The workflow illustrated in figure 9.7 is a generalization of how you'd use Docker to create an image and prepare it for distribution. You should be familiar with using `docker build` to create an image and `docker save` or `docker export` to create an image file. You can perform each of these operations with a single command.

You can use any file transport once you have an image in file form. One custom component not show in figure 9.7 is the mechanism that uploads an image to the transport. That mechanism may be a folder that is watched by a file-sharing tool like Dropbox. It could also be a piece of custom code that runs periodically, or in response to a new file, and uses FTP or HTTP to push the file to a remote server. Whatever the mechanism, this general component will require some effort to integrate.

The figure also shows how a client would ingest the image and use it to build a container after the image has been distributed. Similar to image origins, clients require some process or mechanism to acquire the image from a remote source. Once clients have the image file, they can use the `docker load` or `import` commands to complete the transfer.

It doesn't make sense to measure manual image distribution against individual selection criteria. Using a non-Docker distribution channel gives you full control. It will be up to you to determine how your options measure against the criteria shown in table 9.4.

Table 9.4 Performance of custom image distribution infrastructure.

Criteria	Rating	Notes
Cost	Good	Distribution costs are driven by bandwidth, storage, and hardware needs. Hosted distribution solutions like cloud storage will bundle these costs and generally scale down price per unit as your usage increases. But hosted solutions bundle in the cost of personnel and several other benefits that you may not need, driving up the price compared to a mechanism that you own.
Visibility	Good	Like private registries, most manual distribution methods are special and take more effort to advertise than well-known registries. Examples might include using popular websites or other well-known file distribution hubs.
Transport speed/size	Good	Whereas transport speed depends on the transport, file sizes are dependent on your choice of using layered images or flattened images. Remember, layered images maintain the history of the image, container-creation metadata, and old files that might have been deleted or overridden. Flattened images contain only the current set of files on the file system.
Availability control	Best	If availability control is an important factor for your case, you can use a transport mechanism that you own.
Longevity control	Bad	Using proprietary protocols, tools, or other technology that is neither open nor under your control will impact longevity control. For example, distributing image files with a hosted file-sharing service like Dropbox will give you no longevity control. On the other hand, swapping USB drives with your friend will last as long as the two of you decide to use USB drives.

Table 9.4 Performance of custom image distribution infrastructure.

Criteria	Rating	Notes
Access control	Bad	You could use a transport with the access control features you need or use file encryption. If you built a system that encrypted your image files with a specific key, you could be sure that only a person or people with the correct key could access the image.
Artifact integrity	Bad	Integrity validation is a more expensive feature to implement for broad distribution. At a minimum, you'd need a trusted communication channel for advertising cryptographic file signatures.
Secrecy	Good	You can implement content secrecy with cheap encryption tools. If you need meta-secrecy (where the exchange itself is secret) as well as content secrecy, then you should avoid hosted tools and make sure that the transport that you use provides secrecy (HTTPS, SFTP, SSH, or offline).
Requisite experience	Good	Hosted tools will typically be designed for ease of use and require a lesser degree of experience to integrate with your workflow. But you can use simple tools that you own as easily in most cases.

All the same criteria apply to manual distribution, but it's difficult to discuss them without the context of a specific transportation method.

9.4.1 A sample distribution infrastructure using the File Transfer Protocol

Building a fully functioning example will help you understand exactly what goes into a manual distribution infrastructure. This section will help you build an infrastructure with the File Transfer Protocol.

FTP is less popular than it used to be. The protocol provides no secrecy and requires credentials to be transmitted over the wire for authentication. But the software is freely available and clients have been written for most platforms. That makes FTP a great tool for building your own distribution infrastructure. Figure 9.8 illustrates what you'll build.

The example in this section uses two existing images. The first, `dockerinaction/ch9_ftpd`, is a specialization of the `centos:6` image where `vsftpd` (an FTP daemon) has been installed and configured for anonymous write access. The second image, `dockerinaction/ch9_ftp_client`, is a specialization of a popular minimal Alpine Linux image. An FTP client named `LFTP` has been installed and set as the entrypoint for the image.

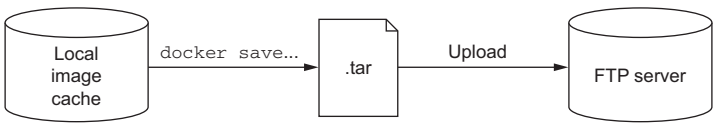


Figure 9.8 An FTP publishing infrastructure

To prepare for the experiment, pull a known image from Docker Hub that you want to distribute. In the example, the `registry:2` image is used:

```
docker pull registry:2
```

Once you have an image to distribute, you can begin. The first step is building your image distribution infrastructure. In this case, that means running an FTP server:

```
docker run -d --name ftp-transport -p 21:12 dockerinaction/ch9_ftpd
```

This command will start an FTP server that accepts FTP connections on TCP port 21 (the default port). Don't use this image in any production capacity. The server is configured to allow anonymous connections write access under the pub/incoming folder. Your distribution infrastructure will use that folder as an image distribution point.

The next thing you need to do is export an image to the file format. You can use the following command to do so:

```
docker save -o ./registry.2.tar registry:2
```

Running this command will export the `registry:2` image as a structured image file in your current directory. The file will retain all the metadata and history associated with the image. At this point, you could inject all sorts of phases like checksum generation or file encryption. This infrastructure has no such requirements, and you should move along to distribution.

The `dockerinaction/ch9_ftp_client` image has an FTP client installed and can be used to upload your new image file to your FTP server. Remember, you started the FTP server in a container named `ftp-transport`. If you're running the container on your computer, you can use container linking to reference the FTP server from the client; otherwise, you'll want to use host name injection (see chapter 5), a DNS name of the server, or an IP address:

```
docker run --rm --link ftp-transport:ftp_server \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; put registry.2.tar; exit' ftp_server
```

This command creates a container with a volume bound to your local directory and linked with your FTP server container. The command will use LFTP to upload a file named registry.2.tar to the server located at ftp_server. You can verify that you uploaded the image by listing the contents of the FTP server's folder:

```
docker run --rm --link ftp-transport:ftp_server \
   -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e "cd pub/incoming; ls; exit" ftp_server
```

The registry image is now available for download to any FTP client that knows about the server and can access it over the network. But that file may never be overridden in the current FTP server configuration. You'd need to come up with your own versioning scheme if you were going to use a similar tool in production.

Advertising the availability of the image in this scenario requires clients to periodically poll the server using the last command you ran. You could alternatively build some website or send an email notifying clients about the image, but that all happens outside the standard FTP transfer workflow.

Before moving on to evaluating this distribution method against the selection criteria, consume the registry image from your FTP server to get an idea of how clients would need to integrate.

First, eliminate the registry image from your local image cache and the file from your local directory:

```
rm registry.2.tar
docker rmi registry:2
```

Need to remove any registry containers first

Then download the image file from your FTP server:

```
docker run --rm --link ftp-transport:ftp_server \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; get registry.2.tar; exit' ftp_server
```

At this point you should once again have the registry.2.tar file in your local directory. You can reload that image into your local cache with the `docker load` command:

```
docker load -i registry.2.tar
```

This is a minimal example of how a manual image publishing and distribution infrastructure might be built. With just a bit of extension you could build a production-quality, FTP-based distribution hub. In its current configuration this example matches against the selection criteria, as shown in table 9.5.

Table 9.5 Performance of a sample FTP-based distribution infrastructure

Criteria	Rating	Notes
Cost	Good	This is a low-cost transport. All the related software is free. Bandwidth and storage costs should scale linearly with the number of images hosted and the number of clients.
Visibility	Worst	The FTP server is running in an unadvertised location with a non-standard integration workflow. The visibility of this configuration is very low.
Transport speed/size	Bad	In this example, all the transport happened between containers on the same computer, so all the commands finished quickly. If a client connects to your FTP service over the network, then speeds are directly impacted by your upload speeds. This distribution method will download redundant artifacts and won't download components of the image in parallel. Overall, this method isn't bandwidth-efficient.
Availability control	Best	You have full availability control of the FTP server. If it becomes unavailable, you're the only person who can restore service.

Table 9.5 Performance of a sample FTP-based distribution infrastructure

Criteria	Rating	Notes
Longevity control	Best	You can use the FTP server created for this example as long as you want.
Access control	Worst	This configuration provides no access control.
Artifact integrity	Worst	The network transportation layer does provide file integrity between endpoints. But it's susceptible to interception attacks, and there are no integrity protections between file creation and upload or between download and import.
Secrecy	Worst	This configuration provides no secrecy.
Requisite experience	Good	All requisite experience for implementing this solution has been provided here. If you're interested in extending the example for production, you'll need to familiarize yourself with `vsftpd` configuration options and `SFTP`

In short, there's almost no real scenario where this transport configuration is appropriate. But it helps illustrate the different concerns and basic workflows that you can create when you work with image as files. The only more flexible and potentially complicated image publishing and distribution method involves distributing image sources.

9.5 Image source distribution workflows

When you distribute image sources instead of images, you cut out all the Docker distribution workflow and rely solely on the Docker image builder. As with manual image publishing and distribution, source-distribution workflows should be evaluated against the selection criteria in the context of a particular implementation.

Using a hosted source control system like Git on GitHub will have very different traits from using a file backup tool like `rsync`. In a way, source-distribution workflows have a superset of the concerns of manual image publishing and distribution workflows. You'll have to build your workflow but without the help of the `docker save`, `load`, `export`, or `import` commands. Producers need to determine how they will package their sources, and consumers need to understand how those sources are packaged as well as how to build an image from them. That expanded interface makes source-distribution workflows the most flexible and potentially complicated distribution method. Figure 9.9 shows image source distribution on the most complicated end of the spectrum.

Image source distribution is one of the most common methods, despite having the most potential for complication. The reason is that the expanded interface has been standardized by popular version-control software.

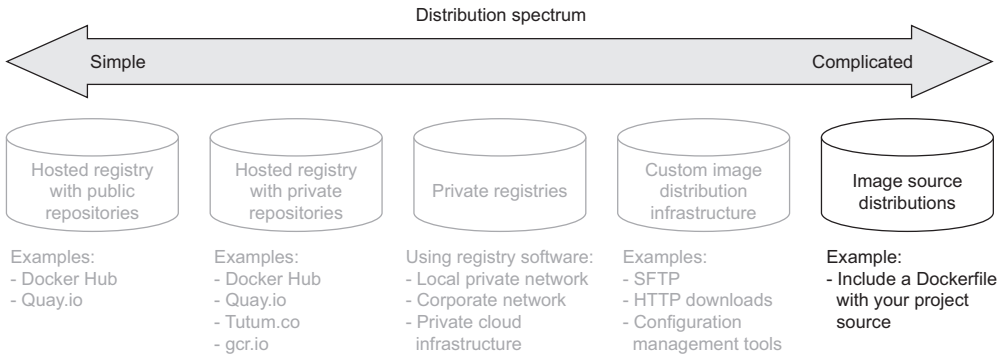


Figure 9.9 Using existing infrastructure to distribute image sources

9.5.1 Distributing a project with Dockerfile on GitHub

Using Dockerfile and GitHub to distribute image sources is almost identical to setting up automated builds on hosted Docker image repositories. All the steps for using Git to integrate your local Git repository with a repository on GitHub are the same. The only difference comes in that you don't create a Docker Hub account or repository. Instead, your image consumers will clone your GitHub repository directly and use `docker build` to build your image locally.

