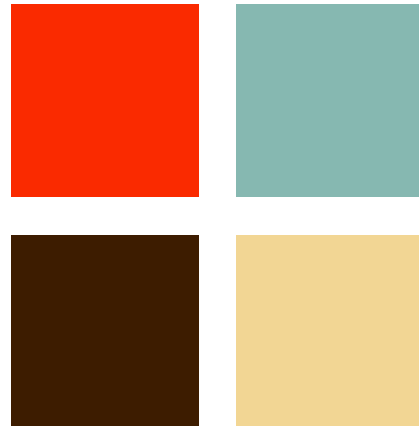
9/9

The overhead is more significant too. Depending on your setup it's not uncommon to waste 10% of resources in running operating systems.

However, the operating system overhead is only part of the issue.

3. Poor resource allocation

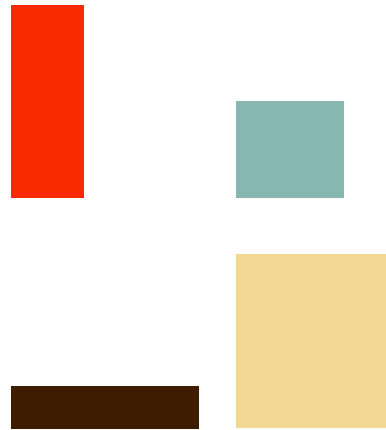
You have probably realised that when you break your service into smaller components, each of them comes with different resource requirements. Some components such as data processing and data mining applications are CPU intensive. Others, such as servers for real-time applications might use more memory than CPU.



1/5

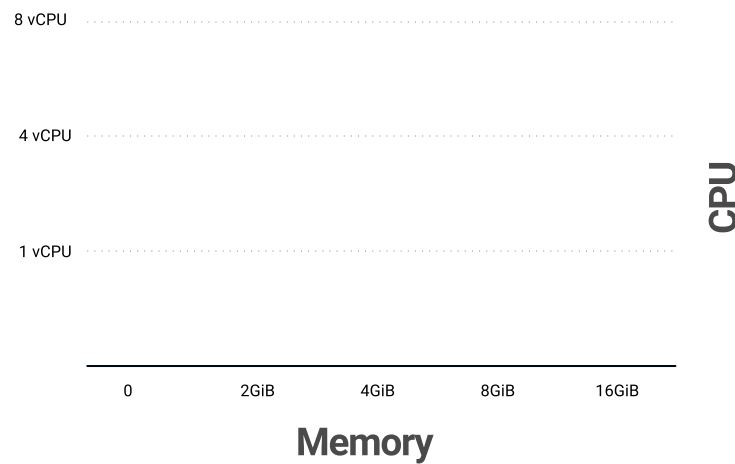
When you develop apps as a collection of smaller components, you realised that no two apps are alike.

WHY CONTAINERS?



2/5

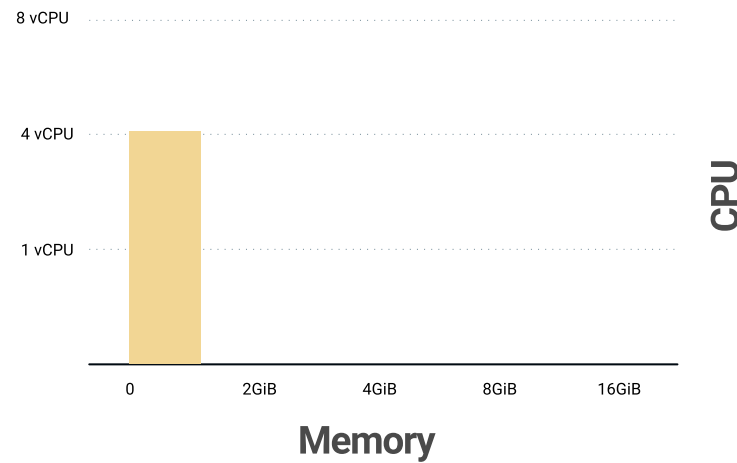
They all look different. Some of them are CPU intensive other require more memory. And you might have apps requiring specific hardware such as GPUs.



3/5

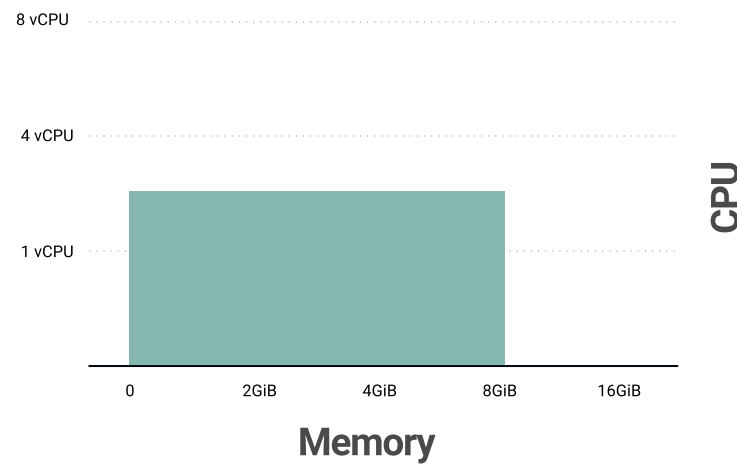
You can imagine being able to measure those apps.

WHY CONTAINERS?



4/5

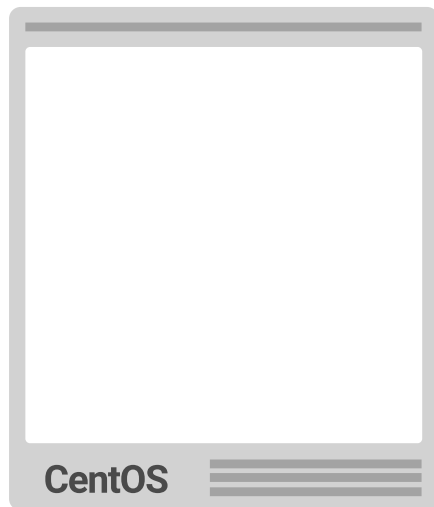
Some of them could be more CPU heavy than others.



5/5

Or you could have components that use similar CPU resources but a lot more memory.

Ideally, you should strive to use the right virtual machine that fits for your component's requirements. If you have an app that uses 800MiB of memory, you don't want to use a virtual machine that has 2GiB. You're probably fine with one that has 1GiB. In practice, this is not always the case. It's much easier to select a single virtual machine that is good enough in 80% of the cases and use it all the times. *The result?* **You waste hundreds of megabytes of RAM and plenty of CPU cycles in underutilised hardware.**



1/5

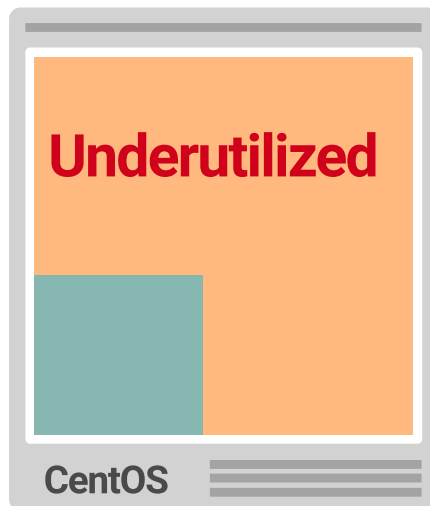
Imagine having a virtual machine with 4 GiB of memory and 4 vCPU.

WHY CONTAINERS?



2/5

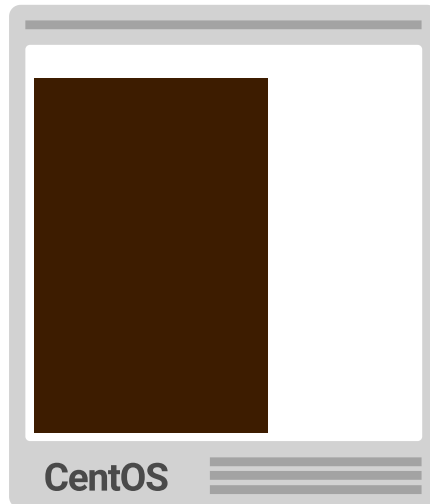
You can deploy an application that uses only 1GiB of memory and 1 vCPU, but you're wasting 3/4 of the resources.



3/5

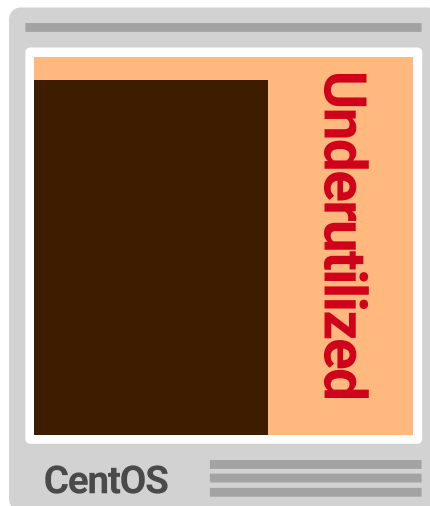
You can deploy an application that uses only 1GiB of memory and 1 vCPU, but you're wasting 3/4 of the resources.

WHY CONTAINERS?



4/5

It's common to use the same virtual machine's spec for all apps. Sometime you might be lucky and minimise the waste in resources.



5/5

It's common to use the same virtual machine's spec for all apps. Sometime you might be lucky and minimise the waste in resources.

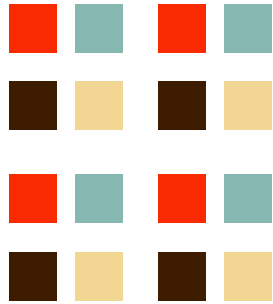
Some companies utilise as little as 10% of their allocated resources. *Can you imagine allocating hundreds of servers, but effectively using entirely only a dozen of them?* What you need is something to break free from the fixed resource allocation of a virtual machine and regain the resource you don't use.



1/3

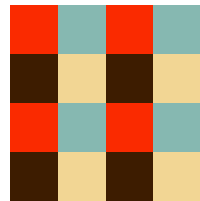
What you want is to get rid of those wrappers around your applications. You don't want to run virtual machines; you only want to run your apps.

WHY CONTAINERS?



2/3

What you want is to get rid of those wrappers around your applications. You don't want to run virtual machines; you only want to run your apps.



3/3

Ideally, if you didn't have to deal with the wrapper, you could pack all of your apps tightly together and save a lot of space.

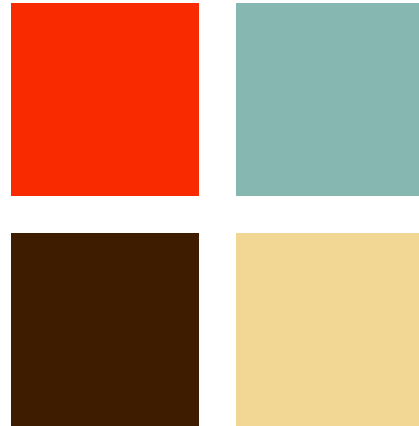
But why are those components so different in specs anyway?!

Why can't you have only lovely squares?

4. Proliferation of languages and frameworks

When you develop smaller and isolated components, you can pick "the right tool for the job" to build your application. *Java and Spring boot for the backend. Node.js and Express for the front-end. Rails and Resque for processing background tasks. You name it.* You can imagine the proliferation of libraries and languages at scale. And even if you try to force only a few libraries and frameworks, it might still not be enough. Two applications sharing the same JVM runtime might rely on a different set of dependencies and libraries and have different resource requirements.

WHY CONTAINERS?



1/5

Imagine developing a service as a collection of smaller components.



2/5

You could pick a different language for each component. You're not limited to use only one like in the case of the monolith.

WHY CONTAINERS?



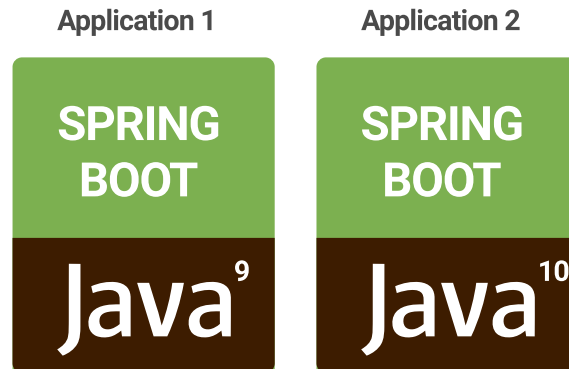
3/5

If you have hundreds of smaller components, you could have several languages, libraries as well as different versions of the same languages.



4/5

And even if you standardise on a single language, you still face the challenge of having different versions of the runtime of dependent libraries.



5/5

Take for example a web app built using Spring Boot and a service broker that uses ZeroMQ with Spring Boot. They're both running on the same version of the JVM, but one requires a binary to be installed on the host.

Traditionally the environment and the application were developed and packaged separately. You had to rely on tools such as Puppet or Ansible to configure your virtual machine with the right dependencies. The approach is still valid, but it doesn't scale when you have hundreds or thousands of components to package. Ideally, **you should keep all of the dependencies alongside your component**. You shouldn't treat the infrastructure as an afterthought. *But how do you keep dependencies and applications together?*

5. Containers in the shipping industry

Not long ago, the shipping industry had a similar challenge. *How do you ship goods in cargo around the world? Do you load individual items of all shapes and sizes? How do you know all the items that belong to the same importer?* Manually keeping track of things is hard. So the shipping industry turned to containers. A container is more convenient to load and deliver because it has a standard size. It would be great if you could use a container for your components too.

6. Linux containers

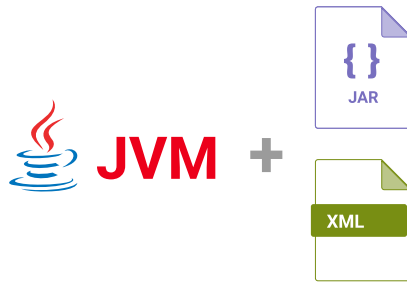
You could pack your applications and all of the dependencies in a standard package to be deployed. **Enter Linux containers.** A Linux container is like a cargo container, but it encapsulates all files, binaries, and libraries necessary to run your process.



