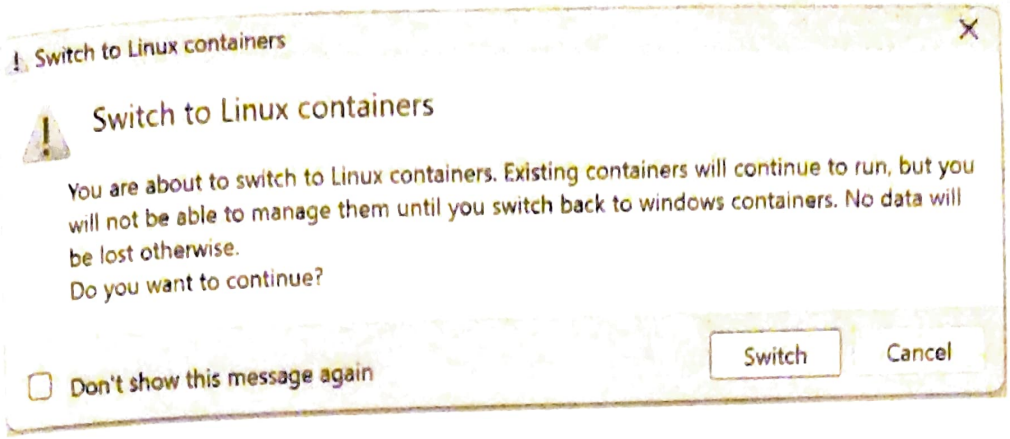

Figure 3-2. Switch to Linux containers confirmation

You can easily switch back and forth if you need to use both Linux and Windows containers.

Chocolatey installation

You can also install the Docker CLI tools using the popular Chocolatey (*https://docs.chocolatey.org/en-us/choco/setup*) package management system for Windows. If you take this approach, you should consider installing Vagrant for creating and managing your Linux VM. We'll discuss that shortly in "Non-Linux VM-Based Server" on page 40.

The Docker website (*https://docs.docker.com/engine/install*) has installation directions for additional environments online.

Docker Server

The Docker server is a separate binary from the client and is used to manage most of the work for which Docker is typically used. Next we will explore the most common ways to manage the Docker server.

Docker Desktop and Docker Community Edition already set up the server for you, so if you took that route, you do not need to do anything else besides ensuring that the server (`dockerd`) is running. On Windows and macOS, this typically just means starting the Docker application. On Linux, you may need to run the following `systemctl` commands to start the server.

systemd-Based Linux

Current Fedora and Ubuntu releases use `systemd` (*https://www.freedesktop.org/wiki/Software/systemd*) to manage processes on the system. Because you have already installed Docker, you can ensure that the server starts every time you boot the system by typing this:

```
$ sudo systemctl enable docker
```

This tells `systemd` to enable the `docker` service and start it when the system boots or switches into the default run level. To start the Docker server, type the following:

```
$ sudo systemctl start docker
```

Non-Linux VM-Based Server

If you are using Microsoft Windows or macOS in your Docker workflow, you will need a VM so that you can set up a Docker server for testing. Docker Desktop is convenient because it sets up this VM for you using the native virtualization technology on these platforms. If you are running an older version of Windows or cannot use Docker Desktop for other reasons, you should investigate Vagrant (*https://www.vagrantup.com*) to help you create and manage your Docker server Linux VM.

In addition to using Vagrant, you can also use other virtualization tools, like Lima on macOS (*https://github.com/lima-vm/lima*) or any standard hypervisor, to set up a local Docker server, depending on your preferences and needs.

Vagrant

Vagrant provides support for multiple hypervisors and can often be leverage
mimic even the most complex environments.

A common use case for leveraging Vagrant during
is to support testing on images that match your pro
Vagrant supports everything from broad distributions lik
Linux (*https://www.redhat.com/en/technologies/linux-platform*
Ubuntu (*https://ubuntu.com*) to finely focused at
CoreOS (*https://getfedora.org/en/coreos*).

