Image Registries

Glyn Normington

May 29, 2019

This document provides a formal model of some aspects of *Docker* image registries.

I am indebted to Chris Frost and Steve Powell for their helpful feedback.

Contents

1 Introduction

This document provides a formal model of image registries.

Docker Inc. [1][1] introduced *container images* and *registries* to hold them and these were later standardised as part of the Open Container Initiative [2]. The reader is assumed to have a basic understanding of how images are used.

This document models image references, repositories, and registries. It covers digests and tags.

The Z specification language is used to describe the model, but sufficient English text is also provided that readers who do not know Z should be able to understand what's going on. Appendix B contains a summary of the Z notation. For more information about Z, please consult the Z Manual [3]. The model was type checked using `fuzz` [4].

2 Fundamentals

Images are opaque blobs as far as we are concerned here - the decomposition into layers is ignored. Similarly, cryptographic hashes (or *hexes*, to use the terminology of the OCI Distribution specification), tags, and (registry) hostnames and paths are modelled in terms of their usage, but not their implementation.

$$[Image, Hex, Tag, Hostname, Path]$$

There is a special reserved tag.

$$| \; Latest : Tag$$

[1]See the bibliography in Appendix A.

3 Content Digests

A content digest is a combination of a cryptographic hash function (such as `SHA-256`) or "algorithm", and the hash output by such a function.

```
┌─ ContentDigest ─────────────────────────
│ alg : Image → Hex
│ hash : Hex
└─────────────────────────────────────────
```

The algorithm in a content digest is represented as a string which names one of a limited set of algorithms, but we ignore that here and pretend the algorithm is part of the content digest.

The idea is that a content digest d identifies an image i if and only if applying the hash function (from the content digest) to the image produces the hash output in the content digest:

$$d.alg\ i = d.hash$$

The limited set of algorithms is chosen so that given an image i with a content digest d, it is extremely difficult, if not impossible, to construct a distinct image j with the same content digest:

$$i \neq j \land d.alg\ i = d.hash = d.alg\ j$$

An optional content digest is modelled as a datatype.

$$OptionalContentDigest ::= None \mid Dig\langle\langle ContentDigest\rangle\rangle$$

We add a convenient function for extracting the content digest from an optional content digest which is "present".

$$Cd == Dig^{\sim}$$

4 Repositories

A repository is a collection of images indexed by content digest and by tag.

$$
\begin{array}{|l}
\textit{Repo} \\
\hline
\mathit{cd} : \mathit{ContentDigest} \pinj \mathit{Image} \\
\mathit{tag} : \mathit{Tag} \pfun \mathit{Image} \\
\hline
\forall d : \operatorname{dom} \mathit{cd} \bullet d.\mathit{alg}\,(\mathit{cd}\ d) = d.\mathit{hash} \\
\operatorname{ran} \mathit{tag} \subseteq \operatorname{ran} \mathit{cd} \\
\hline
\end{array}
$$

The content digests identify the corresponding images and there is only one content digest per image (since this is chosen when the image added to the repository, as we shall soon see). Each image identified by a tag is also identified by a content digest.

Initially, a repository is empty.

$$
\begin{array}{|l}
\textit{RepoInit} \\
\hline
\mathit{Repo}' \\
\hline
\mathit{cd}' = \varnothing \\
\mathit{tag}' = \varnothing \\
\hline
\end{array}
$$

An image is added to a repository by *pushing* it. The operation takes any image and any tag and returns a content digest of the image.

$$
\begin{array}{|l}
\textit{RepoPush} \\
\hline
\Delta \mathit{Repo} \\
i? : \mathit{Image} \\
t? : \mathit{Tag} \\
d! : \mathit{ContentDigest} \\
\hline
\mathit{tag}' = \mathit{tag} \oplus \{t? \mapsto i?\} \\
\mathit{cd}' = \mathit{cd} \oplus \{d! \mapsto i?\} \\
\hline
\end{array}
$$

The tag may be omitted, in which case it defaults to *Latest*, although this is not modelled. Note that the invariant of *Repo′* ensures that the output digest identifies the input image. There is, in theory, some non-determinism in the choice of algorithm, but for OCI compliant implementations, `SHA-256` is used everywhere.