Supposing a producer had an existing project, Dockerfile, and GitHub repository, their distribution workflow would look like this:

```
git init
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git add Dockerfile
# git add *whatever other files you need for the image*
git commit -m "first commit"
git remote add origin https://github.com/<your username>/<your repo>.git
git push -u origin master
```

Meanwhile, a consumer would use a general command set that looks like this:

```
git clone https://github.com/<your username>/<your repo>.git
cd <your-repo>
docker build -t <your username>/<your repo> .
```

These are all steps that a regular Git or GitHub user is familiar with, as shown in table 9.6.

Table 9.6 Performance of image source distribution via GitHub

Criteria	Rating	Notes
Cost	Best	There's no cost if you're using a public GitHub repository.
Visibility	Best	GitHub is a highly visible location for open source tools. It provides excellent social and search components, making project discovery simple.

Table 9.6 Performance of image source distribution via GitHub

Criteria	Rating	Notes
Transport speed/size	Good	By distributing image sources, you can leverage other registries for base layers. Doing so will reduce the transportation and storage burden. GitHub also provides a content delivery network (CDN). That CDN is used to make sure clients around the world can access projects on GitHub with low network latency.
Availability control	Worst	Relying on GitHub or other hosted version-control providers eliminates any availability control.
Longevity control	Bad	Although Git is a popular tool and should be around for a while, you forego any longevity control by integrating with GitHub or other hosted version-control providers.
Access control	Good	GitHub or other hosted version-control providers do provide access control tools for private repositories.
Artifact integrity	Good	This solution provides no integrity for the images produced as part of the build process, or of the sources after they have been cloned to the client machine. But integrity is the whole point of version-control systems. Any integrity problems should be apparent and easily recoverable through standard Git processes.
Secrecy	Worst	Public projects provide no source secrecy.
Requisite Experience	Good	Image producers and consumers need to be familiar with Dockerfile, the Docker builder, and the Git tooling.

Image source distribution is divorced from all Docker distribution tools. By relying only on the image builder, you're free to adopt any distribution toolset available. If you're locked into a particular toolset for distribution or source control, this may be the only option that meets your criteria.

9.6 Summary

This chapter covers various software distribution mechanisms and the value contributed by Docker in each. A reader that has recently implemented a distribution channel, or is currently doing so, might take away additional insights into their solution. Others will learn more about available choices. In either case, it is important to make sure that you have gained the following insights before moving on:

- Having a spectrum of choices illustrates your range of options.
- You should always use a consistent set of selection criteria in order to evaluate your distribution options and determine which method you should use.
- Hosted public repositories provide excellent project visibility, are free, and require very little experience to adopt.
- Consumers will have a higher degree of trust in images generated by automated builds because a trusted third party builds them.

- Hosted private repositories are cost-effective for small teams and provide satisfactory access control.
- Running your own registry enables you to build infrastructure suitable for special use cases without abandoning the Docker distribution facilities.
- Distributing images as files can be accomplished with any file-sharing system.
- Image source distribution is flexible but only as complicated as you make it. Using popular source-distribution tools and patterns will keep things simple.

10 Running customized registries

This chapter covers

- Working directly with the Registry API
- Building a central registry
- Registry authentication tools
- Configuring a registry for scale
- Integrating through notifications

Chapter 9 covers several methods of distributing Docker images, one of them involving running a Docker registry. A Docker registry is a flexible image distribution component that's useful on its own or as part of larger complex systems. For that reason, understanding how to configure your own registry will help you get the most out of Docker.

Someone developing software that integrates with a Docker registry may want to run a local instance to develop against. They might also use it as a staging environment for their project. A development team might deploy their own central registry to share their work and streamline integrations. A company may run one or more centralized registries that are backed by durable artifact storage. These could be used to control external image dependencies or for managing deployment artifacts.

Figure 10.1 illustrates these configurations. This chapter covers all these use cases, scaling approaches, and an introduction to the Registry API itself. By the end of this chapter you will be able to launch an appropriately configured registry for any use case.

Any program that implements the Registry API is a registry. This chapter uses the Distribution (docker/distribution) project. The project is available on Docker Hub in the registry repository.

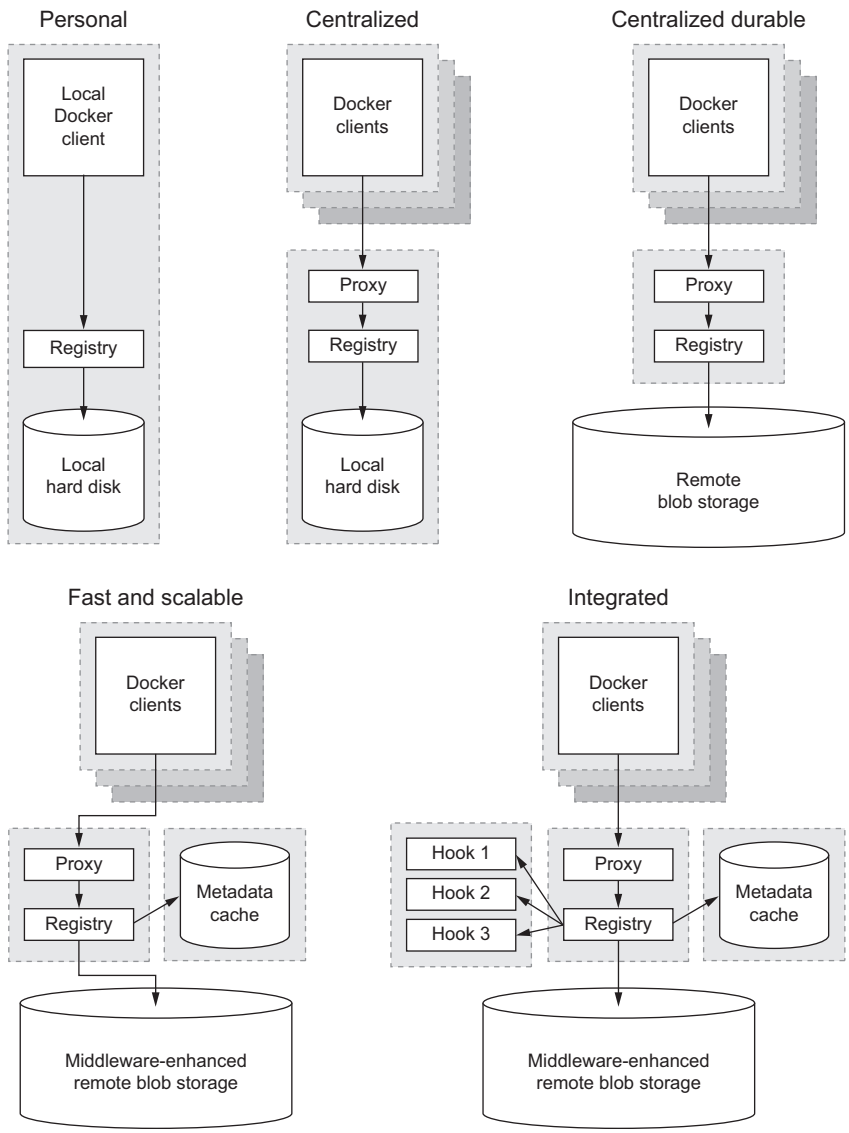


Figure 10.1 Registry configurations ranging from personal to reliable, scalable, and integrated

10.1 Running a personal registry

Launching a personal registry is great if you're just getting started or using your own for development purposes. That material is introduced in chapter 9, but this section demonstrates how to use the Registry API to access your new registry.

10.1.1 Reintroducing the Image

Over the course of this chapter you'll launch and relaunch containers providing a registry several times. In each instance you're going to use specializations of the Distribution project as provided by the registry:2 repository.

A personal registry rarely requires customization. In such a use case, you can use the official image. Pull the image and launch a personal registry to get started:

```
docker run -d --name personal_registry \
    -p 5000:5000 --restart=always \
    registry:2
```

TIP At the time of this writing, the `latest` tag refers to the last version of the registry as implementing the V1 Registry API. The examples in this chapter require the V2 API unless otherwise noted.

The Distribution project runs on port 5000, but clients make no assumptions about locations and attempt connecting to port 80 (HTTP) by default. You could map port 80 on your host to port 5000 on the container, but in this case you should map port 5000 directly. Anytime you connect to the registry, you'll need to explicitly state the port where the registry is running.

The container you started from the registry image will store the repository data that you send to it in a managed volume mounted at /var/lib/registry. This means you don't have to worry about the data being stored on the main layered file system.

An empty registry is a bit boring, so tag and push an image into it before moving on. Use the registry image itself, but in order to differentiate the example, use a different repository name:

```
docker tag registry:2 localhost:5000/distribution:2
docker push localhost:5000/distribution:2
```

The `push` command will output a line for each image layer that's uploaded to the registry and finally output the digest of the image. If you want, you can remove the `local` tag for localhost:5000/distribution:2 and then try pulling from your registry:

```
docker rmi localhost:5000/distribution:2
docker pull localhost:5000/distribution:2
```

All these commands are covered in chapter 2 and chapter 7. The difference is that in those chapters you're working with hosted registries. This example highlights that your knowledge, scripts, and automation infrastructure are portable between hosted solutions and your custom infrastructure when run your own registry. Using the

command-line tools like this is great for scripts and users, but you'll want to use the API directly if you're developing software to integrate with a registry.

10.1.2 Introducing the V2 API

The V2 Registry API is RESTful. If you're unfamiliar with RESTful APIs, it's enough to know that a RESTful API is a patterned use of Hypertext Transfer Protocol (HTTP) and its primitives to access and manipulate remote resources. There are several excellent online resources and books on the subject. In the case of a Docker registry, those resources are tags, manifests, blobs, and blob uploads. You can find the full Registry specification at https://docs.docker.com/registry/spec/api/.

Because RESTful APIs use HTTP, you might take some time to familiarize yourself with the details of that protocol. The examples in this chapter are complete, and you won't need any deep knowledge of HTTP in order to follow along, but you'll get much more from the experience if you're comfortable with what's happening behind the scenes.

In order to exercise any RESTful service you'll need an HTTP client. The truly savvy reader may know how to use a raw TCP connection to issue HTTP requests, but most prefer not to bother with the low-level details of the protocol. Although web browsers are capable of making HTTP requests, command-line tools will give you the power to fully exercise a RESTful API.

The examples in this chapter use the `cURL` command-line tool. Because this is a book about Docker, you should use `cURL` from inside a container. Using `cURL` in this way will also work for both Docker native and Boot2Docker users. Prepare an image for this purpose (and practice your Dockerfile skills):

```
FROM gliderlabs/alpine:latest
LABEL source=dockerinaction
LABEL category=utility
RUN apk --update add curl
ENTRYPOINT ["curl"]
CMD ["--help"]
```

From curl.df

```
docker build -t dockerinaction/curl -f curl.df .
```

With your new `dockerinaction/curl` image, you can issue the `cURL` commands in the examples without worrying about whether `cURL` is installed or what version is installed on your computer. Celebrate your new image and get started with the Registry API by making a simple request to your running registry:

```
docker run --rm --net host dockerinaction/curl -Is
    http://localhost:5000/v2/
```

Note the /v2/

That request will result in the following output:

```
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
```

This command is used to validate that the registry is running the V2 Registry API and returns the specific API version in the HTTP response headers. The last component of that request, `/v2/`, is a prefix for every resource on the V2 API.

If you accidentally issued this request to a registry that was running the V1 API, the output would look something like this:

```
HTTP/1.1 404 NOT FOUND
Server: gunicorn/19.1.1
Connection: keep-alive
Content-Type: text/html
Content-Length: 233
```

> **HTTP DETAIL** This command used an `HTTP HEAD` request to retrieve only the response headers. A successful `GET` request for this resource will return the same headers and response body with an empty document.

