Running a Server Container

In this lesson, you will be introduced to server containers and learn how to run long-lived containers.

WE'LL COVER THE FOLLOWING

- Running a long-lived container
- Listening for Incoming Network Connections
- Wrapping It Up

We just saw how to run short-lived containers. They usually do some processing and display some output. However, there's a very common use for long-lived containers: server containers. Whether you want to host a web application, an API, or a database, you want a container that listens for incoming network connections and is potentially long-lived.

> A word of warning: it's best not to think about containers as long-lived, even when they are. Don't store information inside the containers. In other words, ensure your containers are stateless, not stateful. A stateless container never stores its state when it is run while stateful containers store some information about their state each time they are run. We'll see later on how and where to store your container's state. Docker containers are very stable, but the reason for having stateless containers is that this allows for easy scaling up and recovery. More about that later.

In short, a server container

- is long-lived
- listens for incoming network connections

How can we manage this? Read on.

Running a long-lived container

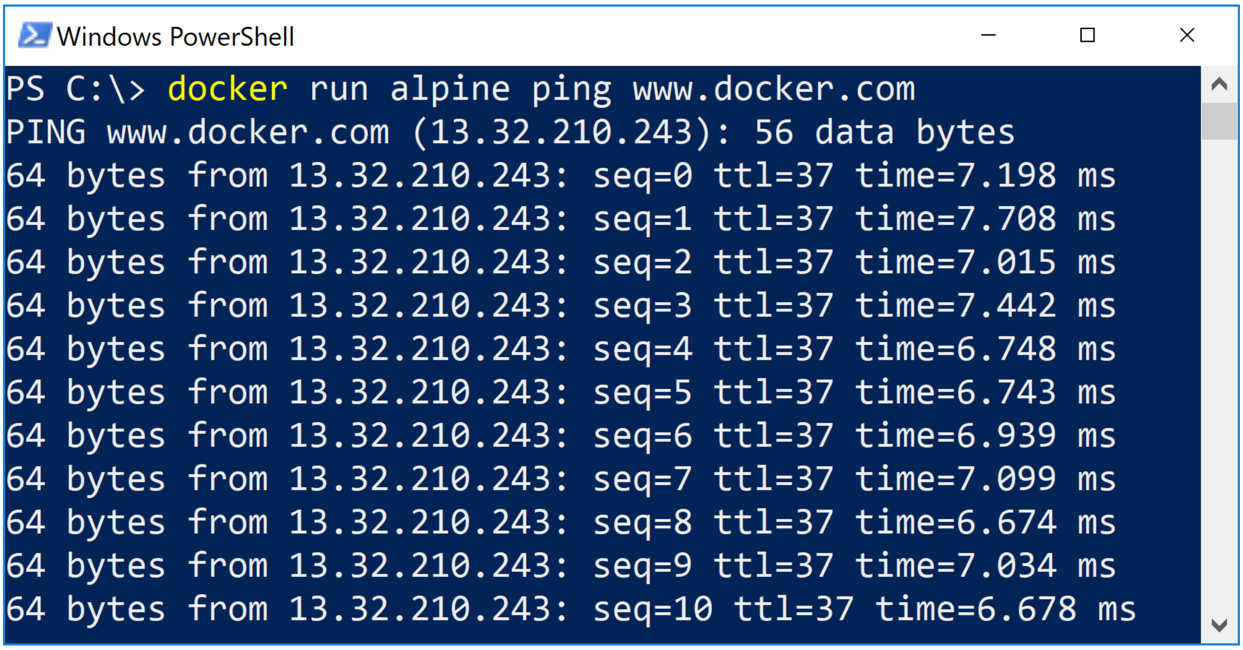
Up until now, we remained connected to the container from our command line using the *docker run* command, making it impractical for running long-lived containers.

To disconnect while allowing the long-lived container to continue running in the background, we use the *-d* or *–detach* switch on the *docker run* command.

Running a container as detached means that you immediately get your command-line back and the standard output from the container is not redirected to your command-line anymore.