An image is retrieved from a repository by *pulling* it.

We can pull an image using a content digest.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RepoPullByDigest}} \\
\Xi \mathit{Repo} \\
d? : \mathit{OptionalContentDigest} \\
i! : \mathit{Image} \\
\hline
d? \in \operatorname{dom} \mathit{Cd} \\
i! = \mathit{cd}(\mathit{Cd}\ d?) \\
\hline
\end{array}
$$

The content digest is provided and is used to identify the output image.

Alternatively, we can pull and image using a tag.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RepoPullByTag}} \\
\Xi \mathit{Repo} \\
t? : \mathit{Tag} \\
d? : \mathit{OptionalContentDigest} \\
i! : \mathit{Image} \\
\hline
d? = \mathit{None} \\
i! = \mathit{tag}\ t? \\
\hline
\end{array}
$$

The content digest is not provided. The tag is used to identify the output image.

A successful pull operation uses either a content digest or a tag.

$$\mathit{RepoPullOk} \mathrel{\widehat{=}} \mathit{RepoPullByDigest} \lor \mathit{RepoPullByTag}$$

If a content digest is provided, the image is pulled by digest and the tag is ignored. Failure cases, such as “not found”, are omitted from the model.

5 Registries

A registry is a collection of repositories indexed by path.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Registry}} \\
\hline
\textit{repo} : \textit{Path} \mathrel{+\!\!\!\!\rightarrow} \textit{Repo} \\
\hline
\end{array}$$

An empty registry has no repositories.

$$\textit{EmptyRegistry} \mathrel{\widehat{=}} [\textit{Registry} \mid \textit{repo} = \varnothing]$$

Initially a registry has no repositories.

$$\textit{RegistryInit} \mathrel{\widehat{=}} \textit{EmptyRegistry}'$$

An empty repository may be added at a given path.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RegistryAddRepoOk}} \\
\hline
\Delta \textit{Registry} \\
p? : \textit{Path} \\
\hline
p? \notin \operatorname{dom} \textit{repo} \\
\exists\, \textit{RepoInit} \bullet \textit{repo}' = \textit{repo} \cup \{p? \mapsto \theta \textit{Repo}'\} \\
\hline
\end{array}$$

Some registry implementations, such as Amazon's ECR, require a repository to be created before an image can be pushed to the repository. Other registry implementations such as Docker's registry and Google's GCR will create a repository automatically when the first image is pushed to it. We do not model automatic repository creation.

We define a promotion schema[2] which operates on a single repository in a registry.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{RegistryPromote}} \\ \hline
\Delta \mathit{Registry} \\
\Delta \mathit{Repo} \\
p? : \mathit{Path} \\ \hline
p? \in \operatorname{dom} \mathit{repo} \\
\theta \mathit{Repo} = \mathit{repo}\ p? \\
\mathit{repo}' = \mathit{repo} \oplus \{p? \mapsto \theta \mathit{Repo}'\} \\ \hline
\end{array}$$

The path must be valid and the registry is preserved except for the repository identified by the path which may be updated.

We then promote the repository push and pull operations to operate on a registry.

$$\begin{array}{l}
\mathit{RegistryPush} \mathrel{\widehat=} \exists\, \Delta \mathit{Repo} \bullet \mathit{RepoPush} \land \mathit{RegistryPromote} \\
\mathit{RegistryPullOk} \mathrel{\widehat=} \exists\, \Delta \mathit{Repo} \bullet \mathit{RepoPullOk} \land \mathit{RegistryPromote}
\end{array}$$

Registries are accessed remotely in a network via their hostnames.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Net}} \\ \hline
\mathit{reg} : \mathit{Hostname} \pfun \mathit{Registry} \\ \hline
\end{array}$$

Initiallly, there are no registries in the network.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NetInit}} \\ \hline
\mathit{Net}' \\ \hline
\mathit{reg}' = \varnothing \\ \hline
\end{array}$$

We can add an empty registry to the network.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NetAddRegistryOk}} \\ \hline
\Delta \mathit{Net} \\
h? : \mathit{Hostname} \\ \hline
h? \notin \operatorname{dom} \mathit{reg} \\
\exists\, \mathit{EmptyRegistry} \bullet \mathit{reg}' = \mathit{reg} \cup \{h? \mapsto \theta \mathit{EmptyRegistry}\} \\ \hline
\end{array}$$

[2]Promotion schemas are used to turn operations on a particular type into operations on a collection of the type which operate on a single member of the collection and leave the rest unchanged.

