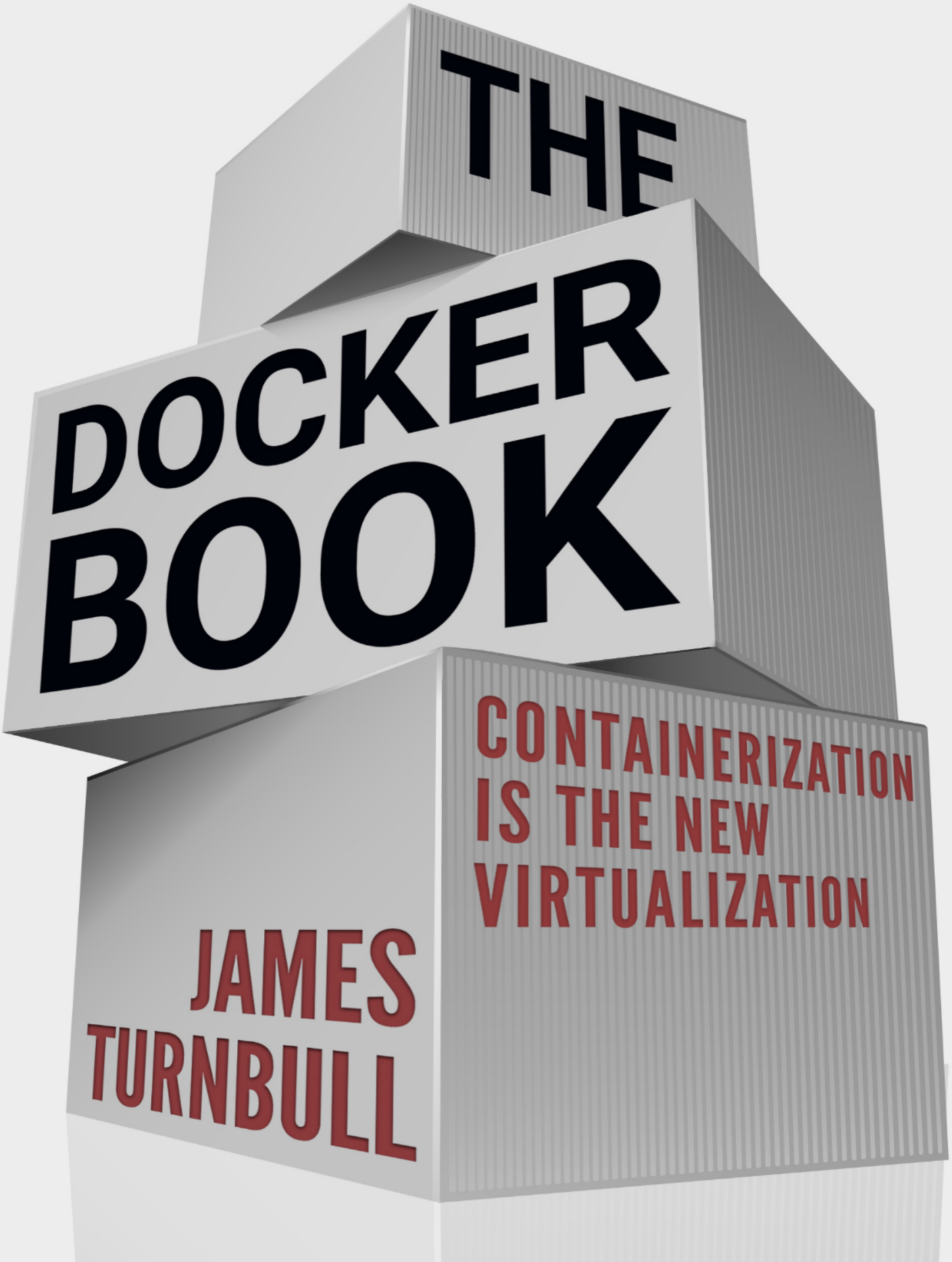
SIMPLE | SCALABLE | UP-TO-DATE



The Docker Book

James Turnbull

August 26, 2019

Version: v18.09.2 (c2c5fa8)

Website: The Docker Book

Some rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording, or otherwise, for commercial purposes without the prior permission of the publisher.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit here.

© Copyright 2016 - James Turnbull < >

ISBN 978-0-9888202-0-3

Table of contents

[illegible]

In Chapter 2, we learned how to install Docker. In Chapter 3, we learned how to use a variety of commands to manage Docker containers, including the [illegible] command.

Let's see the [illegible] command again.

```
[illegible]

[illegible]
```

This command will launch a new container called [illegible] from the [illegible] image and open a Bash shell.

In this chapter, we're going to explore Docker images: the building blocks from which we launch containers. We'll learn a lot more about Docker images, what they are, how to manage them, how to modify them, and how to create, store,

and share your own images. We'll also examine the repositories that hold images and the registries that store repositories.

[illegible]

Let's continue our journey with Docker by learning a bit more about Docker images. A Docker image is made up of filesystems layered over each other. At the base is a boot filesystem, [illegible], which resembles the typical Linux/Unix boot filesystem. A Docker user will probably never interact with the boot filesystem. Indeed, when a container has booted, it is moved into memory, and the boot filesystem is unmounted to free up the RAM used by the [illegible] disk image.

So far this looks pretty much like a typical Linux virtualization stack. Indeed, Docker next layers a root filesystem, [illegible], on top of the boot filesystem. This [illegible] can be one or more operating systems (e.g., a Debian or Ubuntu filesystem).

In a more traditional Linux boot, the root filesystem is mounted read-only and then switched to read-write after boot and an integrity check is conducted. In the Docker world, however, the root filesystem stays in read-only mode, and Docker takes advantage of a union mount to add more read-only filesystems onto the root filesystem. A union mount is a mount that allows several filesystems to be mounted at one time but appear to be one filesystem. The union mount overlays the filesystems on top of one another so that the resulting filesystem may contain files and subdirectories from any or all of the underlying filesystems.

Docker calls each of these filesystems images. Images can be layered on top of one another. The image below is called the parent image and you can traverse each layer until you reach the bottom of the image stack where the final image is called the base image. Finally, when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. This is where whatever processes we want our Docker container to run will execute.

This sounds confusing, so perhaps it is best represented by a diagram.

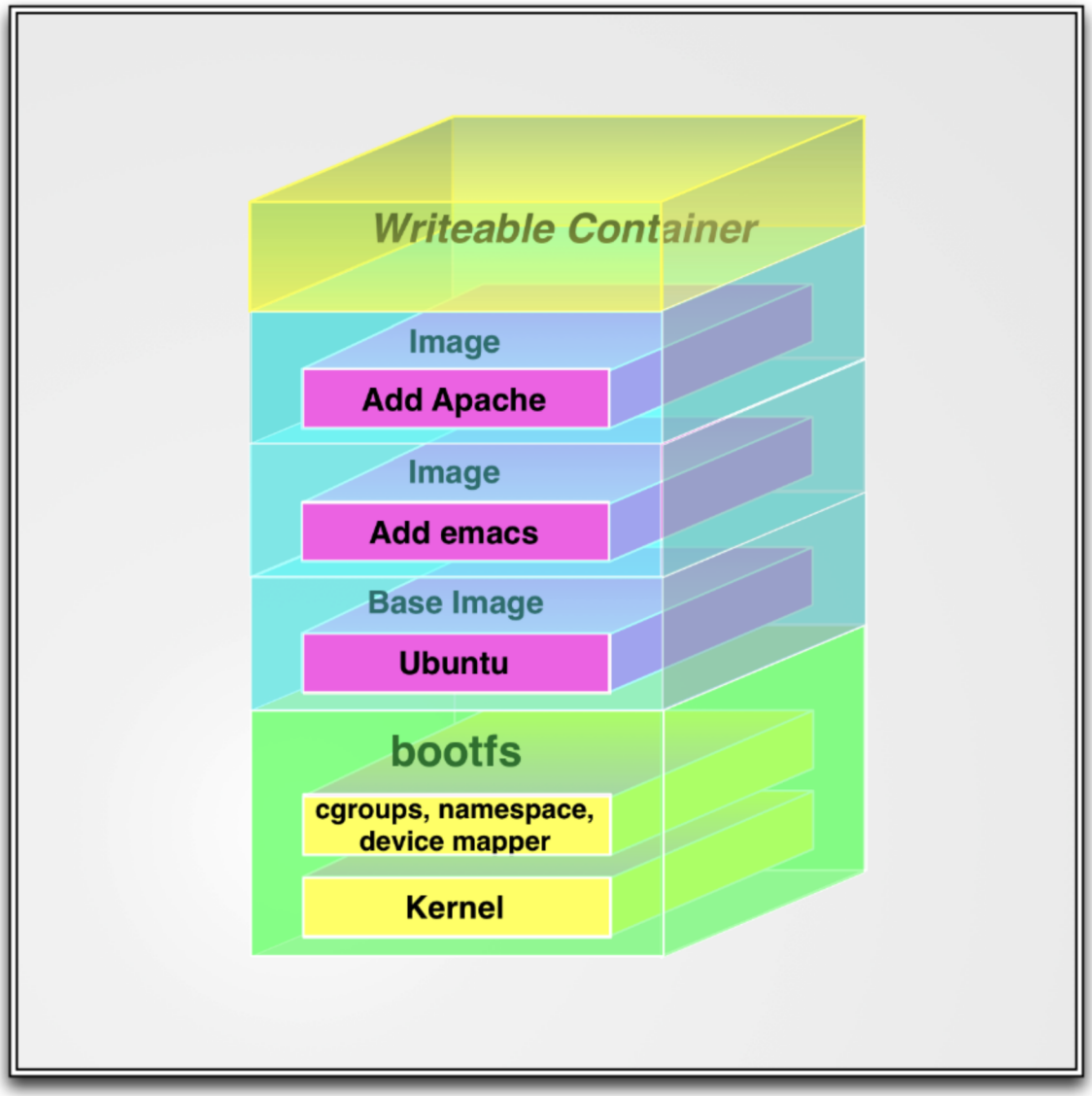


Figure 1.1: The Docker filesystem layers

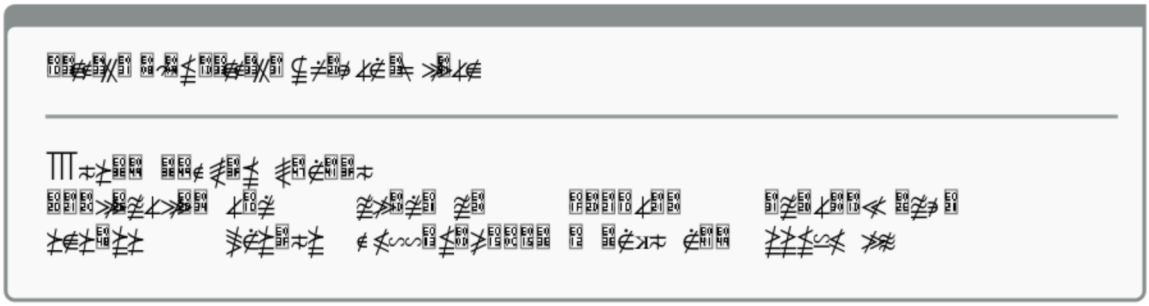
When Docker first starts a container, the initial read-write layer is empty. As changes occur, they are applied to this layer; for example, if you want to change

a file, then that file will be copied from the read-only layer below into the read-write layer. The read-only version of the file will still exist but is now hidden underneath the copy.

This pattern is traditionally called “copy on write” and is one of the features that makes Docker so powerful. Each read-only image layer is read-only; this image never changes. When a container is created, Docker builds from the stack of images and then adds the read-write layer on top. That layer, combined with the knowledge of the image layers below it and some configuration data, form the container. As we discovered in the last chapter, containers can be changed, they have state, and they can be started and stopped. This, and the image-layering framework, allows us to quickly build images and run containers with our applications and services.

[illegible]

Let’s get started with Docker images by looking at what images are available to us on our Docker host. We can do this using the [illegible] command.



We see that we’ve got an image, from a repository called [illegible]. So where does this image come from? Remember in Chapter 3, when we ran the [illegible] command, that part of the process was downloading an image? In our case, it’s the [illegible] image.

Local images live on our local Docker host in the directory. Each image will be inside a directory named for your storage driver; for example, or . You'll also find all your containers in the directory.

That image was downloaded from a repository. Images live inside repositories, and repositories live on registries. The default registry is the public registry managed by Docker, Inc., Docker Hub.

The Docker registry code is open source. You can also run your own registry, as we'll see later in this chapter. The Docker Hub product is also available as a commercial "behind the firewall" product called Docker Trusted Registry, formerly Docker Enterprise Hub.

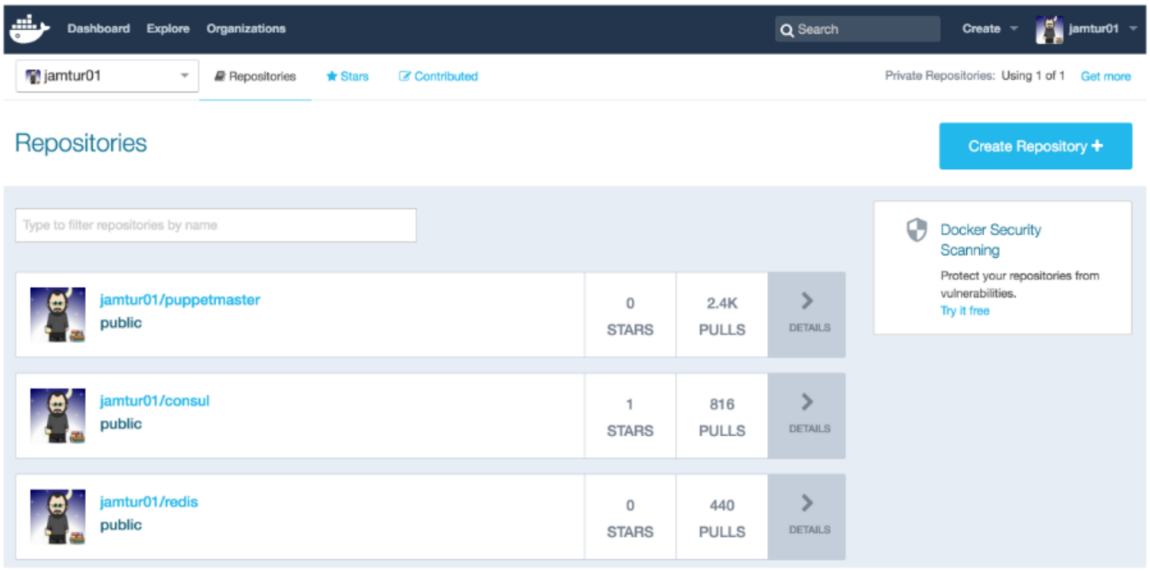
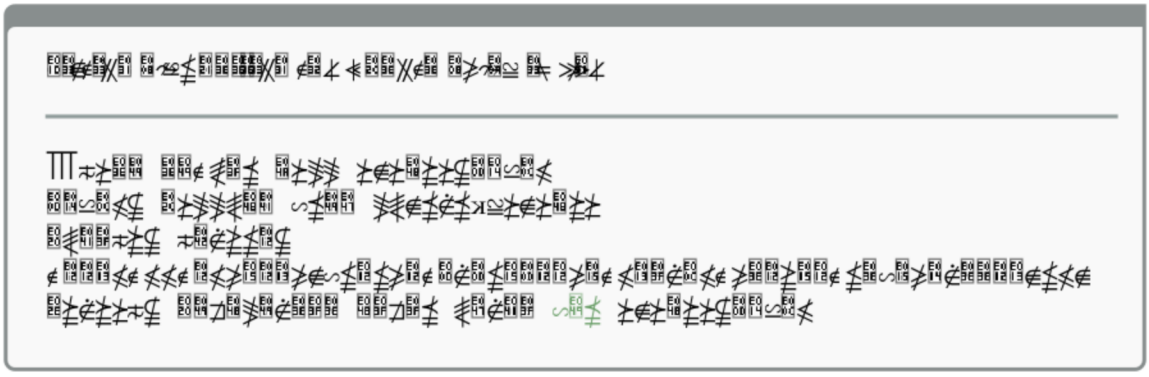


Figure 1.2: Docker Hub

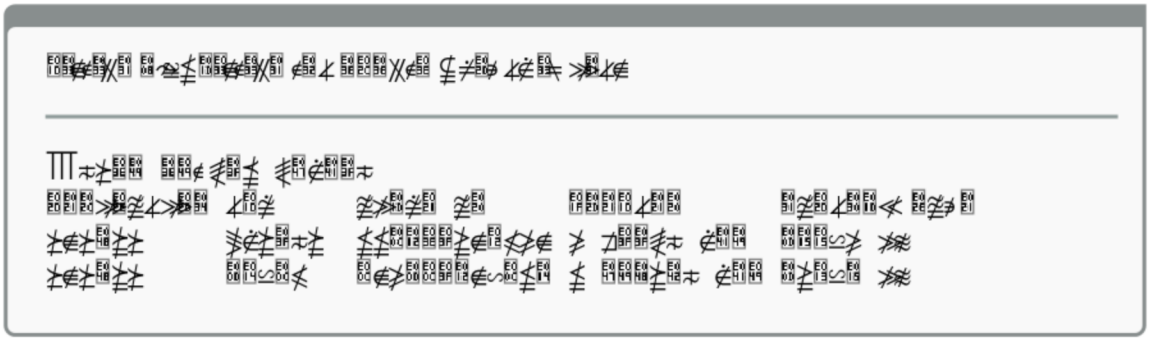
Inside Docker Hub (or on a Docker registry you run yourself), images are stored in repositories. You can think of an image repository as being much like a Git repository. It contains images, layers, and metadata about those images.

Each repository can contain multiple images (e.g., the [illegible] repository contains images for Ubuntu 12.04, 12.10, 13.04, 13.10, 14.04, 16.04, 18.04). Let's get another image from the [illegible] repository now.



Here we've used the [illegible] command to pull down the Ubuntu 18.04 image from the [illegible] repository.

Let's see what our [illegible] command reveals now.



[illegible] Throughout the book we use the [illegible] image. This is a reasonably heavyweight image, measuring a couple of hundred megabytes in size. If you'd prefer something smaller the Alpine Linux image is recommended as extremely

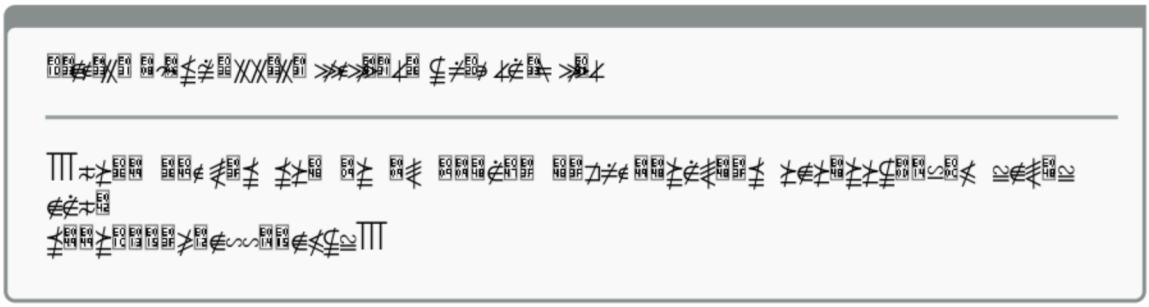
lightweight, generally 5Mb in size for the base image. Its image name is [illegible].