1/4

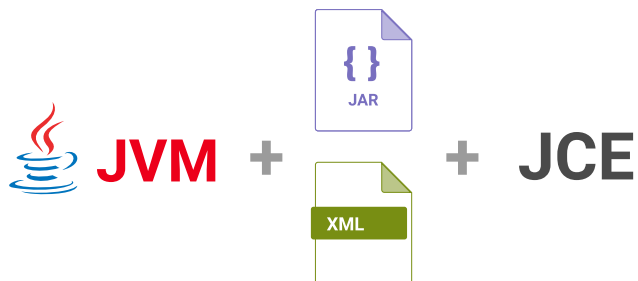
Your application might have dependencies such as a runtime (e.g. JVM), files such as configuration and artefacts, and external dependencies such as ImageMagick or the Java Cryptographic Extension (JCE).

2/4



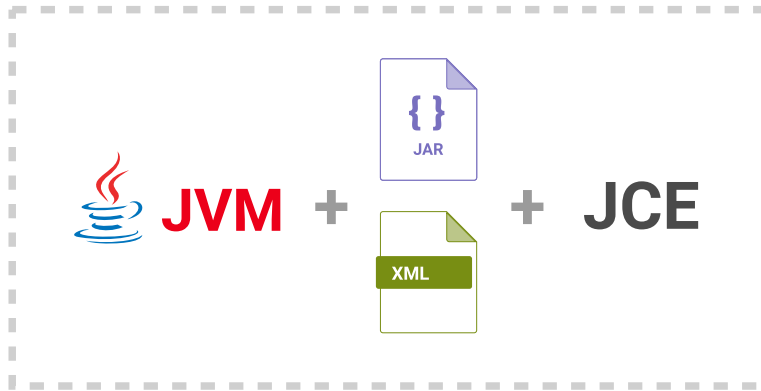
Your application might have dependencies such as a runtime (e.g. JVM), files such as configuration and artefacts, and external dependencies such as ImageMagick or the Java Cryptographic Extension (JCE).

3/4



Your application might have dependencies such as a runtime (e.g. JVM), files such as configuration and artefacts, and external dependencies such as ImageMagick or the Java Cryptographic Extension (JCE).

CONTAINER



4/4

Instead of leaving setting the environments to Puppet or Ansible, you could package all of your assets together in a container.

You might be wondering if this doesn't sound a lot like a virtual machine with an integrated configuration management tool. As you realised early on:

- virtual machines are virtualised environments with an operating system that consumes resources
- when you launch and allocate memory and CPU for a virtual machine, you can't claim unused resources back
- virtual machines are like a computer. They are slow to start, and they emulate hardware

On the other hand:

- **Containers isolate processes.** When you start a container, you're just launching a process on your computer
- The process will **consume only the memory and CPU that it needs**
- **Processes are quick to start**, and there's no need for an operating system

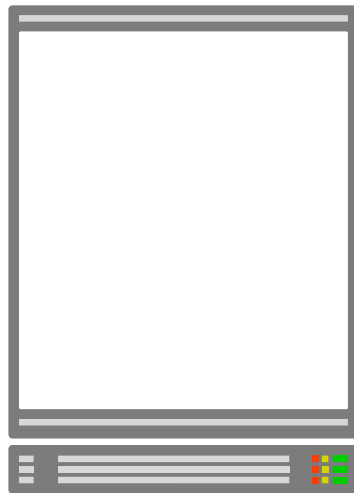
Processes encapsulated in containers, can't see each other. Like if they were packaged in virtual machines. But the isolation provided by containers is different from virtual machines.

7. What are containers made of?

Containers are constructed from two features in the Linux kernel: **control groups** and **namespaces**. **Control groups are a convenient way to limit the CPU or memory that a particular process can use.** As an example, you could say that your component should use only 2GB of memory and one of your four CPU cores. **Namespaces, on the other hand, are in charge of isolating the process and limiting what it can see.** The component can only see the network packets that are directly related to it. It won't be able to see all of the network packets flowing through the network adapter. *Control groups and namespaces are low-level primitives.* With time developers created more and more layers of abstractions to make it easier to control those kernel features. One of the first abstractions was LXC, but the real deal was Docker that was released in 2013. With containers:

- **you only pay for the resources that you use.** No need for an extra operating system for each component

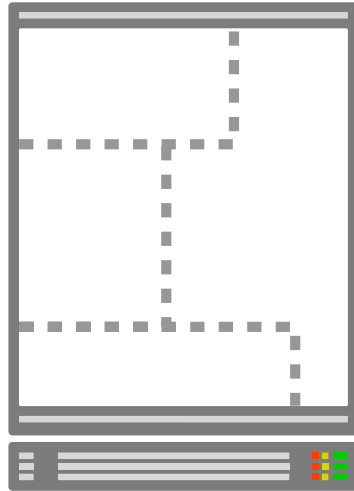
- **you can package and deploy applications with a standard interface.**
You don't have to worry what binary, language or framework is encapsulated in the container
- **multiple isolated processes are running on the same host**



1/3

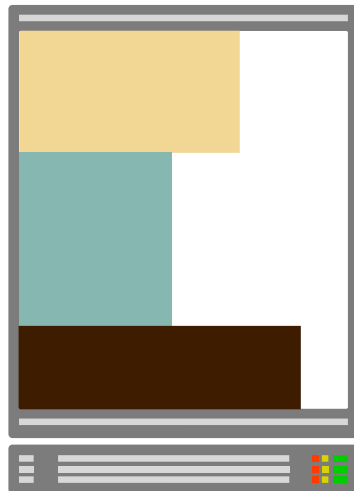
Imagine having a server or a virtual machine.

WHY CONTAINERS?



2/3

With Linux containers, you can define discrete slots for resources such as memory and CPU and have your processes use and see only those resources.



3/3

With Linux containers, you can define discrete slots for resources such as memory and CPU and have your processes use and see only those resources.

You'll see later on how you can leverage Kubernetes to pack your infrastructure efficiently.

Chapter 2

Containers and Docker

Docker uses a *client-server architecture*. The Docker client talks to the Docker daemon using the REST API, over UNIX sockets or a network interface.

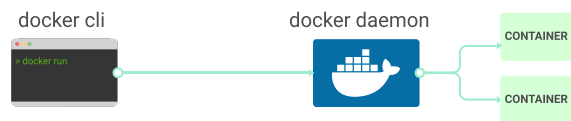
When you want to run a Docker container, the Docker client sends the request to the Docker daemon, and the container runs wherever the daemon is installed. The Docker client and daemon can run on the same computer, or you can connect a Docker client to a remote Docker daemon. When you connect to a remote daemon, the containers are created remotely.

CONTAINERS AND DOCKER



1/8

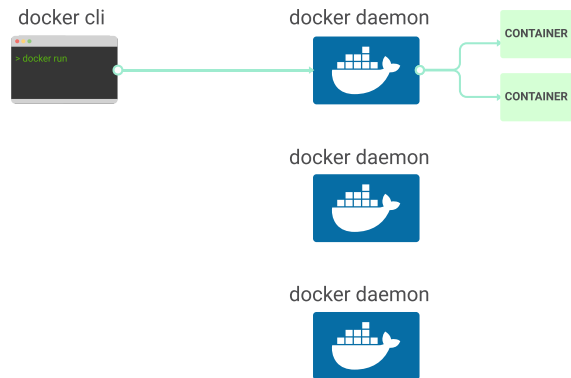
You can connect to a Docker using the REST API, over UNIX sockets or a network interface.



2/8

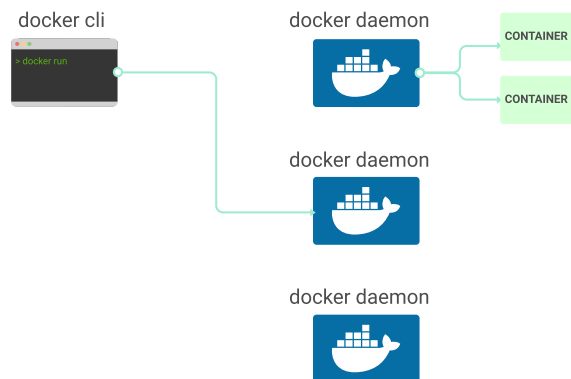
The daemon creates the containers.

CONTAINERS AND DOCKER



3/8

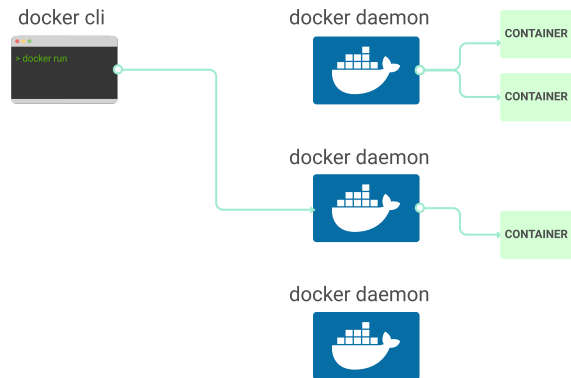
But you could connect to other Docker daemons. Daemon could be located remotely.



4/8

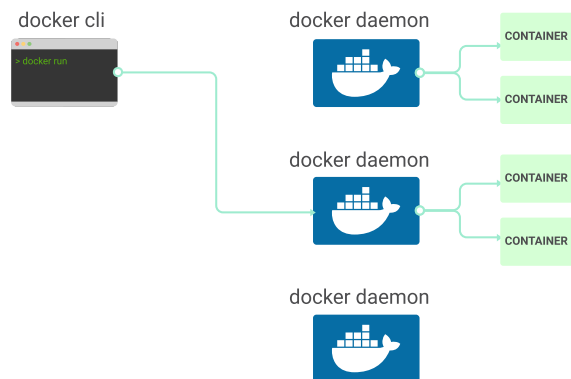
You can change the settings and have your client connecting to a different daemon.

CONTAINERS AND DOCKER



5/8

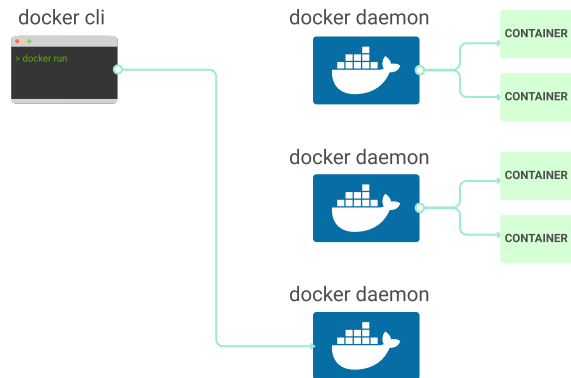
You aren't able to inspect the previous container since the previous Docker daemon created them. You can create more containers, though.



6/8

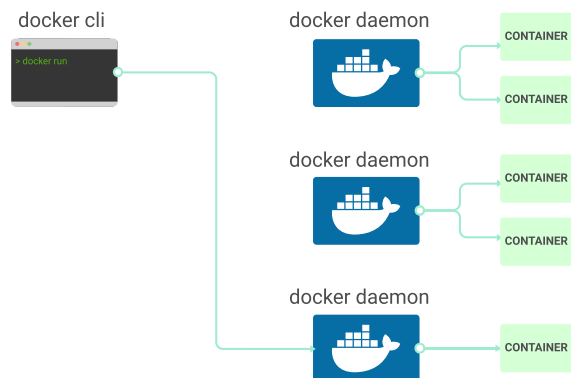
You aren't able to inspect the previous container since the previous Docker daemon created them. You can create more containers, though.

CONTAINERS AND DOCKER



7/8

You could connect to more remote Docker daemons and repeat the process.



8/8


You could connect to more remote Docker daemons and repeat the process.

Connecting to a remote Docker daemon is such a typical operation that you might not even notice about it. If you're using Windows or macOS to run your containers you might wonder how Docker works. *Aren't Linux containers (and Docker) a **Linux feature**? How can you run Docker containers on Windows or macOS?* As you probably already guessed, Docker for Windows and macOS run inside a virtual machine with Linux. The Docker client is connected to the Docker daemon inside the virtual machine. All the containers that you created live inside the Linux virtual machine.

Please note that if you're running Docker for Linux, the Docker daemon can run natively and doesn't use any extra virtual machine.

1. Running containers

You can start a Docker container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti ubuntu bash' in a light gray monospaced font.

```
bash

$ docker run -ti ubuntu bash
```

You can break down the command in the following parts:

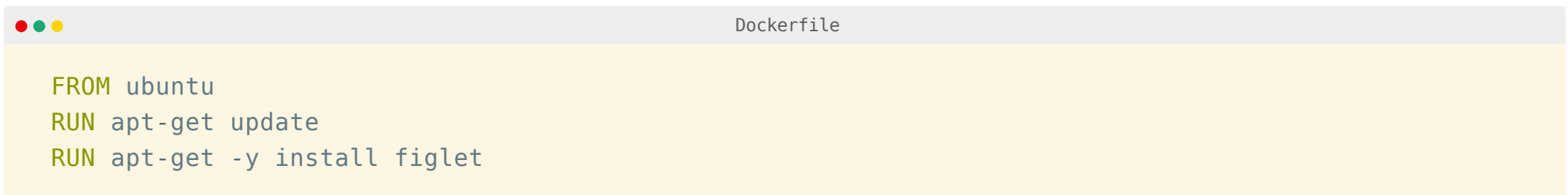
- `docker run` is the command that you use to start a container
- `-ti` means that you wish to run an interactive session
- `ubuntu` is the name of the image
- `bash` is the process you wish to run inside the container

Docker containers are made from Docker images. Images are a collection of files organised as layers. **You could think about Docker images as archives such as a zip or tar archive.** When you launch a container, you run a process such as `bash` with

all the files that you'd generally expect inside an Ubuntu Linux distro. The process is isolated, and everything feels and looks like it's running in an Ubuntu. The reality is that it is running as a process on your computer (or inside a virtual machine if you have Docker for macOS or Windows). *Images are useful, and perhaps you could create your own?*

2. Building Docker images

You learned that Docker images are archives. Also, you probably realised that from a single image you could create several containers. You can keep typing `docker run -ti ubuntu bash` and more `bash` processes are spawned as a consequence. You could think about Docker images as classes in Java. They don't exist until you instantiate them. **When you create an instance of a Docker image, that's called a container.** You can create your personal Docker images by writing a list of instructions. *That list is like a recipe.* You can send that recipe to the daemon, and Docker executes the instructions one by one and creates the image. The instructions are stored in a file called `Dockerfile`:



```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install figlet
```

You can see two commands: `FROM` and `RUN` .