Now that you've used `cURL` to validate that the registry is indeed using the V2 API, you should do something a bit more interesting. The next command will retrieve the list of tags in the distribution repository on your registry:

```
docker run --rm -u 1000:1000 --net host \
    dockerinaction/curl -s http://localhost:5000/v2/distribution/tags/list
```

Running that command should display a result like the following:

```
{"name":"distribution","tags":["2"]}
```

Here you can see that the registry responded to your request with a JSON document that lists the tags in your distribution repository. In this case there's only one, `2`. A JSON document is a structured document of key-value pairs. It uses braces to represent objects (which contain another set of key-value pairs), square brackets to represent lists, and quoted strings to label elements and represent string values.

Run this example again, but this time add another tag to your repository to yield a more interesting result:

```
docker tag \
    localhost:5000/distribution:2 \
    localhost:5000/distribution:two

docker push localhost:5000/distribution:two

docker run --rm \
    -u 1000:1000 \
    --net host \
    dockerinaction/curl \
    -s http://localhost:5000/v2/distribution/tags/list
```

Creative tag name

Run as unprivileged user

Run without network namespace

The `curl` command will return output like the following:

```
{"name":"distribution","tags":["2","two"]}
```

In the output you can clearly see that the repository contains both the tags that you've defined. Each distinct tag that's pushed to this repository on your registry will be listed here.

You can use the personal registry that you created in this section for any testing or personal productivity needs you might have. But the uses are somewhat limited without any changes to the configuration. Even if you have no plans for deploying a centralized registry, you may benefit from some customization to adopt the registry's notifications feature discussed in section 10.5. You'll need to know how to customize the registry image before you can make those changes.

10.1.3 Customizing the Image

The remainder of this chapter explains how to build upon the registry image to grow out of the personal registry and into more advanced use cases. You'll need to know a bit more about the registry image itself before you can do that.

This chapter will specialize the registry image with Dockerfiles. If you're unfamiliar with Dockerfile syntax and haven't read chapter 8, then you should do so now or review the online documentation at https://docs.docker.com/reference/builder.

The first things you need to know about the image are the key components:

- The base image for `registry` is Debian and it has updated dependencies.
- The main program is named `registry` and is available on the `PATH`.
- The default configuration file is config.yml.

Different project maintainers will all have different ideas about the best base image. In the case of Docker, Debian is a frequent choice. Debian has a minimal footprint for a fully featured distribution and takes only approximately 125 MB on your hard drive. It also ships with a popular package manager, so installing or upgrading dependencies should never be an issue.

The main program is named `registry` and is set as the entrypoint for the image. This means that when you start a container from this image, you can omit any command arguments to take the default behavior or add your own arguments directly to the trailing portion of the `docker run` command.

Like all Docker projects, the Distribution project aims to provide a sensible default configuration. That default configuration is located in a file named config.yml. As the name implies, the configuration is written in YAML. If you're not familiar with YAML, there's no reason to worry. YAML is designed to maximize human readability. If you're interested, you'll find several YAML resources at http://yaml.org.

This configuration file is the real star of this chapter. There are several ways that you might inject your own configuration in the image. You might modify the included file directly. A bind-mount volume could be used to override the file with one that you've been writing. In this case, you'll copy in your own file to another location and set a new default command for your new image.

The configuration file contains nine top-level sections. Each defines a major functional component of the registry:

- `version`—This is a required field and specifies the configuration version (not software version).
- `log`—The configuration in this section controls the logging output produced by the Distribution project.
- `storage`—The `storage` configuration controls where and how images are stored and maintained.
- `auth`—This configuration controls in-registry authentication mechanisms.
- `middleware`—The `middleware` configuration is optional. It's used to configure the storage, registry, or repository middleware in use.
- `reporting`—Certain reporting tools have been integrated with the Distribution project. These include Bugsnag and NewRelic. This section configures each of those toolsets.
- `http`—This section specifies how Distribution should make itself available on the network.
- `notifications`—Webhook-style integration with other projects is configured in the `notifications` section.
- `redis`—Finally, configuration for a Redis cache is provided in the `redis` section.

With these components in mind, you should be ready to build advanced registry use cases by customizing the registry image. Remember, the Distribution project is rapidly evolving. If you hit a snag with the instructions in this book, you can always consult the online documentation for reference or check out the project itself at https://github.com/docker/distribution.

10.2 Enhancements for centralized registries

Launching a local copy of the registry image with no modifications works well for personal purposes or testing. When more than one person needs access to the same registry it is called a centralized registry. This section explains how to implement a centralized registry by customizing the official registry image. Figure 10.2 shows the changes involved in growing from a personal registry into a centralized registry.

For more than one person to access the registry, it will need to be available on a network. You can accomplish that easily enough by mapping the registry container to port 80 on the network interface of the computer it's running on (`docker run ... -p 80:5000 ...`). Introducing a dependency on the network introduces a whole set of new vulnerabilities. From snooping to corrupting image transfers, man-in-the-middle attacks can create several problems. Adding transport layer security will protect your system from these and other attacks.

Once clients can access the registry, you'll want to make sure that only the right users can access it. This is called authentication (covered in section 10.2.3).

The most unavoidable issue that any service owner encounters is client compatibility. With multiple clients connecting to your registry, you need to consider what

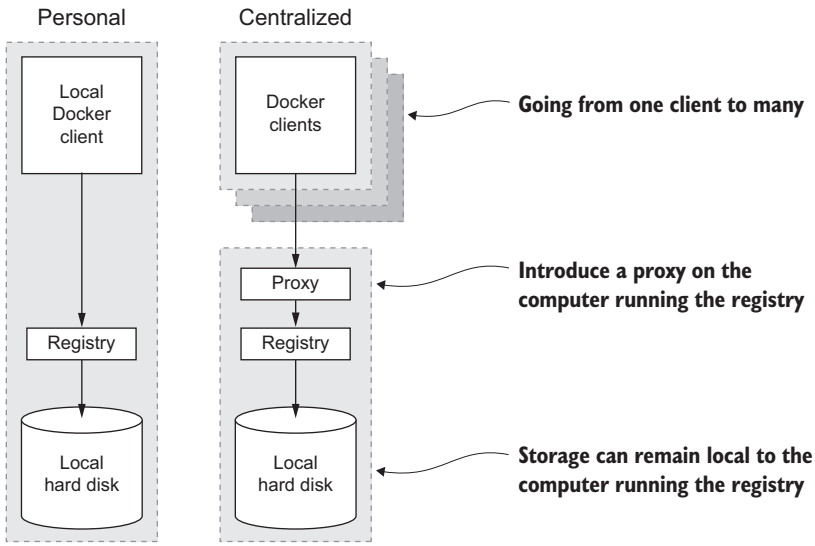


Figure 10.2 Personal versus centralized Docker registry system architecture

versions of Docker they're using. Supporting multiple client versions can be tricky. Section 10.2.4 demonstrates one way to handle the problem.

Before moving on to the next registry type, section 10.2.5 covers best practices for production registry configurations. These include hardening and preventive maintenance steps.

Most of these concerns are focused on interactions with clients. Although the Distribution software has some tools to meet the new needs, adding a proxy to the system introduces the required flexibility to implement all these enhancements.

10.2.1 Creating a reverse proxy

Creating a reverse proxy is a quick task when you use the right base image and Dockerfile to make your customizations. Figure 10.3 illustrates the relationship between the reverse proxy and your registry.

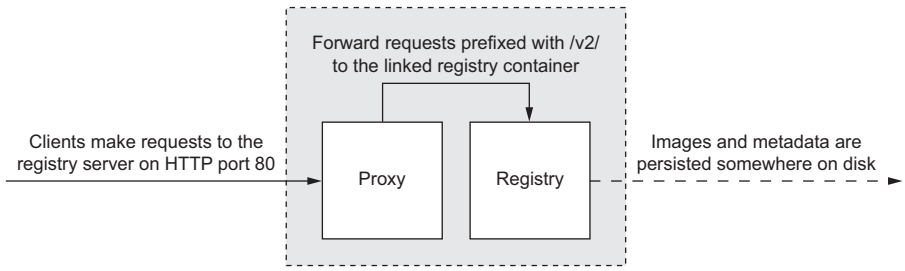


Figure 10.3 Inserting a reverse proxy between clients and a registry

Your reverse proxy configuration will involve two containers. The first will run the NGINX reverse proxy. The second will run your registry. The reverse proxy container will be linked to the registry container on the host alias, `registry`. Get started by creating a new file named basic-proxy.conf and include the following configuration:

From basic-proxy.conf

Link alias requirement

```
upstream docker-registry {
  server registry:5000;
}

server {
  listen 80;
  # Use the localhost name for testing purposes
  server_name localhost;
  # A real deployment would use the real hostname where it is deployed
  # server_name mytotallyawesomeregistry.com;

  client_max_body_size 0;
  chunked_transfer_encoding on;

  # We're going to forward all traffic bound for the registry
  location /v2/ {
    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

Container port requirement

Note /v2/ prefix

Resolves to the upstream

As stated earlier in the book, NGINX is a sophisticated piece of software. Whole books have been dedicated to its use, and so I won't do it the disservice of attempting to explain it here. The bits of this configuration that you should understand are annotated. This configuration will forward all traffic on port 80 for the HTTP host localhost and with the path prefix /v2/ on to http://registry:5000. This configuration will be the base for other modifications you make to your reverse proxy going forward.

Once you have your reverse proxy configuration, you'll want to build a new image. A minimal Dockerfile should suffice. It should start from the latest NGINX image and include your new configuration. The base NGINX image takes care of all the standard things like exposing ports. Create a new file named basic-proxy.df and paste in the following Dockerfile:

From basic-proxy.df

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ./basic-proxy.conf /etc/nginx/conf.d/default.conf
```

TIP In production, you really should include an environment validation script (see section 8.4.1) just to make sure that the containers have been linked correctly. But because you're using an NGINX upstream directive, it will verify that the host can be resolved for you.

At this point you should be ready to build your image. Use the following `docker build` command to do so:

```
docker build -t dockerinaction/basic_proxy -f basic-proxy.df .
```

Now you're ready to put it all together. Your personal registry should already be running from the example in section 10.1.1. Use it again here because it should already be primed with some content. The following commands will create your new reverse proxy and test the connection:

```
docker run -d --name basic_proxy -p 80:80 \
    --link personal_registry:registry \
    dockerinaction/basic_proxy

docker run --rm -u 1000:1000 --net host \
    dockerinaction/curl \
    -s http://localhost:80/v2/distribution/tags/list
```

Start reverse proxy

Link to registry

Run cURL to query your registry through the proxy

A few things are happening quickly, so before you move on, take some time to make sure you understand what's going on here.

You created a personal registry earlier. It's running in a container named `personal_registry` and has exposed port 5000. You also added a few tagged images to a repository named distribution hosted by your personal registry. The first command in this set creates a new container running a reverse proxy that listens on port 80. The reverse proxy container is linked to your registry. Any traffic that the proxy receives on port 80 that's also requesting a path prefixed with /v2/ will be forwarded to port 5000 on your registry container.

Finally, you run a `curl` command from the same host where your Docker daemon is running. That command makes a request to port 80 on the localhost (in this case you need that host name). The request is proxied to your registry (because the path starts with /v2/), and the registry responds with a list of tags contained in the distribution repository.

> **NOTE** This end-to-end test is very similar to what would happen if your proxy and registry were deployed to a different host with some known name. Comments in the basic-proxy.conf file explain how to set the host name for production deployments.