You can see we've now got the [illegible] Ubuntu image and the [illegible] image. This shows us that the [illegible] image is actually a series of images collected under a single repository.

[illegible] We call it the Ubuntu operating system, but really it is not the full operating system. It's a cut-down version with the bare runtime required to run the distribution.

We identify each image inside that repository by what Docker calls tags. Each image is being listed by the tags applied to it, so, for example, [illegible], [illegible], [illegible], or [illegible] and so on. Each tag marks together a series of image layers that represent a specific image (e.g., the [illegible] tag collects together all the layers of the Ubuntu 18.04 image). This allows us to store more than one image inside a repository.

We can refer to a specific image inside a repository by suffixing the repository name with a colon and a tag name, for example:



This launches a container from the [illegible] image, which is an Ubuntu 18.04 operating system.

It's always a good idea to build a container from specific tags. That way we'll know exactly what the source of our container is. There are differences, for example, between Ubuntu 16.04 and 18.04, so it would be useful to specifically state that we're using [illegible] so we know exactly what we're getting.

There are two types of repositories: user repositories, which contain images contributed by Docker users, and top-level repositories, which are controlled by the people behind Docker.

A user repository takes the form of a username and a repository name; for example, [illegible].

- Username: [illegible]
- Repository name: [illegible]

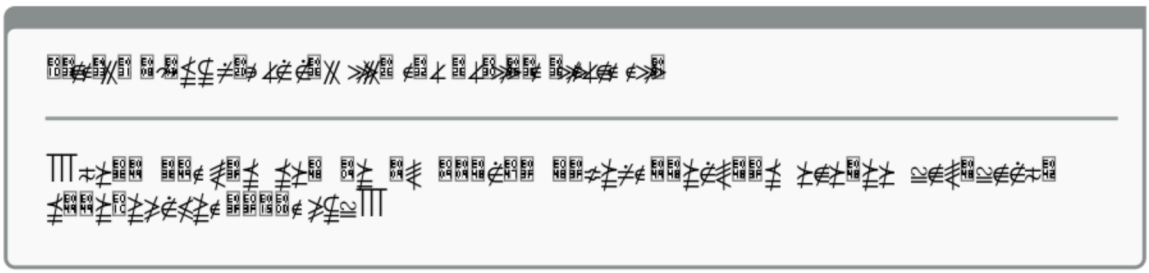
Alternatively, a top-level repository only has a repository name like [illegible]. The top-level repositories are managed by Docker Inc and by selected vendors who provide curated base images that you can build upon (e.g., the Fedora team provides a [illegible] image). The top-level repositories also represent a commitment from vendors and Docker Inc that the images contained in them are well constructed, secure, and up to date.

In Docker 1.8 support was also added for managing the content security of images, essentially signed images. This is currently an optional feature and you can read more about it on the Docker blog.

| [illegible] User-contributed images are built by members of the Docker community. You should use them at your own risk: they are not validated or verified in any way by Docker Inc.

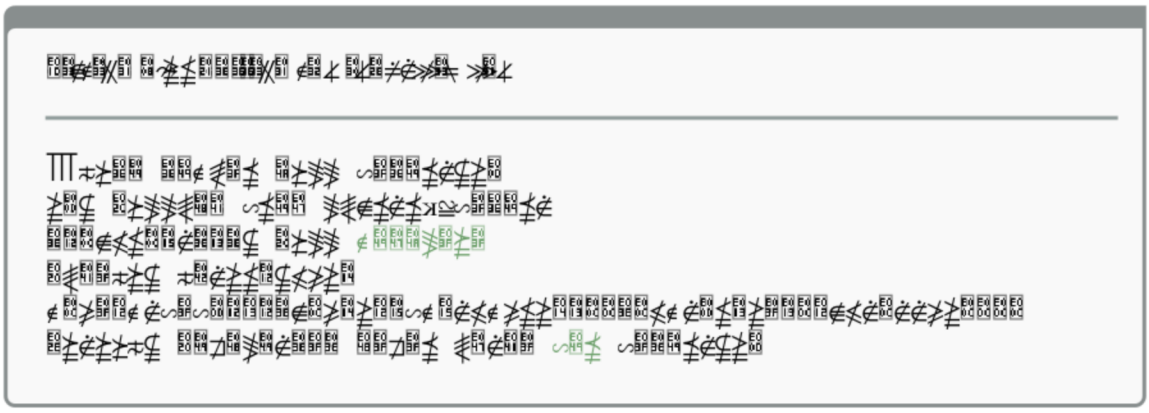
[illegible]

When we run a container from images with the [illegible] command, if the image isn't present locally already then Docker will download it from the Docker Hub. By default, if you don't specify a specific tag, Docker will download the [illegible] tag, for example:



Will download the [illegible] image if it isn't already present on the host.

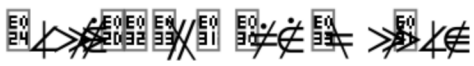
Alternatively, we can use the [illegible] command to pull images down ourselves preemptively. Using [illegible] saves us some time launching a container from a new image. Let's see that now by pulling down the 'fedora:21 base image.



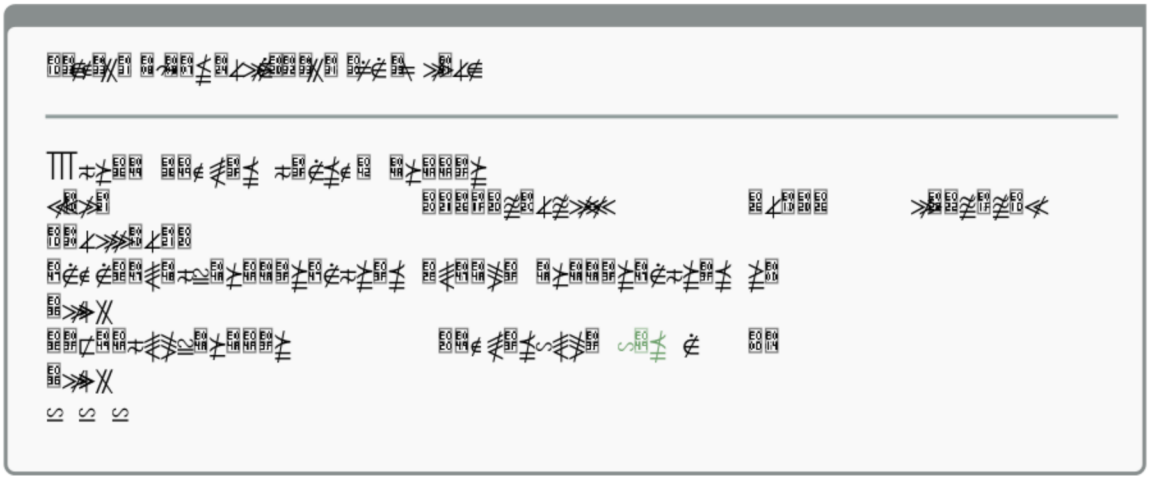
Let's see this new image on our Docker host using the [illegible] command. This time, however, let's narrow our review of the images to only the [illegible] images. To do so, we can specify the image name after the [illegible] command.

We see that the ubuntu:latest image has been downloaded. We could also download another tagged image using the docker pull command.

This would have just pulled the ubuntu:18.04 image.



We can also search all of the publicly available images on Docker Hub using the docker search command:



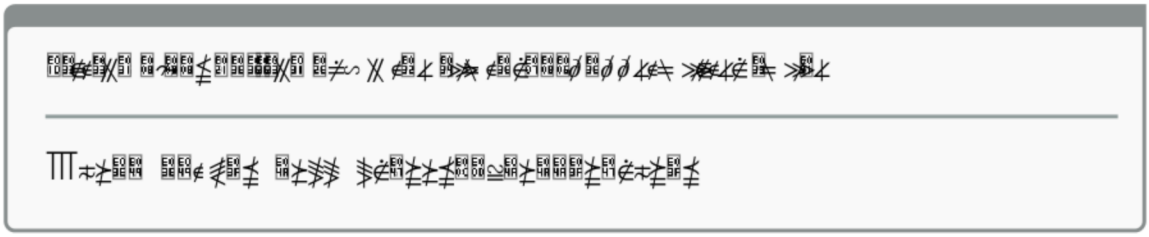
You can also browse the available images online at Docker Hub.

Here, we've searched the Docker Hub for the term [illegible]. It'll search images and return:

- Repository names
- Image descriptions
- Stars - these measure the popularity of an image
- Official - an image managed by the upstream developer (e.g., the [illegible] image managed by the Fedora team)
- Automated - an image built by the Docker Hub's Automated Build process

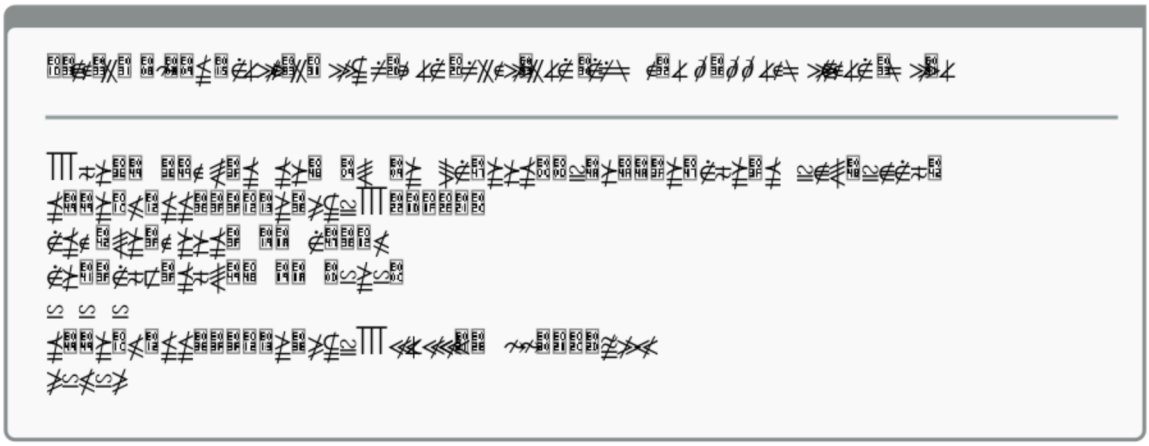
We'll see more about Automated Builds later in this chapter.

Let's pull down an image.



This will pull down the [illegible] image (which, by the way, contains a pre-installed Puppet master server).

We can then use this image to build a new container. Let's do that now using the [illegible] command again.



You can see we've launched a new container from our [illegible] image. We've launched the container interactively and told the container to run the Bash shell. Once inside the container's shell, we've run Facter (Puppet's inventory application), which was pre-installed on our image. From inside the container, we've also run the [illegible] binary to confirm it is installed.

[illegible]

So we've seen that we can pull down pre-prepared images with custom contents. How do we go about modifying our own images and updating and managing them?

There are two ways to create a Docker image:

- Via the [illegible] command
- Via the [illegible] command with a [illegible]

The [illegible] method is not currently recommended, as building with a [illegible] is far more flexible and powerful, but we'll demonstrate it to you for the sake of completeness. After that, we'll focus on the recommended method of building Docker images: writing a [illegible] and using the [illegible] command.

[illegible] We don't generally actually "create" new images; rather, we build new images from existing base images, like the [illegible] or [illegible] images we've already seen. If you want to build an entirely new base image, you can see some information on this in this guide.

[illegible]

A big part of image building is sharing and distributing your images. We do this by pushing them to the Docker Hub or your own registry. To facilitate this, let's start by creating an account on the Docker Hub. You can join Docker Hub here.

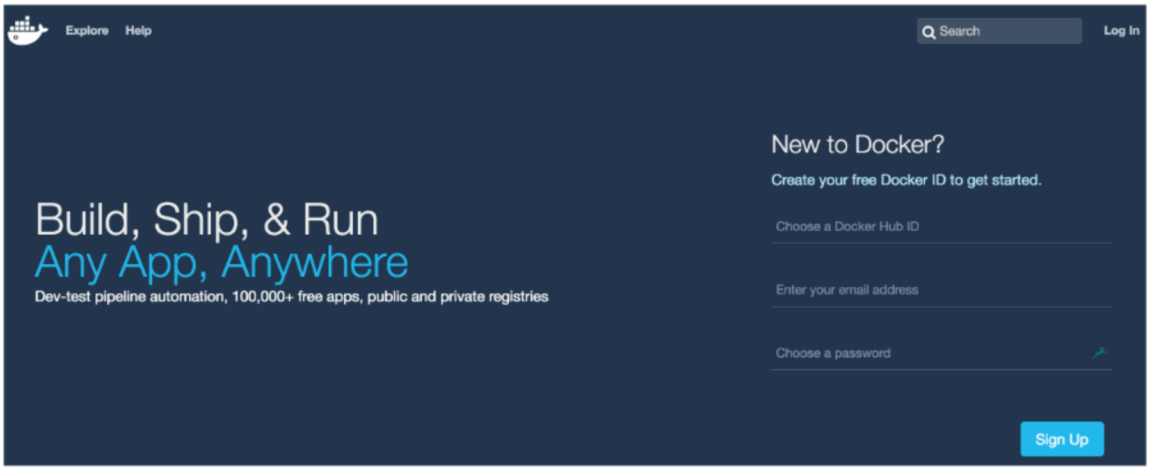
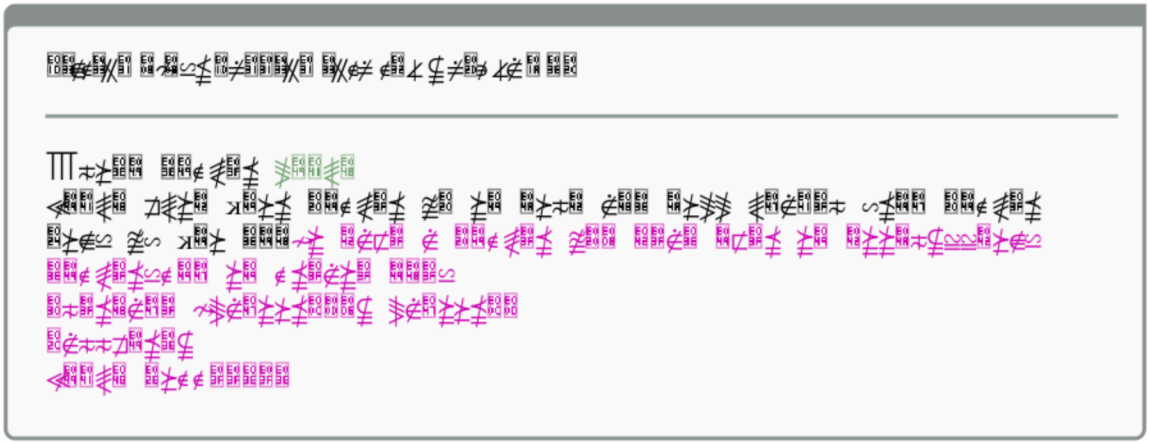


Figure 1.3: Creating a Docker Hub account.

Create an account and verify your email address from the email you'll receive after signing up.

Now let's test our new account from Docker. To sign into the Docker Hub you can use the [illegible] command.



This command will log you into the Docker Hub and store your credentials for future use. You can use the [illegible] command to log out from a registry server.

Your credentials will be stored in the file. Since Docker 1.7.0 this is now .

Creating images from containers

The first method of creating images uses the command. You can think about this method as much like making a commit in a version control system. We create a container, make changes to that container as you would change code, and then commit those changes to a new image.

Let's start by creating a container from the image we've used in the past.

```
```

```
```

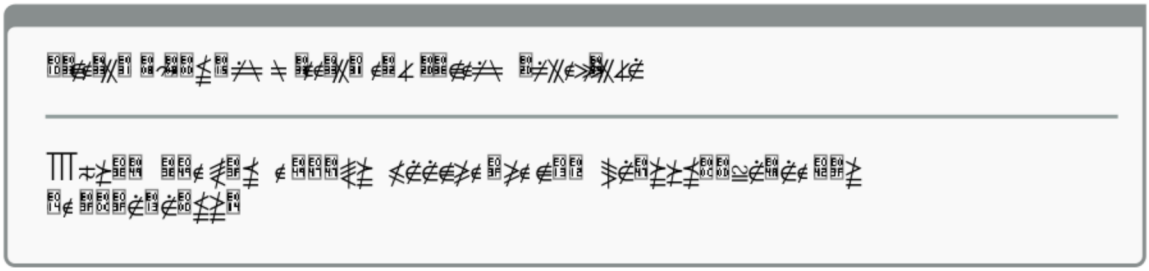
Next, we'll install Apache into our container.

```
```

```
```

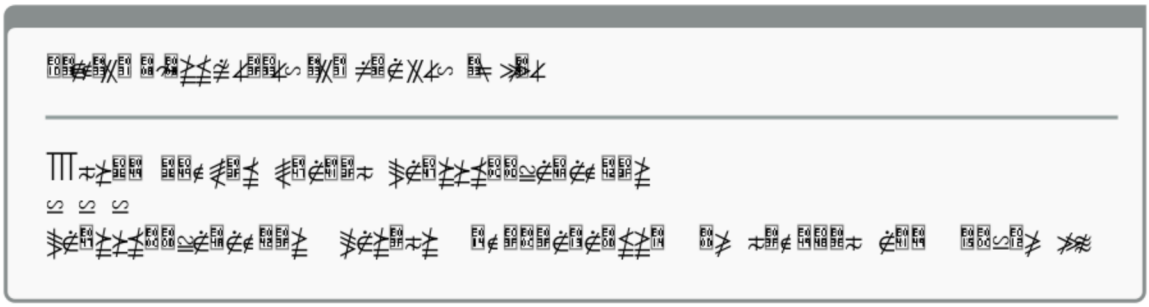
We've launched our container and then installed Apache within it. We're going to use this container as a web server, so we'll want to save it in its current state. That will save us from having to rebuild it with Apache every time we create a

new container. To do this we exit from the container, using the [illegible] command, and use the [illegible] command.

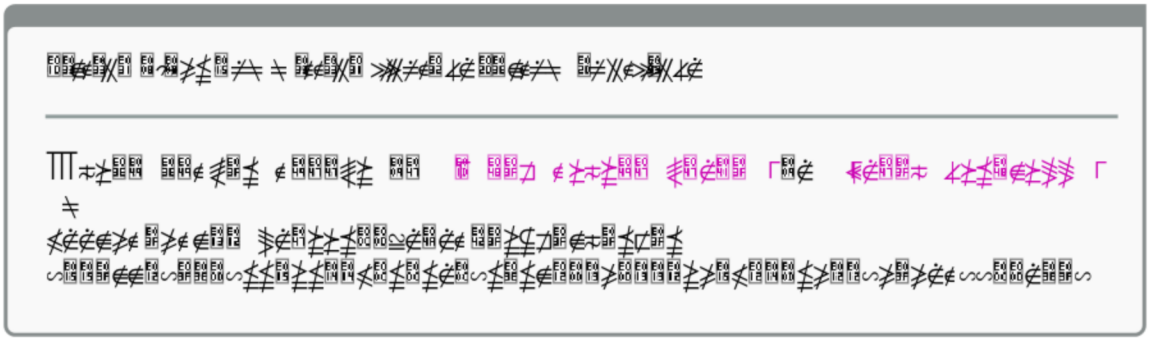


You can see we've used the [illegible] command and specified the ID of the container we've just changed (to find that ID you could use the [illegible] command to return the ID of the last created container) as well as a target repository and image name, here [illegible]. Of note is that the [illegible] command only commits the differences between the image the container was created from and the current state of the container. This means updates are lightweight.

Let's look at our new image.