- `FROM` is used to inherit from a base image. When the image is created, the Ubuntu image with all its files is used as the base
- `RUN` execute a command inside the image

In the example above, you start from a Ubuntu image and install the `figlet` binary — a program to draw ASCII art.

There is a list of [all available commands on the official Docker documentation](#).

Once you create a `Dockerfile` locally, you can ask the Docker daemon to build the image with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker build -t my-image .' in a light gray monospaced font.

```
bash

$ docker build -t my-image .
```


The command reads as follow:

- `docker build` is the command to build an image
- `-t my-image` is the name you wish to give to the image
- `.` is the directory where the `Dockerfile` is located.

It's common practice to have the Dockerfile in the root of the project (hence the `.` dot).

3. Managing containers


You can launch containers with `docker run`, but how do you know they are running? You can list all running containers with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the text '\$ docker ps' in a light gray monospaced font.

```
bash

$ docker ps
```

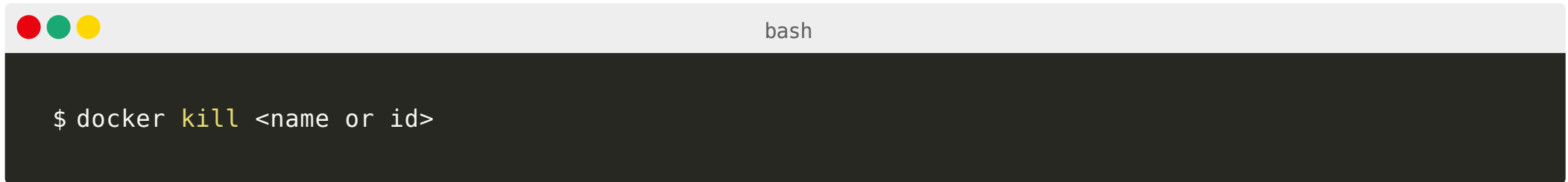
If you wish to stop a container you can use either its name or id like this:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the text '\$ docker stop <name or id>' in a light gray monospaced font.

```
bash

$ docker stop <name or id>
```

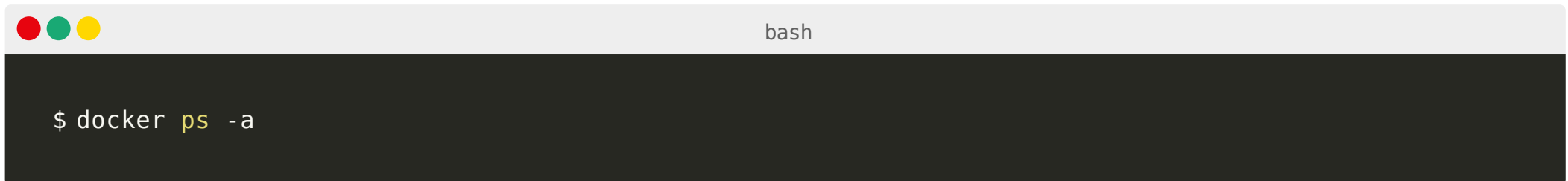
The `docker stop` command waits for the process to shut down gracefully. If you prefer not to wait you can force the process to exit with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker kill <name or id>' in a light gray monospace font.

```
bash

$ docker kill <name or id>
```

Once the container is stopped, it doesn't appear in the `docker ps`. However, you can still see the stopped container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker ps -a' in a light gray monospace font.

```
bash

$ docker ps -a
```

4. Sharing the file system

There are times when you want your containers to share files with your computer. Perhaps you want to edit a python script locally with your IDE and have the container run it. When you wish to share files with a container you can mount that folder inside the container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains a single line of text: '\$ docker run -ti -v /my-computer:/app ubuntu bash'.

```
bash

$ docker run -ti -v /my-computer:/app ubuntu bash
```

You're already familiar with the `docker run` command. The `-v` flag is a shorthand for `--volume` and is designed to mount the local folder `/mycomputer` inside the container as `/app`. Please bear in mind that you can only share files at runtime after the image was created. *So what should you do when you want to package assets such as Java wars inside the container?*

5. COPYing files

You can bundle files, artefacts and assets when you build the image by adding an extra instruction to your `Dockerfile`. The `COPY` command takes two arguments:

1. the location of the folder or file in your computer
2. the location inside the container where the file should be saved



```
FROM ubuntu
COPY ./mycomputer /app-container
```

The `COPY` command is similar to the `-v` (volume) flag you learnt earlier. *When should you use one or the other?* **You should use the `COPY` command every time you wish to bundle your artefacts with the image.** When you start the container, the files are already there, no need to mount them. **You should use the `-v` (or `--volume`) flag when you want to**

share files with the container dynamically. Configuration files are an excellent example of a file that you don't want to bundle at build time because it's environment specific. Instead, you can mount the configuration for a specific environment at runtime when you decide where the container should run.

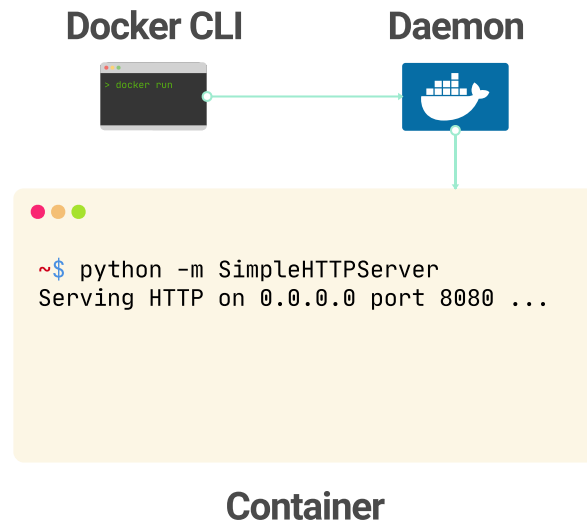
6. Forwarding ports

Imagine you packaged a python web server as a Docker image. The web server exposes port 8080, and if you visit that port, you can receive a response.



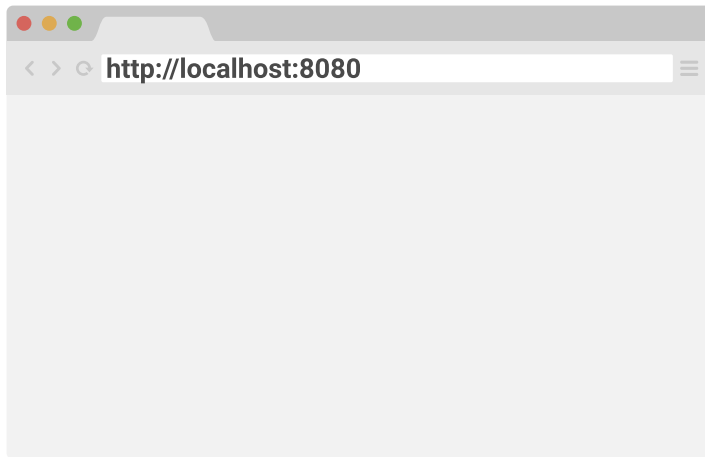
1/2

You have a Docker client connected to a daemon.

**2/2**

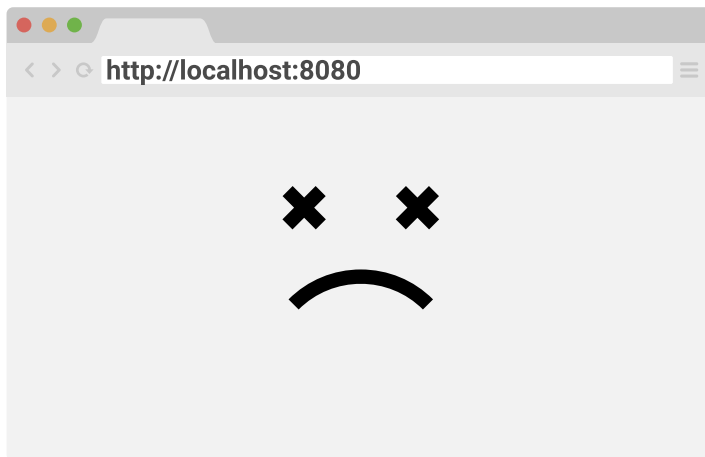
You create a container for a python web-server on port 8080.

You launch the container with `docker run` and open the browser on `http://localhost:8080`. *Do you expect to receive a response from the web server?*



1/2

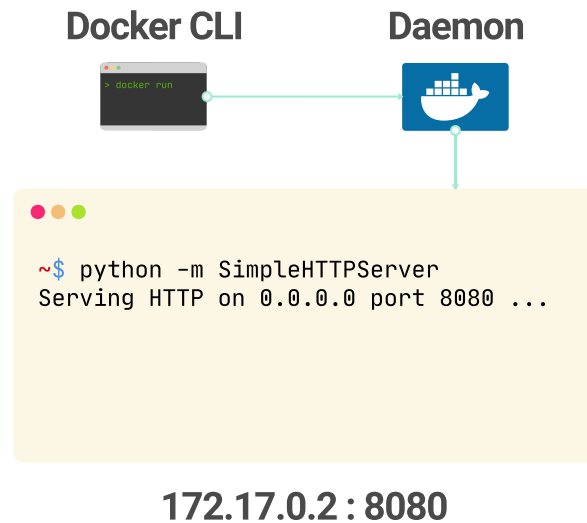
You decide to visit the web server on
`localhost:8080`



2/2

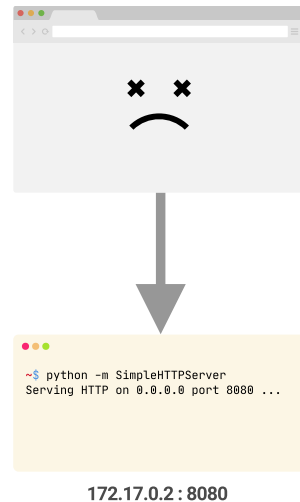
But it doesn't work.

Unfortunately, the page times out. *Why?* When you create a container, Docker assigns an IP address and attaches the container to the default network. **The web server is not bound on localhost, but the container's IP address.**



1/2

When you run a container, the Docker daemon assigns an IP address and attaches it to the network.

**2/2**

The web server is exposed on port 8080 on the IP address of the container. There's no process listening for port 8080 on your computer.

Since forwarding traffic to the container is frequently needed, Docker has a handy feature where you can send all the traffic from your localhost to the container. When you launch your container with `docker run` you can use the `-p` flag — shorthand for *port*.

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text `bash` on the right. The main area of the terminal is dark gray and contains the command `$ docker run -p 8080:8080 python-web`.

Docker automatically creates port forwarding rules to route the traffic from port 8080 on localhost to port 8080 inside the container. You can also change the mapping and have port 3000 on your computer forward the traffic to port 8080 in the container:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -p 3000:8080 python-web' in white text.

```
bash

$ docker run -p 3000:8080 python-web
```

You can bind any port from your container to your host **at run time**.

7. Managing configuration

We already discussed how using volumes is a convenient way to mount files and configuration at run time. There's another way to pass arguments when a container is launched. You can use the `-e` flag, *shorthand for environment*, to inject environment variables in the container.

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti -e HELLO=world bash' in a light-colored monospaced font.

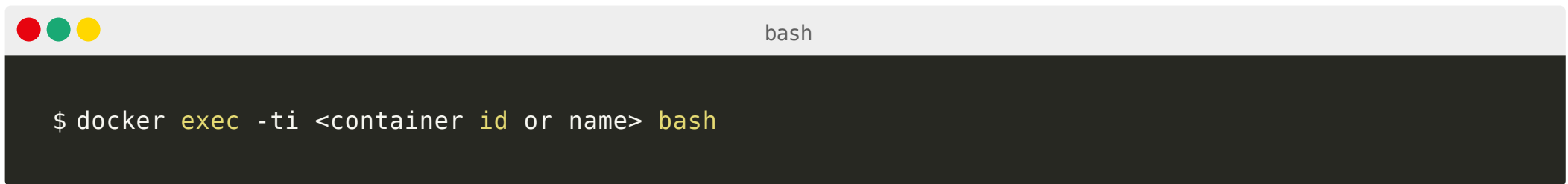
```
bash

$ docker run -ti -e HELLO=world bash
```

You can use the `-e` flag multiple times and inject several environment variables at once. *But how do you test if the environment variable is set up correctly? Can you SSH inside the container and print it?*

8. Attaching to running containers

Unlike virtual machines, you don't need to install SSH to open an interactive session inside the container. Since containers are processes, you can attach to the running process directly. You can use the `docker exec` command to start a session in the running container:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command `$ docker exec -ti <container id or name> bash` in a light gray monospaced font.

```
bash

$ docker exec -ti <container id or name> bash
```

The command above opens a `bash` session inside the container. Like if you were to use SSH. As for the `docker run`, the `-ti` flags have a similar meaning:

- `-t` for terminal
- `-i` for interactive

While easier to remember, the above should read:

`-it` instructs Docker to allocate a pseudo-TTY connected to the container's stdin; creating an interactive bash shell in the container

You've learnt the basics of Linux containers and Docker. It's time to put your knowledge to practice.