We can also remove a registry from the network.

$$
\begin{array}{|l}
\hline
\textit{NetRemoveRegistryOk} \\
\Delta \textit{Net} \\
h? : \textit{Hostname} \\
\hline
h? \in \operatorname{dom} \textit{reg} \\
\textit{reg}' = \{h?\} \mathbin{\lhd\!\!\!-} \textit{reg} \\
\hline
\end{array}
$$

We define a promotion schema which operates on a single registry in a network.

$$
\begin{array}{|l}
\hline
\textit{NetPromote} \\
\Delta \textit{Net} \\
\Delta \textit{Registry} \\
h? : \textit{Hostname} \\
\hline
h? \in \operatorname{dom} \textit{reg} \\
\theta \textit{Registry} = \textit{reg}\ h? \\
\textit{reg}' = \textit{reg} \oplus \{h? \mapsto \theta \textit{Registry}'\} \\
\hline
\end{array}
$$

The hostname must be valid and the network is preserved except for the registry identified by the hostname which may be updated.

Finally, we promote the push and pull operations to work on a network.

$$
\begin{aligned}
&\textit{NetPushOk} \mathrel{\widehat=} \exists\, \Delta \textit{Registry} \bullet \textit{RegistryPush} \land \textit{NetPromote} \\
&\textit{NetPullOk} \mathrel{\widehat=} \exists\, \Delta \textit{Registry} \bullet \textit{RegistryPullOk} \land \textit{NetPromote}
\end{aligned}
$$

Pushing and pulling fail if there is no registry with the input hostname.

6 Image References

An image reference identifies an image in a registry. Let's remind ourselves what image references look like.

An image reference consists of a hostname (with optional port) and a path. The image reference may also contain a tag and/or a digest. The hostname determines the network location of a registry. The path consists of one or more components separated by forward slashes. The first component is sometimes, by convention for certain registries, a user name providing access control to the image.

Let's look at some examples:

- The image name `docker.io/istio/proxyv2` refers to an image with user name `istio` residing in the docker hub registry at `docker.io`.
- The image name `projectriff/builder:v1` is short-hand for `docker.io/projectriff/builder:v1` which refers to an image with user name `projectriff` also residing at `docker.io`. The image has tag `v1`.
- The image name `gcr.io/cf-elafros/knative-releases/github.com/knative/serving/cmd/autoscaler@sha256:deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef` refers to an image with user name `cf-elafros` residing at `gcr.io`. The image has a (made up) `SHA-256` digest.

For the purposes of our model, an image reference consists of a hostname, a path, a tag, and an optional content digest.

Ref

host : *Hostname*
path : *Path*
tag : *Tag*
dig : *OptionalContentDigest*

A tag is always logically present, but if it is omitted from the textual representation of an image reference, it defaults to *Latest*. A content digest may be part of an image reference or may be omitted.

So far the push and pull operations have accumulated several input parameters.

$$\begin{array}{|l}
\hline
\mathit{PushParms} \\
h? : \mathit{Hostname} \\
p? : \mathit{Path} \\
t? : \mathit{Tag} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{PullParms} \\
h? : \mathit{Hostname} \\
p? : \mathit{Path} \\
t? : \mathit{Tag} \\
d? : \mathit{OptionalContentDigest} \\
\hline
\end{array}$$

An image reference is related to push input parameters as follows.

$$\begin{array}{|l}
\hline
\mathit{RefPushParms} \\
r? : \mathit{Ref} \\
\mathit{PushParms} \\
\hline
h? = r?.\mathit{host} \\
p? = r?.\mathit{path} \\
t? = r?.\mathit{tag} \\
r?.\mathit{dig} = \mathit{None} \\
\hline
\end{array}$$

Pushing is not allowed if the image reference has a content digest.