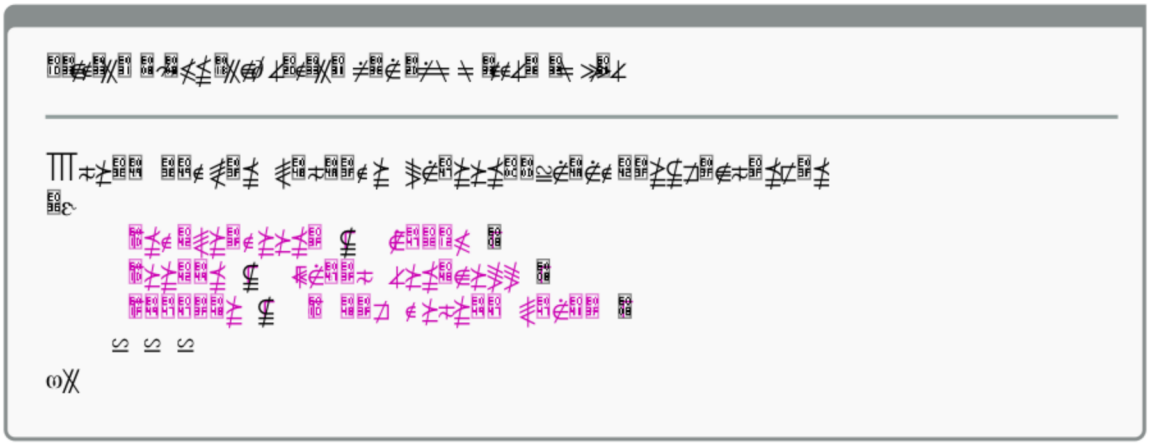


We can also provide some more data about our changes when committing our image, including tags. For example:



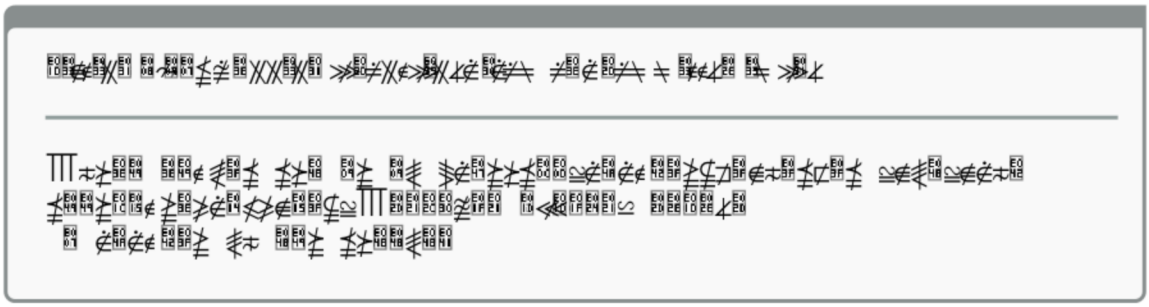
Here, we've specified some more information while committing our new image. We've added the [illegible] option which allows us to provide a commit message explaining our new image. We've also specified the [illegible] option to list the author of the image. We've then specified the ID of the container we're committing. Finally, we've specified the username and repository of the image, [illegible], and we've added a tag, [illegible], to our image.

We can view this information about our image using the [illegible] command.



You can find a full list of the [illegible] flags here.

If we want to run a container from our new image, we can do so using the [illegible] command.



You'll note that we've specified our image with the full tag: [illegible].

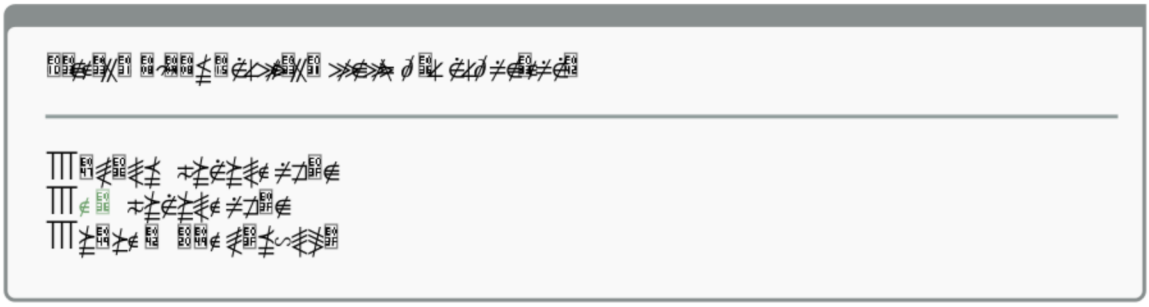
[illegible]

We don't recommend the [illegible] approach. Instead, we recommend that you build images using a definition file called a [illegible] and the [illegible] command. The [illegible] uses a basic DSL (Domain Specific Language) with instructions for building Docker images. We recommend the [illegible] approach over [illegible] because it provides a more repeatable, transparent, and idempotent mechanism for creating images.

Once we have a [illegible] we then use the [illegible] command to build a new image from the instructions in the [illegible].

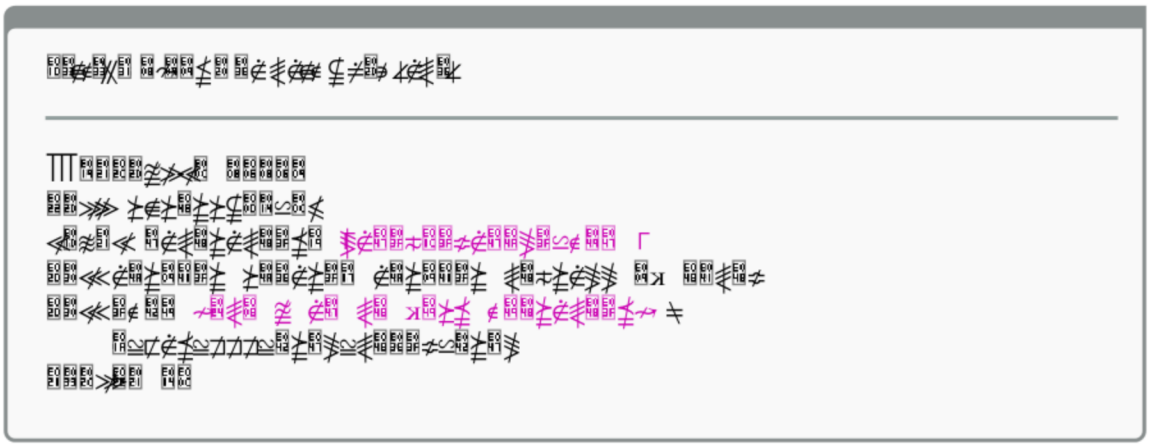
[illegible]

Let's now create a directory and an initial [illegible]. We're going to build a Docker image that contains a simple web server.



We've created a directory called [illegible] to hold our [illegible]. This directory is our build environment, which is what Docker calls a context or build context. Docker will upload the build context, as well as any files and directories contained in it, to our Docker daemon when the build is run. This provides the Docker daemon with direct access to any code, files or other data you might want to include in the image.

We've also created an empty [illegible] file to get started. Now let's look at an example of a [illegible] to create a Docker image that will act as a Web server.



The [illegible] contains a series of instructions paired with arguments. Each instruction, for example [illegible], should be in upper-case and be followed by an argument: [illegible]. Instructions in the [illegible] are processed from the top down, so you should order them accordingly.

Each instruction adds a new layer to the image and then commits the image.

Docker executing instructions roughly follow a workflow:

- Docker runs a container from the image.
- An instruction executes and makes a change to the container.
- Docker runs the equivalent of [illegible] to commit a new layer.
- Docker then runs a new container from this new image.
- The next instruction in the file is executed, and the process repeats until all instructions have been executed.

This means that if your [illegible] stops for some reason (for example, if an instruction fails to complete), you will be left with an image you can use. This is highly useful for debugging: you can run a container from this image interactively and then debug why your instruction failed using the last image created.

[illegible] The [illegible] also supports comments. Any line that starts with a [illegible] is considered a comment. You can see an example of this in the first line of our [illegible].

The first instruction in a [illegible] must be [illegible]. The [illegible]instruction specifies an existing image that the following instructions will operate on; this image is called the base image.

In our sample [illegible] we've specified the [illegible] image as our base image. This specification will build an image on top of an Ubuntu 18.04 base operating system. As with running a container, you should always be specific about exactly from which base image you are building.

Next, we've specified the [illegible]instruction with a value of 'maintainer = "james@example.com", which tells Docker who the author of the image is and what their email address is. This is useful for specifying an owner and contact for an image.

[illegible] This [illegible] instructions replaces the [illegible] instruction which was deprecated in Docker 1.13.0.

We've followed these instructions with two [illegible] instructions. The [illegible] instruction executes commands on the current image. The commands in our example: updating the installed APT repositories and installing the [illegible] package and then creating the [illegible] file containing some example text. As we've discovered, each of these instructions will create a new layer and, if successful, will commit that layer and then execute the next instruction.

By default, the [illegible] instruction executes inside a shell using the command wrapper [illegible]. If you are running the instruction on a platform without a shell or you wish to execute without a shell (for example, to avoid shell string munging), you can specify the instruction in [illegible] format:

```
[illegible]

[illegible]
```

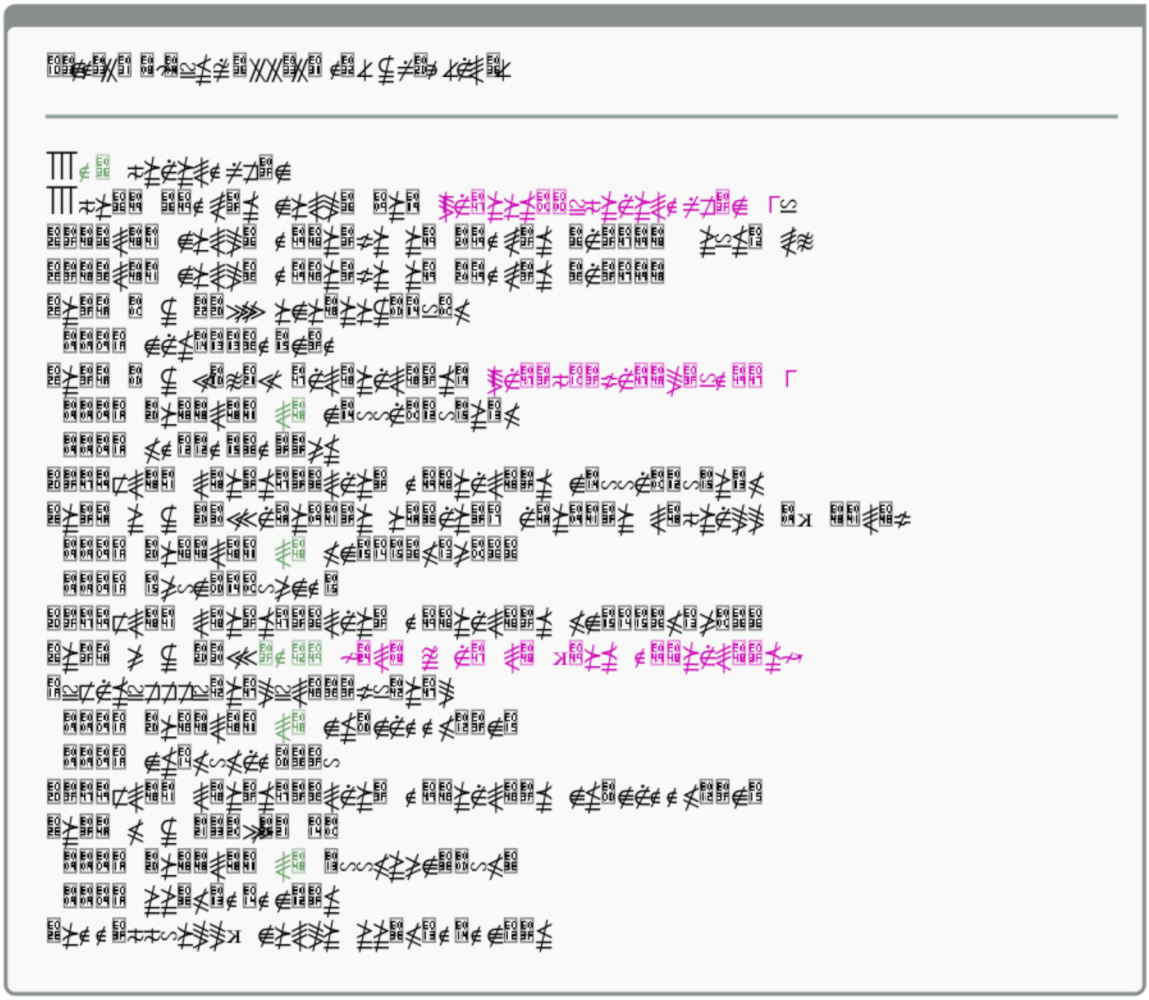
We use this format to specify an array containing the command to be executed and then each parameter to pass to the command.

Next, we've specified the [illegible] instruction, which tells Docker that the application in this container will use this specific port on the container. That doesn't mean you can automatically access whatever service is running on that port (here, port [illegible]) on the container. For security reasons, Docker doesn't open the port automatically, but waits for you to do it when you run the container using the [illegible] command. We'll see this shortly when we create a new container from this image.

You can specify multiple [illegible] instructions to mark multiple ports to be exposed.

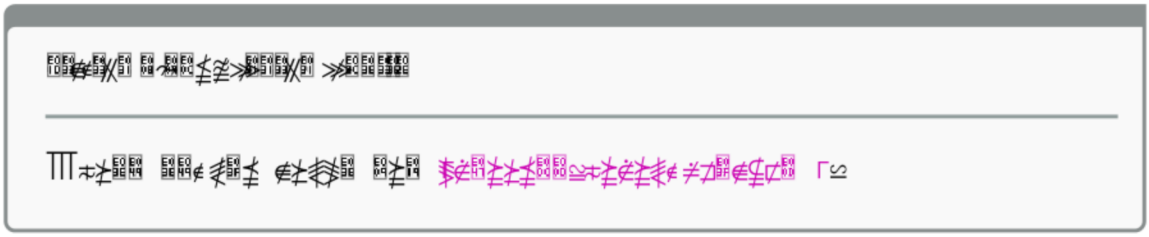
Docker also uses the instruction to help link together containers, which we'll see in Chapter 5. You can expose ports at run time with the command with the option.

All of the instructions will be executed and committed and a new image returned when we run the command. Let's try that now:



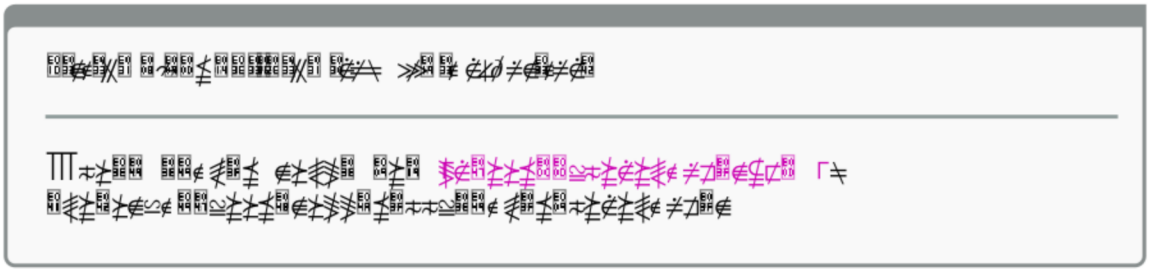
We've used the [illegible] command to build our new image. We've specified the [illegible] option to mark our resulting image with a repository and a name, here the [illegible] repository and the image name [illegible]. I strongly recommend you always name your images to make it easier to track and manage them.

You can also tag images during the build process by suffixing the tag after the image name with a colon, for example:



If you don't specify any tag, Docker will automatically tag your image as .

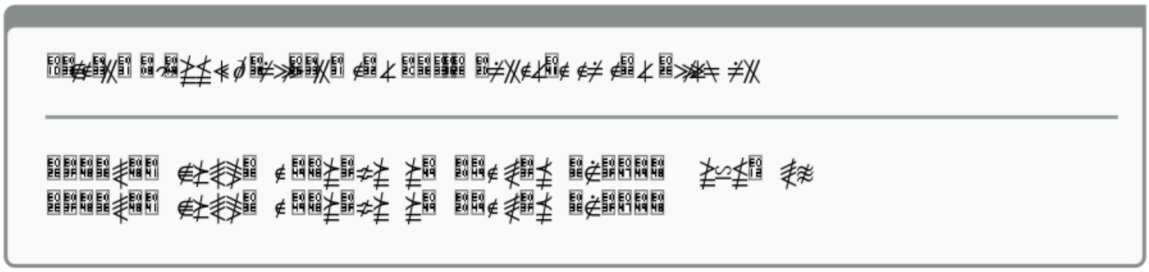
The trailing tells Docker to look in the local directory to find the . You can also specify a Git repository as a source for the as we see here:



Here Docker assumes that there is a located in the root of the Git repository.

Since Docker 1.5.0 and later you can also specify a path to a file to use as a build source using the flag. For example, . The file specified doesn't need to be called but must still be within the build context.

But back to our [illegible] process. You can see that the build context has been uploaded to the Docker daemon.



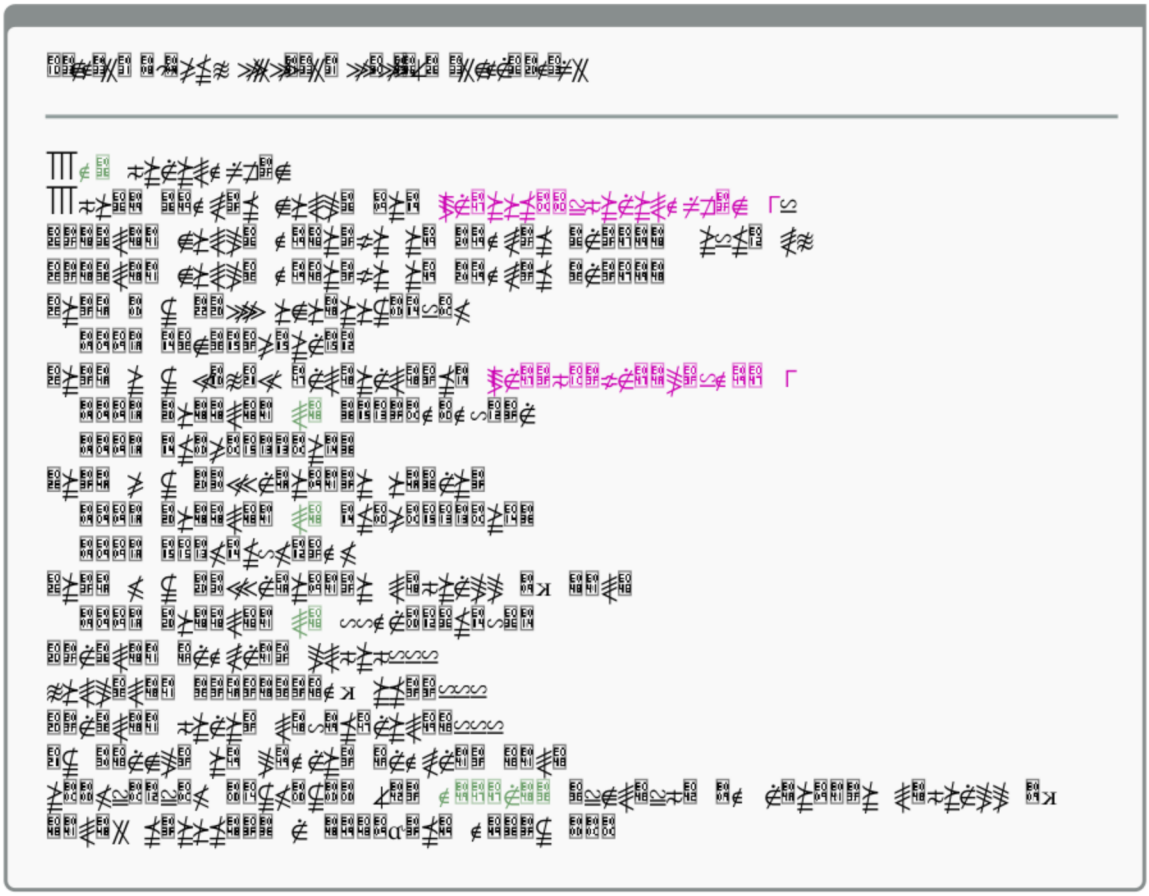
[illegible] If a file named [illegible] exists in the root of the build context then it is interpreted as a newline-separated list of exclusion patterns. Much like a [illegible] file it excludes the listed files from being treated as part of the build context, and therefore prevents them from being uploaded to the Docker daemon. Globbing can be done using Go's filepath.