Suppose I want to run a *ping* command. I can use a Linux alpine container for this:

```
docker run alpine ping www.docker.com
```

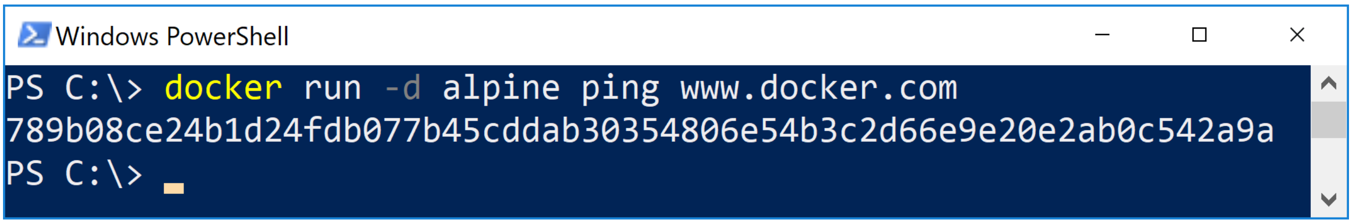


```
PS C:\> docker run alpine ping www.docker.com
PING www.docker.com (13.32.210.243): 56 data bytes
64 bytes from 13.32.210.243: seq=0 ttl=37 time=7.198 ms
64 bytes from 13.32.210.243: seq=1 ttl=37 time=7.708 ms
64 bytes from 13.32.210.243: seq=2 ttl=37 time=7.015 ms
64 bytes from 13.32.210.243: seq=3 ttl=37 time=7.442 ms
64 bytes from 13.32.210.243: seq=4 ttl=37 time=6.748 ms
64 bytes from 13.32.210.243: seq=5 ttl=37 time=6.743 ms
64 bytes from 13.32.210.243: seq=6 ttl=37 time=6.939 ms
64 bytes from 13.32.210.243: seq=7 ttl=37 time=7.099 ms
64 bytes from 13.32.210.243: seq=8 ttl=37 time=6.674 ms
64 bytes from 13.32.210.243: seq=9 ttl=37 time=7.034 ms
64 bytes from 13.32.210.243: seq=10 ttl=37 time=6.678 ms
```

The ping command doesn’t end since it keeps pinging the Docker server. That’s a long-lived container. I can detach from it using the *Ctrl-C* shortcut, and it keeps running in the background. However, it’s best to run it as detached from the beginning:

```
docker run -d alpine ping www.docker.com
```

Note the addition of a *-d* switch. When doing so, the container starts, but we don't see its output. Instead, the *docker run* command returns the ID of the container that was just created:



```
PS C:\> docker run -d alpine ping www.docker.com
789b08ce24b1d24fdb077b45cddab30354806e54b3c2d66e9e20e2ab0c542a9a
PS C:\> _
```

> The container ID is quite long. However, you don't need to write it entirely in your commands. As long as there is no ambiguity, you can use the beginning of the container ID in commands that require the container ID, like *docker logs* or *docker run*. Using the beginning of the container ID comes in handy when you're managing containers manually.

The container is still running. I can see it using a *docker ps* command that outputs something like:

Container ID	Image	Status
789b08ce24b1	alpine	Up 2 minutes

The status is telling us that the container has been running for 2 minutes and is still alive.

