Docker Image Specification v1.0.0

An *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the format of these filesystem changes and corresponding parameters and describes how to create and use them for use with a container runtime and execution tool.

Terminology

This specification uses the following terms:

Layer

Images are composed of *layers*. *Image layer* is a general term which may be used to refer to one or both of the following:

1. The metadata for the layer, described in the JSON format.
2. The filesystem changes described by a layer.

To refer to the former you may use the term *Layer JSON* or *Layer Metadata*. To refer to the latter you may use the term *Image Filesystem Changeset* or *Image Diff*.

Image JSON

Each layer has an associated JSON structure which describes some basic information about the image such as date created, author, and the ID of its parent image as well as execution/runtime configuration like its entry point, default arguments, CPU/memory shares, networking, and volumes.

Image Filesystem Changeset

Each layer has an archive of the files which have been added, changed, or deleted relative to its parent layer. Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

Image ID

Each layer is given an ID upon its creation. It is represented as a hexadecimal encoding of 256 bits, e.g., `a9561eb1b190625c9adb5a9513e72c4dedafc1cb2d4c5236c9a6957ec7dfd5a9`. Image IDs should be sufficiently random so as to be globally unique. 32 bytes read from `/dev/urandom` is sufficient for all practical purposes. Alternatively, an image ID may be derived as a cryptographic hash of image contents as the result is considered indistinguishable from random. The choice is left up to implementors.

Image Parent

Most layer metadata structs contain a `parent` field which refers to the Image from which another directly descends. An image contains a separate JSON metadata file and set of changes relative to the filesystem of its parent image. *Image Ancestor* and *Image Descendant* are also common terms.

Image Checksum

Layer metadata structs contain a cryptographic hash of the contents of the layer's filesystem changeset. Though the set of changes exists as a simple Tar archive, two archives with identical filenames and content will have different SHA digests if the last-access or last-modified times of any entries differ. For this reason, image checksums are generated using the TarSum algorithm which produces a cryptographic hash of file contents and selected headers only. Details of this algorithm are described in the separate TarSum specification.

Tag

A tag serves to map a descriptive, user-given name to any single image ID. An image name suffix (the name component after `:`) is often referred to as a tag as well, though it strictly refers to the full name of an image. Acceptable values for a tag suffix are implementation specific, but they SHOULD be limited to the set of alphanumeric characters `[a-zA-Z0-9]`, punctuation characters `[._-]`, and MUST NOT contain a `:` character.

Repository

A collection of tags grouped under a common prefix (the name component before `:`). For example, in an image tagged with the name `my-app:3.1.4`, `my-app` is the *Repository* component of the name. Acceptable values for repository name are implementation specific, but they SHOULD be limited to the set of alphanumeric characters `[a-zA-Z0-9]`, and punctuation characters `[._-]`, however it MAY contain additional `/` and `:` characters for organizational purposes, with the last

`:` character being interpreted dividing the repository component of the name from the tag suffix component.

Image JSON Description

Here is an example image JSON file:

```
{
    "id": "a9561eb1b190625c9adb5a9513e72c4dedafc1cb2d4c5236c9a6957ec7dfd5a9",
    "parent":
"c6e3cedcda2e3982a1a6760e178355e8e65f7b80e4e5248743fa3549d284e024",
    "checksum":
"tarsum.v1+sha256:e58fcf7418d2390dec8e8fb69d88c06ec07039d651fedc3aa72af9972e7d046

    "created": "2014-10-13T21:19:18.674353812Z",
    "author": "Alyssa P. Hacker &ltalyspdev@example.com&gt",
    "architecture": "amd64",
    "os": "linux",
    "Size": 271828,
    "config": {
        "User": "alice",
        "Memory": 2048,
        "MemorySwap": 4096,
        "CpuShares": 8,
        "ExposedPorts": {
            "8080/tcp": {}
        },
        "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
            "FOO=docker_is_a_really",
            "BAR=great_tool_you_know"
        ],
        "Entrypoint": [
            "/bin/my-app-binary"
        ],
        "Cmd": [
            "--foreground",
            "--config",
            "/etc/my-app.d/default.cfg"
        ],
        "Volumes": {
            "/var/job-result-data": {},
            "/var/log/my-app-logs": {},
        },
        "WorkingDir": "/home/alice",
    }
}
```

Image JSON Field Descriptions

id `string`
Randomly generated, 256-bit, hexadecimal encoded. Uniquely identifies the image.