Next, you can see that each instruction in the [illegible] has been executed with the image ID, [illegible], being returned as the final output of the build process. Each step and its associated instruction are run individually, and Docker has committed the result of each operation before outputting that final image ID.

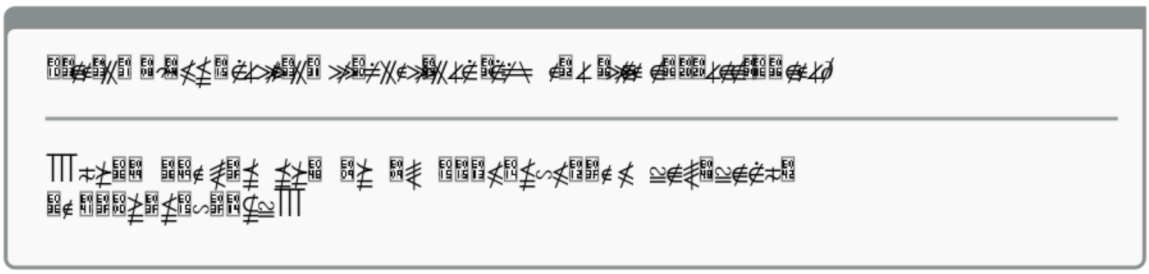
[illegible]

Earlier, we talked about what happens if an instruction fails. Let's look at an example: let's assume that in Step 4 we got the name of the required package wrong and instead called it [illegible].

Let's run the build again and see what happens when it fails.



Let's say I want to debug this failure. I can use the [illegible] command to create a container from the last step that succeeded in my Docker build, in this example using the image ID of [illegible].



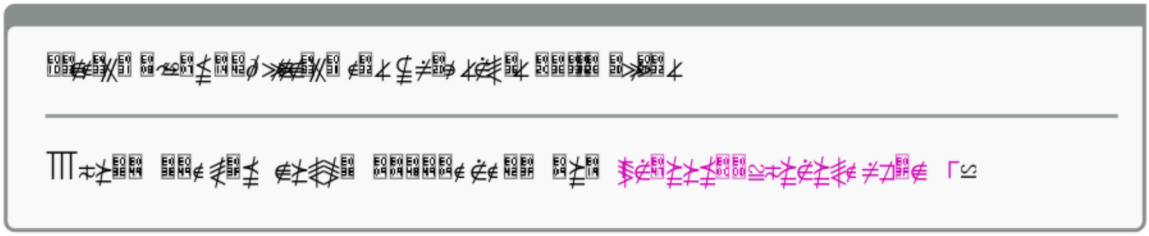
I can then try to run the [illegible] step again with the right package name or conduct some other debugging to determine what went wrong. Once

I've identified the issue, I can exit the container, update my [illegible] with the right package name, and retry my build.

[illegible]

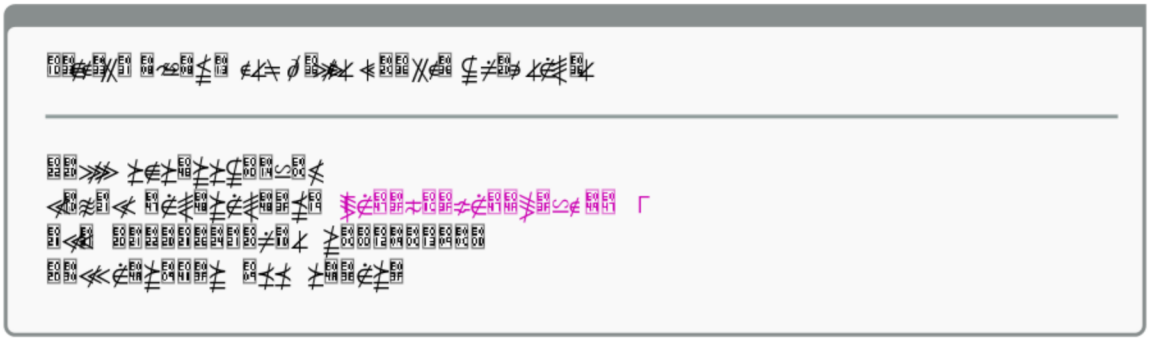
As a result of each step being committed as an image, Docker is able to be really clever about building images. It will treat previous layers as a cache. If, in our debugging example, we did not need to change anything in Steps 1 to 3, then Docker would use the previously built images as a cache and a starting point. Essentially, it'd start the build process straight from Step 4. This can save you a lot of time when building images if a previous step has not changed. If, however, you did change something in Steps 1 to 3, then Docker would restart from the first changed instruction.

Sometimes, though, you want to make sure you don't use the cache. For example, if you'd cached Step 3 above, [illegible], then it wouldn't refresh the APT package cache. You might want it to do this to get a new version of a package. To skip the cache, we can use the [illegible] flag with the [illegible] command..



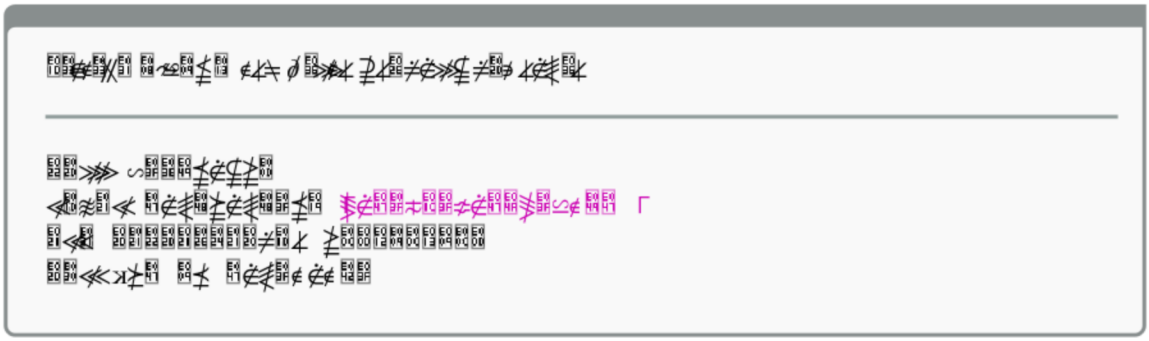
[illegible]

As a result of the build cache, you can build your [illegible]s in the form of simple templates (e.g., adding a package repository or updating packages near the top of the file to ensure the cache is hit). I generally have the same template set of instructions in the top of my [illegible], for example for Ubuntu:



Let's step through this new [illegible]. Firstly, I've used the [illegible] instruction to specify a base image of [illegible]. Next, I've added my [illegible] instruction to provide my contact details. I've then specified a new instruction, [illegible]. The [illegible] instruction sets environment variables in the image. In this case, I've specified the [illegible] instruction to set an environment variable called [illegible], showing when the template was last updated. Lastly, I've specified the [illegible] command in a [illegible] instruction. This refreshes the APT package cache when it's run, ensuring that the latest packages are available to install.

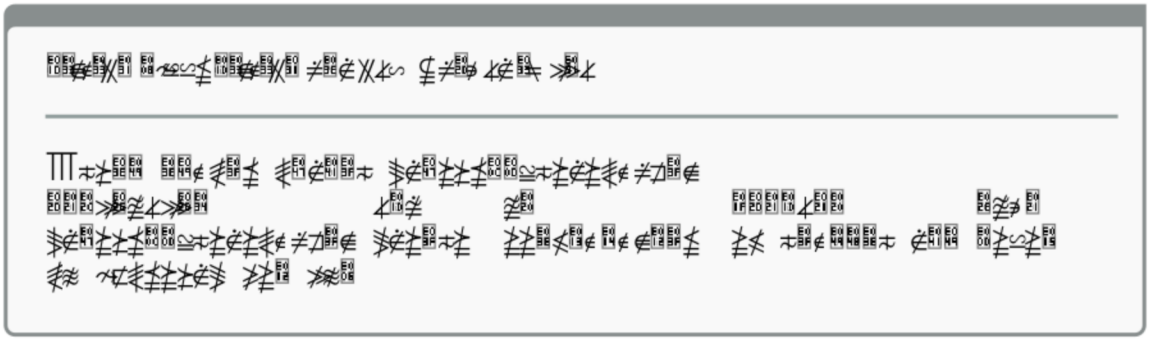
With my template, when I want to refresh the build, I change the date in my [illegible] instruction. Docker then resets the cache when it hits that [illegible] instruction and runs every subsequent instruction anew without relying on the cache. This means my [illegible] instruction is rerun and my package cache is refreshed with the latest content. You can extend this template example for your target platform or to fit a variety of needs. For example, for a [illegible] image we might:



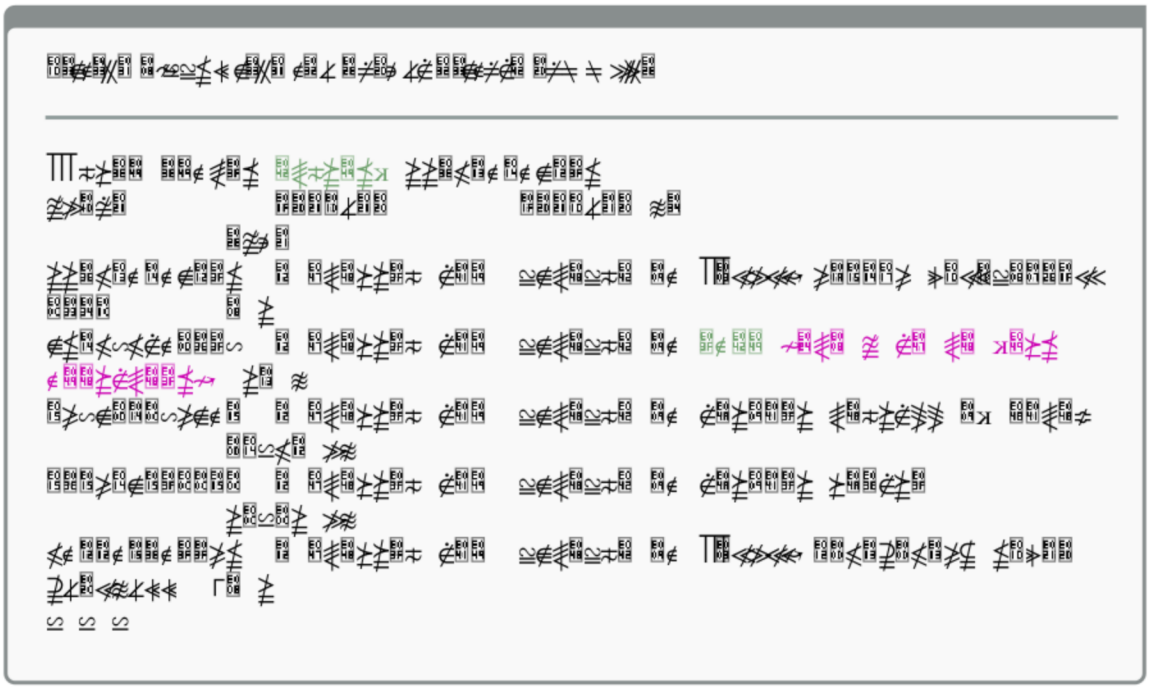
Which performs a similar caching function for Fedora using Yum.

[illegible]

Now let's take a look at our new image. We can do this using the [illegible] command.

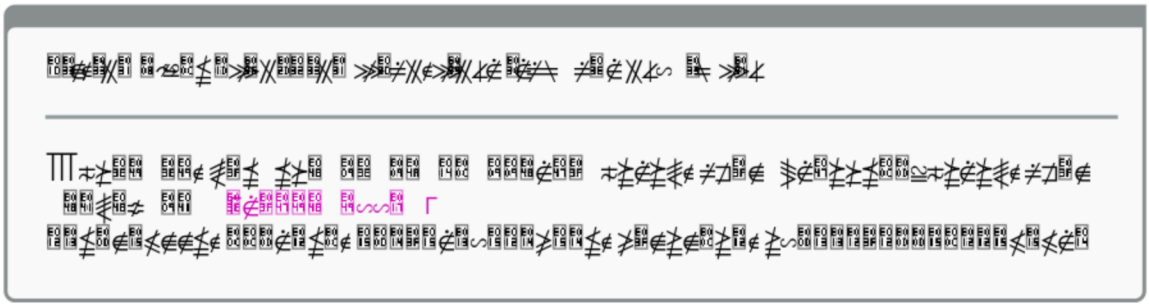


If we want to drill down into how our image was created, we can use the [illegible] command.



We see each of the image layers inside our new [illegible] image and the [illegible] instruction that created them.

Let's launch a new container using our new image and see if what we've built has worked.



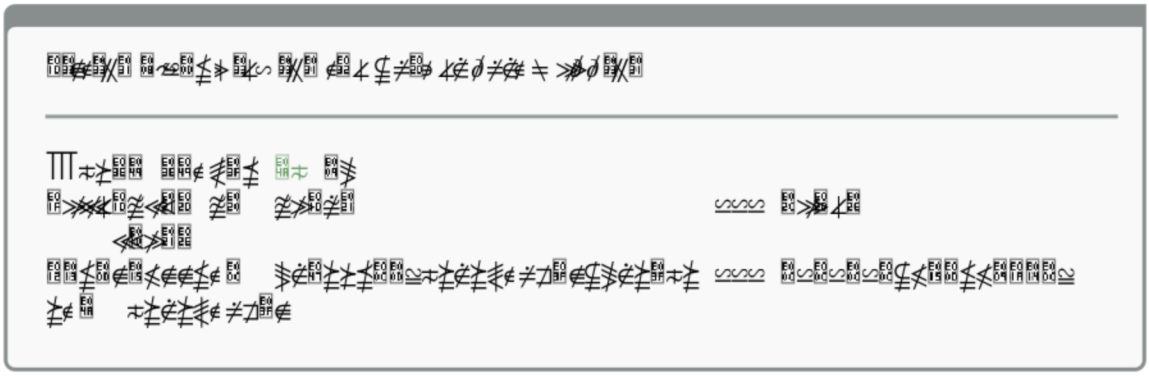
Here I've launched a new container called using the command and the name of the image we've just created. We've specified the option, which tells Docker to run detached in the background. This allows us to run long-running processes like the Nginx daemon. We've also specified a command for the container to run: . This will launch Nginx in the foreground to run our web server.

We've also specified a new flag, . The flag manages which network ports Docker publishes at runtime. When you run a container, Docker has two methods of assigning ports on the Docker host:

- Docker can randomly assign a high port from the range to on the Docker host that maps to port 80 on the container.
- You can specify a specific port on the Docker host that maps to port 80 on the container.

The command will open a random port on the Docker host that will connect to port 80 on the Docker container.

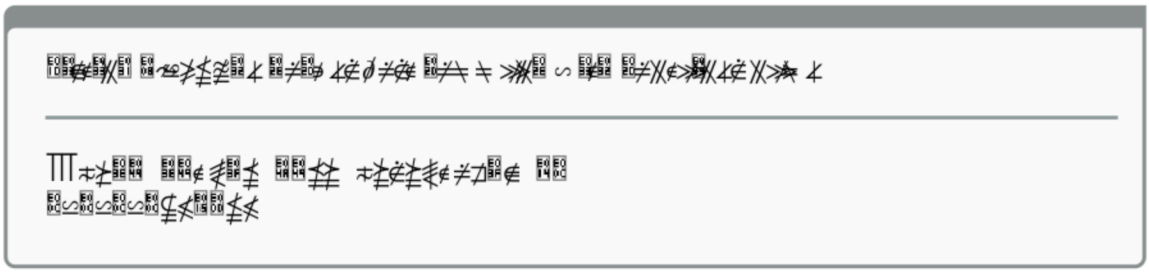
Let's look at what port has been assigned using the command. The flag tells Docker to show us the last container launched.



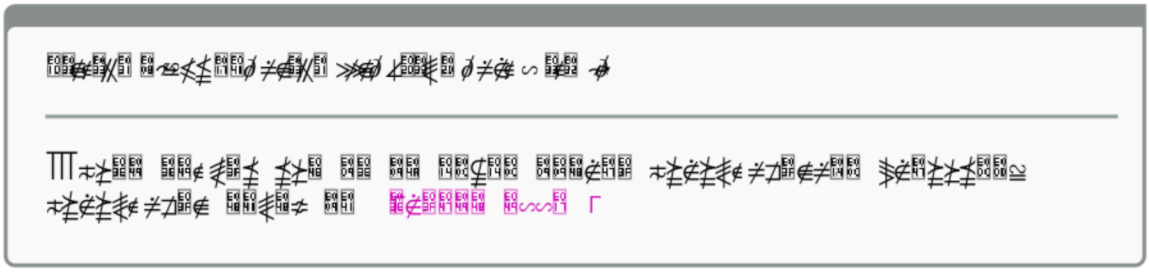
We see that port 49154 is mapped to the container port of 80. We can get the same information with the [illegible] command.

We've specified the container ID and the container port for which we'd like to see the mapping, [illegible], and it has returned the mapped port, [illegible].

Or we could use the container name too.

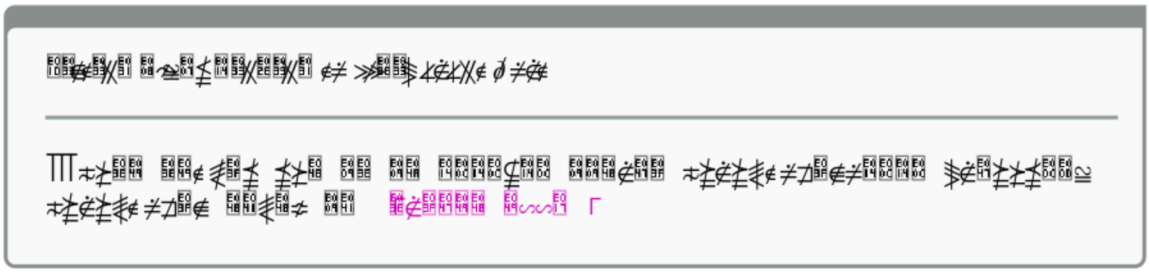


The [illegible] option also allows us to be flexible about how a port is published to the host. For example, we can specify that Docker bind the port to a specific port:



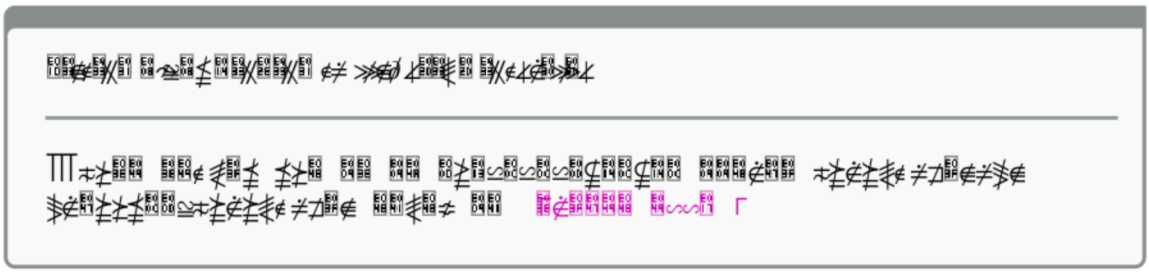
This will bind port 80 on the container to port 80 on the local host. It's important to be wary of this direct binding: if you're running multiple containers, only one container can bind a specific port on the local host. This can limit Docker's flexibility.