The reverse proxy that you're building here doesn't add anything to your system except another hop in the network and a hardened HTTP interface. The next three sections explain how to modify this basic configuration to add TLS, authentication, and multi-version client support.

10.2.2 Configuring HTTPS (TLS) on the reverse proxy

Transport layer security (TLS) provides endpoint identification, message integrity, and message privacy. It's implemented at a layer below HTTP and is what provides the *S* in HTTPS. Using TLS to secure your registry is more than a best practice. The Docker

daemon won't connect to a registry without TLS unless that registry is running on localhost. That makes these steps mandatory for anyone running a centralized registry.

What about SSH tunnels?

Readers who have experience with TLS may already know that the power the public key infrastructure provides comes with expense and complexity. A cheap and arguably less complex way to secure your registry's network traffic is to enable connections only through Secure Shell (SSH).

SSH uses similar security techniques as TLS but lacks the third-party trust mechanism that makes TLS scale to large numbers of users. But SSH does provide a protocol for tunneling network traffic.

To secure your registry with SSH, you'd install an SSH server (OpenSSH) on the same machine as the registry. With the SSH server in place, map the registry only to the loopback interface (localhost) on the machine. Doing so will restrict inbound registry traffic to what comes through the SSH server.

When clients want to use your registry in this configuration, they'll create an SSH tunnel. That tunnel binds a local TCP port and forwards traffic to it over an SSH connection between your computer and the remote SSH server and out to some destination host and port. To put this in context, clients create a tunnel that allows them to treat your registry as if it were running locally. A client would use a command line like this to create the tunnel:

```
ssh -f -i my_key user@ssh-host -L 4000:localhost:5000 -N
```

With the tunnel created, the client would use the registry as if it were running locally on port 4000.

Using SSH tunnels might work if you're running a centralized registry for a small team. A prerequisite to their use is user account management and authentication (so you can solve the authentication issue at the same time). But the practice doesn't scale well because of account management overhead and generally requires a higher degree of user sophistication than HTTPS.

An HTTPS endpoint is different from the HTTP endpoint in three ways. First, it should listen on TCP port 443. Second, it requires signed certificate and private key files. Last, the host name of the server and the proxy configuration must match the one used to create the certificate. In this example, you're going to create a self-signed certificate for the localhost name. Such a certificate won't work well for a real registry, but there are many guides available to help you replace that certificate with one issued by a certificate authority. Figure 10.4 illustrates the new HTTPS protection in relation to your proxy and registry.

The first step in making this design a reality is to generate a private and public key pair and a self-signed certificate. Without Docker you'd need to install OpenSSL and

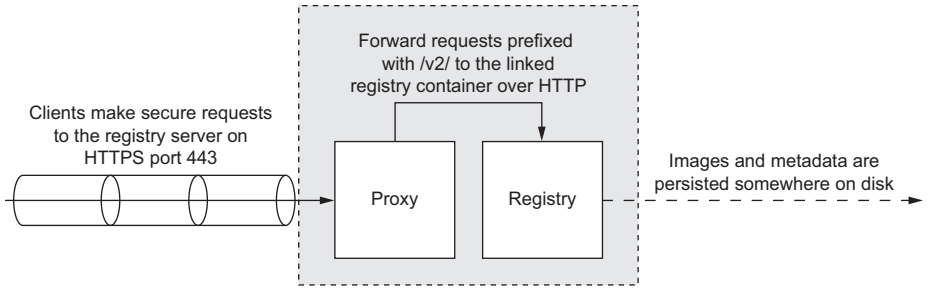


Figure 10.4 Adding an HTTPS (TLS) endpoint to the proxy

run three complicated commands. With Docker (and a public image created by CenturyLink) you can do the whole thing with one command:

```
docker run --rm -e COMMON_NAME=localhost -e KEY_NAME=localhost \
    -v "$(pwd)":/certs centurylink/openssl
```

This command will generate a 4096-bit RSA key pair and store the private key file and self-signed certificate in your current working directory. The image is publicly available and maintained with an automated build on Docker Hub. It's fully auditable, so the more paranoid are free to validate (or re-create) the image as needed. Of the three files that are created, you'll use two. The third is a certificate-signing request (CSR) and can be removed.

The next step is to create your proxy configuration file. Create a new file named tls-proxy.conf and copy in the following configuration. Again, relevant lines are annotated:

```
upstream docker-registry {                          From tls-proxy.conf
  server registry:5000;
}

server {
  listen 443 ssl;                                   Note use of port 443 and "ssl"
  server_name localhost                             Named localhost

  client_max_body_size 0;
  chunked_transfer_encoding on;

  ssl_certificate /etc/nginx/conf.d/localhost.crt;  Note SSL configuration
  ssl_certificate_key /etc/nginx/conf.d/localhost.key;

  location /v2/ {
    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

The differences between this configuration and the basic proxy configuration include the following:

- Listening on port 443
- Registration of an SSL certificate
- Registration of an SSL certificate key

You should note that this proxy configuration is set to use the same registry on port 5000. Running multiple proxies against the same registry is no different from the registry's perspective than running multiple clients.

The last step before you put it all together is the creation of a Dockerfile. This time, in addition to copying the proxy configuration, you'll also need to copy the certificate and key file into the image. The following Dockerfile uses the multisource form of the `COPY` directive. Doing so eliminates the need for multiple layers that you might otherwise create as the result of multiple `COPY` directives. Create a new file named tls-proxy.df and insert the following lines:

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ["./tls-proxy.conf", \
      "./localhost.crt", \
      "./localhost.key", \
      "/etc/nginx/conf.d/"]
```

Copy new certificate

Copy private key

Build your new image with the following `docker build` command:

```
docker build -t dockerinaction/tls_proxy -f tls-proxy.df .
```

Now put it all together by starting your proxy and testing it with `curl`:

```
docker run -d --name tls-proxy -p 443:443  \
    --link personal_registry:registry \
    dockerinaction/tls_proxy

docker run --rm \
    --net host \
    dockerinaction/curl -ks \
    https://localhost:443/v2/distribution/tags/list
```

Note port 443

Note link

Note "k" flag

Note "https" and "443"

This command should list both tags for the distribution repository in your personal registry:

```
{"name":"distribution","tags":["2","two"]}
```

The `curl` command in this example uses the `-k` option. That option will instruct `curl` to ignore any certificate errors with the request endpoint. It's required in this case because you're using a self-signed certificate. Aside from that nuance, you're successfully making a request to your registry over HTTPS.

10.2.3 Adding an authentication layer

There are three mechanisms for authentication included with the Distribution project itself. These are appropriately named silly, token, and htpasswd. As an alternative to configuring authentication directly on the Distribution project, you can configure various authentication mechanisms in the reverse proxy layer (see section 10.2.2).

The first, silly, is completely insecure and should be ignored. It exists for development purposes only and may be removed in later versions of the software.

The second, token, uses JSON web token (JWT) and is the same mechanism that's used to authenticate with Docker Hub. It's a sophisticated approach to authentication that enables the registry to validate that a caller has authenticated with a third-party service without any back-end communication. The key detail to take away from this is that users don't authenticate with your registry directly. Using this mechanism requires that you deploy a separate authentication service.

There are a few open source JWT authentication services available but none that can be recommended for production use. Until the JWT ecosystem matures, the best course of action is to use the third authentication mechanism, htpasswd.

htpasswd is named for an open source program that ships with the Apache Web Server utilities. htpasswd is used to generate encoded username and password pairs where the password has been encrypted with the bcrypt algorithm. When you adopt the htpasswd authentication form, you should be aware that passwords are sent from the client to your registry unencrypted. This is called HTTP basic authentication. Because HTTP basic sends passwords over the network, it's critical that you use this authentication form in tandem with HTTPS.

There are two ways to add htpasswd authentication to your registry: at the reverse proxy layer and on the registry itself. In either case, you'll need to create a password file with htpasswd. If you don't have htpasswd installed, you can do so using Docker. Create an image from the following Dockerfile (named htpasswd.df) and build command:

```
FROM debian:jessie
LABEL source=dockerinaction
LABEL category=utility
RUN apt-get update && \
    apt-get install -y apache2-utils
ENTRYPOINT ["htpasswd"]
```

Build your image once you have the Dockerfile:

```
docker build -t htpasswd -f htpasswd.df .
```

With your new image available, you can create a new entry for a password file like so:

```
docker run -it --rm htpasswd -nB <USERNAME>
```

It's important to replace `<USERNAME>` with the username you want to create and use the `-nB` flags for htpasswd. Doing so will display the output in your terminal and use

the bcrypt algorithm. The program will prompt you for your password twice and then generate the password file entry. Copy the result into a file named registry.password. The result should look something like this:

```
registryuser:$2y$05$mfQjXkprC94Tjk4IQz4vOOK6q5VxUhsxC6zajd35ys1O2J2x1aLbK
```

Once you have a password file, you can implement HTTP Basic authentication in NGINX by simply adding two lines to the configuration file presented in section 10.2.2. Create tls-authproxy.conf and add these lines:

```
# filename: tls-auth-proxy.conf
upstream docker-registry {
  server registry:5000;
}

server {
  listen 443 ssl;
  server_name localhost

  client_max_body_size 0;
  chunked_transfer_encoding on;

  # SSL
  ssl_certificate /etc/nginx/conf.d/localhost.crt;
  ssl_certificate_key /etc/nginx/conf.d/localhost.key;

  location /v2/ {
    auth_basic "registry.localhost";
    auth_basic_user_file /etc/nginx/conf.d/registry.password;

    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

Authentication realm

Password file

Now create a new Dockerfile named tls-auth-proxy.df:

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ["./tls-auth-proxy.conf", \
      "./localhost.crt", \
      "./localhost.key", \
      "./registry.password", \
      "/etc/nginx/conf.d/"]
```

With that change, you could use the rest of the instructions in section 10.2.2 to rebuild your registry to process HTTP basic authentication. Rather than repeat that work, it's more worthwhile to configure Distribution to use TLS and HTTP basic.

Adding TLS and basic authentication directly to your registry is useful if you want to tighten security on your personal registry. In production it's likely more suitable to terminate the TLS connection at your proxy layer.

The following configuration file (named tls-auth-registry.yml) adds TLS and HTTP basic authentication to an otherwise default Distribution container:

```
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    filesystem:
        rootdirectory: /var/lib/registry
    cache:
        layerinfo: inmemory
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    tls:
        certificate: /localhost.crt
        key: /localhost.key
    debug:
        addr: localhost:5001
auth:
    htpasswd:
        realm: registry.localhost
        path: /registry.password
```

TLS configuration

Authentication configuration

The annotated text shows both sections of the configuration that have been changed. The first is the `http` section. A subsection has been added named `tls`. The `tls` section has two properties, `certificate` and `key`. The values of these properties are paths to the certificate and key file that you generated in section 10.2.2. You'll need to either copy these files into the image or use volumes to mount them into the container at runtime. It's always a bad idea to copy key files into an image for anything other than testing purposes.