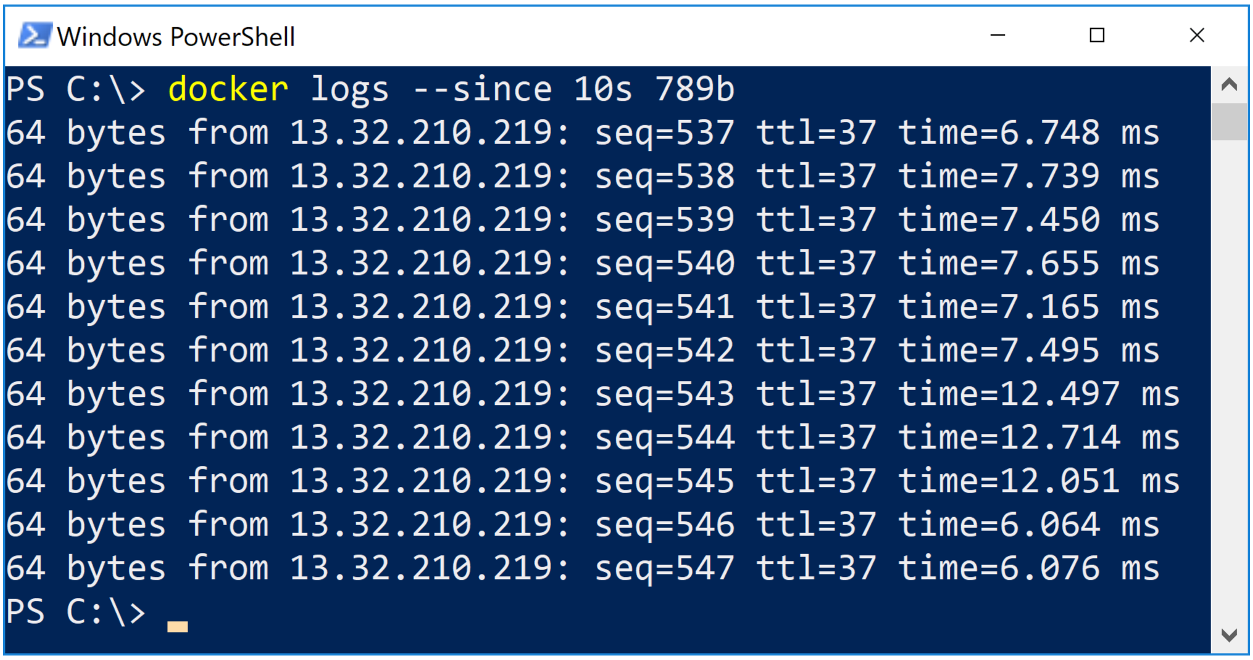
I can interact with the running container using the commands we saw above: *docker logs* to see its output, *docker inspect* to get detailed information and even *docker stop* in order to kill it.

Let's look at the standard output of the container using the following command (note I only use the beginning of the container ID):

```
docker logs 789b
```

The above command prints the whole standard output of the container from its beginning, which may be lengthy. But we can get a portion of the output using the *–from*, *–until*, or *–tail* switches. Let's see the most recent 10 seconds of logs for our running container:

```
docker logs --since 10s 789b
```



```
PS C:\> docker logs --since 10s 789b
64 bytes from 13.32.210.219: seq=537 ttl=37 time=6.748 ms
64 bytes from 13.32.210.219: seq=538 ttl=37 time=7.739 ms
64 bytes from 13.32.210.219: seq=539 ttl=37 time=7.450 ms
64 bytes from 13.32.210.219: seq=540 ttl=37 time=7.655 ms
64 bytes from 13.32.210.219: seq=541 ttl=37 time=7.165 ms
64 bytes from 13.32.210.219: seq=542 ttl=37 time=7.495 ms
64 bytes from 13.32.210.219: seq=543 ttl=37 time=12.497 ms
64 bytes from 13.32.210.219: seq=544 ttl=37 time=12.714 ms
64 bytes from 13.32.210.219: seq=545 ttl=37 time=12.051 ms
64 bytes from 13.32.210.219: seq=546 ttl=37 time=6.064 ms
64 bytes from 13.32.210.219: seq=547 ttl=37 time=6.076 ms
PS C:\>
```

> In real-world applications with multiple running containers, you would typically redirect your containers’ output to log management services. That being said, it can still be useful to get the last output of a container for debugging purposes.

A long-running container is bound to run for quite some time, but for now, I’m going to stop and clean up that container. So, I use the following commands:

```
docker stop 789b
docker rm 789b
docker ps -a
```

The last command is here so that I can confirm that the container is completely gone.

Listening for Incoming Network Connections

By default, a container runs in isolation, and as such, it doesn’t listen for incoming connections on the machine where it is running. You must explicitly open a port on the host machine and map it to a port on the container.