Chapter 3

Installing Docker

You can find the instructions to install Docker for your operating system below:

- [Windows 10](#)
- [macOS](#)
- [Ubuntu](#)

Please note that only Windows 10 Pro is supported. If you need help with older versions of Windows, please head over to our detailed guide on [how to install Docker and minikube on Windows](#).

1. Windows 10

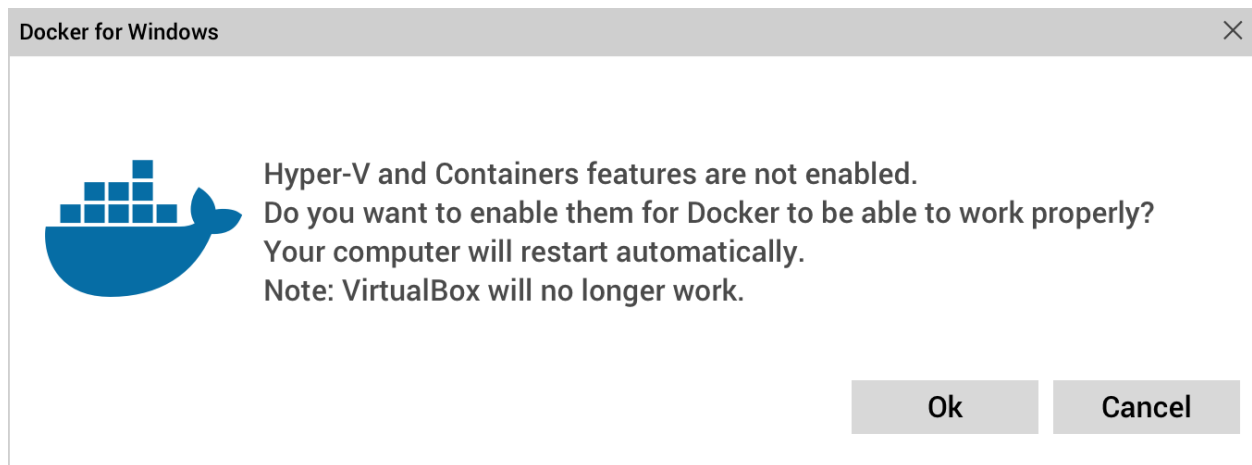
You can download Docker for Windows with:

—□ X

PowerShell

```
choco install docker-desktop -y
```

Docker will ask you if you wish to enable Hyper-V and restart your computer (if you haven't already enabled it before).



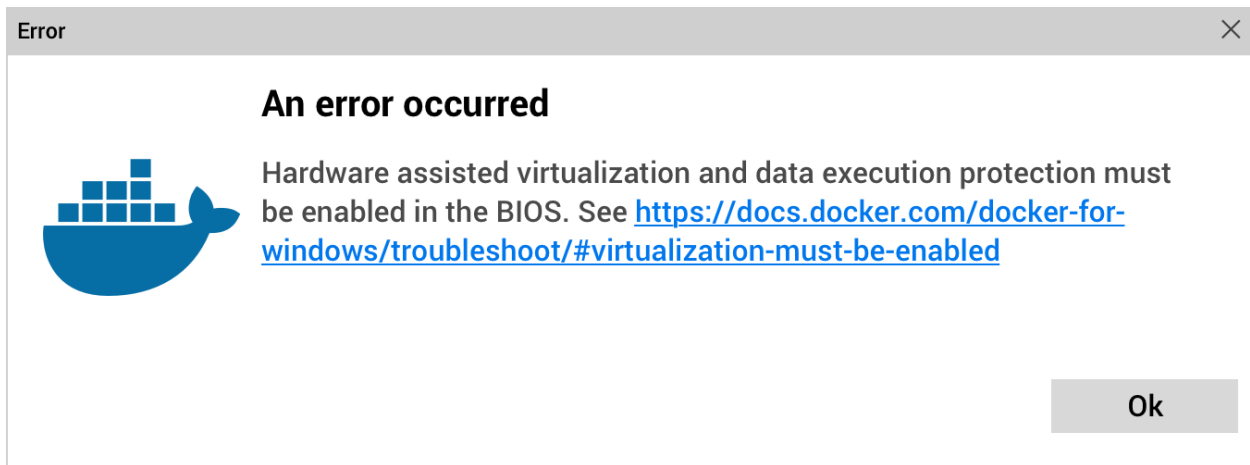
Enabling Hyper-V in Docker

Yes, you do!

You should be aware that Docker requires VT-X/AMD-v virtual hardware extension to be enabled before you can run any container. Depending on your computer, you may need to reboot and enable it in your BIOS.

If you're unsure VT-X/AMD-v was enabled, don't worry. If you don't have it, Docker will greet you with the following error message:

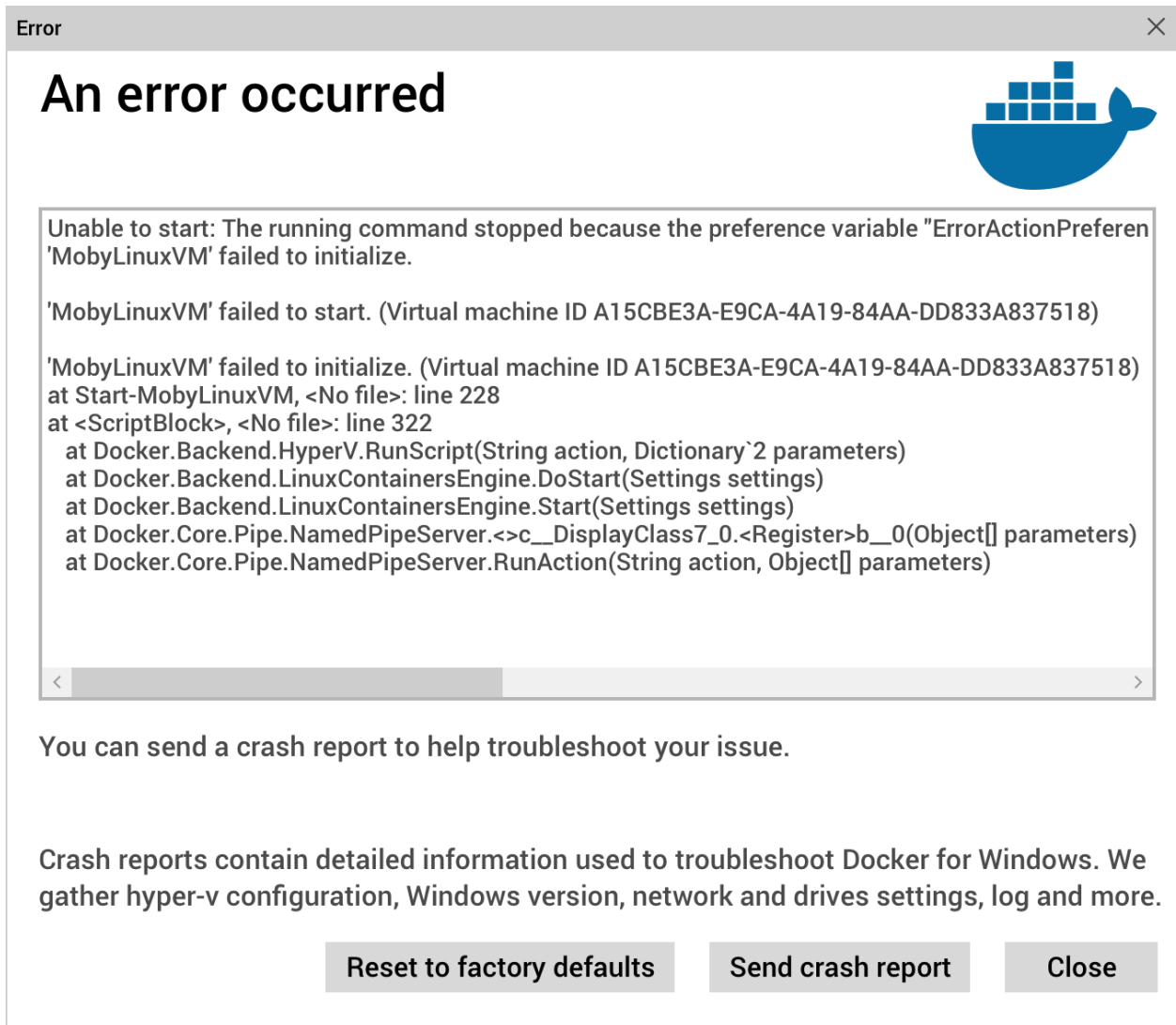
Hardware assisted virtualization and data execution protection must be enabled in the BIOS.



You should enable VT-X/AMD-v

Another common error has to do with the Hyper-V hypervisor not being enabled. If you experience this error:

Unable to start: The running command stopped because the preference variable "ErrorActionPreference" or common parameter is set to Stop: 'MobyLinuxVM' failed to start. Failed to start the virtual machine 'MobyLinuxVM' because one of the Hyper-V components is not running.



You should enable Hyper-V with Docker for Windows

You should enable Hyper-V. Open a new command prompt as an administrator and type the following:

—□X

PowerShell

```
bcdedit /set hypervisorlaunchtype auto
```

You should reboot your machine and Docker should finally start.

But how do you know if Docker is working?

Open a new command prompt and type:

—□X

PowerShell

```
docker ps
```

If everything works as expected, you should see an empty list of containers running.

If your Docker daemon isn't running, you're probably banging your head against this error:

—□X

PowerShell

```
docker ps
```

```
error during connect: Get http://%2F%2F.%2Fpipe%2Fdocker_engine/v1.37/containers/json:
open ///.//pipe/docker_engine: The system cannot find the file specified. In the default
daemon configuration on Windows, the docker client must be run elevated to connect. This
error may also indicate that the docker daemon is not running.
```

The error above is suggesting that your Docker installation is not behaving normally and wasn't able to start.

You should start your Docker daemon before you connect to it.

Testing your Docker installation

You should test that your Docker installation was successful.

In your terminal type:

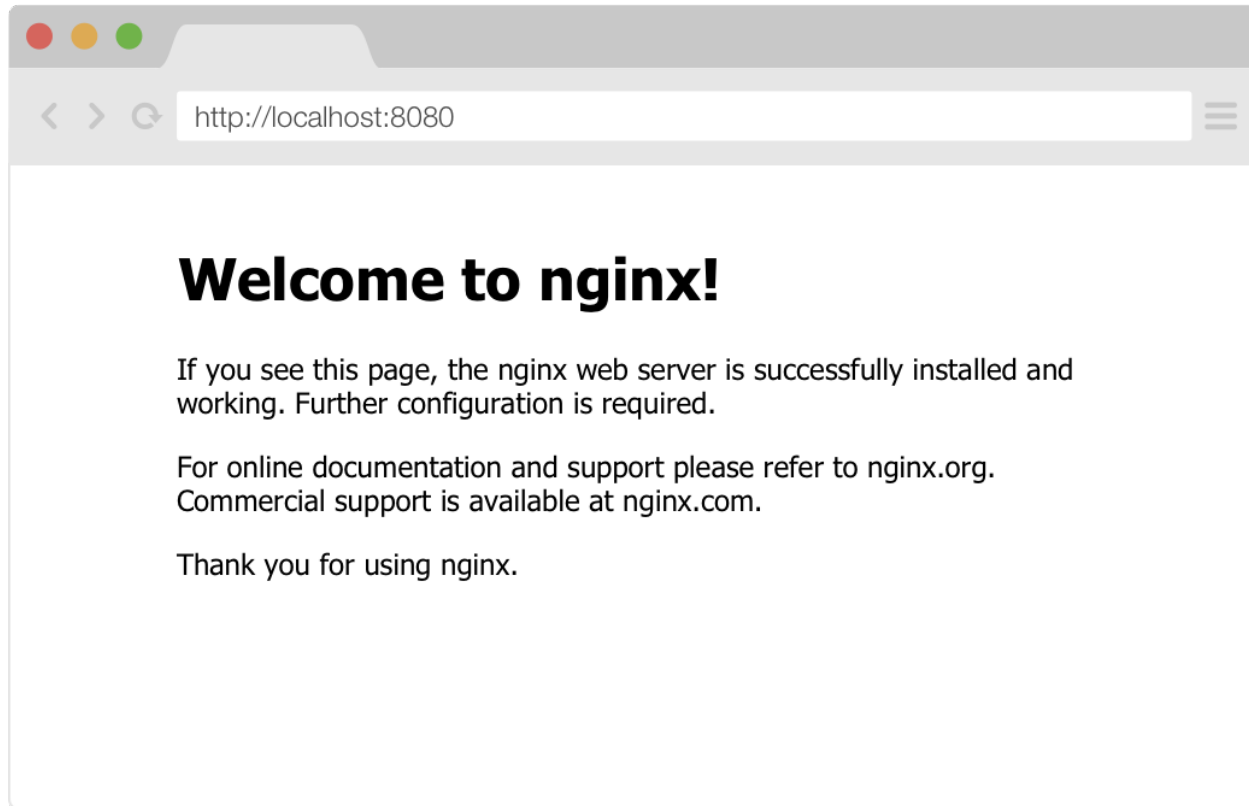
—□X

PowerShell

```
docker run -ti -p 8080:80 nginx
```

Once Docker has completed downloading the image, you should visit <http://localhost:8080/>.

You should see the Nginx welcome page.




Nginx Hello World

Congratulations!

The installation was successful.