The second new section is `auth`. As you can see, the new htpasswd section uses two properties. The first, `realm`, simply defines the HTTP authentication realm. It's just a string. The second, `path`, is the location of the registry.password file that you created with htpasswd. Put all these things together with a quick Dockerfile (named tls-auth-registry.df):

```
# Filename: tls-auth-registry.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use
```

```
# Setting it early will enable layer caching if the
# tls-auth-registry.yml changes.
CMD ["/tls-auth-registry.yml"]
COPY ["./tls-auth-registry.yml", \
      "./localhost.crt", \
      "./localhost.key", \
      "./registry.password", \
      "/"]
```

Again, copying your key file into the image for production is a bad idea. Use volumes instead. The previous Dockerfile copies all your specialization material into the root directory for demonstration purposes. Build and launch the new secured registry with `docker build` and `docker run`:

```
docker build -t dockerinaction/secure_registry -f tls-auth-registry.df  .
```

```
docker run -d --name secure_registry \
    -p 5443:5000 --restart=always \
    dockerinaction/secure_registry
```

If you secure the registry itself with TLS, you may encounter problems when you install a reverse proxy. The reason is that application-level proxy software (like NGINX or Apache httpd) operates at the HTTP level. It needs to inspect request traffic in order to know how it needs to be routed or to route traffic from a specific client consistently to the same upstream host. Such a proxy would see encrypted traffic only if the TLS session was terminated by the registry. A functioning solution would either terminate the TLS session at the proxy layer (as you did earlier) or use a proxy (load-balancer) that operates at a lower network layer (like layer 4). For more information about network layers, look up information on the OSI model.

One example where your proxy would need to inspect the request content in order to route correctly is if you need to support multiple client versions. The Registry API changed with the release of Docker 1.6. If you want to support both Registry APIs, then you'll need to implement an API-aware proxy layer.

10.2.4 Client compatibility

The registry protocol changed dramatically between version 1 and version 2. Docker clients older than version 1.6 can't talk to version 2 registries. Distinguishing between version 1- and version 2-compatible clients and subsequently directing requests to compatible registry services are simple with our proxy in place.

For the sake of clarity, the examples in this section omit any HTTPS or authentication. But you're encouraged to combine the relevant features to build a proxy to suit your needs. Figure 10.5 shows the proxy and routing configuration that you'll build to support multiple client versions.

Like the previous proxies that you've built, this modification requires three steps:

- Create an NGINX configuration file (dual-client-proxy.conf).
- Create a brief Dockerfile (dual-client-proxy.df).
- Build the new image.

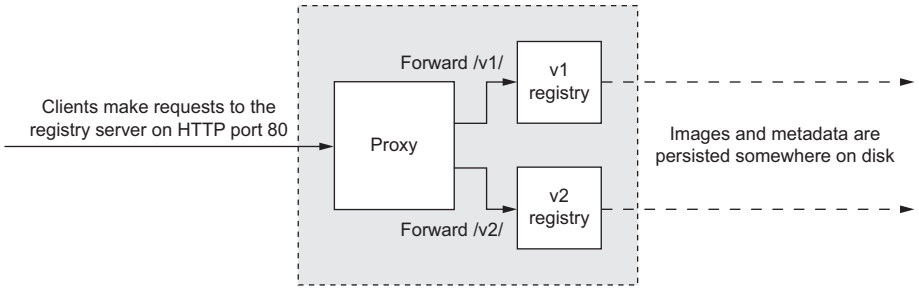


Figure 10.5 Routing clients based on requested Registry API version

Start by placing the new proxy configuration in a file named dual-client-proxy.conf and include the following:

```
upstream docker-registry-v2 {
  server registry2:5000;
}
upstream docker-registry-v1 {
  server registry1:5000;
}

server {
  listen 80;
  server_name localhost;

  client_max_body_size 0;
  chunked_transfer_encoding on;

  location /v1/ {
    proxy_pass                      http://docker-registry-v1;
    proxy_set_header  Host          $http_host;
    proxy_set_header  X-Real-IP     $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout              900;
  }

  location /v2/ {
    proxy_pass                      http://docker-registry-v2;
    proxy_set_header  Host          $http_host;
    proxy_set_header  X-Real-IP     $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout              900;
  }
}
```

V2 registry upstream

V1 registry upstream

V1 URL prefix

V1 upstream routing

V2 URL prefix

V2 upstream routing

The only notable difference from the basic proxy configuration is the inclusion of a second upstream server and a second location specification for URLs starting with

/v1/. Next, create a new Dockerfile named dual-client-proxy.df and include the following directives:

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ./dual-client-proxy.conf /etc/nginx/conf.d/default.conf
```

Last, create your new image:

```
docker build -t dual_client_proxy -f dual-client-proxy.df .
```

Before you can start a proxy that forwards traffic for both v1 and v2 Registry API requests, you'll need to have a v1 registry running. The following command will pull the 0.9.1 registry software and start it in a container:

```
docker run -d --name registry_v1 registry:0.9.1
```

With both versions of the registry running, you can finally create your dual-API support proxy. The following commands will create the proxy and link it to both registries and then test the v1 and v2 APIs:

```
docker run -d --name dual_client_proxy \
    -p 80:80 \
    --link personal_registry:registry2 \
    --link registry_v1:registry1 \
    dual_client_proxy

docker run --rm -u 1000:1000 \
    --net host \
    dockerinaction/curl -s http://localhost:80/v1/_ping

docker run --rm -u 1000:1000 \
    --net host \
    dockerinaction/curl -Is http://localhost:80/v2/
```

Test v1 from host

Test v2 from host

As time goes on, fewer and fewer clients will require v1 registry support. But it's always possible that another API change could happen in the future. Proxies are the best way to handle those situations.

10.2.5 Before going to production

Production configurations should typically differ from development or test configurations in a few specific ways. The most prominent difference is related to secret materials management, followed by log tuning, debug endpoints, and reliable storage.

Secrets such as private keys should never be committed to an image. It's easy to move or manipulate images in such a way that secrets can be revealed. Instead, any system that is serious about the security of secrets should only store secrets in read-protected memory or something more robust like a software vault or hardware security module.

The Distribution project makes use of a few different secrets:

- TLS private key
- SMTP username and password

- Redis secret
- Various remote storage account ID and key pairs
- Client state signature key

It's important that these not be committed to your production registry configuration or included with any image that you might create. Instead, consider injecting secret files by bind-mounting volumes that are on mounted tmpfs or RAMDisk devices and setting limited file permissions. Secrets that are sourced directly from the configuration file can be injected using environment variables.

Environment variables prefixed with `REGISTRY_` will be used as overrides to the configuration loaded by the Distribution project. Configuration variables are fully qualified and underscore-delimited for indentation levels. For example, the client state secret in the configuration file at

```
http:
    secret: somedefaultsecret
```

can be overridden using an environment variable named `REGISTRY_HTTP_SECRET`. If you want to start a container running the Distribution project in production, you should inject that secret using the `-e` flag on the `docker run` command:

```
docker run -d -e REGISTRY_HTTP_SECRET=<MY_SECRET> registry:2
```

There are a growing number of centralized secret management and secret distribution projects. You should invest some time in selecting one for your production infrastructure.

In production a logging configuration that's set too sensitive can overwhelm disk or log-handling infrastructure. Dial down the log level by setting an environment variable. Set `REGISTRY_LOG_LEVEL` to `error` or `warn`:

```
docker run -d -e REGISTRY_LOG_LEVEL=error registry:2
```

The next production configuration difference is simple. Disable the debug endpoint. This can be accomplished with environment variable configuration overriding. Setting `REGISTRY_HTTP_DEBUG` to an empty string will ensure Distribution doesn't start a debug endpoint:

```
docker run -d -e REGISTRY_HTTP_DEBUG='' registry:2
```

When you deploy a registry to a production environment, you'll likely need to move storage off the local file system. The biggest issue with local file system storage is specialization. Every image stored in a registry that uses local storage specializes the computer where it's running. That specialization reduces the durability of the registry. If the hard drive storing the registry data crashes or that one machine becomes inoperable, then you may experience data loss or at best reduced availability.

Using local storage in production is appropriate when volatility is acceptable or for use cases involving reproducible images. Aside from those specific cases, you need data durability, and so you'll need a durable storage back end.

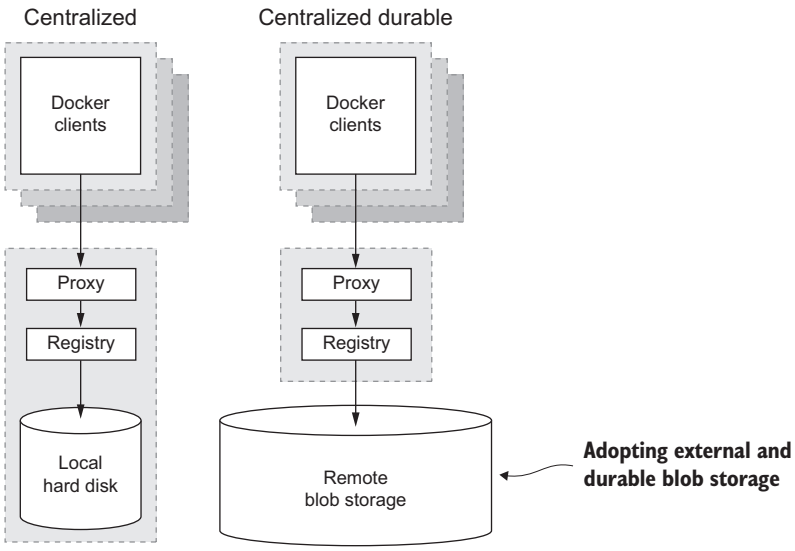


Figure 10.6 How to improve the durability of a centralized registry

10.3 Durable blob storage

Blob is short for *binary large object*. A registry deals in image layers (blobs) and metadata. The term is relevant because there are several popular blob storage services and projects. To adopt durable blob storage, you'll need to make a change to your registry's configuration file and build a new image. Figure 10.6 shows how to grow from a centralized to a durable centralized registry.

The Distribution project currently supports popular blob storage back ends in addition to the local file system. The section of the configuration file that deals with storage is appropriately named `storage`. That mapping has four conflicting properties that define the storage back end. Only one of these properties should be present in a valid registry configuration:

- `filesystem`
- `azure`
- `s3`
- `rados`

The `filesystem` property is used by the default configuration and has only a single property, `rootdirectory`, that specifies the base directory to use for local storage. For example, the following is a sample from the default configuration:

```
storage:
    filesystem:
        rootdirectory: /var/lib/registry
```

The other storage properties configure the Distribution project for integration with various distributed blob storage services and are covered over the remainder of this section, starting with Azure.

10.3.1 Hosted remote storage with Microsoft's Azure

Azure is the name of Microsoft's cloud services product family. One service in that family is a blob storage service named Storage. If you have an Azure account, you can use the Storage service for your registry blob storage. You can learn more about the service on the website: http://azure.microsoft.com/services/storage/.

In order to adopt Azure for your blob storage, you need to use the `azure` property and set three subproperties: `accountname`, `accountkey`, and `container`. In this context, `container` refers to an Azure Storage container, not a Linux container.

A minimal Azure configuration file might be named azure-config.yml and include the following configuration:

```
# Filename: azure-config.yml
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    azure:                                                    <-- Azure-specific fields
        accountname: <your account name>
        accountkey: <your base64 encoded account key>
        container: <your container>
        realm: core.windows.net
    cache:
        layerinfo: inmemory
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

Replace the text in angle brackets with configuration for your account. The `realm` property should be set to the realm where you want to store your images. See the official Azure Storage documentation for details. `realm` is not a required property and will default to `core.windows.net`.

You can pack the new configuration into a layer over the original registry image with the following Dockerfile. You could name it azure-config.df:

```
# Filename: azure-config.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
```

```
# Set the default argument to specify the config file to use
# Setting it early will enable layer caching if the
# azure-config.yml changes.
CMD ["/azure-config.yml"]
COPY ["./azure-config.yml","/azure-config.yml"]
```

And you can build it with the following `docker build` command:

```
docker build -t dockerinaction/azure-registry -f azure-config.df .
```

With an Azure Storage back end, you can build a durable registry that scales in terms of expense instead of technical complexity. That trade-off is one of the strong points of hosted remote blob storage. If you're interested in using a hosted solution, you might consider the more mature AWS Simple Storage Service.