To avoid this, we could bind to a different port.

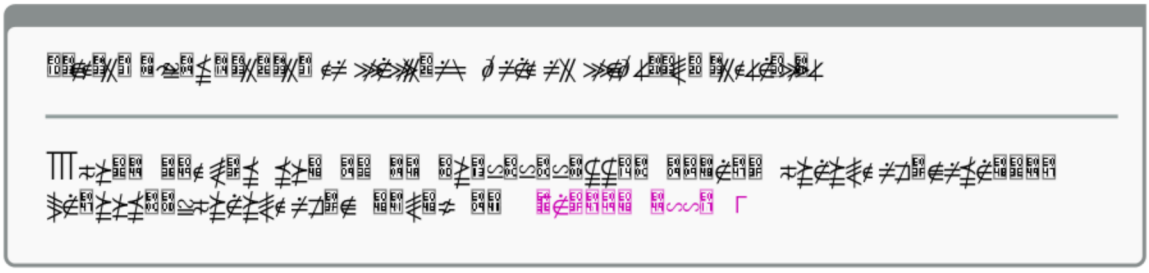


This would bind port 80 on the container to port 8080 on the local host.

We can also bind to a specific interface.



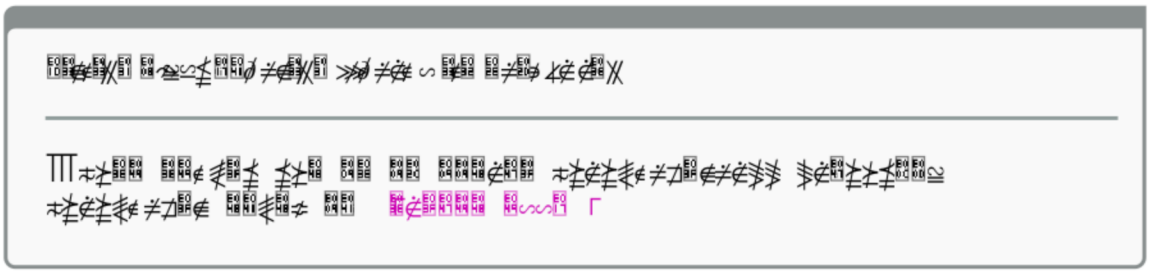
Here we've bound port 80 of the container to port 80 on the interface on the local host. We can also bind to a random port using the same structure.



Here we've removed the specific port to bind to on . We would now use the or command to see which random port was assigned to port 80 on the container.

You can bind UDP ports by adding the suffix to the port binding.

Docker also has a shortcut, , that allows us to publish all ports we've exposed via instructions in our .



This would publish port 80 on a random port on our local host. It would also publish any additional ports we had specified with other instructions in the that built our image.

You can find more information on port redirection here.

With this port number, we can now view the web server on the running container using the IP address of our host or the [illegible] on [illegible].

[illegible] You can find the IP address of your local host with the [illegible] or [illegible] command.

[illegible]

[illegible]

Now we've got a simple Docker-based web server.

[illegible]

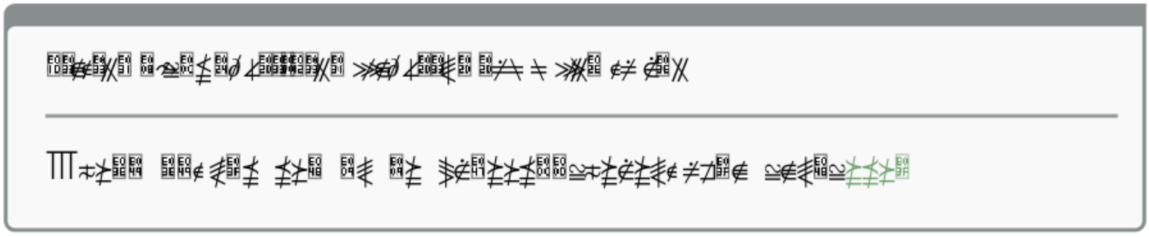
We've already seen some of the available [illegible] instructions, like [illegible] and [illegible]. But there are also a variety of other instructions we can put in our [illegible]. These include [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible], [illegible] and [illegible]. You can see a full list of the available [illegible] instructions here.

We'll also see a lot more [illegible]s in the next few chapters and see how to build some cool applications into Docker containers.

[illegible]

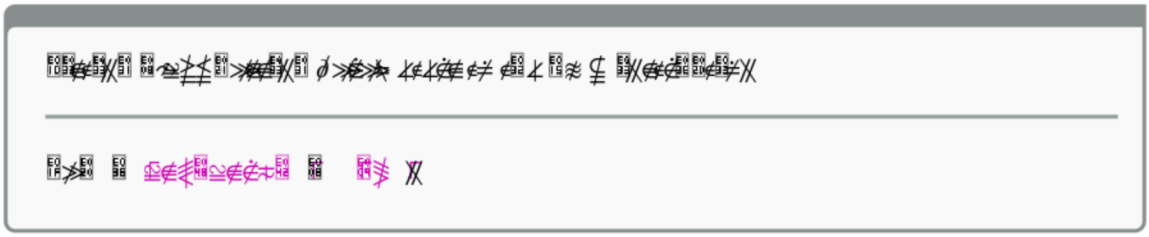
The [illegible] instruction specifies the command to run when a container is launched. It is similar to the [illegible] instruction, but rather than running the command when the

container is being built, it will specify the command to run when the container is launched, much like specifying a command to run when launching a container with the [illegible] command, for example:



This would be articulated in the [illegible] as:

You can also specify parameters to the command, like so:



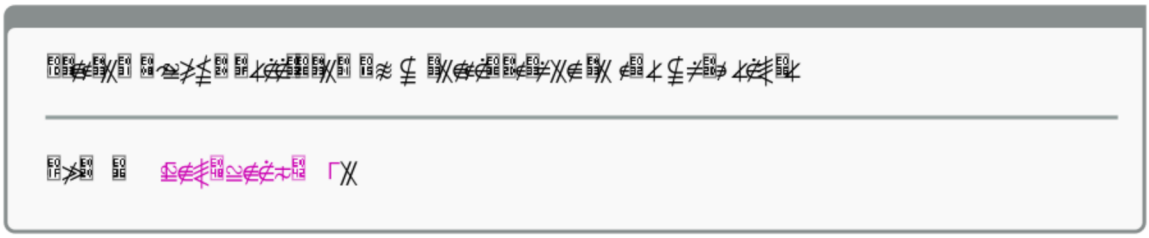
Here we're passing the [illegible] flag to the [illegible] command.

[illegible] You'll note that the command is contained in an array. This tells Docker to run the command 'as-is'. You can also specify the [illegible] instruction without an array, in which case Docker will prepend [illegible] to the command. This may result in unexpected behavior when the command is executed. As a result, it is recommended that you always use the array syntax.

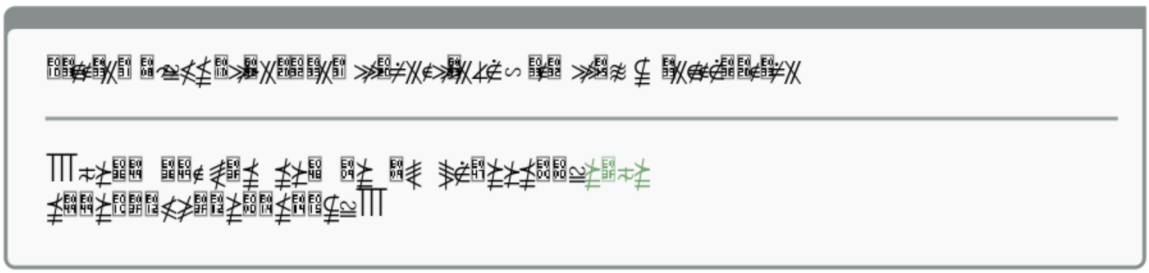
Lastly, it's important to understand that we can override the [illegible] instruction using the [illegible] command. If we specify a [illegible] in our [illegible] and one on the [illegible] command line, then the command line will override the [illegible]'s [illegible] instruction.

[illegible] It's also important to understand the interaction between the [illegible] instruction and the [illegible] instruction. We'll see some more details of this below.

Let's look at this process a little more closely. Let's say our [illegible] contains the [illegible]:



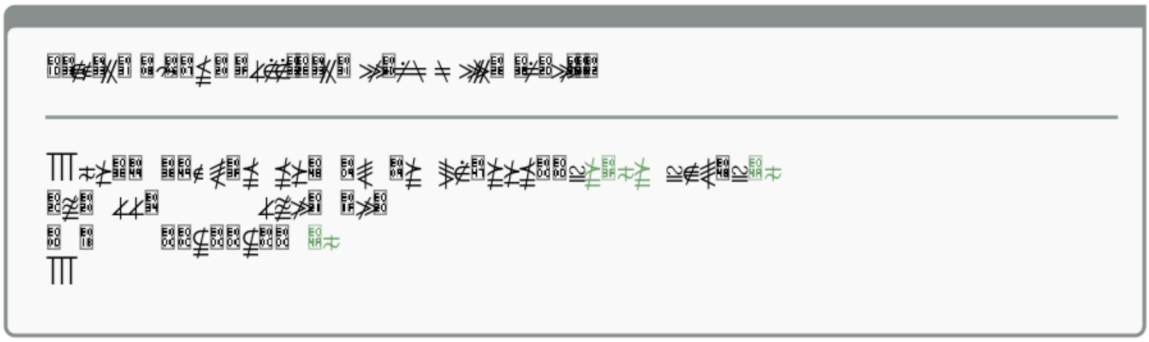
We can build a new image (let's call it [illegible]) using the [illegible] command and then launch a new container from this image.



Notice something different? We didn't specify the command to be executed at the

end of the [illegible]. Instead, Docker used the command specified by the [illegible] instruction.

If, however, I did specify a command, what would happen?



You can see here that we have specified the [illegible] command to list running processes. Instead of launching a shell, the container merely returned the list of running processes and stopped, overriding the command specified in the [illegible] instruction.

[illegible] You can only specify one [illegible] instruction in a [illegible]. If more than one is specified, then the last [illegible] instruction will be used. If you need to run multiple processes or commands as part of starting a container you should use a service management tool like Supervisor.

[illegible]

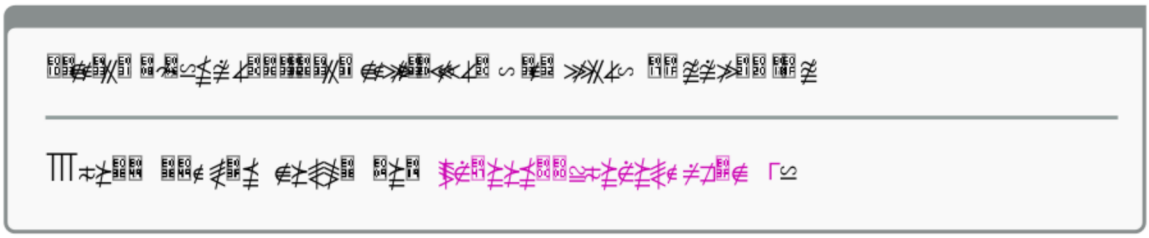
Closely related to the [illegible] instruction, and often confused with it, is the [illegible] instruction. So what's the difference between the two, and why are they both needed? As we've just discovered, we can override the [illegible] instruction on the [illegible] command line. Sometimes this isn't great when we want a container to behave in a certain way. The [illegible] instruction provides a command

that isn't as easily overridden. Instead, any arguments we specify on the [illegible] command line will be passed as arguments to the command specified in the [illegible]. Let's see an example of an [illegible] instruction.

Like the [illegible] instruction, we also specify parameters by adding to the array. For example:

[illegible] As with the [illegible] instruction above, you can see that we've specified the [illegible] command in an array to avoid any issues with the command being prepended with [illegible].

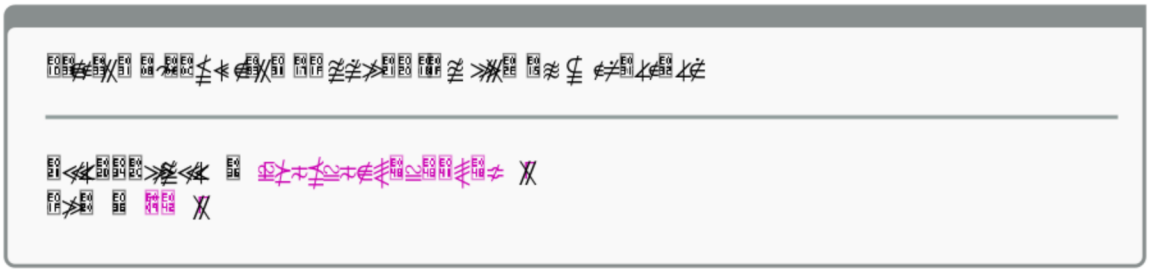
Now let's rebuild our image with an [illegible] of [illegible].



And then launch a new container from our [illegible] image.

We've rebuilt our image and then launched an interactive container. We specified the argument [illegible] This argument will be passed to the command specified in the [illegible] instruction, which will thus become [illegible] This command would then launch the Nginx daemon in the foreground and leave the container running as a web server.

We can also combine [illegible] and [illegible] to do some neat things. For example, we might want to specify the following in our [illegible].



Now when we launch a container, any option we specify will be passed to the Nginx daemon; for example, we could specify [illegible] as we did above to run the daemon in the foreground. If we don't specify anything to pass to the container, then the [illegible] is passed by the [illegible] instruction and returns the Nginx help text: [illegible].

This allows us to build in a default command to execute when our container is run combined with overridable options and flags on the [illegible] command line.

[illegible] If required at runtime, you can override the [illegible] instruction using

the [illegible] command with [illegible] flag.

[illegible]

The [illegible] instruction provides a way to set the working directory for the container and the [illegible] and/or [illegible] to be executed when a container is launched from the image.

We can use it to set the working directory for a series of instructions or for the final container. For example, to set the working directory for a specific instruction we might:

```
[illegible]

[illegible]
[illegible]
[illegible]
[illegible]
```

Here we've changed into the [illegible] directory to run [illegible] and then changed into the [illegible] directory prior to specifying our [illegible] instruction of [illegible].

You can override the working directory at runtime with the [illegible] flag, for example:

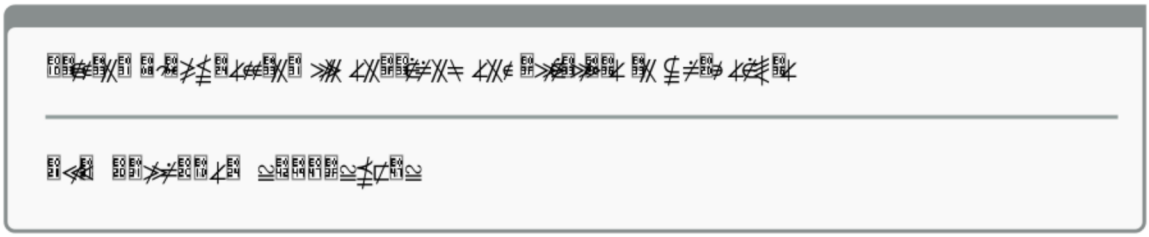
```
[illegible]

[illegible]
```

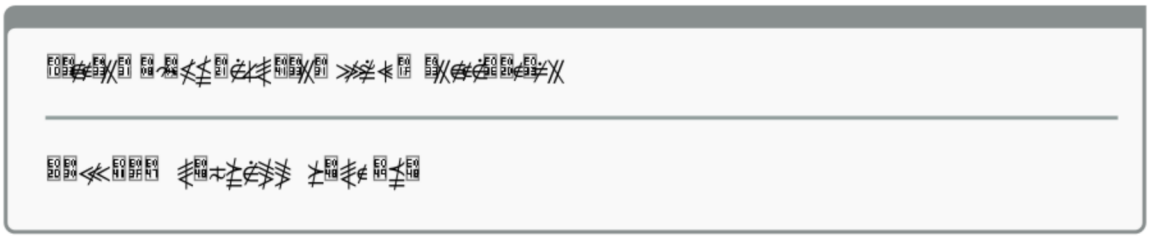
This will set the container's working directory to [illegible].

[illegible]

The [illegible] instruction is used to set environment variables during the image build process. For example:

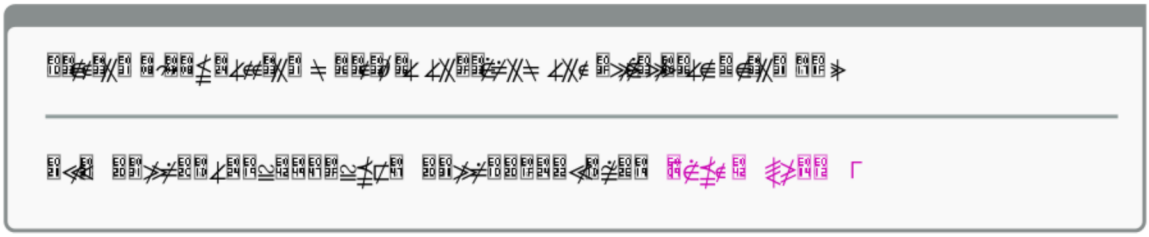


This new environment variable will be used for any subsequent [illegible]instructions, as if we had specified an environment variable prefix to a command like so:

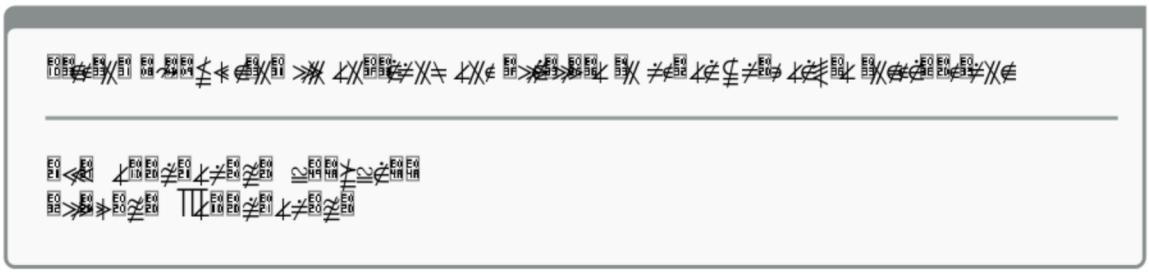


would be executed as:

You can specify single environment variables in an [illegible] instruction or since Docker 1.4 you can specify multiple variables like so:



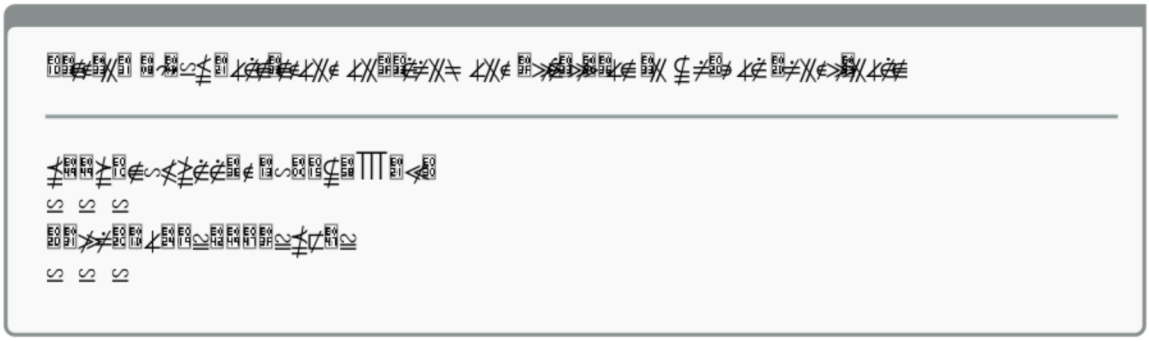
We can also use these environment variables in other instructions.