2. macOS

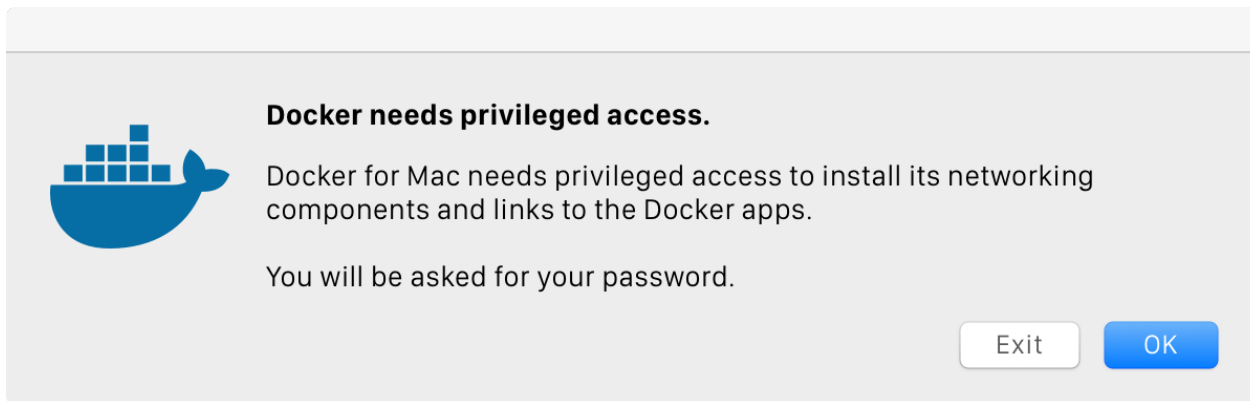
You can download Docker for Mac with:



```
bash

$ brew cask install docker
```

You should start Docker. It will ask permission to install the networking component.



Docker networking component

You should proceed.

You can find Docker in your Applications folder.

Testing your Docker installation

You should test that your Docker installation was successful.

In your terminal type:

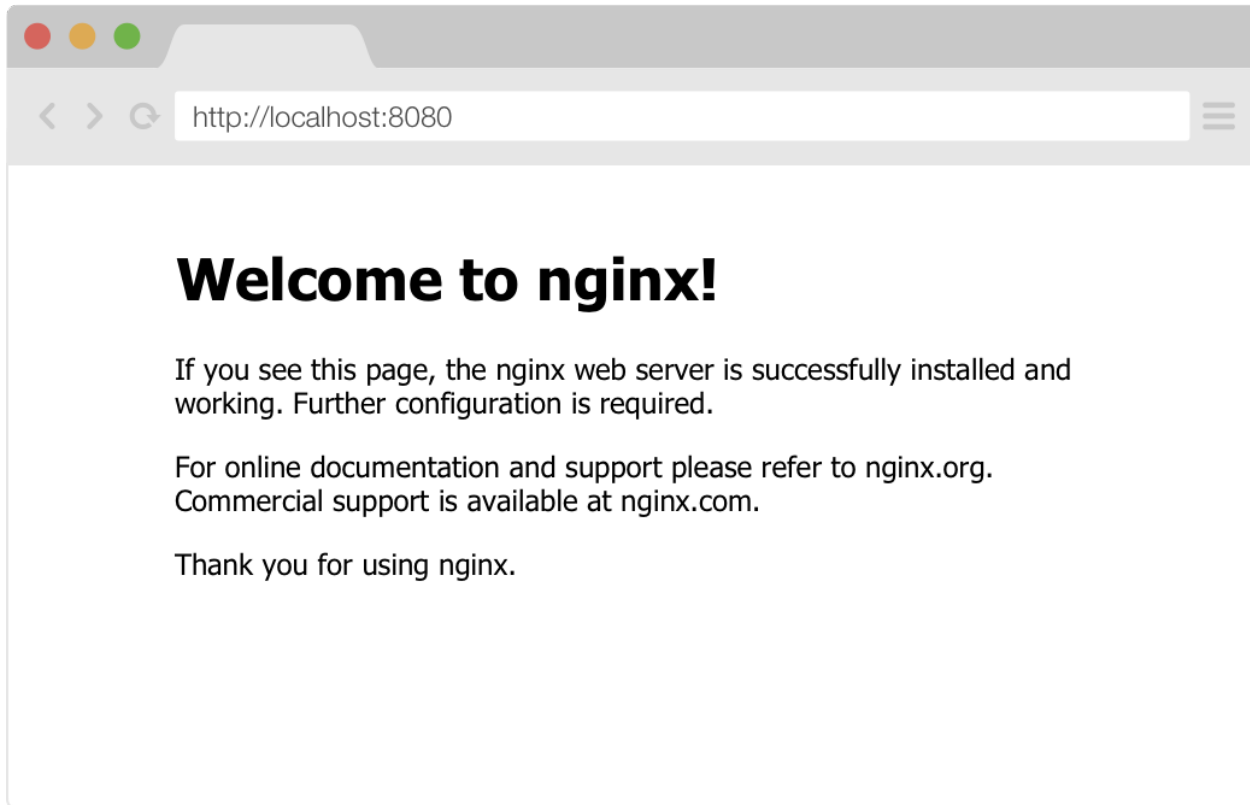


```
bash

$ docker run -ti -p 8080:80 nginx
```

Once Docker has completed downloading the image, you should visit <http://localhost:8080/>.

You should see the Nginx welcome page.



Nginx Hello World

Congratulations!

The installation was successful.

3. Ubuntu

Update the `apt` package index:



```
bash

$ sudo apt-get update -qq
```

Install packages to allow apt to use a repository over HTTPS:



```
bash

$ sudo apt-get install -qqy \
  apt-transport-https \
  ca-certificates \
  curl \
  software-properties-common
```

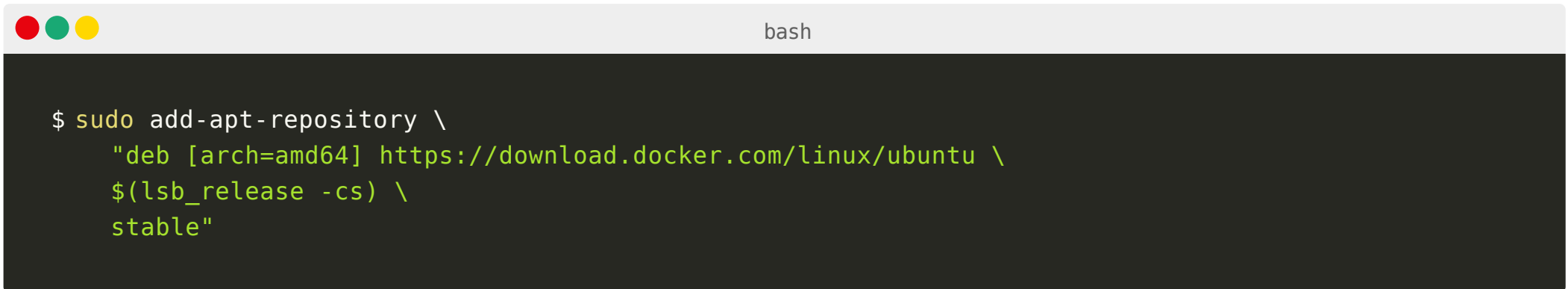
Add Docker's official GPG key:



```
bash

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

Use the following command to add the stable repository.



```
bash

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


Update the apt package index again:



```
bash

$ sudo apt-get update -qq
```

Finally, install the latest version of Docker:



```
bash

$ sudo apt-get install -qqy docker-ce
```

Make sure that there's a group for Docker:



```
bash

$ sudo groupadd docker
```

And your user belongs to it:



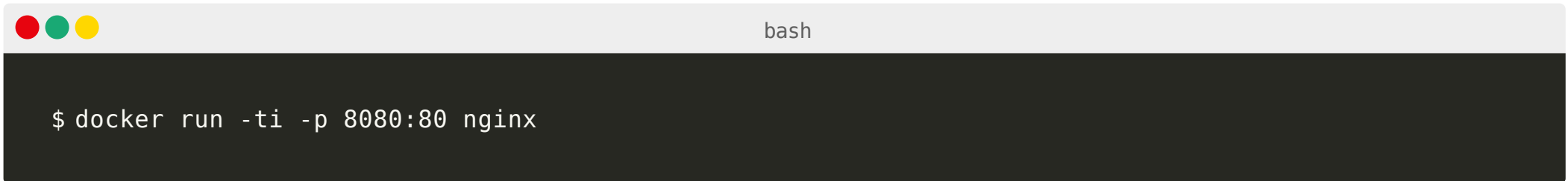
```
$ sudo usermod -aG docker $USER
```

You log out and log back in to enable the changes.

Testing your Docker installation

You should test that your Docker installation was successful.

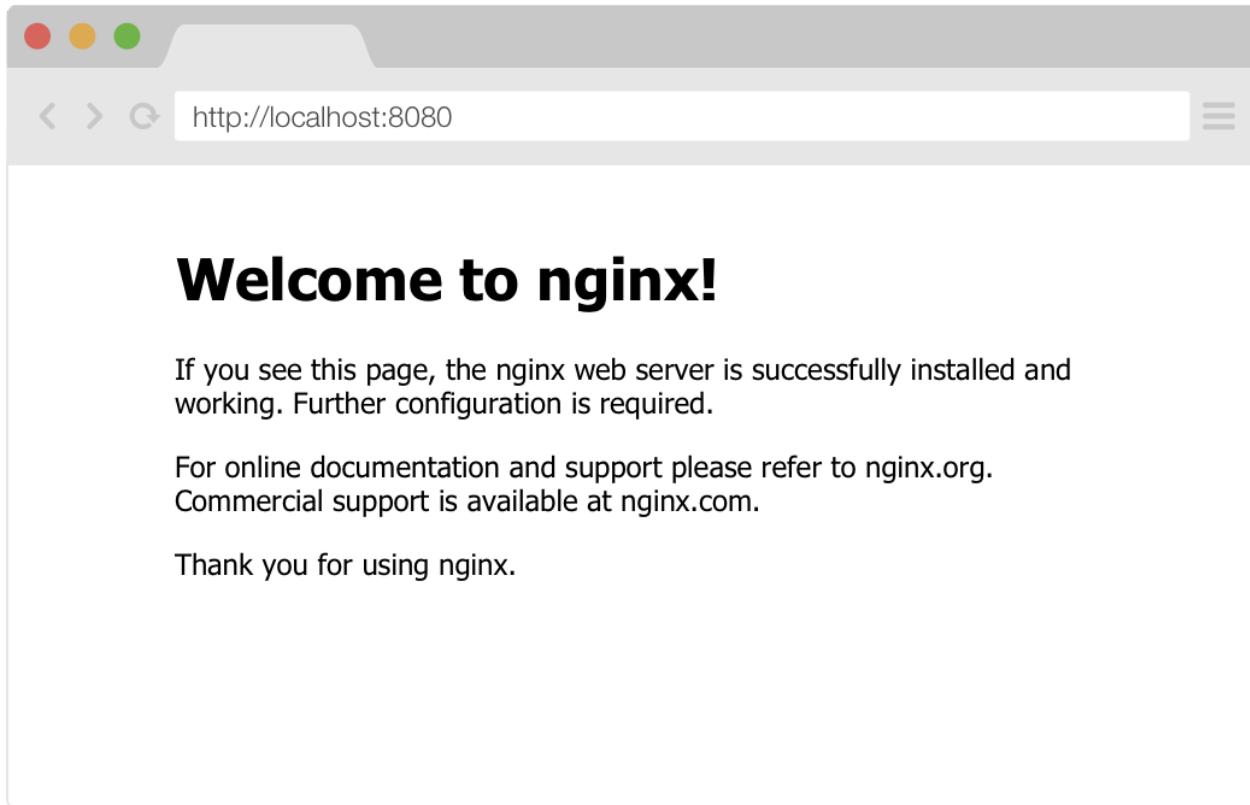
In your terminal type:



```
$ docker run -ti -p 8080:80 nginx
```

Once Docker has completed downloading the image, you should visit <http://localhost:8080/>.

You should see the Nginx welcome page.



Nginx Hello World

Congratulations!

The installation was successful.

Chapter 4

Getting started with Docker

Before you start, make sure that you have followed the instructions to install and test Docker on your computer.

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' in the center. The main area of the terminal is dark gray and contains the text '\$ docker version' in a light gray monospaced font.

```
bash

$ docker version
```

1. Running Docker containers

You can start a Docker container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti node:8.12.0-slim' in white text.

```
bash

$ docker run -ti node:8.12.0-slim
```

You can break down the command in the following parts:

- `docker run` is the command that you use to start a container
- `-ti` means that you wish to run an interactive session
- `node` is the name of the image
- `:8.12.0-slim` is the particular version of the image you wish to use

If this is the first time that you're using that particular image, Docker will download the image from the registry.