10.3.2 Hosted remote storage with Amazon's Simple Storage Service

Simple Storage Service (S3) from AWS offers several features in addition to blob storage. You can configure blobs to be encrypted at rest, versioned, access audited, or made available through AWS's content delivery network (see section 10.4.2).

Use the `s3` storage property to adopt S3 as your hosted remote blob store. There are four required subproperties: `accesskey`, `secretkey`, `region`, and `bucket`. These are required to authenticate your account and set the location where blob reads and writes will happen. Other subproperties specify how the Distribution project should use the blob store. These include `encrypt`, `secure`, `v4auth`, `chunksize`, and `rootdirectory`.

Setting the `encrypt` property to `true` will enable data at rest encryption for the data your registry saves to S3. This is a free feature that enhances the security of your service.

The `secure` property controls the use of HTTPS for communication with S3. The default is `false` and will result in the use of HTTP. If you're storing private image material, you should set this to `true`.

The `v4auth` property tells the registry to use version 4 of the AWS authentication protocol. In general this should be set to `true` but defaults to `false`.

Files greater than 5 GB must be split into smaller files and reassembled on the service side in S3. But chunked uploads are available to files smaller than 5 GB and should be considered for files greater than 100 MB. File chunks can be uploaded in parallel, and individual chunk upload failures can be retired individually. The Distribution project and its S3 client perform file chunking automatically, but the `chunksize` property sets the size beyond which files should be chunked. The minimum chunk size is 5 MB.

Finally, the `rootdirectory` property sets the directory within your S3 bucket where the registry data should be rooted. This is helpful if you want to run multiple registries from the same bucket:

```
# Filename: s3-config.yml
version: 0.1
```

```
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    cache:
        layerinfo: inmemory
    s3:                                          <-- S3 configuration
        accesskey: <your awsaccesskey>
        secretkey: <your awssecretkey>
        region: <your bucket region>
        bucket: <your bucketname>
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

If you've created a configuration file named s3-config.yml and provided your account access key, secret, bucket name, and region, you can pack the updated registry configuration into a new image just like you did for Azure with the following Dockerfile:

```
# Filename: s3-config.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use
# Setting it early will enable layer caching if the
# s3-config.yml changes.
CMD ["/s3-config.yml"]
COPY ["./s3-config.yml","/s3-config.yml"]
```

And you can build it with the following `docker build` command:

```
docker build -t dockerinaction/s3-registry -f s3-config.df .
```

Both S3 and Azure are offered under a use-based cost model. There's no up-front cost to get started, and many smaller registries will be able to operate within the free tier of either service.

If you aren't interested in a hosted data service and don't hesitate in the face of some technical complexity, then you might alternatively consider running a Ceph storage cluster and the RADOS blob storage back end.

10.3.3 Internal remote storage with RADOS (Ceph)

Reliable Autonomic Distributed Object Store (RADOS) is provided by a software project named Ceph (http://ceph.com). Ceph is the software that you'd use to build your own Azure Storage or AWS S3-like distributed blob storage service. If you have a budget, time, and a bit of expertise, you can deploy your own Ceph cluster and save money over the long term. More than money, running your own blob storage puts you in control of your data.

If you decide to go that route, you can use the `rados` storage property to integrate with your own storage:

```
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    cache:
        layerinfo: inmemory
storage:
    rados:                          <-- RADOS configuration
        poolname: radospool
        username: radosuser
        chunksize: 4194304
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

The three subproperties are `poolname`, `username`, and `chunksize`. The `username` property should be self-explanatory, but `poolname` and `chunksize` are interesting.

Ceph stores blobs in pools. A *pool* is configured with certain redundancy, distribution, and behavior. The pool a blob is stored in will dictate how that blob is stored across your Ceph storage cluster. The `poolname` property tells Distribution which pool to use for blob storage.

Ceph chunks are similar to but not the same as S3 chunks. Chunks are important to Ceph's internal data representation. An overview of the Ceph architecture can be found here: http://ceph.com/docs/master/architecture/. The default chunk size is 4 MB, but if you need to override that value, you can do so with the `chunksize` property.

Adopting a distributed blob storage system is an important part of building a durable registry. If you intend on exposing your registry to the world, you will need enhancements for fast and scalable registries. The next section explains how to implement those enhancements.

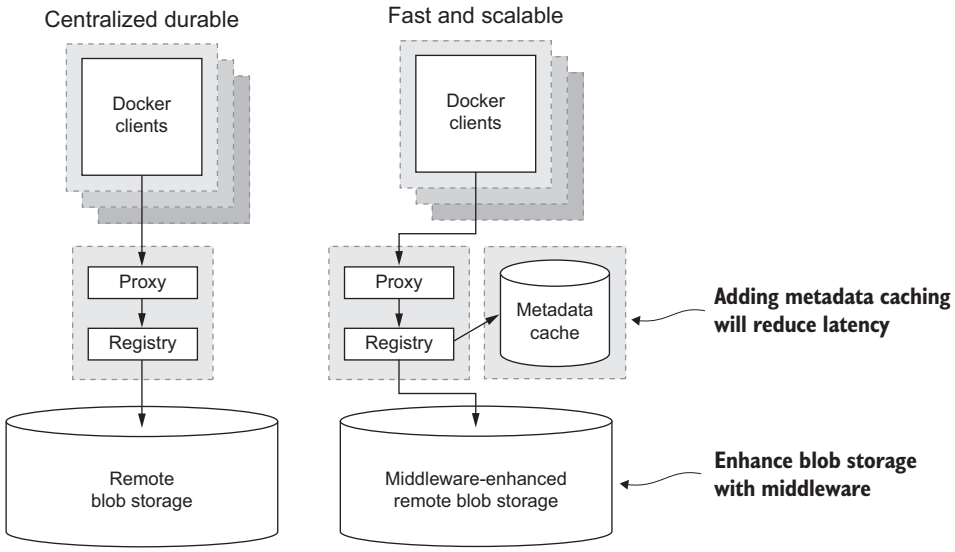


Figure 10.7 The fast and scalable architecture introduces a metadata cache and middleware.

10.4 Scaling access and latency improvements

With the reverse proxy and a durable storage back end in place, you should be able to scale your registry horizontally. But doing so introduces additional latency overhead. If you need to scale your registry to serve thousands of transactions per second, then you'll need to implement a caching strategy. You may even consider using a content delivery network like Amazon CloudFront.

As illustrated in figure 10.7, this section introduces two new components that will help you achieve low-latency response times.

Most readers won't buy additional machines for the sole purpose of running the exercises, so these examples will let you use separate containers instead of separate machines. If you're reading along while implementing your multi-machine registry and need information about linking software between hosts, consult chapter 5.

10.4.1 Integrating a metadata cache

When you need low-latency data retrieval, a cache is the first tool to reach for. The Distribution project can cache repository metadata using either an in-memory map or Redis. Redis (http://redis.io) is a popular, open source, key-value cache and data-structure server.

The in-memory map option is appropriate for smaller registries or development purposes, but using a dedicated caching project like Redis will help improve the reliability of your cache and reduce average latencies.

The Distribution configuration for metadata caching is set with the `cache` subproperty of the `storage` property. Cache has one subproperty named `blobdescriptor` with two potential values, `inmemory` and `redis`. If you're using `inmemory`, then setting that value is the only configuration required, but if you're using Redis, you need to provide additional connection pool configuration.

The top-level `redis` property has only one requisite subproperty, `addr`. The `addr` property specifies the location of the Redis server to use for the cache. The server can be running on the same computer or a different one, but if you use the localhost name here, it must be running in the same container or another container with a joined network. Using a known host alias gives you the flexibility to delegate that linkage to a runtime configuration. In the following configuration sample, the registry will attempt to connect to a Redis server at redis-host on port 6379:

```
# Filename: redis-config.yml
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
storage:
    cache:                                  <-- Cache configuration
        blobdescriptor: redis
    s3:
        accesskey: <your awsaccesskey>
        secretkey: <your awssecretkey>
        region: <your bucket region>
        bucket: <your bucketname>
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
redis:                                      <-- Redis-specific details
    addr: redis-host:6379
    password: asecret
    dialtimeout: 10ms
    readtimeout: 10ms
    writetimeout: 10ms
    pool:
        maxidle: 16
        maxactive: 64
        idletimeout: 300s
```

The `redis` configuration in this example sets several optional properties. The `password` property defines the password that will be passed to the Redis `AUTH` command on connection. The `dialtimeout`, `readtimeout`, and `writetimeout` properties specify timeout values for connecting to, reading from, or writing to the Redis server. The last property, `pool`, has three subproperties that define the attributes for the connection pool.

The minimum pool size is specified with the `maxidle` property. The maximum pool size is set with the `maxactive` property. Finally, the time from the last use of an active connection to the moment it's a candidate to be flipped into an idle state is specified with the `idletimeout` property. Any connections that are flipped to idle when the current number of idle connections has reached the maximum will be closed.

Use dummy values in place of secrets in order to produce environment-agnostic images. Properties like `password` should be overridden at runtime using environment variables.

The cache configuration will help reduce latency associated with serving registry metadata, but serving image blobs remains inefficient. By integrating with remote blob storage like S3, a registry becomes a streaming bottleneck during image transfer. Streaming connections are tricky because the connections tend to be long-lived relative to metadata queries. Things are made even trickier when long-lived connections are made through the same load-balancing infrastructure as short-lived connections.

You can try out this configuration yourself by building a registry with this configuration and linking in a Redis container:

```
docker run -d --name redis redis
docker build -t dockerinaction/redis-registry -f redis-config.df .
docker run -d --name redis-registry \
    --link redis:redis-host -p 5001:5000 \
    dockerinaction/redis-registry
```

The next section explains how you can use a content delivery network (CDN) and registry middleware to streamline blob downloads.

10.4.2 Streamline blob transfer with storage middleware

Middleware, in the Distribution project, acts like advice or decorators for a registry, repository, or storage driver. At present, Distribution ships with a single storage middleware. It integrates your registry and S3 storage back end with AWS CloudFront. CloudFront is a content delivery network.

CDNs are designed for geographically aware network access to files. This makes CloudFront a perfect solution to the remaining scale issue caused by adopting durable and distributed blob storage.

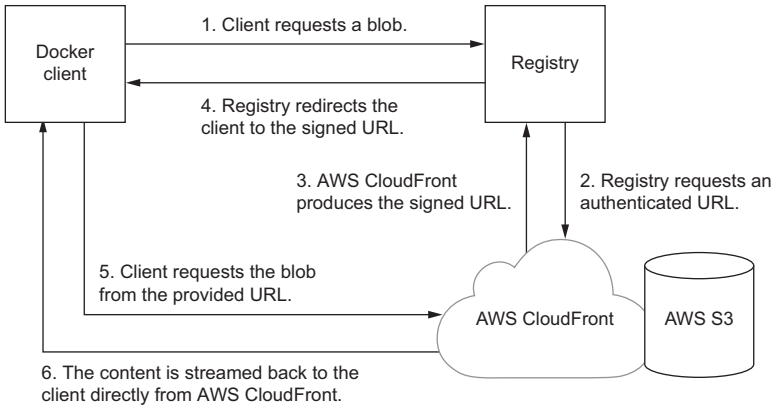


Figure 10.8 Offloading streaming blob traffic with AWS CloudFront storage middleware

Downloading a file is streamlined with the CloudFront middleware enabled and S3 in use for your storage back end. Figure 10.8 illustrates how data flows through the integrated configuration.