You can easily install Vagrant on most platfo
package (*https://www.vagrantup.com/downloa*

systemd-Based Linux

Current Fedora and Ubuntu releases use `systemd` (*https://www.freedesktop.org/wiki/Software/systemd*) to manage processes on the system. Because you have already installed Docker, you can ensure that the server starts every time you boot the system by typing this:

```
$ sudo systemctl enable docker
```

This tells `systemd` to enable the `docker` service and start it when the system boots or switches into the default run level. To start the Docker server, type the following:

```
$ sudo systemctl start docker
```

Non-Linux VM-Based Server

If you are using Microsoft Windows or macOS in your Docker workflow, you will need a VM so that you can set up a Docker server for testing. Docker Desktop is convenient because it sets up this VM for you using the native virtualization technology on these platforms. If you are running an older version of Windows or cannot use Docker Desktop for other reasons, you should investigate Vagrant (*https://www.vagrantup.com*) to help you create and manage your Docker server Linux VM.

In addition to using Vagrant, you can also use other virtualization tools, like Lima on macOS (*https://github.com/lima-vm/lima*) or any standard hypervisor, to set up a local Docker server, depending on your preferences and needs.

Vagrant

Vagrant provides support for multiple hypervisors and can often be leveraged to mimic even the most complex environments.

A common use case for leveraging Vagrant during Docker development is to support testing on images that match your production environment. Vagrant supports everything from broad distributions like Red Hat Enterprise Linux (*https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux*) and Ubuntu (*https://ubuntu.com*) to finely focused atomic host distributions like Fedora CoreOS (*https://getfedora.org/en/coreos*).

You can easily install Vagrant on most platforms by downloading a self-contained package (*https://www.vagrantup.com/downloads.html*).

This Vagrant example is not secure and is not intended to be a recommendation. Instead, it is simply a demonstration of the basic requirements needed to set up a *remote* Docker server VM and make use of it. Securing the server is of critical importance.

Using Docker Desktop for development is often a better option, when possible.

You will need to have a hypervisor, like one of the following, fully installed on your system:

- VirtualBox (*https://www.virtualbox.org/wiki/Downloads*)
  - Freely available
  - Supports multiplatforms on most architectures
- VMware Workstation Pro/Fusion (*https://oreil.ly/4uNsR*)[2]
  - Commercial software
  - Supports multiplatforms on most architectures
- HyperV (*https://oreil.ly/agPTI*)[3]
  - Commercial software
  - Supports Windows on most architectures
- KVM (*https://www.linux-kvm.org*)
  - Freely available
  - Supports Linux on most architectures

By default, Vagrant assumes that you are using the VirtualBox hypervisor, but you can change it by using the `--provider` flag (*https://learn.hashicorp.com/tutorials/vagrant/getting-started-providers*) when using the `vagrant` command.

In the following example, you will create a Ubuntu-based Docker host running the Docker daemon. Then you will create a host directory with a name similar to *docker-host* and move into that directory:

```
$ mkdir docker-host
$ cd docker-host
```

In order to use Vagrant, you need to find a Vagrant Box (VM image) (*https://app.vagrantup.com/boxes/search*) that is compatible with your provisioner and architecture. In this example, we will use a Vagrant Box for the Virtual Box hypervisor.

2 Full URL: *https://www.vmware.com/products/workstation-pro.html*

3 Full URL: *https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v*

Virtual Box only works on Intel/AMD x86(64) systems, and the Vagrant Box we are using is specifically built for AMD64 systems.