2. Docker registries

The Node.js Docker image that you just downloaded comes from [Docker Hub](#). Docker Hub is a public registry. You can create an account and upload your public images. If you prefer to upload your images to a private registry, there're alternatives such as the premium version of Docker Hub, Quay.io, Bintray and Artifactory. Also, each major cloud provider offers private Docker registry:

- Amazon Web Services provides the Elastic Container Registry (ECR)
- Azure has the Azure Container Registry (ACR)
- Google Cloud Platform provides the Google Container Registry (GCR)

3. Running commands inside containers

The previous command ran the `node` executable by default. If you wish to run another command other than the default command inside the container you can do so with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti node:8.12.0-slim bash' in a light gray monospaced font. The word 'bash' in the command is highlighted in yellow.

```
$ docker run -ti node:8.12.0-slim bash
```

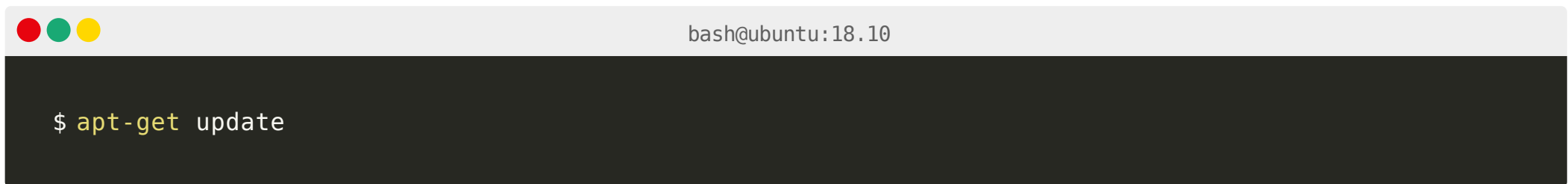
You can now interact with the shell inside the container. But containers are handy when it comes to testing executables. You might not always want to install binaries on your computer. Containers are excellent as a volatile environment where you can test your ideas. Let's assume you want to install and try `figlet` — a binary that displays words as ASCII art. You could launch an Ubuntu container with:



```
bash

$ docker run -ti ubuntu:18.10
```

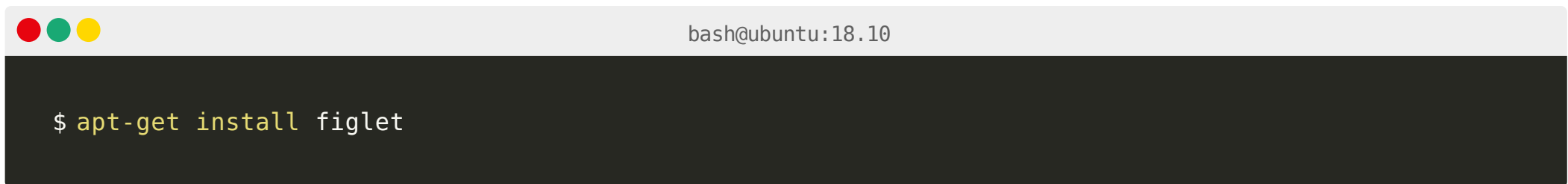
The container doesn't have an updated list of repositories for `apt`. You can fix that with:



```
bash@ubuntu:18.10

$ apt-get update
```

You can install `figlet` with:



```
bash@ubuntu:18.10

$ apt-get install figlet
```

And you can test it with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash@ubuntu:18.10' on the right. The main area of the terminal is dark gray and contains the text '\$ figlet hello' in a white monospaced font.

```
bash@ubuntu:18.10  
  
$ figlet hello
```

You didn't have to install anything on your computer to test `figlet` other than Docker. *Where is the container gone? Is it destroyed?*

4. Start, stop, list, kill

You can list all running and stopped containers on your computer with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker ps -a' in a light gray monospace font.

```
bash

$ docker ps -a
```

If you omit the `-a` flag, Docker will list only the running containers.

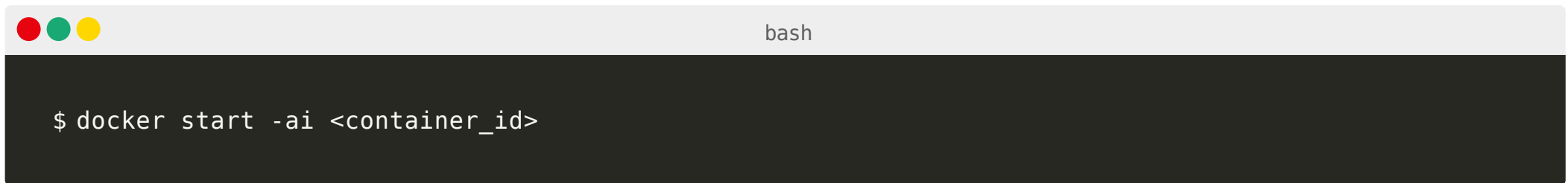
You probably already noticed that the Ubuntu and Node.js containers are stopped. You can delete the Node.js container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command `$ docker rm <container_id>` in white text.

```
bash

$ docker rm <container_id>
```

And you can start the Ubuntu container again with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command `$ docker start -ai <container_id>` in white text.

```
bash

$ docker start -ai <container_id>
```

The `-ai` flag is used to attach the input and output to your terminal session.

All the commands you run inside containers block the terminal. As an example, the following container displays the time and is always in the foreground:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains the command '\$ docker run -ti jpetazzo/clock' in white text.

```
bash

$ docker run -ti jpetazzo/clock
```

But you can start the same container in the background, as a daemon, with the `-d` flag.

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains the command '\$ docker run -d jpetazzo/clock' in white text.

```
bash

$ docker run -d jpetazzo/clock
```

Where is the output gone? You can list all the running containers with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains the command '\$ docker ps' in white text.

```
bash

$ docker ps
```


You can tail the logs of the container running as a daemon with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker logs <container id>' in white text.

```
bash

$ docker logs <container id>
```

Once you're satisfied, you can stop a running container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker stop <container id>' in white text.

```
bash

$ docker stop <container id>
```

The `docker stop` command waits for the process to terminate gracefully. If you don't mind terminating a process abruptly you can force stopping a running container with:



A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' in the center. The main area of the terminal is dark gray and contains the command `$ docker kill <container id>` in a light gray monospaced font.

```
bash

$ docker kill <container id>
```

5. Terminology

In Docker, it's easy to confuse the terms containers and images.

- **containers** are created from images. You can have multiple containers instantiated from the same image.
- **images** are similar to templates. When you want to run a container, you select and copy the template to run it.

An image can have multiple containers. A container can only have one image. If you are still confused, you can think about images as Java classes. You can't run a Java class by itself. You should be instantiating it first. An instance of an image is a container.

Chapter 5

Creating images

If the standard Docker images hosted on a public or private registry don't cut it, you can create your images. Docker builds images automatically by reading the instructions from a `Dockerfile` — a text file that contains all commands needed to build a given image.

1. Your first Dockerfile

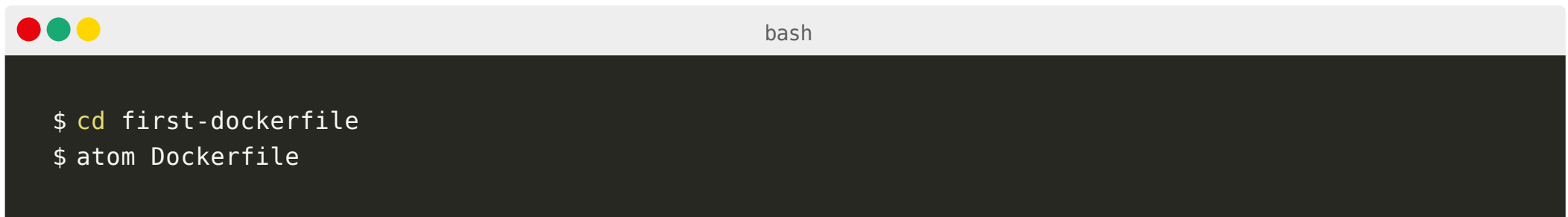
Let's assume you want to create a simple image with the `figlet` binary packaged as part of it. Let's start with creating a temporary directory:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ mkdir first-dockerfile' in a light gray monospaced font.

```
bash

$ mkdir first-dockerfile
```

You should create a `Dockerfile` inside it:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains two commands in a light gray monospaced font: '\$ cd first-dockerfile' followed by '\$ atom Dockerfile' on the next line.

```
bash

$ cd first-dockerfile
$ atom Dockerfile
```


Please note the capital D in `Dockerfile`. Also, the file has no extension.

The content should be:

```
Dockerfile

FROM ubuntu:18.10
RUN apt-get update
RUN apt-get -y install figlet
```

The `Dockerfile` contains three instructions:

- `FROM` is the base image. Your image will inherit from the stock Ubuntu image
- `RUN` is used to run command during the build phase — i.e. when the image is built

2. Build the image

You can build your first image with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker build -t first .' in a light gray monospaced font.


```
bash

$ docker build -t first .
```

The command reads as follow:

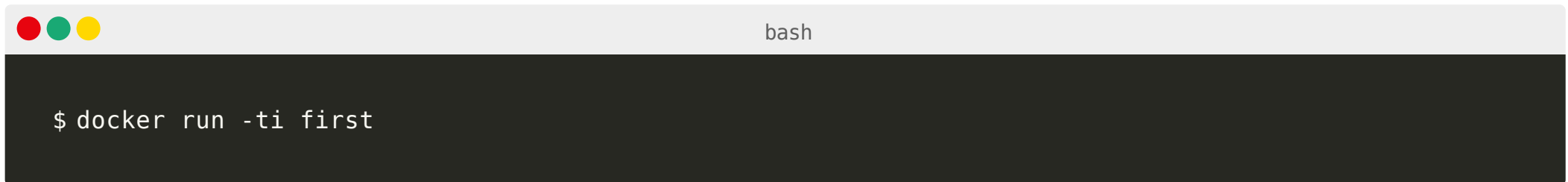
- `docker build` is the command to build an image
- `-t first` is the name you wish to give to the image
- `.` is the directory where the `Dockerfile` is located. It's common practice to have the Dockerfile in the root of the project. You could move the Dockerfile to a `build` directory. If you do that the command will change to `docker build -t first ./build`.

As soon as you type the command, Docker executes the steps written in the Dockerfile. One by one. You can verify that the image was built correctly with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command `$ docker images | grep first` in a light gray monospaced font.

```
bash
$ docker images | grep first
```

Finally, you can run the image with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command `$ docker run -ti first` in a light gray monospaced font.

```
bash
$ docker run -ti first
```

Inside the container you can test that the `figlet` binary comes preinstalled:

CREATING IMAGES

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' in the center. The main area of the terminal is dark gray and contains the command '\$ figlet "it works"' in a light green monospaced font.

```
$ figlet "it works"
```

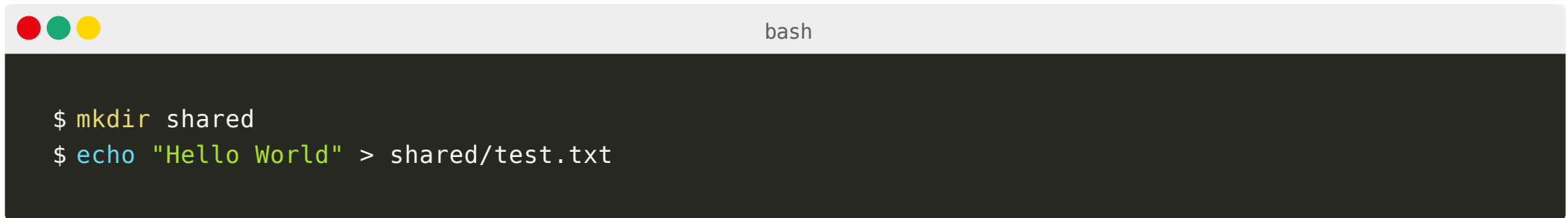
Chapter 6

Managing state

Containers are isolated processes that share the kernel with their host. But most of the time, you want to share more than the kernel. Sharing files is an everyday use case.

1. Sharing the file system

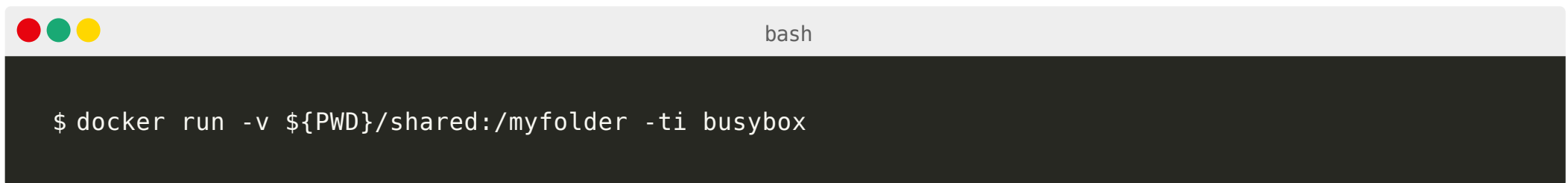
On your computer create a folder and a dummy file:



```
bash

$ mkdir shared
$ echo "Hello World" > shared/test.txt
```

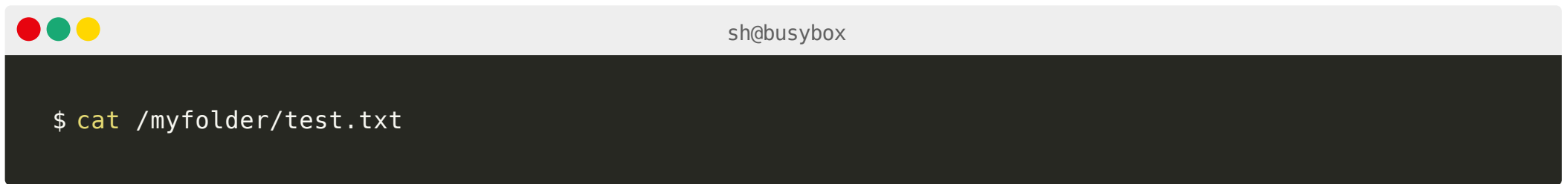
Mount the directory `./shared` as `/myfolder` inside the container:



```
bash

$ docker run -v ${PWD}/shared:/myfolder -ti busybox
```

The `-v` flag is used to map a local folder on the host, `./shared`, to a remote folder inside the container `/myfolder`. The `${PWD}` environment variable is a convenient way to provide the full path of that `./shared` folder. Once you're inside the container you should check that the content of the file is identical:

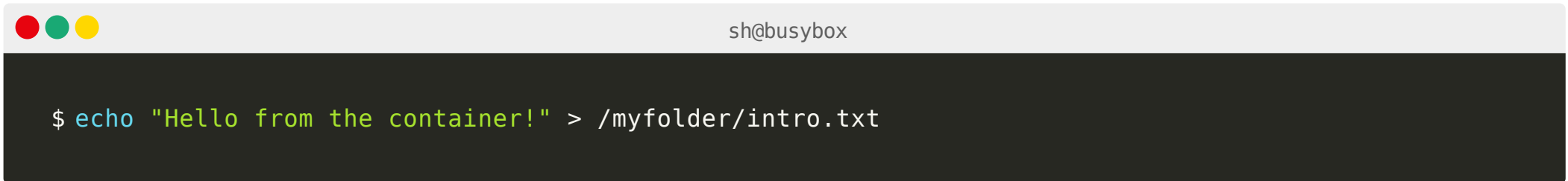
A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'sh@busybox' on the right. The main area of the terminal is dark gray and contains the command '\$ cat /myfolder/test.txt' in a light-colored monospace font.

```
sh@busybox

$ cat /myfolder/test.txt
```