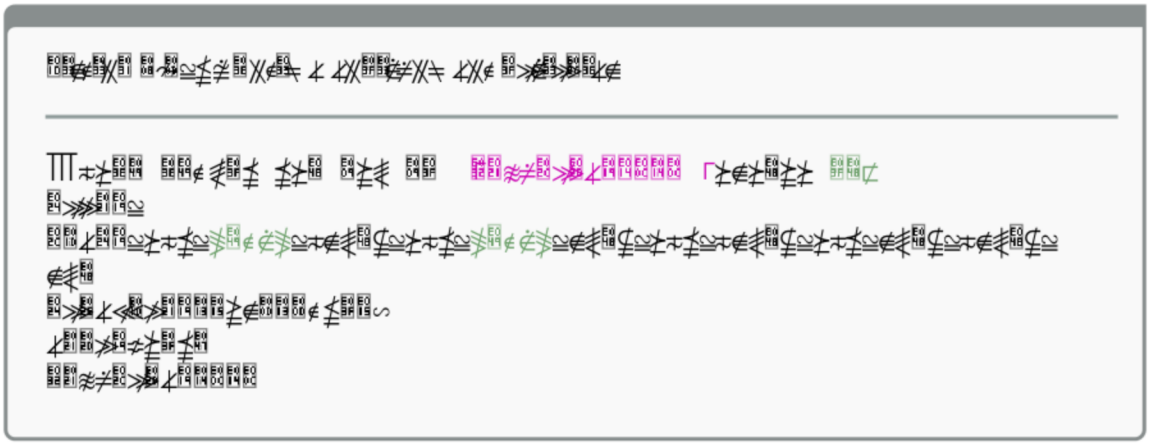
Here we've specified a new environment variable, , and then used its value in a instruction. Our instruction would now be set to .

TIP You can also escape environment variables when needed by prefixing them with a backslash.

These environment variables will also be persisted into any containers created from your image. So, if we were to run the command in a container built with the instruction we'd see:



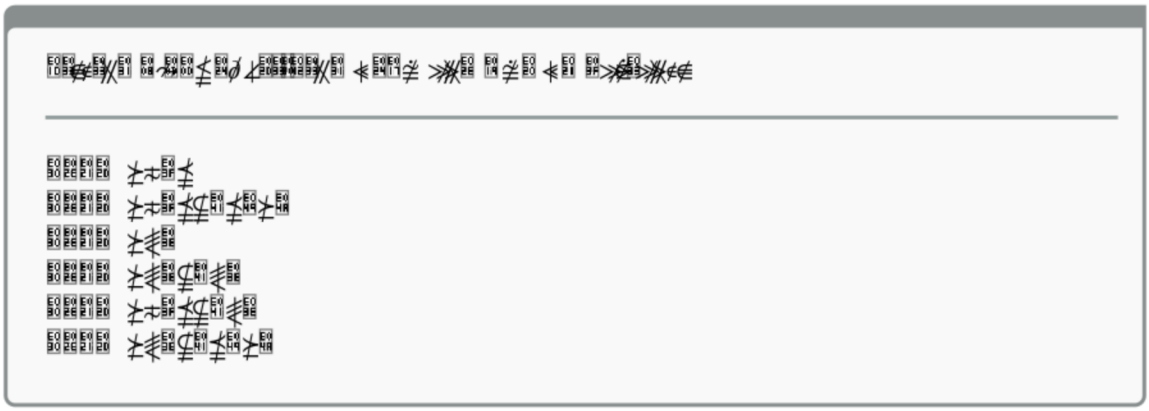
You can also pass environment variables on the command line using the flag. These variables will only apply at runtime, for example:



Now our container has the environment variable set to .

The instruction specifies a user that the image should be run as; for example:

This will cause containers created from the image to be run by the [illegible] user. We can specify a username or a UID and group or GID. Or even a combination thereof, for example:



You can also override this at runtime by specifying the [illegible] flag with the [illegible] command.

The default user if you don't specify the [illegible] instruction is [illegible].

[illegible]

The [illegible] instruction adds volumes to any container created from the image. A volume is a specially designated directory within one or more containers that bypasses the Union File System to provide several useful features for persistent or shared data:

- Volumes can be shared and reused between containers.
- A container doesn't have to be running to share its volumes.
- Changes to a volume are made directly.
- Changes to a volume will not be included when you update an image.

- Volumes persist even if no containers use them.

This allows us to add data (like source code), a database, or other content into an image without committing it to the image and allows us to share that data between containers. This can be used to do testing with containers and an application's code, manage logs, or handle databases inside a container. We'll see examples of this in Chapters 5 and 6.

You can use the [illegible] instruction like so:

[illegible]

[illegible]

This would attempt to create a mount point [illegible] to any container created from the image.

[illegible] Also useful and related is the [illegible] command. This allows you to copy files to and from your containers. You can read about it in the Docker command line documentation.

Or we can specify multiple volumes by specifying an array:

[illegible]

[illegible]

We'll see a lot more about volumes and how to use them in Chapters 5 and 6. If you're curious you can read more about volumes in the Docker volumes documentation.

[illegible]

The [illegible] instruction adds files and directories from our build environment into our image; for example, when installing an application. The [illegible] instruction specifies a source and a destination for the files, like so:

```
[illegible] [illegible] [illegible] [illegible]

[illegible] [illegible] [illegible]
```

This [illegible] instruction will copy the file [illegible] from the build directory to [illegible] in the image. The source of the file can be a URL, filename, or directory as long as it is inside the build context or environment. You cannot [illegible] files from outside the build directory or context.

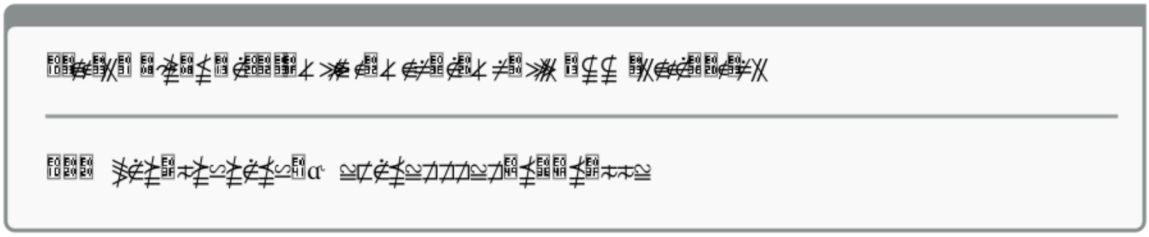
When [illegible]'ing files Docker uses the ending character of the destination to determine what the source is. If the destination ends in a [illegible], then it considers the source a directory. If it doesn't end in a [illegible], it considers the source a file.

The source of the file can also be a URL; for example:

```
[illegible] [illegible] [illegible] [illegible]

[illegible] [illegible] [illegible]
```

Lastly, the [illegible] instruction has some special magic for taking care of local [illegible] archives. If a [illegible] archive (valid archive types include gzip, bzip2, xz) is specified as the source file, then Docker will automatically unpack it for you:



This will unpack the [illegible] archive into the [illegible] directory. The archive is unpacked with the same behavior as running [illegible] with the [illegible] option: the output is the union of whatever exists in the destination plus the contents of the archive. If a file or directory with the same name already exists in the destination, it will not be overwritten.

[illegible] Currently this will not work with a tar archive specified in a URL. This is somewhat inconsistent behavior and may change in a future release.

Finally, if the destination doesn't exist, Docker will create the full path for us, including any directories. New files and directories will be created with a mode of 0755 and a UID and GID of 0.

[illegible] It's also important to note that the build cache can be invalidated by [illegible] instructions. If the files or directories added by an [illegible] instruction change then this will invalidate the cache for all following instructions in the [illegible].

[illegible]

The [illegible] instruction is closely related to the [illegible] instruction. The key difference is that the [illegible] instruction is purely focused on copying local files from the build context and does not have any extraction or decompression capabilities.

[illegible]

[illegible]

This will copy files from the [illegible] directory to the [illegible] directory.

The source of the files must be the path to a file or directory relative to the build context, the local source directory in which your [illegible] resides. You cannot copy anything that is outside of this directory, because the build context is uploaded to the Docker daemon, and the copy takes place there. Anything outside of the build context is not available. The destination should be an absolute path inside the container.

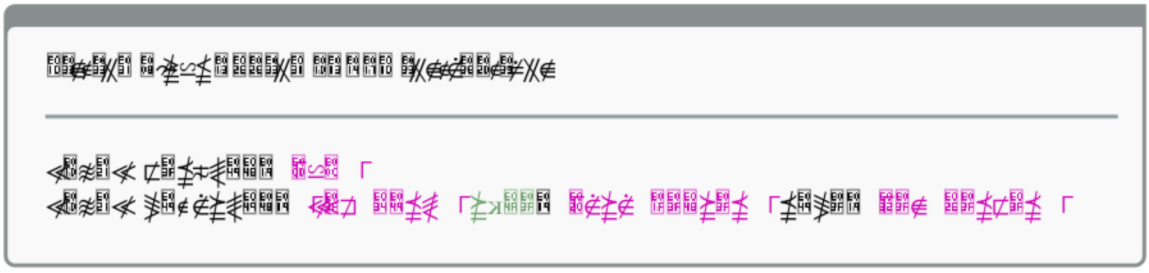
Any files and directories created by the copy will have a UID and GID of 0.

If the source is a directory, the entire directory is copied, including filesystem metadata; if the source is any other kind of file, it is copied individually along with its metadata. In our example, the destination ends with a trailing slash [illegible], so it will be considered a directory and copied to the destination directory.

If the destination doesn't exist, it is created along with all missing directories in its path, much like how the [illegible] command works.

[illegible]

The [illegible]instruction adds metadata to a Docker image. The metadata is in the form of key/value pairs. Let's see an example.



The instruction is written in the form of . You can specify one item of metadata per label or multiple items separated with white space. We recommend combining all your metadata in a single instruction to save creating multiple layers with each piece of metadata. You can inspect the labels on an image using the command..

Here we see the metadata we just defined using the instruction.

The instruction was introduced in Docker 1.6.

[illegible]

The [illegible] instruction sets the system call signal that will be sent to the container when you tell it to stop. This signal can be a valid number from the kernel syscall table, for instance 9, or a signal name in the format [illegible], for instance [illegible].

[illegible] The [illegible] instruction was introduced in Docker 1.9.

[illegible]

The [illegible] instruction defines variables that can be passed at build-time via the [illegible] command. This is done using the [illegible] flag. You can only specify build-time arguments that have been defined in the [illegible].

```
[illegible]

[illegible]
[illegible]
```

The second [illegible] instruction sets a default, if no value is specified for the argument at build-time then the default is used. Let's use one of these arguments in a [illegible] now.

As the image is built the variable will be set to and the variable will inherit the default value of .

Note At this point you're probably thinking - this is a great way to pass secrets like credentials or keys. Don't do this. Your credentials will be exposed during the build process and in the build history of the image.

Docker has a set of predefined variables that you can use at build-time without a corresponding instruction in the .

To use these predefined variables, pass them using the flag to the command.

The instruction was introduced in Docker 1.9 and you can read more about it in the Docker documentation.

The instruction

The instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is and on Windows is .

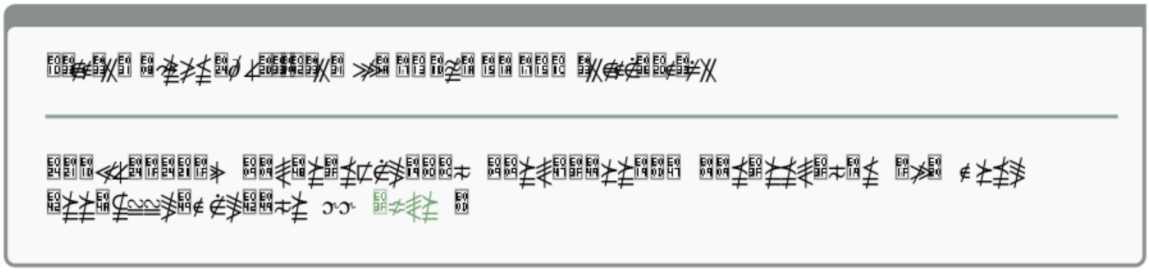
The instruction is useful on platforms such as Windows where there are multiple shells, for example running commands in the or environments. Or when need to run a command on Linux in a specific shell, for example Bash.

The instruction can be used multiple times. Each new instruction overrides all previous instructions, and affects any subsequent instructions.

The instruction

The instruction tells Docker how to test a container to check that it is still working correctly. This allows you to check things like a web site being served or an API endpoint responding with the correct data, allowing you to identify issues that appear, even if an underlying process still appears to be running normally.

When a container has a health check specified, it has a health status in addition to its normal status. You can specify a health check like:



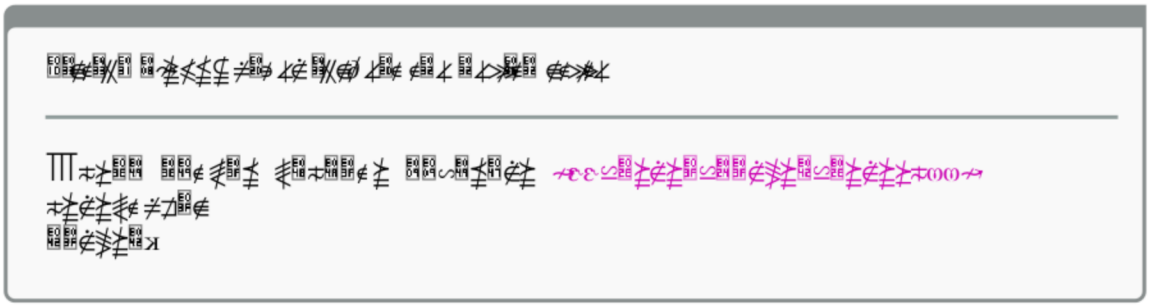
The HEALTHCHECK instruction contains options and then the command you wish to run itself, separated by a CMD keyword.

We've first specified three default options:

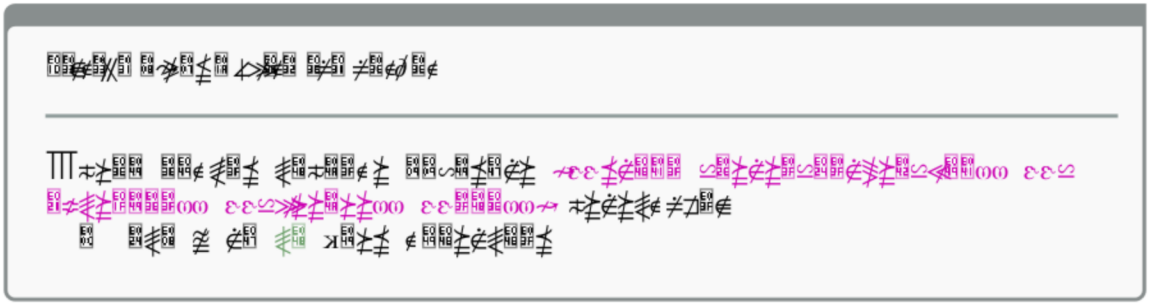
- --interval - defaults to 30 seconds. This is the period between health checks. In this case the first health check will run 10 seconds after container launch and subsequently every 10 seconds.
- --timeout - defaults to 30 seconds. If the health check takes longer the timeout then it is deemed to have failed.
- --retries - defaults to 3. The number of failed checks before the container is marked as unhealthy.

The command after the CMD keyword can be either a shell command or an exec array, for example as we've seen in the ENTRYPOINT instruction. The command should exit with 0 to indicate health or 1 to indicate an unhealthy state. In our CMD we're executing curl on the localhost. If the command fails we're exiting with an exit code of 1, indicating an unhealthy state.

We can see the state of the health check using the docker inspect command.



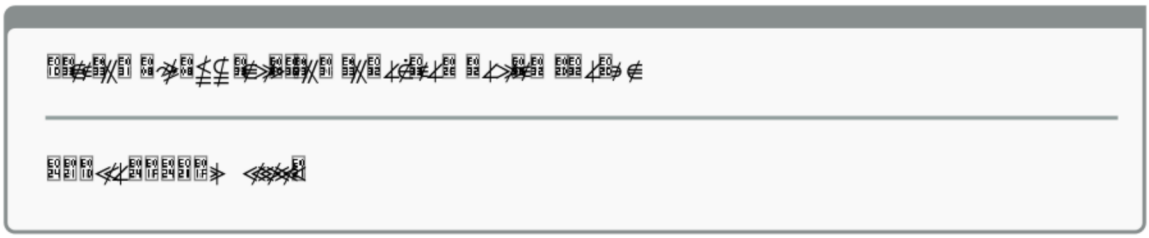
The health check state and related data is stored in the [illegible] namespace and includes current state as well as a history of previous checks and their output. The output from each health check is also available via [illegible].



Here we're iterating through the array of [illegible] entries in the [illegible] output.

There can only be one [illegible] instruction in a [illegible]. If you list more than one then only the last will take effect.

You can also disable any health checks specified in any base images you may have inherited with the instruction:



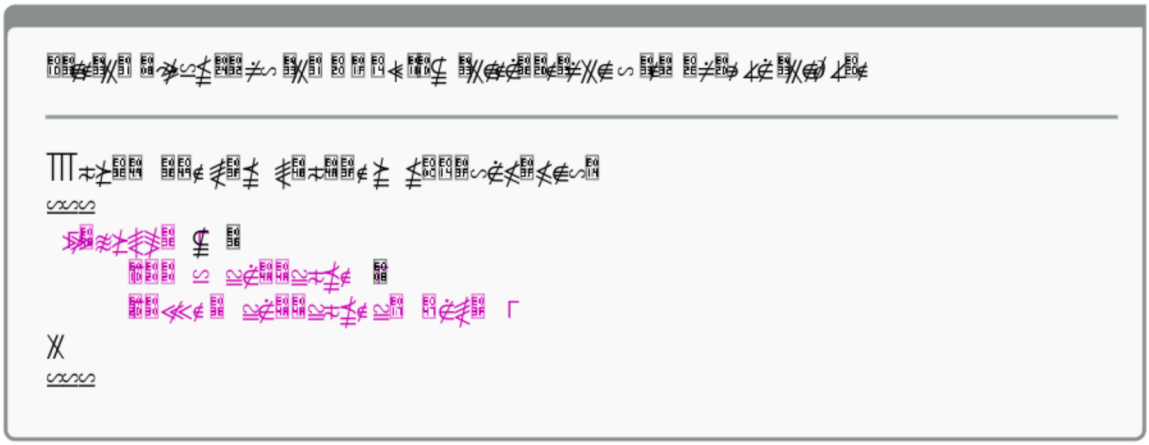
[illegible] This instruction was added in Docker 1.12.