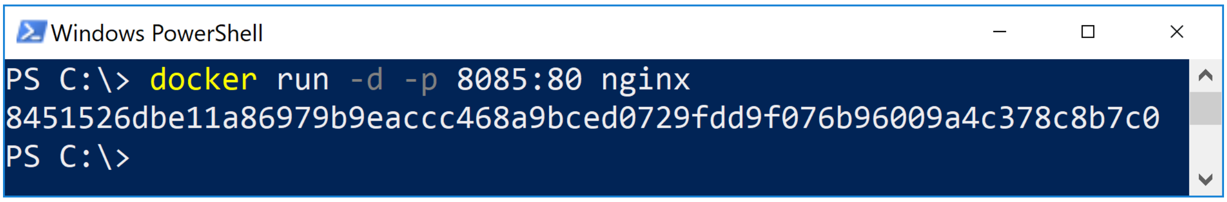
Suppose I want to run the NGINX web server. It listens for incoming HTTP requests on port 80 by default. If I simply run the server, my machine does not

route incoming requests to it unless I use the *-p* switch on the *docker run* command.

The *-p* switch takes two parameters; the incoming port you want to open on the host machine, and the port to which it should be mapped inside the container. For instance, here is how I state that I want my machine to listen for incoming connections on port 8085 and route them to port 80 inside a container that runs NGINX:

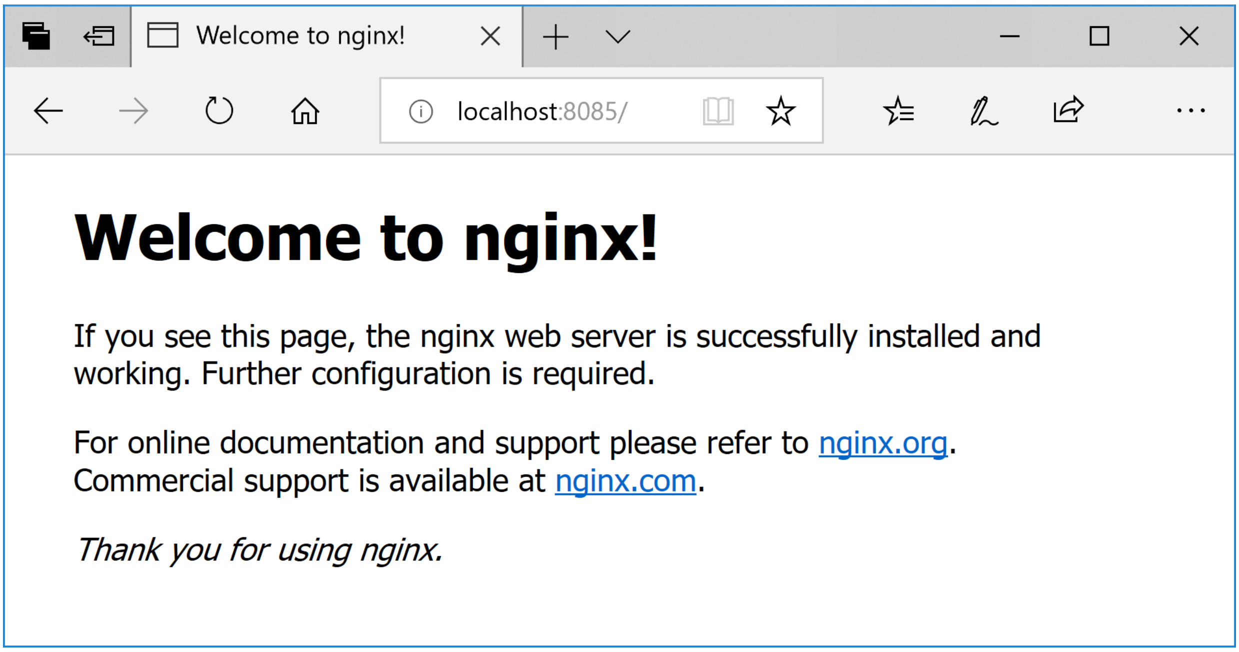
```
docker run -d -p 8085:80 nginx
```

Note the *-d* switch. It's not mandatory, but since I'm running a server container it's a good idea to keep in the background. An NGINX server container starts and I get its ID:



```
PS C:\> docker run -d -p 8085:80 nginx
8451526dbe11a86979b9eaccc468a9bced0729fdd9f076b96009a4c378c8b7c0
PS C:\>
```