2. Bidirectional sync

What happens when you write in the mounted directory? From within the container create a file:

A terminal window with a light gray title bar containing three colored circles (red, green, yellow) on the left and the text 'sh@busybox' on the right. The terminal area has a dark background and shows a command being entered: '\$ echo "Hello from the container!" > /myfolder/intro.txt'.

```
sh@busybox

$ echo "Hello from the container!" > /myfolder/intro.txt
```

If you exit from the container, you should notice that the same file is now present on your host:

A terminal window with a light gray title bar containing three colored circles (red, green, yellow) on the left and the text 'bash' on the right. The terminal area has a dark background and shows a command being entered: '\$ cat shared/intro.txt'.

```
bash

$ cat shared/intro.txt
```

3. Baking files in the image

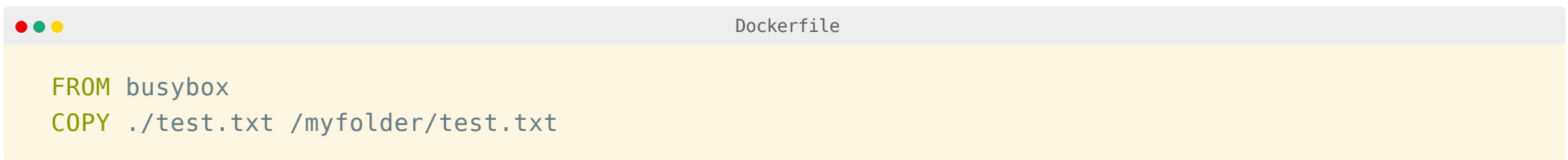
You can only mount volumes when you run a container. *But how do you copy your jars, executables or scripts inside the image during the build phase?* Docker has a particular command that you can place in your `Dockerfile` to do just that. Create a `Dockerfile` in the shared directory as follow:



```
bash

$ atom shared/Dockerfile
```

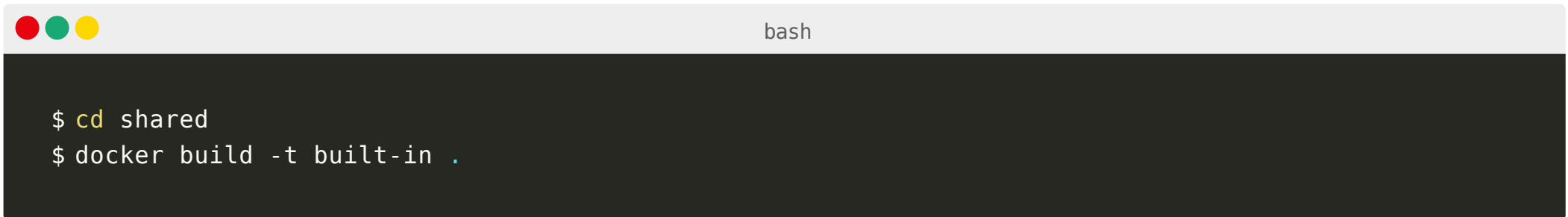
And save the following content:



```
Dockerfile

FROM busybox
COPY ./test.txt /myfolder/test.txt
```

You can build your image with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains two lines of text: '\$ cd shared' and '\$ docker build -t built-in .' in a light gray monospace font.

```
bash

$ cd shared
$ docker build -t built-in .
```

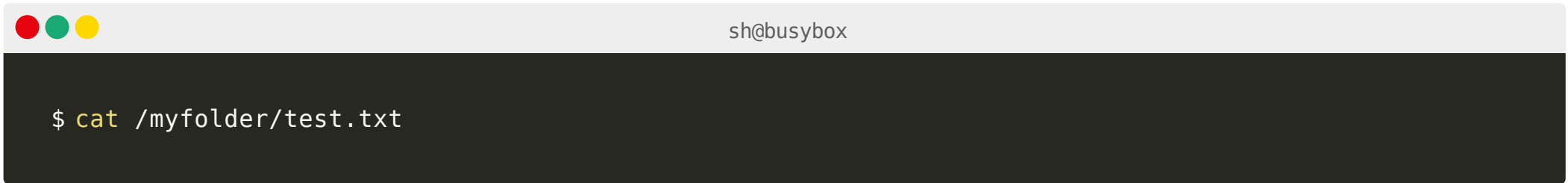
You can test the image with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains one line of text: '\$ docker run -ti built-in' in a light gray monospace font.

```
bash

$ docker run -ti built-in
```

As soon as you are inside the container you can retrieve the content of the file:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'sh@busybox' on the right. The main area is dark gray and contains the command '\$ cat /myfolder/test.txt' in a light-colored monospace font.

```
sh@busybox

$ cat /myfolder/test.txt
```


What happens when you edit the file? You can overwrite the file from within the container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'sh@busybox' on the right. The main area is dark gray and contains the command '\$ echo "Edited" > /myfolder/test.txt' in a light-colored monospace font.

```
sh@busybox

$ echo "Edited" > /myfolder/test.txt
```

If you exit from the container and retrieve the file from the host you will notice that the file hasn't changed:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains the command '\$ cat test.txt' in a light-colored monospace font.

```
bash

$ cat test.txt
```

Indeed, the file was changed in the file system of the container. There wasn't any sharing between host and container.

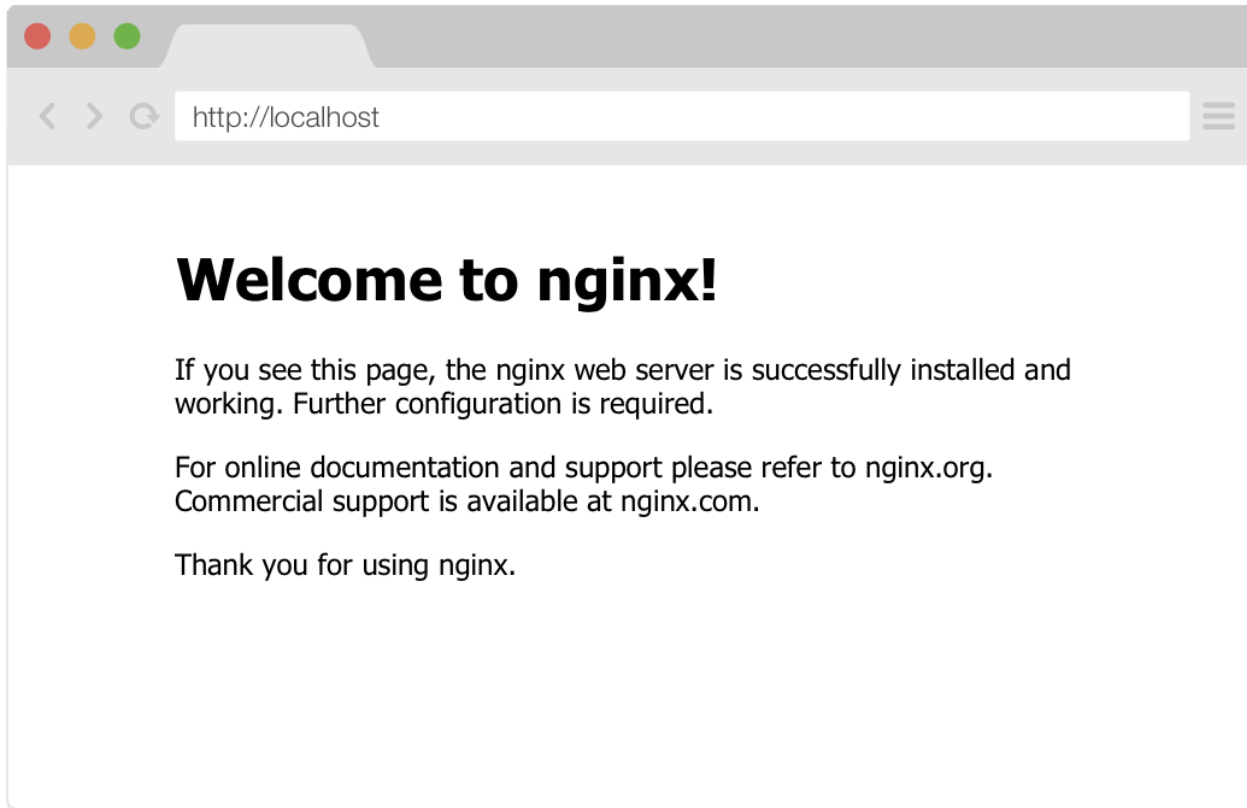
Chapter 7

Networking

Docker containers have IP addresses assigned to them. By default, they are also attached to a bridge network and can talk to each other. Docker containers can consume services from the internet too. *But can you connect to a Docker container?*

1. Connecting to the container

On Docker Hub, there's an image for Nginx — a popular web server. Nginx is usually used as a reverse proxy, a web server or a load balancer, but you will be using just for the nice *Hello World* page that is served by default.



Nginx Hello World

By default, Nginx starts on port 80. You can start the container with:

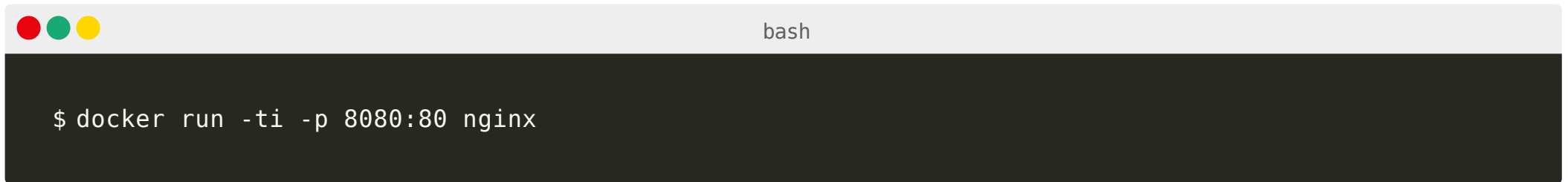
A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti nginx:1.15.3-alpine' in white text.

```
$ docker run -ti nginx:1.15.3-alpine
```

What do you expect to see when you visit <http://localhost>? Are you greeted by the Nginx "Welcome to nginx!" page? Not this time. The port is opened on the container, but you're visiting localhost on port 80 on your computer. How do you visit port 80 on the container itself? Perhaps you need some port forwarding.

2. Bind container ports to the host

You can bind a port from the container to your host with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti -p 8080:80 nginx' in a light gray monospaced font.

```
bash

$ docker run -ti -p 8080:80 nginx
```

The flag `-p` instructs Docker to bind the port 8080 on your host to the port 80 on the container. You can verify the bindings with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker ps' in a light gray monospaced font.

```
bash

$ docker ps
```

The `PORTS` column suggests that the application is bound correctly. You can visit your application at <http://localhost:8080>.

Note how the application still thinks it's running on port 80.

Chapter 8

Packaging an application

You know most of the concepts needed to create containers. It's time to practice by packaging a real application.

1. Hello World, Node.js

In this section, you will create a simple Node.js *Hello World* application. Create a new directory:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ mkdir -p docker-hello-world' in a light gray monospaced font.

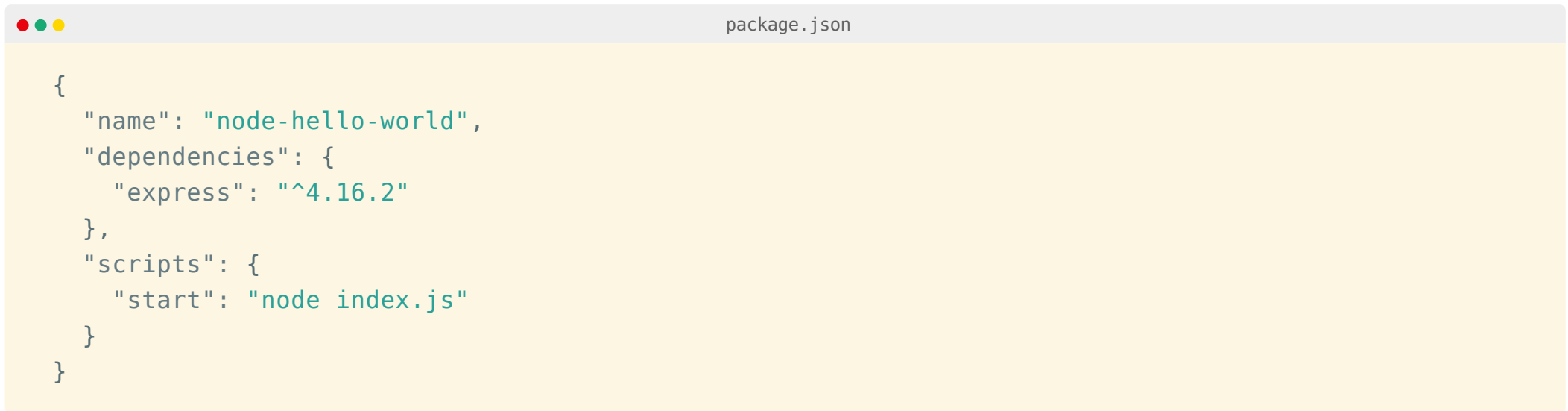
```
$ mkdir -p docker-hello-world
```

Inside that directory, create the following resources. Create an `index.js` file with the following content:



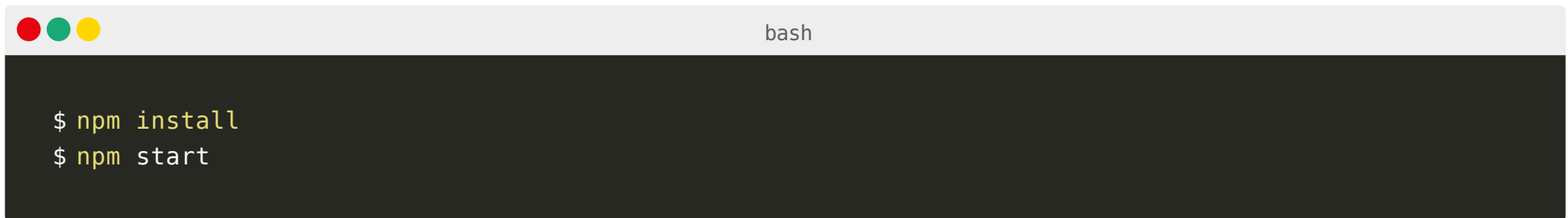
```
const express = require('express')
const app = express()
const port = process.env.PORT || 8080;
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.get('/version', (req, res) => {
  res.send(process.env.VERSION || 'No version')
})
app.listen(port, () => console.log(`Listening on port ${port}!`))
```

Create a `package.json` file with the following content:



```
{
  "name": "node-hello-world",
  "dependencies": {
    "express": "^4.16.2"
  },
  "scripts": {
    "start": "node index.js"
  }
}
```

If you have Node.js installed locally, you can run the application with:



```
bash

$ npm install
$ npm start
```

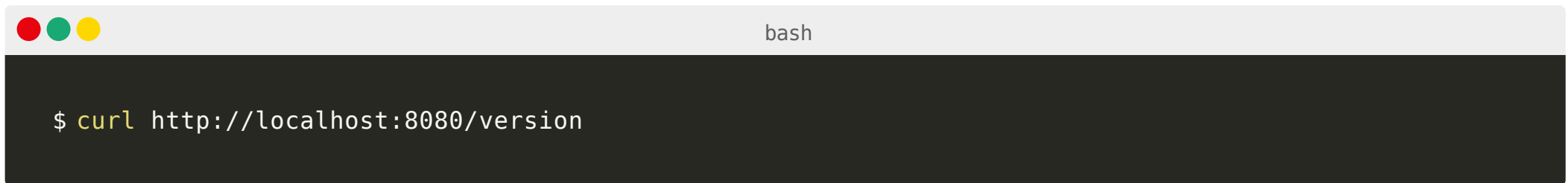
The application is available at <http://localhost:8080>. In another terminal, try to retrieve the home page with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the text '\$ curl http://localhost:8080/' in a light gray monospace font.

```
bash

$ curl http://localhost:8080/
```


You should be greeted by *Hello World*. The application has a second route. You can request the version of the application with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the text '\$ curl http://localhost:8080/version' in a light gray monospace font.

```
bash

$ curl http://localhost:8080/version
```

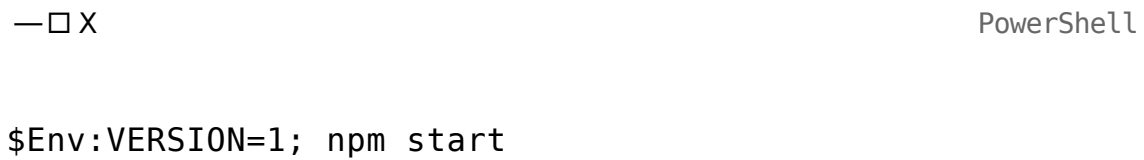
If there's no environment variable set for `VERSION`, the application replies with *No version*. You can stop the application and relaunch it with:



```
bash

$ VERSION=1 npm start
```


Or if you're on Windows:



```
PowerShell

$Env:VERSION=1; npm start
```

You can verify that the version has changed with:



```
bash

$ curl http://localhost:8080/version
```

2. Creating an image

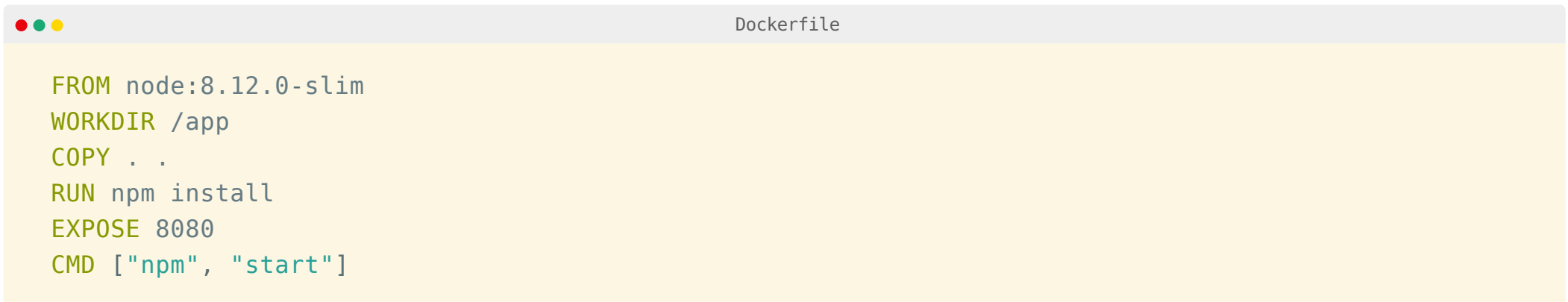
You should create a `Dockerfile` with a list of commands to build your image.

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text "bash" on the right. The main area of the terminal is dark gray and contains the text "\$ atom Dockerfile" in white.

```
bash

$ atom Dockerfile
```

It should have the following content:

A code editor window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text "Dockerfile" on the right. The main area has a light yellow background and contains Dockerfile instructions in a monospaced font, with some words highlighted in green and blue.


```
Dockerfile

FROM node:8.12.0-slim
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 8080
CMD ["npm", "start"]
```

Take your time and inspect to inspect the `Dockerfile` .

- `FROM node:8.12.0-slim` is used to select the base image from which your image inherits from
- `WORKDIR /app` sets the current directory inside the container to be `/app` .
- `COPY . .` copies all the files on your current directory in your host to the container current directory `/app`
- `RUN npm install` installs all the dependencies inside the container
- `EXPOSE 8080` doesn't expose any port. The command is equivalent to a comment
- `CMD ["npm", "start"]` when the container starts, it executes this command by default

You can create the image with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker build -t hello-world .' in white text.

```
bash

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

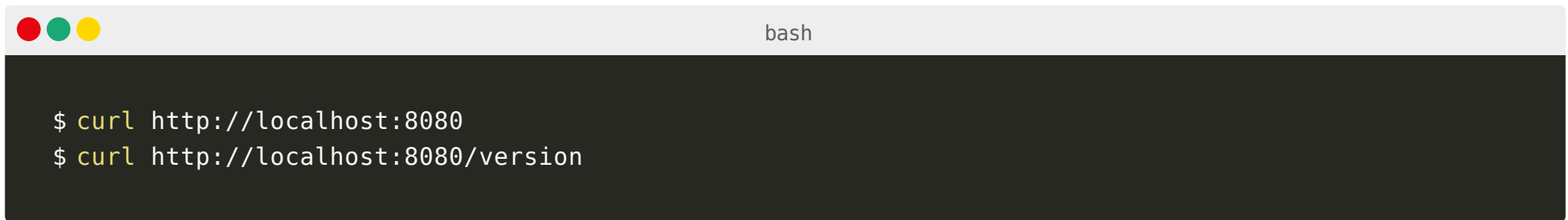
You can run the application with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti -p 8080:8080 hello-world' in white text.

```
bash

$ docker run -ti -p 8080:8080 hello-world
```

You can test the routes for the application with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains two commands in white text: '\$ curl http://localhost:8080' and '\$ curl http://localhost:8080/version'.

```
bash

$ curl http://localhost:8080
$ curl http://localhost:8080/version
```

3. Configurations

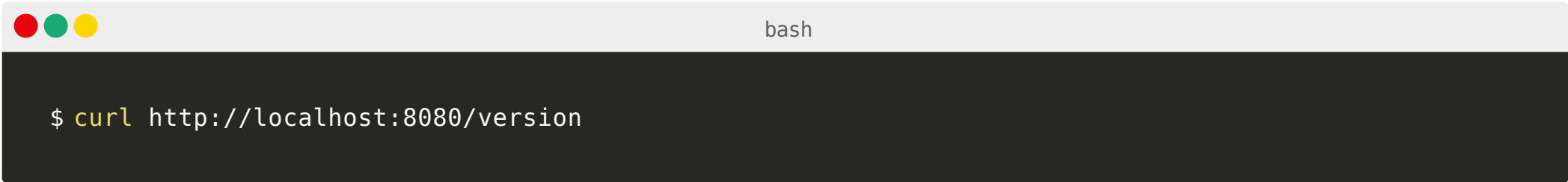
If you wish to inject a `VERSION` environment variable into your container you can with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains a white text command.

```
$ docker run -ti -p 8080:8080 -e VERSION=1 hello-world
```

Please note that you can inject multiple environment variables by adding the flag `-e` multiple times.

You can verify that the version was successfully updated with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area is dark gray and contains a white text command.

```
$ curl http://localhost:8080/version
```

Injecting environment variables in containers is used to decouple configuration from the application. In other words, you should create images for containers that can be run in all environments such as development, preproduction and production. If you need to read secrets or configs such as a database username or password, you should do so outside of the container. You shouldn't write your secrets in your image since you could expose production secrets in all your environments.

Chapter 9

Debugging

Few helpful tips to debug your containers.

1. Attaching to a running container

You can inspect the environment inside a container by attaching to it. Connecting to a running container is like being able to SSH into it. Let's start by launching an Ubuntu container with few environment variables:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker run -ti -e ENV=TEST ubuntu:18.10' in white text.

```
bash

$ docker run -ti -e ENV=TEST ubuntu:18.10
```

In another terminal, you can retrieve the id of the container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker ps' in white text.

```
bash

$ docker ps
```

You can attach to a running container with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ docker exec -ti <container_id> bash' in a light gray monospace font.

```
bash

$ docker exec -ti <container_id> bash
```

To inspect the value of `ENV` you can execute the following command from within the container:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash@container' on the right. The main area of the terminal is dark gray and contains the command '\$ echo \$ENV' in a light gray monospace font.

```
bash@container

$ echo $ENV
```

In this particular case, you decided to run the `bash` binary. But you could also run other commands such as `apt-cache` :



```
$ docker exec -ti <container_id> apt-cache
```

2. Common problems with attaching to containers

When running or attaching to running containers, you may not be able to execute the `bash` binary:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text "bash" on the right. The main area is dark gray and contains the command "\$ docker run -ti busybox bash" in a light gray monospace font.

```
bash

$ docker run -ti busybox bash
```

The command fails with an error. Indeed, there's no `bash` in busybox. But there's `sh` :

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text "bash" on the right. The main area is dark gray and contains the command "\$ docker run -ti busybox sh" in a light gray monospace font.

```
bash

$ docker run -ti busybox sh
```

In other words, you can only execute binaries that are part of your image.

Chapter 10

Labs

Congratulations, you made it this far! In this section you can test your Docker knowledge by solving five challenges of increasing difficulty. The challenges are designed to make you think. However, you might need a hint from time to time. Feel free to [ping us on Slack](#).

Chapter 11

Inspecting a container

You can find the container `learnk8s/labs-docker:inspect` on the official registry. The container has a `hello.txt` file in the `/app` folder.

Can you retrieve the content of that file?

Chapter 12

Attention to details

If you try to download and run `learnk8s/labs-docker:details` you will notice that there are some missing details. Can you provide the missing arguments and run the container? Running the container will reveal the code to solve the challenge.

Chapter 13

Building walls

A container image with a TCP server is available at `learnk8s/labs-docker:walls`. You should find the port of the server and send a specific payload to it to unlock the challenge.

Chapter 14

Encrypted message

You should create a `Dockerfile` that inherits from `learnk8s/labs-docker:encrypted`. The `Dockerfile` should create the following folders:

- `/data/folder1`
- `/home/ubuntu/Documents`

You should create a `message.txt` and copy it to `/home/ubuntu/Documents/message.txt` inside the container. The content of `/home/ubuntu/Documents/message.txt` should be:

ENCRYPTED MESSAGE

```
message.txt

rmUCPf4e7NySq91PSS8svLakkC79JkJe033KN2nr3TkPdcbmmz/bSrUatoskX5Pj78Qntd2cornRAjvF
snkgLAIiNF5cElGQD2cYI7fYN7WksfuE3pmNgSuuYJBtSTduT/Tc1IeoV/6xycjQ1cW1A+Hz1yG2qpqI
K5k+CaGnXbMco0W1/BEjMxC6Nh7QAtJSxpM+7s5Fb7cXpMVJuRSrQ/Briy8DNvqeD2TIiGAukvXzB9V
NNaYaClsvBPTpfNdgncae3do3FXjDXw4FltSzcm+Wwp+an1YNG0fXL+/qaV2A3Xsf2Cw1fVWI7KTx+M4
M4yFDB6D0Yl8mCrk8ecxllaeF2RaPn1eRk47efQhT1z0n0zvIltvLZbXf+4f+KoCI0q3cqLoBsA3FRDA
fDEWpQ05TMxnE2Ev6C2nLMdWFLv4WD6bTp5hxlMMbPdudz07rvJ52WcEz5m0hjoCCGXneX94YvPEM4t5C
lfXRenn7mV7Wxjn1TDSH8nzWwnxWsg1ELquYxqmnmnXZ6P4zN/K1atNue28sxaoTD6uHLCFDqY5MDbHw
e8u+HtrWpwfb026b/hLcWe3JoHk0kDR+Qwg04xnpQsFjkJzljjQp0pN0V+ICmsoRgoE106/IB608cGtG
F8E0EgBBHExkeiyYs49/La2FqBKRu5UTZ7bHVbzbLNY=
```

If you followed the instruction correctly, running the container will reveal the code to unlock the challenge.

Chapter 15

Debugging from scratch

The `learnk8s/labs-docker:scratch` container inherits from the `SCRATCH` image. Can you retrieve the content of the `/app/hello.txt` file?