An image reference is related to pull input parameters as follows.

$$\begin{array}{|l}
\hline
\mathit{RefPullParms} \\
r? : \mathit{Ref} \\
\mathit{PullParms} \\
\hline
h? = r?.\mathit{host} \\
p? = r?.\mathit{path} \\
t? = r?.\mathit{tag} \\
d? = r?.\mathit{dig} \\
\hline
\end{array}$$

Push and pull can then be reframed to take an image reference instead of the corresponding input parameters.

$$\mathit{RefPushOk} \mathrel{\widehat{=}} \exists\, \mathit{PushParms} \bullet \mathit{NetPushOk} \land \mathit{RefPushParms}$$
$$\mathit{RefPullOk} \mathrel{\widehat{=}} \exists\, \mathit{PullParms} \bullet \mathit{NetPullOk} \land \mathit{RefPullParms}$$

A Bibliography

[1] Various authors, *Docker Inc.*, `https://www.docker.com/`.

[2] Various authors, *Open Container Initiative*, `https://www.opencontainers.org/`.

[3] Mike Spivey, *The Z Manual*, `https://www.cs.umd.edu/mvz/handouts/z-manual.pdf`.

[4] Mike Spivey, *The fuzz type-checker for Z*, `https://bitbucket.org/Spivey/fuzz`.

B Z Notation

Numbers:

$\mathbb{N}$	Natural numbers {0,1,...}

Propositional logic and the schema calculus:

$\ldots \land \ldots$	And	$\langle\!\langle \ldots \rangle\!\rangle$	Free type injection
$\ldots \lor \ldots$	Or	$[\ldots]$	Given sets
$\ldots \Rightarrow \ldots$	Implies	$', ?, !, _0 \ldots _9$	Schema decorations
$\forall .. \mid .. \bullet ..$	For all	$\ldots \vdash \ldots$	theorem
$\exists .. \mid .. \bullet ..$	There exists	$\theta \ldots$	Binding formation
$\ldots \setminus \ldots$	Hiding	$\lambda \ldots$	Function definition
$\ldots \widehat{=} \ldots$	Schema definition	$\mu \ldots$	Mu-expression
$\ldots == \ldots$	Abbreviation	$\Delta \ldots$	State change
$\ldots ::= \ldots \mid \ldots$	Free type definition	$\Xi \ldots$	Invariant state change

Sets and sequences:

$\{\ldots\}$	Set	$\ldots \setminus \ldots$	Set difference
$\{.. \mid .. \bullet ..\}$	Set comprehension	$\bigcup \ldots$	Distributed union
$\mathbb{P} \ldots$	Set of subsets of	$\# \ldots$	Cardinality
$\varnothing$	Empty set	$\ldots \subseteq \ldots$	Subset
$\ldots \times \ldots$	Cartesian product	$\ldots \subset \ldots$	Proper subset
$\ldots \in \ldots$	Set membership	$\ldots$ partition $\ldots$	Set partition
$\ldots \notin \ldots$	Set non-membership	seq	Sequences
$\ldots \cup \ldots$	Union	$\langle \ldots \rangle$	Sequence
$\ldots \cap \ldots$	Intersection	disjoint $\ldots$	Disjoint sequence of sets

Functions and relations:

$\ldots \leftrightarrow \ldots$	Relation	$\ldots^{*}$	Reflexive-transitive closure
$\ldots ⇸ \ldots$	Partial function	$\ldots \llparenthesis \ldots \rrparenthesis$	Relational image
$\ldots \rightarrow \ldots$	Total function	$\ldots \oplus \ldots$	Functional overriding
$\ldots ⤔ \ldots$	Partial injection	$\ldots \lhd \ldots$	Domain restriction
$\ldots \rightarrowtail \ldots$	Injection	$\ldots \rhd \ldots$	Range restriction
dom $\ldots$	Domain	$\ldots ⩤ \ldots$	Domain subtraction
ran $\ldots$	Range	$\ldots ⩥ \ldots$	Range subtraction
$\ldots \mapsto \ldots$	maplet		
$\ldots^{\sim}$	Relational inverse		

Axiomatic descriptions:

Declarations

Predicates

Schema definitions:

SchemaName

Declaration

Predicates

Decorations:

(*undecorated*)	Input state	?	Input parameter
$'$	Output state	!	Output parameter