Now I can run a browser and query that server using the `http://localhost:8085` URL:



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

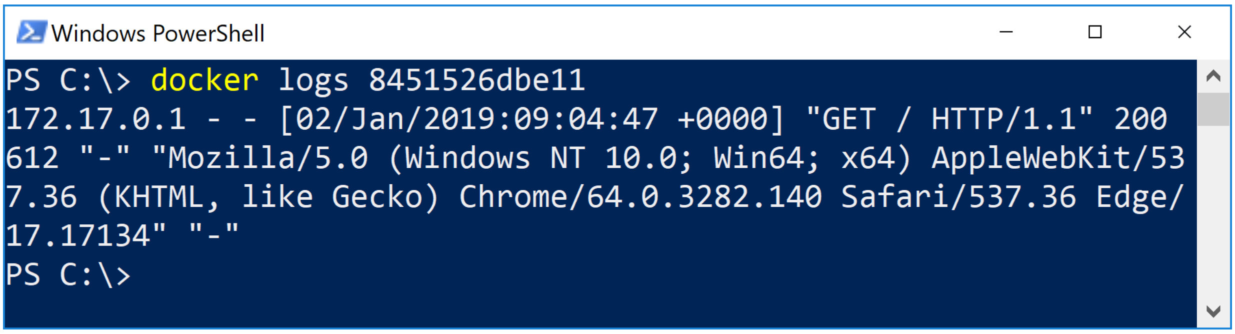
Thank you for using nginx.

Since that container is running in the background, its output isn't displayed on my terminal. However, I can still get it using a *docker logs* command:

my terminal. However, I can still get it using a *docker logs* command.

```
docker logs 8451526dbe11
```

We can see a trace of the browser’s HTTP request that NGINX received:



```
PS C:\> docker logs 8451526dbe11
172.17.0.1 - - [02/Jan/2019:09:04:47 +0000] "GET / HTTP/1.1" 200
612 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/
17.17134" "-"
PS C:\>
```

The NGINX container continues to run and serve incoming requests on port 8085. I can see it using the *docker ps* command. Let’s kill it so that I keep my machine free of unused containers:

```
docker stop 8451
docker rm 8451
```

Wrapping It Up

Did you notice we now have essentially the equivalent of a brand-new server? This means we can install whatever we want on it and trash it whenever we like.