Rather than streaming blobs from S3 back to your registry and subsequently back to the requesting client, integration with CloudFront lets you redirect clients to authenticated CloudFront URLs directly. This eliminates network overhead associated with image download for your local network. It also offloads long-lived connections to the appropriately designed CloudFront service.

Enabling the CloudFront middleware is as simple as adding the appropriate configuration. The following sample is complete with S3, Redis, and CloudFront:

```
# Filename: scalable-config.conf
version: 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
storage:
    cache:
        blobdescriptor: redis
    s3:
        accesskey: <your awsaccesskey>
        secretkey: <your awssecretkey>
        region: <your bucket region>
        bucket: <your bucketname>
```

```
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
redis:
    addr: redis-host:6379
    password: asecret
    dialtimeout: 10ms
    readtimeout: 10ms
    writetimeout: 10ms
    pool:
        maxidle: 16
        maxactive: 64
        idletimeout: 300s
middleware:                                  <- Middleware configuration
    storage:
        - name: cloudfront
          options:
              baseurl: <https://my.cloudfronted.domain.com/>
              privatekey: </path/to/pem>
              keypairid: <cloudfrontkeypairid>
              duration: 3000
```

The `middleware` property and `storage` subproperty are a bit different from other configurations you've seen so far. The `storage` subproperty contains a list of named storage middleware, each with its own set of options specific to the middleware.

In this sample, you're using the middleware named `cloudfront` and setting its `baseurl`, `privatekey` path, `keypairid` name, and the `duration` over which the authenticated URLs are valid. Consult the CloudFront user documentation (http://aws.amazon.com/cloudfront) for the correct settings for your account.

Once you've added a configuration specific to your AWS account and CloudFront distribution, you can bundle the configuration with a Dockerfile and deploy any number of high-performance registry containers. With the proper hardware and configuration you could scale to thousands of image pulls or pushes per second.

All that activity generates useful data. A component like your registry should be integrated with the rest of your systems to react to events or centrally collect data. The Distribution project makes these integrations possible with notifications.

10.5 Integrating through notifications

Launching your own registry can help you build your own distribution infrastructure, but to do so you need to integrate it with other tools. Notifications are a simple webhook-style integration tool. When you provide an endpoint definition in the registry configuration file, the registry will make an HTTP request and upload a

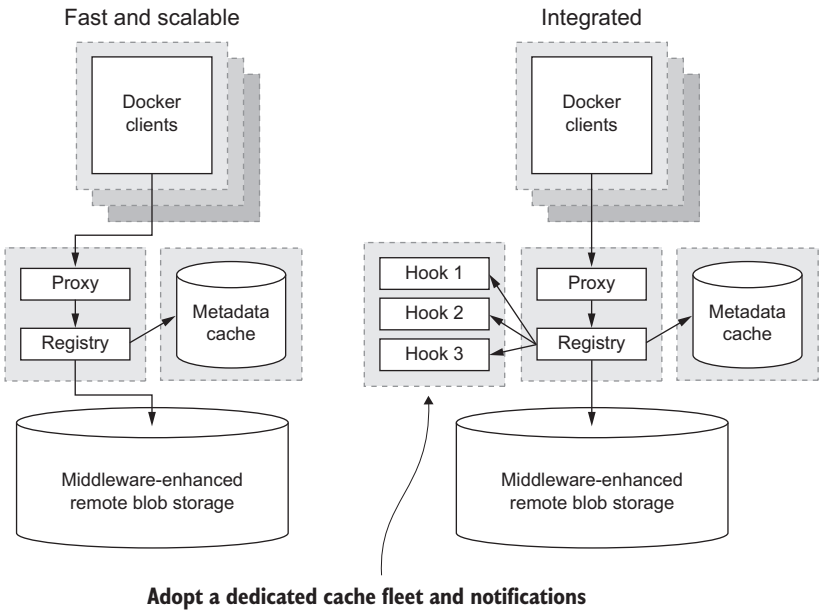


Figure 10.9 Notifications, reporting, and a dedicated metadata cache

JSON-encoded event for each push or pull event on the registry. Figure 10.9 shows how notifications integrate your system architecture.

When Distribute is configured to send notifications, any valid push or pull event triggers the delivery of JSON documents describing the event to configured endpoints. This is the primary integration mechanism for the Distribute project.

Notifications can be used to collect usage metrics, trigger deployments, trigger image rebuilds, send email, or do anything else you can think of. You might use a notification integration to post messages to your IRC channel, regenerate user documentation, or trigger an indexing service to scan the repository. In this example, the last of the chapter, you'll integrate the Distribution project with the Elasticsearch project (https://github.com/elastic/elasticsearch) and a web interface to create a fully searchable database of registry events.

Elasticsearch is a scalable document index and database. It provides all the functionality required to run your own search engine. Calaca is a popular open source web interface for Elasticsearch. In this example, you'll run each of these in its own container, a small pump implemented with Node.js, and a Distribution registry configured to send notifications to the pump. Figure 10.10 shows the integration you will build in this example.

To build this system, you'll use the official Elasticsearch image from Docker Hub and two images provided for this example. All this material is open source, so if you're

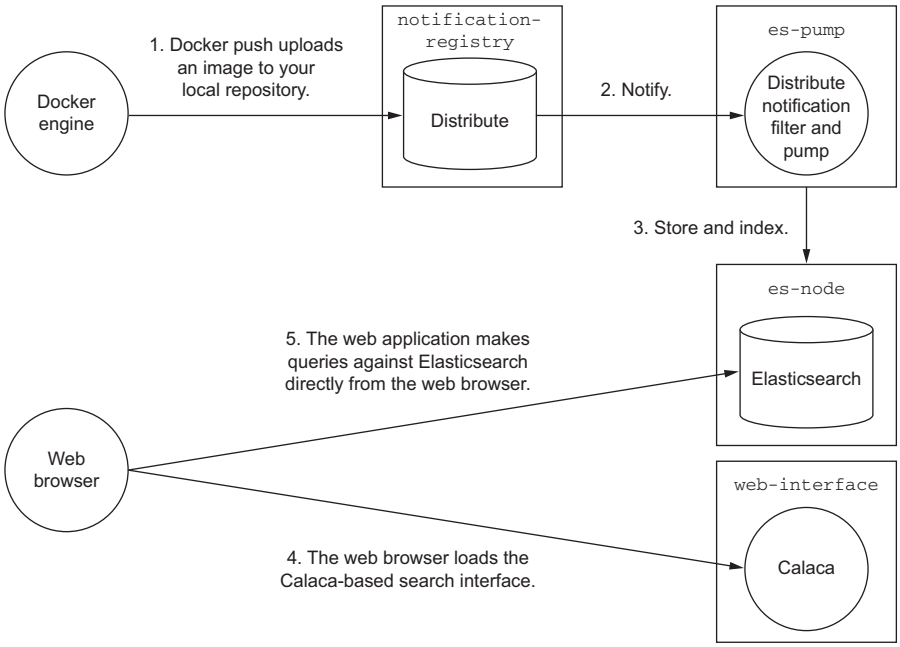


Figure 10.10 Integrating Distribute with Elasticsearch

interested in the mechanics at work, please inspect the repositories. Details of the integration and Distribution configuration will be covered here. Prepare for the example by pulling the required images from Docker Hub:

```
docker pull elasticsearch:1.6
docker pull dockerinaction/ch10_calaca
docker pull dockerinaction/ch10_pump
```

Briefly, the `dockerinaction/ch10_calaca` image contains a basic Calaca release that has been configured to use an Elasticsearch node running on localhost. The name is important in this case to comply with cross-origin resource sharing (CORS) rules. The `dockerinaction/ch10_pump` image contains a small Node.js service. The service listens for notifications and forwards notifications that contain pull or push actions on repository manifests. This represents a small subset of the types of notifications sent by the registry.

Every valid action on a registry results in a notification, including the following:

- Repository manifest uploads and downloads
- Blob metadata requests, uploads, and downloads

Notifications are delivered as JSON objects. Each notification contains a list of events. Each event contains properties that describe the event. The following stub shows the available properties:

```
{ "events": [{
  "id": "921a9db6-1703-4fe4-9dda-ea71ad0014f1",
  "timestamp": ...
  "action": "push",
  "target": {
    "mediaType": ...
    "length": ...
    "digest": ...
    "repository": ...
    "url": ...
  },
  "request": {
    "id": ...
    "addr": ...
    "host": ...
    "method": ...
    "useragent": ...
  },
  "actor": {},
  "source": {
    "addr": ...
    "instanceID": ...
  }
}]}
```

The service in `dockerinaction/ch10_pump` inspects each element in the event list and then forwards appropriate events to the Elasticsearch node.

Now that you have an idea of how the pump works, start up the Elasticsearch and pump containers:

```
docker run -d --name elasticsearch -p 9200:9200 \
    elasticsearch:1.6 -Des.http.cors.enabled=true

docker run -d --name es-pump -p 8000 \
    --link elasticsearch:esnode \
    dockerinaction/ch10_pump
```

Containers created from the Elasticsearch image can be customized without creating a whole image by passing environment variables to the Elasticsearch program itself. In the previous command, you enable CORS headers so that you can integrate this container with Calaca. With these components in place, any number of Distribution instances might send notifications to the `es-pump` container. All the relevant data will be stored in Elasticsearch.

Next, create a container to run the Calaca web interface:

```
docker run -d --name calaca -p 3000:3000 \
    dockerinaction/ch10_calaca
```

Notice that the container running Calaca doesn't need a link to the Elasticsearch container. Calaca uses a direct connection from your web browser to the Elasticsearch node. In this case, the provided image is configured to use the Elasticsearch node running on localhost. If you're running VirtualBox, the next step can be tricky.

VirtualBox users have not technically bound their `elasticsearch` container's port to localhost. Instead it's bound to the IP address of their VirtualBox virtual machine. You can solve this problem with the VBoxManage program included with VirtualBox. Use the program to create port-forwarding rules between your host network and the default virtual machine. You can create the rules you need with two commands:

```
VBoxManage controlvm "$(docker-machine active)" natpf1 \
    "tcp-port9200,tcp,,9200,,9200"
VBoxManage controlvm "$(docker-machine active)" natpf1 \
    "tcp-port3000,tcp,,3000,,3000"
```

These commands create two rules: forward port 9200 on localhost to port 9200 of the default virtual machine, and do the same for port 3000. Now VirtualBox users can interact with these ports in the same way that native Docker users can.

At this point, you should be ready to configure and launch a new registry. For this example, start from the default registry configuration and simply add a `notifications` section. Create a new file and copy in the following configuration:

```
# Filename: hooks-config.yml
version: 0.1
log:
    level: debug
    formatter: text
    fields:
        service: registry
        environment: staging
storage:
    filesystem:
        rootdirectory: /var/lib/registry
    maintenance:
        uploadpurging:
            enabled: true
            age: 168h
            interval: 24h
            dryrun: false
http:
    addr: 0.0.0.0:5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
notifications:
    endpoints:
        - name: webhookmonitor
          disabled: false
          url: http://webhookmonitor:8000/
          timeout: 500
          threshold: 5
          backoff: 1000
```

Notification configuration

The last section, `notifications`, specifies a list of endpoints to notify. The configuration of each endpoint includes a name, URL, attempt timeout, attempt threshold, and backoff time. You can also disable individual endpoints without removing the configuration by setting the `disabled` attribute to `false`. In this case, you've defined a single endpoint at webhookmonitor on port 8000. If you were deploying this to a distributed environment, webhookmonitor might be set to resolve to a different host. In this example, webhookmonitor is an alias for the container running the pump.