[illegible]

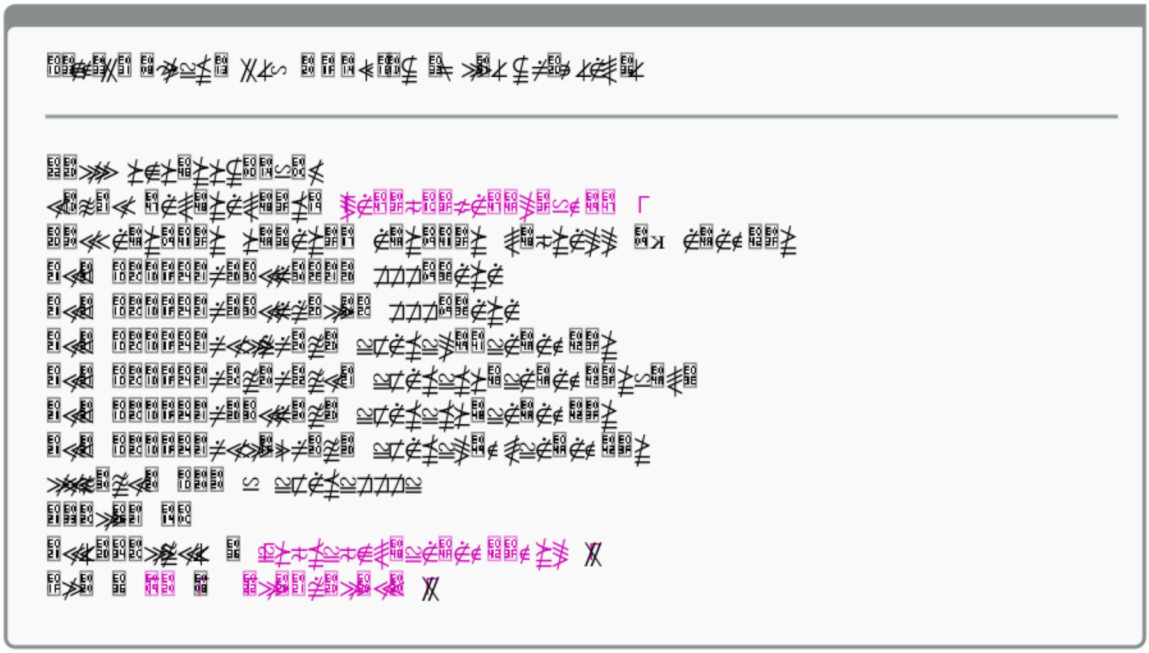
The [illegible] instruction adds triggers to images. A trigger is executed when the image is used as the basis of another image (e.g., if you have an image that needs source code added from a specific location that might not yet be available, or if you need to execute a build script that is specific to the environment in which the image is built).

The trigger inserts a new instruction in the build process, as if it were specified right after the [illegible] instruction. The trigger can be any build instruction. For example:

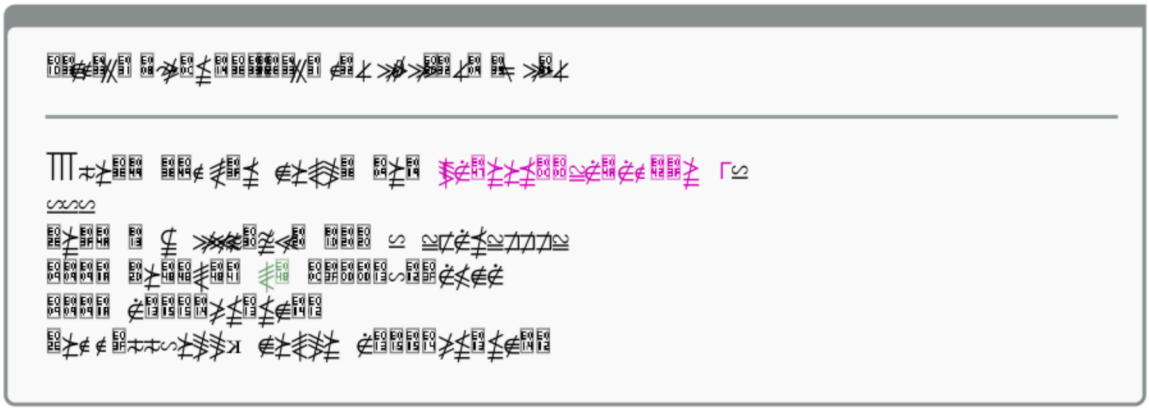
This would add an [illegible] trigger to the image being created, which we see when we run [illegible] on the image.



For example, we'll build a new [illegible] for an Apache2 image that we'll call [illegible].

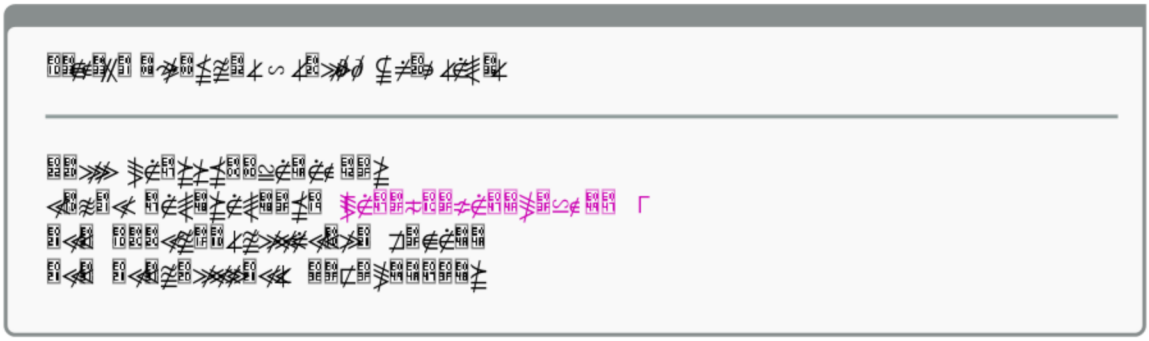


Now we'll build this image.

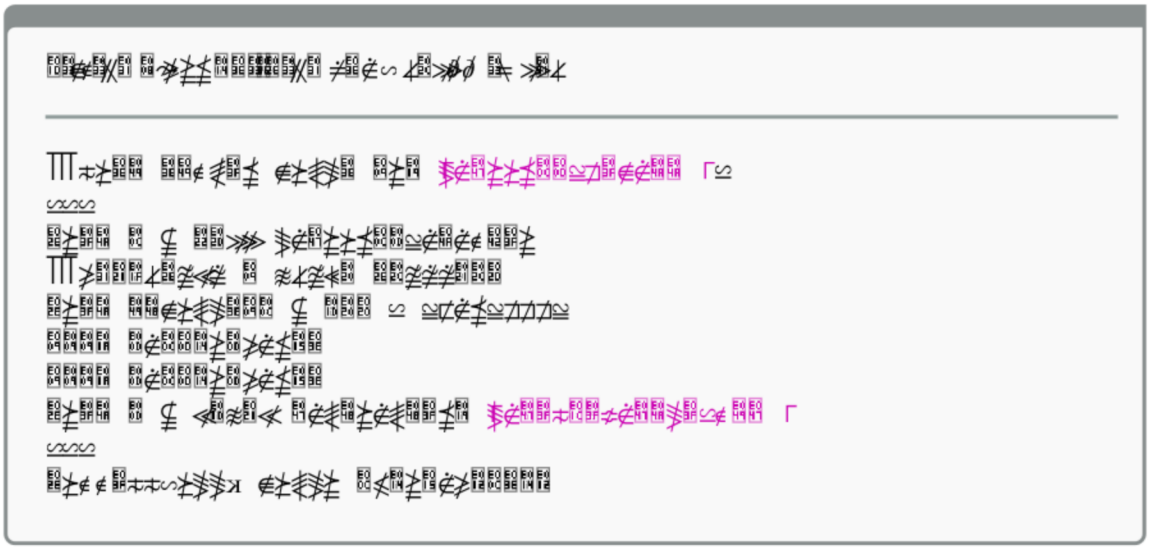


We now have an image with an instruction that uses the instruction to add the contents of the directory we're building from to the directory in our image. This could readily be our generic web application template from which I build web applications.

Let's try this now by building a new image called from the following :



Let's look at what happens when I build this image.



We see that straight after the instruction, Docker has inserted the instruction, specified by the trigger, and then proceeded to execute the remaining steps. This would allow me to always add the local source and, as I've done here, specify some configuration or build information for each application; hence, this becomes a useful template image.

The triggers are executed in the order specified in the parent image and are only inherited once (i.e., by children and not grandchildren). If we built another image from this new image, a grandchild of the image, then the triggers would not be executed when that image is built.

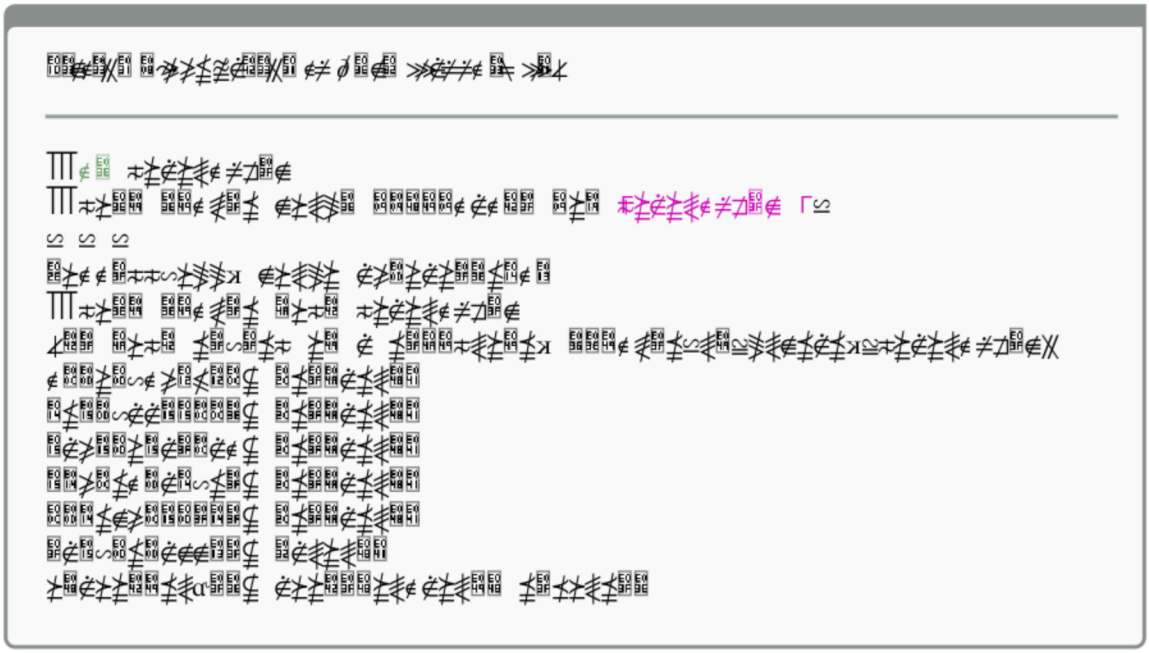
There are several instructions you can't , , and itself. This is done to prevent Inception-like recursion in builds.

Once we've got an image, we can upload it to the Docker Hub. This allows us to make it available for others to use. For example, we could share it with others in our organization or make it publicly available.

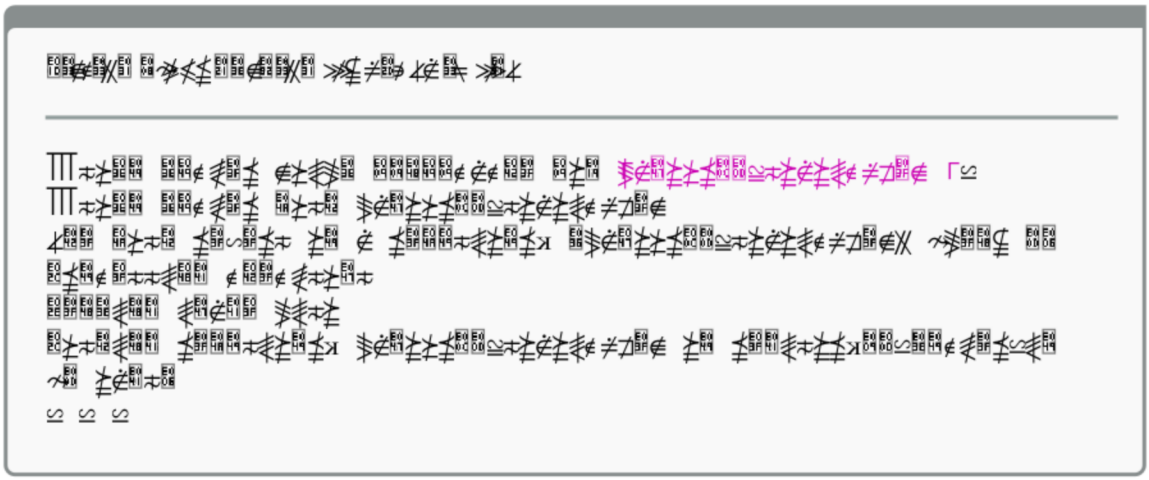
The Docker Hub also has the option of private repositories. These are a paid-for feature that allows you to store an image in a private repository that is only available to you or anyone with whom you share it. This allows you to have private images containing proprietary information or code you might not want to share publicly.

We push images to the Docker Hub using the command.

Let's build an image without a user prefix and try and push it now.



What's gone wrong here? We've tried to push our image to the repository , but Docker knows this is a root repository. Root repositories are managed only by the Docker, Inc., team and will reject our attempt to write to them as unauthorized. Let's try again, rebuilding our image with a user prefix and then pushing it.



This time, our push has worked, and we've written to a user repository,

. We would write to your own user ID, which we created earlier, and to an appropriately named image (e.g.,).

We can now see our uploaded image on the Docker Hub.

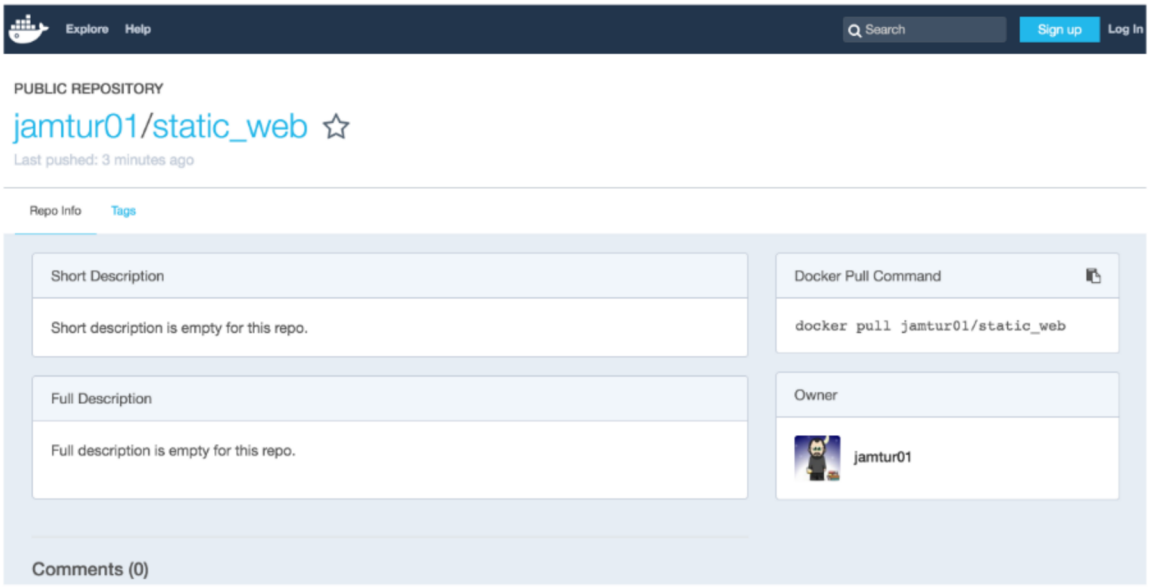


Figure 1.4: Your image on the Docker Hub.

You can find documentation and more information on the features of the Docker Hub here.

In addition to being able to build and push our images from the command line, the Docker Hub also allows us to define Automated Builds. We can do so by connecting a GitHub or BitBucket repository containing a to the Docker

Hub. When we push to this repository, an image build will be triggered and a new image created. This was previously also known as a Trusted Build.

TIP Automated Builds also work for private GitHub and BitBucket repositories.

The first step in adding an Automated Build to the Docker Hub is to connect your GitHub account or BitBucket to your Docker Hub account. To do this, navigate to Docker Hub, sign in, click on your profile link, then click the Add Repository button.

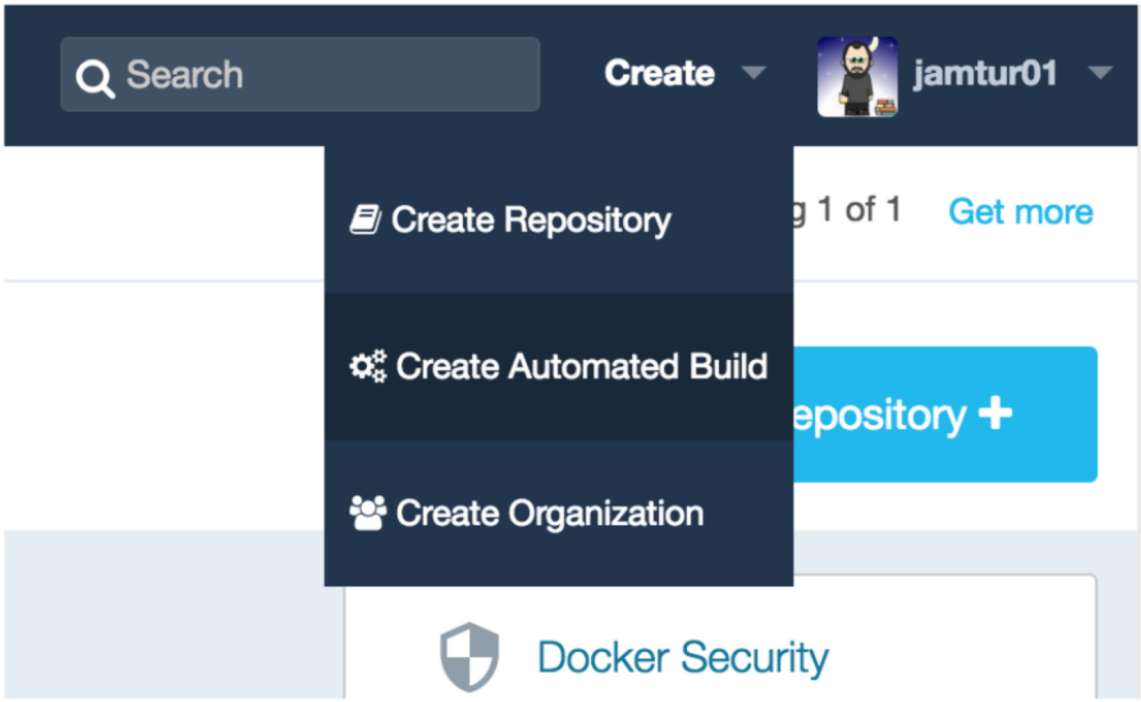


Figure 1.5: The Add Repository button.