Go ahead and create a new file called *Vagrantfile* with the following contents in it:

```
puts (<<-EOT)
  ------------------------------------------------------------
  [WARNING] This exposes an unencrypted Docker TCP port on the VM!!

  This is NOT secure and may expose your system to significant risk
  if left running and exposed to the broader network.
  ------------------------------------------------------------

EOT

$script = <<-SCRIPT
echo \'{"hosts": ["tcp://0.0.0.0:2375", "unix:///var/run/docker.sock"]}\' | \
sudo tee /etc/docker/daemon.json
sudo mkdir -p /etc/systemd/system/docker.service.d
echo -e \"[Service]\nExecStart=\nExecStart=/usr/bin/dockerd\" | \
sudo tee /etc/systemd/system/docker.service.d/docker.conf
sudo systemctl daemon-reload
sudo systemctl restart docker
SCRIPT

Vagrant.configure(2) do |config|

  # Pick a compatible Vagrant Box
  config.vm.box = 'bento/ubuntu-20.04'

  # Install Docker if it is not already on the VM image
  config.vm.provision :docker

  # Configure Docker to listen on an unencrypted local port
  config.vm.provision "shell",
    inline: $script,
    run: "always"

  # Port-forward the Docker port to
  # 12375 (or another open port) on our host machine
  config.vm.network "forwarded_port",
    guest: 2375,
    host: 12375,
    protocol: "tcp",
    auto_correct: true

end
```

You can retrieve a complete copy of this file by running this:

```
$ git clone https://github.com/bluewhalebook/\
docker-up-and-running-3rd-edition.git --config core.autocrlf=input
$ cd docker-up-and-running-3rd-edition/chapter_03/vagrant
$ ls Vagrantfile
```

You may need to remove the "\" in the `git clone` command and reassemble the URL into a single line. It is there because the command is too long for the standard printed page, and this should work in a standard Unix shell as long as there are no leading or trailing spaces in either line.

Ensure that you are in the directory with the *Vagrantfile*, and then run the following command to start the Vagrant VM.

This setup is provided as a simple example. It is not secure and should not be left running without ensuring that the server cannot be accessed from the broader network.

Docker maintains documentation on how to secure your Docker endpoint with SSH or TLS client certificates (*https://docs.docker.com/engine/security/protect-access*) and provides some additional information about the attack surface of the Docker daemon (*https://docs.docker.com/engine/security/#docker-daemon-attack-surface*).

```
$ vagrant up
…
Bringing machine 'default' up with 'virtualbox' provider…
==> default: Importing base box 'bento/ubuntu-20.04'…
==> default: Matching MAC address for NAT networking…
==> default: Checking if box 'bento/ubuntu-20.04' version '…' is up to date…
==> default: A newer version of the box 'bento/ubuntu-20.04' for provider…
==> default: available! You currently have version '…'. The latest is version
==> default: '202206.03.0'. Run `vagrant box update` to update.
==> default: Setting the name of the VM: vagrant_default_1654970697417_18732
==> default: Clearing any previously set network interfaces…
…
==> default: Running provisioner: docker…
    default: Installing Docker onto machine…
==> default: Running provisioner: shell…
    default: Running: inline script
    default: {"hosts": ["tcp://0.0.0.0:2375", "unix:///var/run/docker.sock"]}
    default: [Service]
    default: ExecStart=
    default: ExecStart=/usr/bin/dockerd
```

On macOS, you may see an error like this:

```
VBoxManage: error: Details: code NS_ERROR_FAILURE
(0x80004005), component MachineWrap, interface IMachine
```

This is due to the security features in macOS. A quick search should lead you to an online post that describes the fix (*https://scriptcrunch.com/solved-vboxmanage-error-component-machinewrap*).

Once the VM is running, you should be able to connect to the Docker server by running the following command and telling the Docker client where it should connect to with the -H argument:

```
$ docker -H 127.0.0.1:12375 version
Client:
 Cloud integration: v1.0.24
 Version:           20.10.14
 API version:       1.41
…

Server: Docker Engine - Community
 Engine:
  Version:          20.10.17
  API version:      1.41 (minimum version 1.12)
…
```

The output will provide you with version information about the various components that make up the Docker client and server.