Once you've saved the configuration file, you can start the new registry container and see notifications in action. The following command will create a registry. It uses the base image, injects the configuration using a bind-mount volume, and sets the configuration file to use with the last argument. The command creates a link to the pump and assigns it to the webhookmonitor alias. Finally, it binds the registry to port 5555 on localhost (or the Boot2Docker IP address):

```
docker run -d --name ch10-hooks-registry -p 5555:5000 \
    --link es-pump:webhookmonitor \
    -v "$(pwd)"/hooks-config.yml:/hooks-config.yml \
    registry:2 /hooks-config.yml
```

With that last component running, you're ready to test the system. Test the Calaca container first. Open a web browser and navigate to http://localhost:3000/. When you do, you should see a simple web page titled calaca and a large search box. Nothing that you search for will have any results at this point because no notifications would have been sent to Elasticsearch yet. Push and pull images from your repository to see Elasticsearch and notifications at work.

Tag and push one of your existing images into your new registry to trigger notifications. You might consider using the `dockerinaction/curl` image that you created earlier in the chapter. It's a small image, and the test will be fast:

```
docker tag dockerinaction/curl localhost:5555/dockerinaction/curl
docker push localhost:5555/dockerinaction/curl
```

If you named the `cURL` image something different, you'll need to use that name instead of the one provided here. Otherwise, you should be ready to search. Head back to Calaca in your web browser and type `curl` in the search box.

As you finish typing `curl`, a single search result should appear. The results listed here correspond to registry events. Any valid push or pull will trigger the creation of one such event in Elasticsearch. Calaca has been configured to display the repository name, the event type, the timestamp on the event, and finally the raw notification. If you push the same repository again, then there should be two events. If you pull the repository instead, there will be a third event for the `curl` search term, but the type will be `pull`. Try it yourself:

```
docker pull localhost:5555/dockerinaction/curl
```

The raw notifications are included in the search results to help you get creative with searching the events. Elasticsearch indexes the whole document, so any field on the event is a potential search term. Try using this example to build interesting queries:

- Search for `pull` or `push` to see all pull or push events.
- Search for a particular repository prefix to get a list of all events for that prefix.
- Track activity on specific image digests.
- Discover clients by requesting an IP address.
- Discover all repositories accessed by a client.

This long example should reinforce the potential of a Distribution-based registry as a key component in your release or deployment workflow. The example should also serve as a reminder of how Docker can reduce the barrier to working with any containerized technology.

The most complex part of setting up this example was creating containers, linking containers, and injecting configuration with volumes. In chapter 11 you'll learn how to set up and iterate on this example using a simple YAML file and a single command.

10.6 Summary

This chapter dives deep into building Docker registries from the Distribution project. This information is important both for readers who intend to deploy their own registries and for readers who want to develop a deeper understanding of the primary image distribution channel. The specific material covered is summarized in the following points:

- A Docker registry is defined by the API it exposes. The Distribution project is an open source implementation of the Registry API v2.
- Running your own registry is as simple as starting a container from the `registry:2` image.
- The Distribution project is configured with a YAML file.
- Implementing a centralized registry with several clients typically requires implementing a reverse proxy, adopting TLS, and adding an authentication mechanism.
- Authentication can be offloaded to a reverse proxy or implemented by the registry itself.
- Although other authentication mechanisms are available, HTTP basic authentication is the simplest to configure and the most popular.
- Reverse proxy layers can help ease Registry API changes for mixed client versions.
- Inject secrets in production configurations with bind-mount volumes and environment variable configuration overrides. Do not commit secrets to images.
- Centralized registries should consider adopting remote blob storage like Azure, S3, or Ceph.

- Distribution can be configured to scale by creating a metadata cache (Redis-based) or adopting the Amazon Web Services CloudFront storage middleware.
- It's simple to integrate Distribution with the rest of your deployment, distribution, and data infrastructure using notifications.
- Notifications push event data to configured endpoints in JSON form.

Part 3

Multi-Container and Multi-Host Environments

If part 1 is focused on the isolation provided by containers, this part is focused on their composition. Most valuable systems are composed of two or more components. Simple management of multiple components is more important than ever due to the rise of large-scale server software, service-oriented-architectures, microservices, and now the Internet-of-Things. In building these systems, we've adopted tools like cluster computing, orchestration, and service discovery. These are difficult and nuanced problems in themselves. This final part of *Docker in Action* will introduce three other Docker tools and demonstrate how to use Docker in the wild.

11 Declarative environments with Docker Compose

This chapter covers

- Using Docker Compose
- Manipulating environments and iterating on projects
- Scaling services and cleaning up
- Building declarative environments

Have you ever joined a team with an existing project and struggled to get your development environment set up or IDE configured? If someone asked you to provision a test environment for their project, could you enumerate all the questions you'd need to ask to get the job done? Can you imagine how painful it is for development teams and system administrators to resynchronize when environments change? All of these are common and high-effort tasks. They can be time-intensive while adding little value to a project. In the worst case, they give rise to policies or procedures that limit developer flexibility, slow the iteration cycle, and bring paths of least resistance to the forefront of technical decision making.

This chapter introduces you to Docker Compose (also called Compose) and how you can use it to solve these common problems.

11.1 Docker Compose: up and running on day one

Compose is a tool for defining, launching, and managing services, where a *service* is defined as one or more replicas of a Docker container. Services and systems of services are defined in YAML files (http://yaml.org) and managed with the command-line program `docker-compose`. With Compose you can use simple commands to accomplish these tasks:

- Build Docker images
- Launch containerized applications as services
- Launch full systems of services
- Manage the state of individual services in a system
- Scale services up or down
- View logs for the collection of containers making a service

Compose lets you stop focusing on individual containers and instead describe full environments and service component interactions. A Compose file might describe four or five unique services that are interrelated but should maintain isolation and may scale independently. This level of interaction covers most of the everyday use cases for system management. For that reason, most interactions with Docker will be through Compose.

By this point you've almost certainly installed Docker, but you may not have installed Compose. You can find up-to-date installation instructions for your environment at https://docs.docker.com/compose/install/. Official support for Windows has not been implemented at the time of this writing. But many users have successfully installed Compose on Windows through pip (a Python package manager). Check the official site for up-to-date information. You may be pleasantly surprised to find that Compose is a single binary file and that installation instructions are quite simple. Take the time to install Compose now.

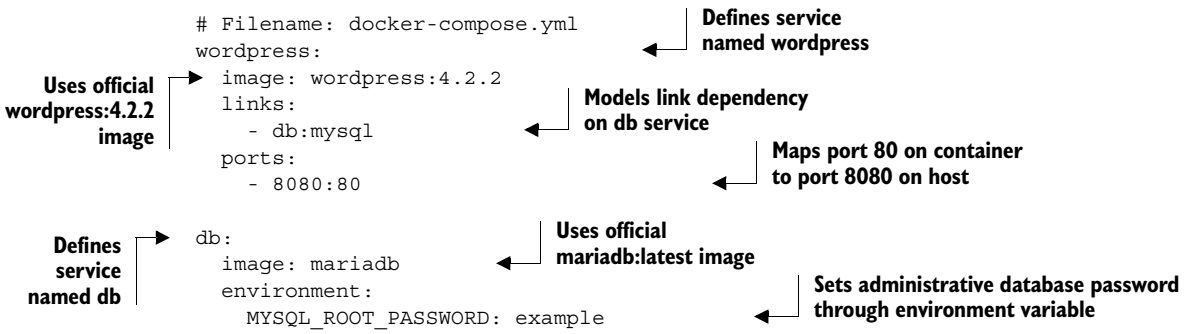
The best way to develop an appreciation for any tool is to use it. The rest of this section will get you started with a few situational examples.

11.1.1 Onboarding with a simple development environment

Suppose you've started a new job as a software developer with a forward-looking team that owns a mature project. If you've been in a similar situation before, you may anticipate that you're going to spend a few days installing and configuring your IDE and getting an operational development environment running on your workstation. But on your first day at this job, your peers give you three simple instructions to get started:

1. Install Docker.
2. Install Docker Compose.
3. Install and use Git to clone the development environment.

Rather than ask you to clone a development environment here, I'll have you create a new directory named wp-example and copy the following docker-compose.yml file into that directory:




```
# Filename: docker-compose.yml
wordpress:
  image: wordpress:4.2.2
  links:
    - db:mysql
  ports:
    - 8080:80

db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

As you may be able to tell from examining the file, you're going to launch a WordPress service and an independent database. This is an iteration of a basic example from chapter 2. Change to the directory where you created the docker-compose.yml file and start it all up with the following command:

```
docker-compose up
```

This should result in output similar to the following:

```
Creating wpexample_db_1...
Creating wpexample_wordpress_1...
...
```

You should be able to open http://localhost:8080/ (or replace "localhost" with your virtual machine's IP address) in a web browser and discover a fresh WordPress installation. This example is fairly simple but does describe a multi-service architecture. Imagine a typical three- or four-tier web application that consists of a web server, application code, a database, and maybe a cache. Launching a local copy of such an environment might typically take a few days—longer if the person doing the work is less familiar with some of the components. With Compose, it's as simple as acquiring the docker-compose.yml file and running `docker-compose up`.

When you've finished having fun with your WordPress instance, you should clean up. You can shut down the whole environment by pressing Ctrl-C (or Control-C). Before you remove all the containers that were created, take a moment to list them with both the `docker` and `docker-compose` commands:

```
docker ps
docker-compose ps
```

Using `docker` displays a list of two (or more) containers in the standard fashion. But listing the containers with `docker-compose` includes only the list of containers that are defined by the docker-compose.yml in the current directory. This form is more

refined and succinct. Filtering the list in this way also helps you focus on the containers that make up the environment you're currently working on. Before moving on, take the time to clean up your environment.

Compose has an `rm` command that's very similar to the `docker rm` command. The difference is that `docker-compose rm` will remove all services or a specific service defined by the environment. Another minor difference is that the `-f` option doesn't force the removal of running containers. Instead, it suppresses a verification stage.

So, the first step in cleaning up is to stop the environment. You can use either `docker-compose stop` or `docker-compose kill` for this purpose. Using `stop` is preferred to `kill` for reasons explained in part 1. Like other Compose commands, these can be passed a service name to target for shutdown.

Once you've stopped the services, clean up with the `docker-compose rm` command. Remember, if you omit the `-v` option, volumes may become orphaned:

```
docker-compose rm -v
```

Compose will display a list of the containers that are going to be removed and prompt you for verification. Press the Y key to proceed. With the removal of these containers, you're ready to learn how Compose manages state and tips for avoiding orphan services while iterating.

This WordPress sample is trivial. Next, you'll see how you might use Compose to model a much more complicated environment.

11.1.2 A complicated architecture: distribution and Elasticsearch integration

At the end of chapter 10, you create a much more complicated example. You launch four related components that together provide a Docker registry that's configured to pump event data into an Elasticsearch instance and provide a web interface for searching those events. See figure 11.1.

Setting up the example required image builds and careful accounting while linking containers together. You can quickly re-create the example by cloning an existing environment from version control and launching it with Compose:

```
git clone https://github.com/dockerinaction/ch11_notifications.git
cd ch11_notifications
docker-compose up -d
```

When you run the last command, Docker will spring to life building various images and starting containers. It differs from the first example in that you use the `-d` option. This option launches the containers in detached mode. It operates exactly like the `-d` option on the `docker run` command. When the containers are detached, the log output of each of the containers will not be streamed to your terminal.

If you need to access that data, you could use the `docker logs` command for a specific container, but that does not scale well if you're running several containers. Instead, use the `docker-compose logs` command to get the aggregated log stream for