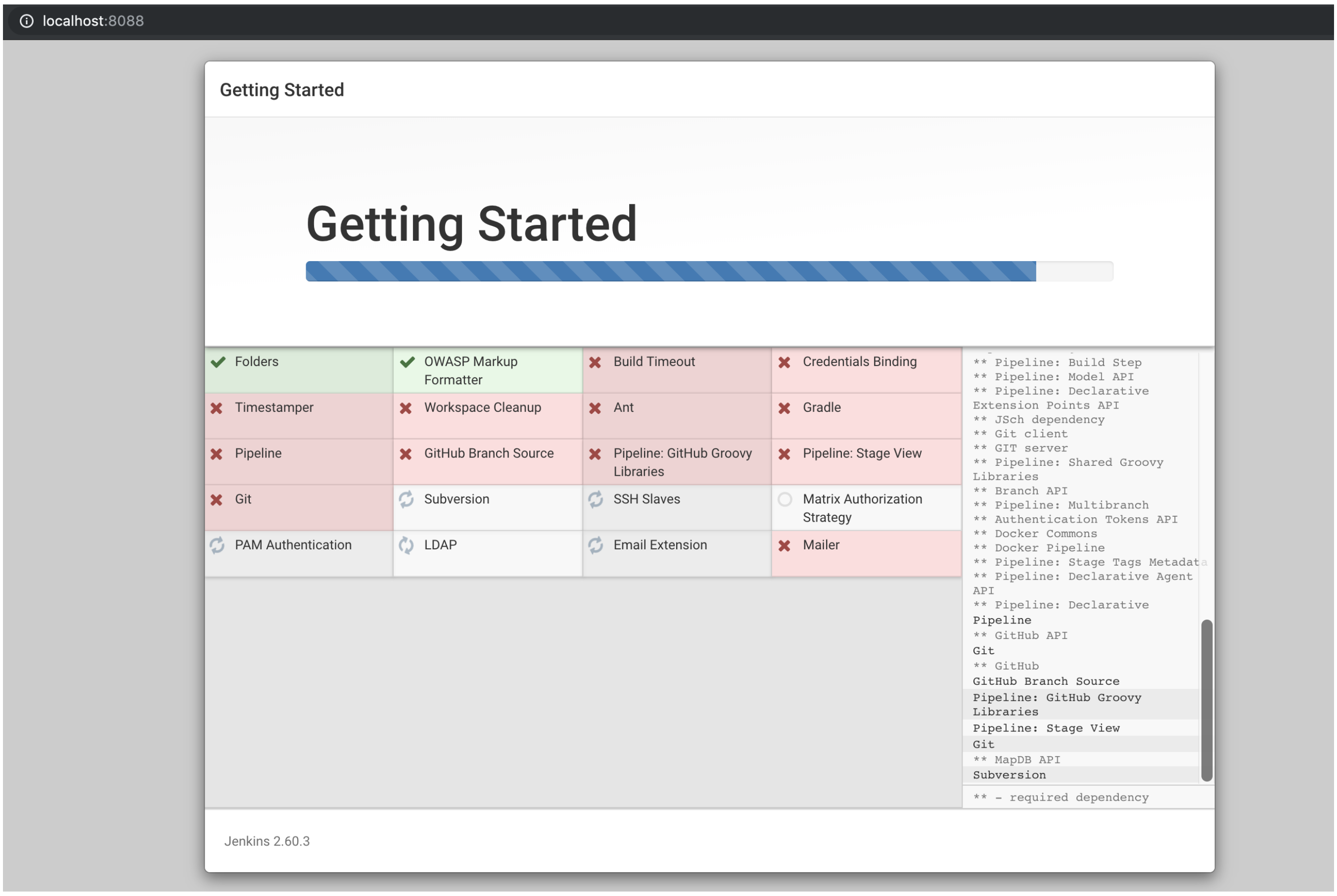
One thing I particularly like about containers is that they allow me to use any software without polluting my machine. Usually, you would hesitate before trying a new piece of software on your machine since it means installing several dependencies that may interfere with existing software and be leftover should you change your mind and uninstall the main software. Thanks to containers, I can even try big pieces of server software without polluting my machine.

Let me run a Jenkins server to illustrate that point. Jenkins is a full continuous integration server coded using Java. Thanks to Docker I don’t need to install Java or any dependency on my machine in order to run a Jenkins server. Jenkins listens by default on port 8080, so I can go away and type:

```
docker run -p 8088:8080 jenkins
```

> Note that I could add a *-d* switch since this is a long-running process. However, I am not using it here because I want to directly see the verbose output. For a real deployment, I would use the *-d* switch and inspect the output with the *docker logs* command when needed.

Then I can point my browser to `http://localhost:8088` and finish the setup:



localhost:8088

Getting Started

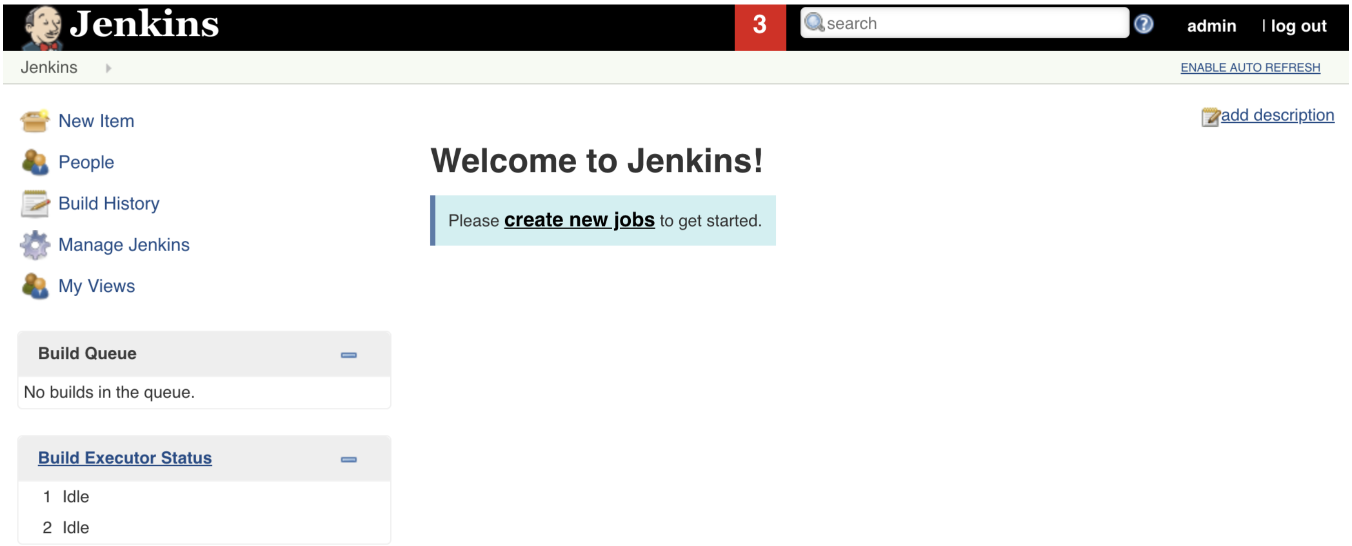
Getting Started

✓ Folders	✓ OWASP Markup Formatter	✗ Build Timeout	✗ Credentials Binding
✗ Timestamper	✗ Workspace Cleanup	✗ Ant	✗ Gradle
✗ Pipeline	✗ GitHub Branch Source	✗ Pipeline: GitHub Groovy Libraries	✗ Pipeline: Stage View
✗ Git	↻ Subversion	↻ SSH Slaves	○ Matrix Authorization Strategy
↻ PAM Authentication	↻ LDAP	↻ Email Extension	✗ Mailer

```
** Pipeline: Build Step
** Pipeline: Model API
** Pipeline: Declarative
Extension Points API
** JSch dependency
** Git client
** GIT server
** Pipeline: Shared Groovy
Libraries
** Branch API
** Pipeline: Multibranch
** Authentication Tokens API
** Docker Commons
** Docker Pipeline
** Pipeline: Stage Tags Metadata
** Pipeline: Declarative Agent
API
** Pipeline: Declarative
Pipeline
** GitHub API
Git
** GitHub
GitHub Branch Source
Pipeline: GitHub Groovy
Libraries
Pipeline: Stage View
Git
** MapDB API
Subversion
** - required dependency
```

Jenkins 2.60.3

And I get a full-blown Jenkins:



Welcome to Jenkins!

Please **create new jobs** to get started.

Build Queue

No builds in the queue.

Build Executor Status

1 Idle

2 Idle

Should I decide not to continue with Jenkins and try another continuous integration server, I can simply run the *docker stop* and *docker rm* commands. I could also run two separate Jenkins servers by just executing the *docker run* command again using another port.

Such isolation and ease of use at a very low resource cost is an enormous advantage of containers. Now that you saw how easy containers make managing server software on a single developer machine, imagine how powerful this is going to be on server machines. Thanks to containers, the Ops part of DevOps becomes smooth.

> When using such images, you could wonder about where the data is stored. Docker uses volumes for this, and we'll cover volumes later in this chapter. Also, databases may be needed for storing data, and those may be run in containers as well. For now, don't worry about that since we need to learn other things first.

Before we move on to volumes, try the exercise in the next lesson.