Passing in the IP address and port every time you want to run a Docker command is not ideal, but luckily Docker can be set up to know about multiple Docker servers by using the `docker context` command. To start, let's check and see what context is currently in use. Take note of the entry that has an asterisk (*) next to it, which designates the current context:

```
$ docker context list
NAME        TYPE …  DOCKER ENDPOINT                  …
default *   moby …  unix:///var/run/docker.sock …
…
```

You can create a new context for the Vagrant VM and then make it active by running the following sequence of commands:

```
$ docker context create vagrant --docker host=tcp://127.0.0.1:12375
vagrant
Successfully created context "vagrant"

$ docker context use vagrant
vagrant
```

If you re-list all the contexts now, you should see something like this:

```
$ docker context list
NAME        TYPE … DOCKER ENDPOINT                …
default     moby … unix:///var/run/docker.sock …
vagrant *   moby … tcp://127.0.0.1:12375          …
…
```

With your current context set to vagrant, running `docker version` without the additional `-H` argument will still connect to the correct Docker server and return the same information as before.

To connect to a shell on the Vagrant-based VM, you can run the following:

```
$ vagrant ssh
…
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-91-generic x86_64)
…
vagrant@vagrant:~$ exit
```

Until you have time to secure this setup, it is best to go ahead and shut down the VM and set your context back to its original state:

```
$ vagrant halt
…
==> default: Attempting graceful shutdown of VM…

$ docker version
Cannot connect to … daemon at tcp://127.0.0.1:12375. Is the … daemon running?

$ docker context use default
default
```

If you are using macOS, you might want to take a look at Colima (*https://github.com/abiosoft/colima*), which makes it very easy to spin up and manage a flexible Docker or Kubernetes VM.

Testing the Setup

Once you have a working client and server set up, you are ready to test that everything is working. You should be able to run any one of the following commands on your local system to tell the Docker daemon to download the latest official container for that distribution and then launch it with a running Unix shell process.

This step is important to ensure that all the pieces are properly installed and communicating with one another as expected. It shows off one of the features of Docker: we can run containers based on any Linux distribution we like. In the next few steps, we'll run Linux containers based on Ubuntu, Fedora, and Alpine Linux. You don't need to run them all to prove that this works; running one of them will suffice.

Exploring the Docker Server

Although the Docker server is often installed, enabled, and run automatically, it's useful to see that running the Docker daemon manually on a Linux system (*https://docs.docker.com/engine/reference/commandline/dockerd*) can be as simple as typing something like this:

```
$ sudo dockerd -H unix:///var/run/docker.sock \
  --config-file /etc/docker/daemon.json
```

This section assumes that you are on the actual Linux server or VM that is running the Docker daemon. If you are using Docker Desktop on Windows or Mac, you won't be able to easily interact with the `dockerd` executable, as it is intentionally hidden from the end user, but we'll show you a trick in just a moment.

This command starts the Docker daemon, creates and listens to a Unix domain socket (`-H unix:///var/run/docker.sock`), and reads in the rest of the configuration from */etc/docker/daemon.json*. You're not likely to have to start the Docker server yourself, but that's what's going on behind the scenes. On non-Linux systems, you will typically have a Linux-based VM that hosts the Docker server. Docker Desktop sets up this VM for you in the background.

If you already have Docker running, executing the daemon again will fail because it can't use the same network port twice.

In most cases, it is very easy to SSH into your new Docker server and take a look around, but the seamless experience of Docker Desktop on a non-Linux system means it is often not apparent that Docker Desktop is leveraging a local VM on which to run the Docker daemon. Because the Docker Desktop VM is designed to be very small and very stable, it does not run an SSH daemon and is, therefore, a bit tricky to access.

If you are curious or just ever have a need to access the underlying VM, you can do it, but it requires a little advanced knowledge. We will talk about the command `nsenter` in much more detail in "nsenter" on page 334, but for now, if you would like to see the VM (or underlying host), you can run these commands:

```
$ docker container run --rm -it --privileged --pid=host debian \
  nsenter -t 1 -m -u -n -i sh

/ # cat /etc/os-release
PRETTY_NAME="Docker Desktop"
```