You will see a page that shows your options for linking to either GitHub or Bit-

Bucket. Click the [illegible] button under the GitHub logo to initiate the account linkage. You will be taken to GitHub and asked to authorize access for Docker Hub.

On Github you have two options: [illegible] and [illegible]. Select [illegible], and click [illegible] to complete the authorization. You may be prompted to input your GitHub password to confirm the access.

From here, you will be prompted to select the organization and repository from which you want to construct an Automated Build.

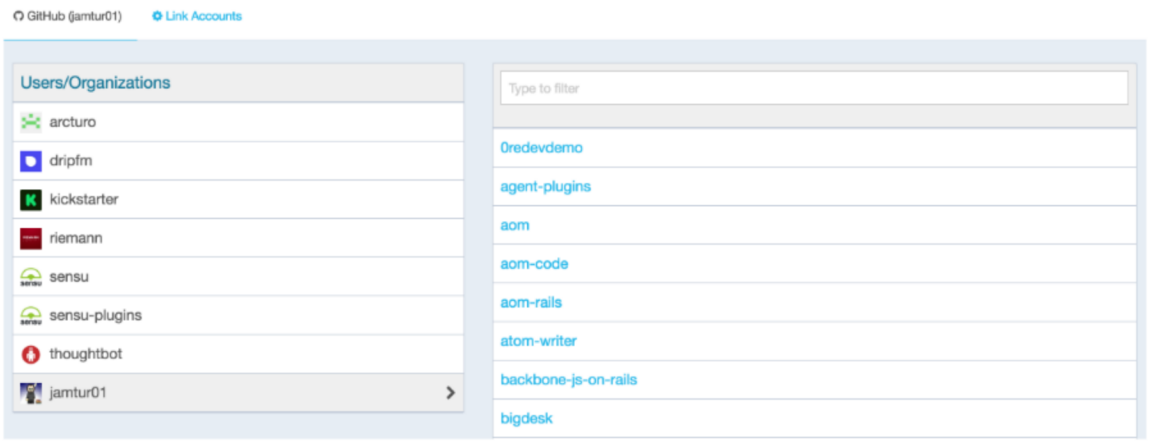


Figure 1.6: Selecting your repository.

Select the repository from which you wish to create an Automated Build and then configure the build.

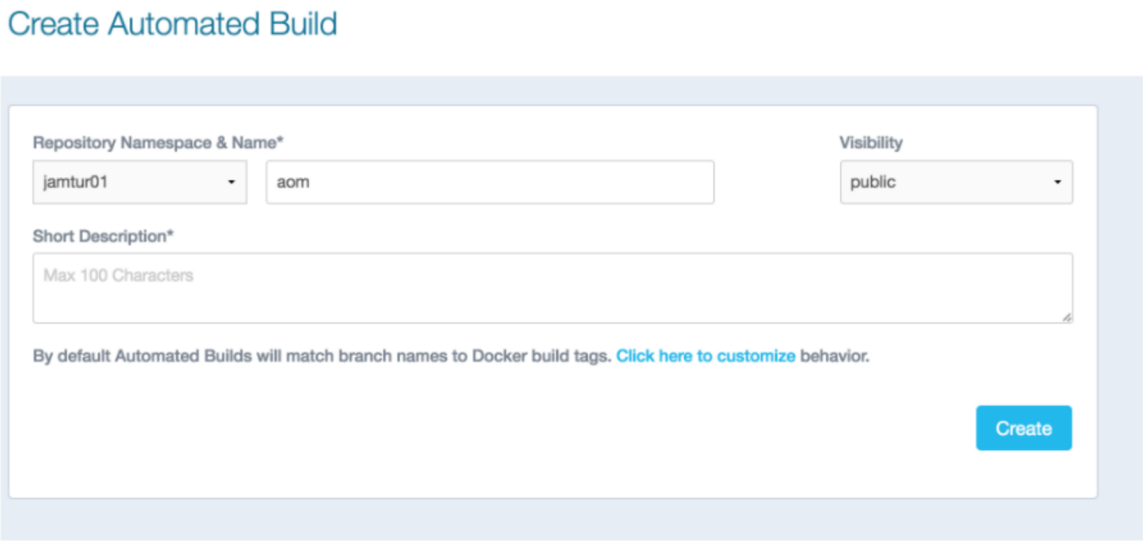


Figure 1.7: Configuring your Automated Build.

Specify the default branch you wish to use, and confirm the repository name.

Specify a tag you wish to apply to any resulting build, then specify the location of the [illegible]. The default is assumed to be the root of the repository, but you can override this with any path.

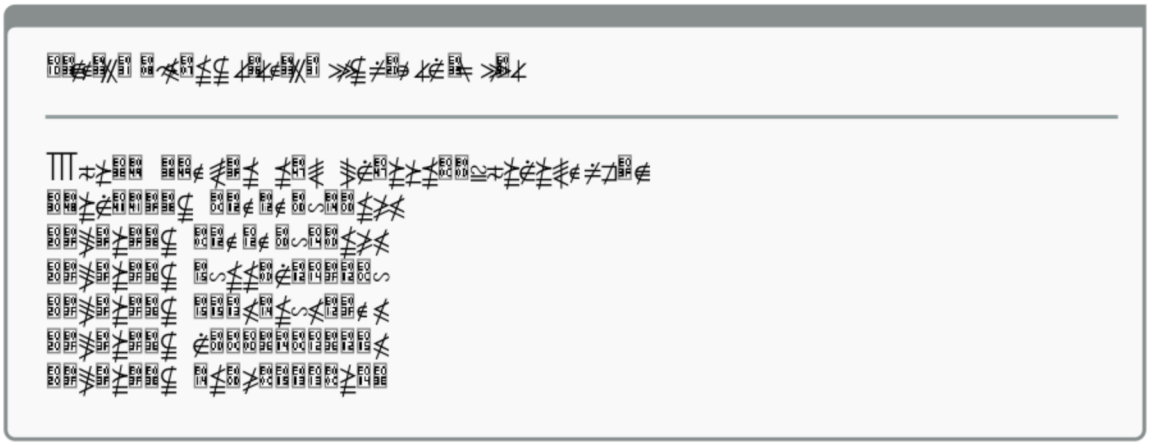
Finally, click the [illegible] button to add your Automated Build to the Docker Hub.

You will now see your Automated Build submitted. Click on the [illegible] link to see the status of the last build, including log output showing the build process and any errors. A build status of [illegible] indicates the Automated Build is up to date. An [illegible] status indicates a problem; you can click through to see the log output.

[illegible] You can't push to an Automated Build using the [illegible] command. You can only update it by pushing updates to your GitHub or BitBucket repository.

[illegible]

We can also delete images when we don't need them anymore. To do this, we'll use the [illegible] command.



Here we've deleted the [illegible] image. You can see Docker's layer filesystem at work here: each of the [illegible] lines represents an image layer being deleted. If a running container is still using an image then you won't be able to delete it. You'll need to stop all containers running that image, remove them and then delete the image.

[illegible] This only deletes the image locally. If you've previously pushed that image to the Docker Hub, it'll still exist there.

If you want to delete an image's repository on the Docker Hub, you'll need to sign in and delete it there using the [illegible] button.

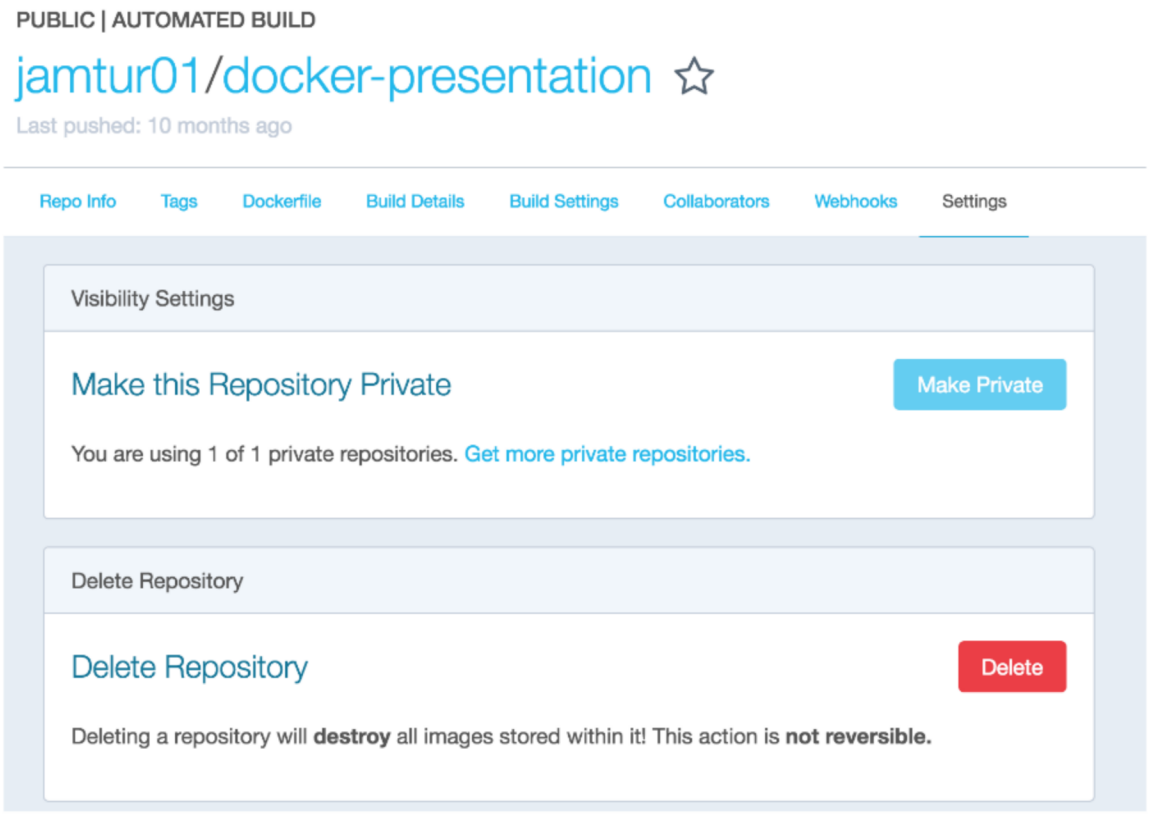
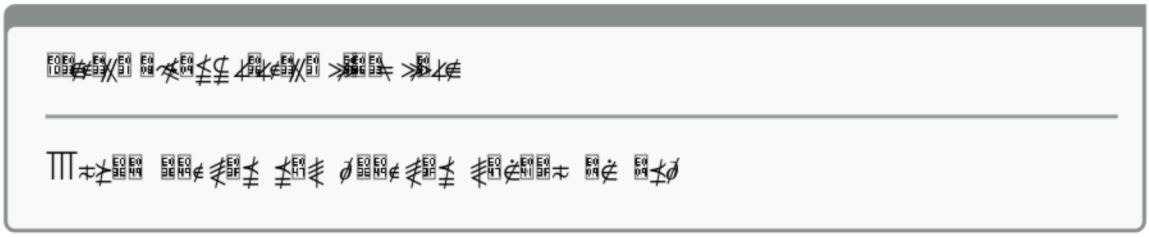


Figure 1.8: Deleting a repository.

We can also delete more than one image by specifying a list on the command line.

or, like the [illegible] command cheat we saw in Chapter 3, we can do the same with the [illegible] command:

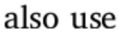
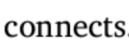


Running your own Docker registry

Having a public registry of Docker images is highly useful. Sometimes, however, we are going to want to build and store images that contain information or data that we don't want to make public. There are two choices in this situation:

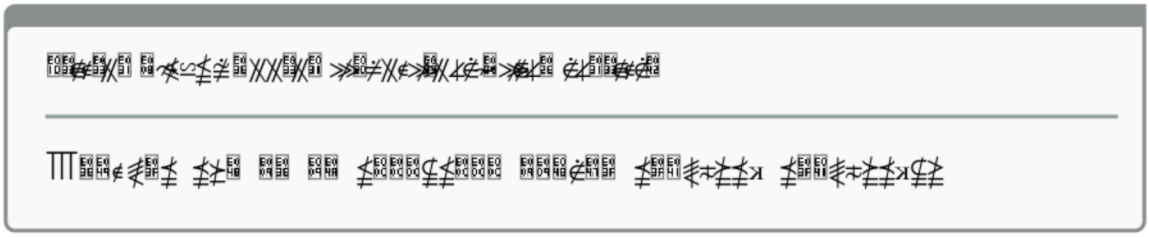
- Make use of private repositories on the Docker Hub.
- Run your own registry behind the firewall.

The team at Docker, Inc., have open-sourced the code they use to run a Docker registry, thus allowing us to build our own internal registry. The registry does not currently have a user interface and is only made available as an API service.

 If you're running Docker behind a proxy or corporate firewall you can also use the  ,  , options to control how Docker connects.

Installing a registry from a Docker container

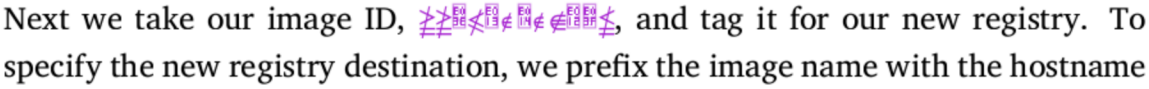
Installing a registry from a Docker container is simple. Just run the Docker-provided container like so:



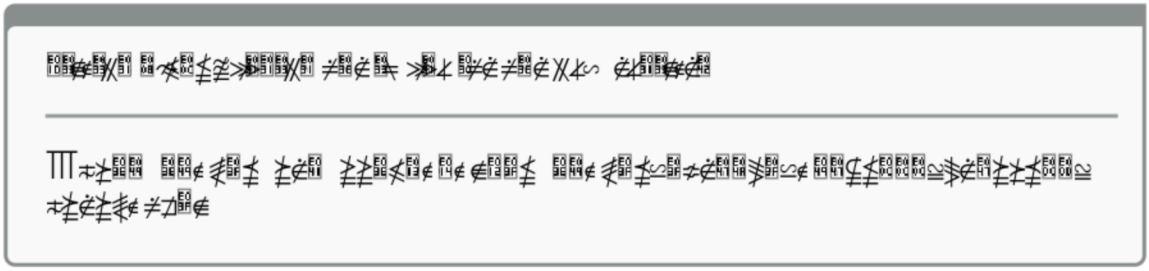
This will launch a container running version 2.0 of the registry application and bind port 5000 to the local host.

If you're running an older version of the Docker Registry, prior to 2.0, you can use the Migrator tool to upgrade to a new registry.

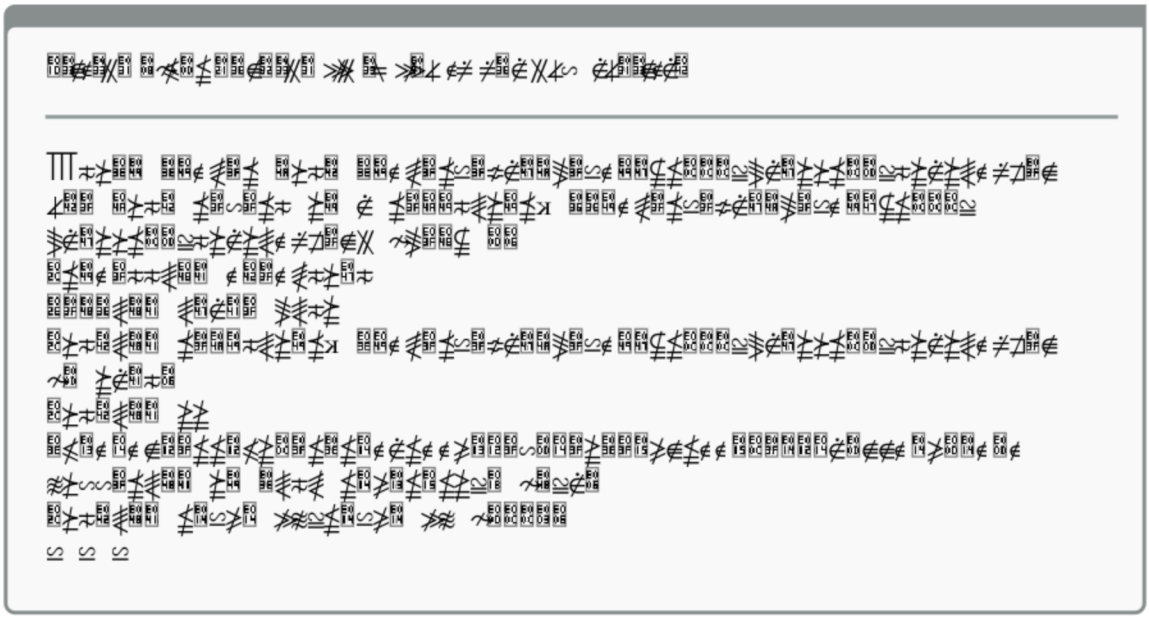
So how can we make use of our new registry? Let's see if we can upload one of our existing images, the  image, to our new registry. First, let's identify the image's ID using the  command.

Next we take our image ID,  , and tag it for our new registry. To specify the new registry destination, we prefix the image name with the hostname and port of our new registry. In our case, our new registry has a hostname of

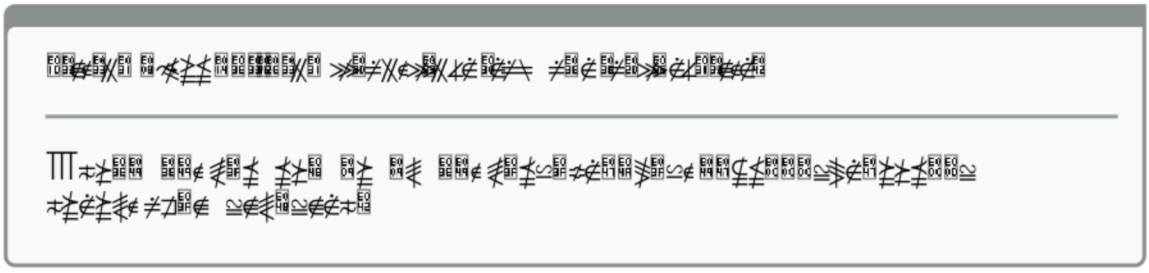
.



After tagging our image, we can then push it to the new registry using the command:



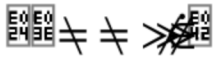
The image is then posted in the local registry and available for us to build new containers using the command.



This is the simplest deployment of the Docker registry behind your firewall. It doesn't explain how to configure the registry or manage it. To find out details like configuring authentication, how to manage the backend storage for your images and how to manage your registry see the full configuration and deployments details in the Docker Registry deployment documentation.

There are a variety of other services and companies out there starting to provide custom Docker registry services.

The Quay service provides a private hosted registry that allows you to upload both public and private containers. Unlimited public repositories are currently free. Private repositories are available in a series of scaled plans. The Quay product has recently been acquired by CoreOS and will be integrated into that product.



In this chapter, we've seen how to use and interact with Docker images and the basics of modifying, updating, and uploading images to the Docker Hub. We've

also learned about using a [illegible] to construct our own custom images. Finally, we've discovered how to run our own local Docker registry and some hosted alternatives. This gives us the basis for starting to build services with Docker.

We'll use this knowledge in the next chapter to see how we can integrate Docker into a testing workflow and into a Continuous Integration lifecycle.

© Copyright 2016 - James Turnbull < >

ISBN 978-0-9888202-0-3