parent `string`
ID of the parent image. If there is no parent image then this field should be omitted. A collection of images may share many of the same ancestor layers. This organizational structure is strictly a tree with any one layer having either no parent or a single parent and zero or more descendant layers. Cycles are not allowed and implementations should be careful to avoid creating them or iterating through a cycle indefinitely.

created `string`
ISO-8601 formatted combined date and time at which the image was created.

author `string`
Gives the name and/or email address of the person or entity which created and is responsible for maintaining the image.

architecture `string`
The CPU architecture which the binaries in this image are built to run on. Possible values include:

- 386
- amd64
- arm

More values may be supported in the future and any of these may or may not be supported by a given container runtime implementation.

os `string`
The name of the operating system which the image is built to run on. Possible values include:

- darwin
- freebsd
- linux

More values may be supported in the future and any of these may or may not be supported by a given container runtime implementation.

checksum `string`
Image Checksum of the filesystem changeset associated with the image layer.

Size `integer`
The size in bytes of the filesystem changeset associated with the image layer.

config `struct`
The execution parameters which should be used as a base when running a container using the image. This field can be `null`, in which case any execution parameters should be specified at creation of the container.

Container RunConfig Field Descriptions

User `string`
The username or UID which the process in the container should run as. This acts as a default value to use when the value is not specified when creating a container.

All of the following are valid:

- `user`
- `uid`
- `user:group`
- `uid:gid`
- `uid:group`
- `user:gid`

If `group`/`gid` is not specified, the default group and supplementary groups of the given `user`/`uid` in `/etc/passwd` from the container are applied.

Memory `integer`
Memory limit (in bytes). This acts as a default value to use when the value is not specified when creating a container.

MemorySwap `integer`
Total memory usage (memory + swap); set to `-1` to disable swap. This acts as a default value to use when the value is not specified when creating a container.

CpuShares `integer`
CPU shares (relative weight vs. other containers). This acts as a default value to use when the value is not specified when creating a container.

ExposedPorts `struct`
A set of ports to expose from a container running this image. This JSON structure value is unusual because it is a direct JSON serialization of the Go type `map[string]struct{}` and is represented in JSON as an object mapping its keys to an empty object. Here is an example:

```
{
    "8080": {},
    "53/udp": {},
    "2356/tcp": {}
}
```

Its keys can be in the format of:

- `"port/tcp"`
- `"port/udp"`
- `"port"`

with the default protocol being `"tcp"` if not specified. These values act as defaults and are merged with any specified when creating a container.

Env `array of strings`
Entries are in the format of `VARNAME="var value"`. These values act as defaults and are merged with any specified when creating a container.

Entrypoint `array of strings`
A list of arguments to use as the command to execute when the container starts. This value acts as a default and is replaced by an entrypoint specified when creating a container.

Cmd `array of strings`
Default arguments to the entry point of the container. These values act as defaults and are replaced with any specified when creating a container. If an `Entrypoint` value is not specified, then the first entry of the `Cmd` array should be interpreted as the executable to run.

Volumes `struct`
A set of directories which should be created as data volumes in a container running this image. This JSON structure value is unusual because it is a direct JSON serialization of the Go type `map[string]struct{}` and is represented in JSON as an object mapping its keys to an empty object. Here is an example:

```
{
    "/var/my-app-data/": {},
    "/etc/some-config.d/": {},
}
```

WorkingDir `string`
Sets the current working directory of the entry point process in the container. This value acts as a default and is replaced by a working directory specified when creating a container.

Any extra fields in the Image JSON struct are considered implementation specific and should be ignored by any implementations which are unable to interpret them.

Creating an Image Filesystem Changeset

An example of creating an Image Filesystem Changeset follows.

An image root filesystem is first created as an empty directory named with the ID of the image being created. Here is the initial empty directory structure for the changeset for an image with ID `c3167915dc9d` (real IDs are much longer, but this example use a truncated one here for brevity. Implementations need not name the rootfs directory in this way but it may be convenient for keeping record of a large number of image layers.):

```
c3167915dc9d/
```

Files and directories are then created:

```
c3167915dc9d/
    etc/
        my-app-config
    bin/
        my-app-binary
        my-app-tools
```

The `c3167915dc9d` directory is then committed as a plain Tar archive with entries for the following files:

```
etc/my-app-config
bin/my-app-binary
bin/my-app-tools
```

The TarSum checksum for the archive file is then computed and placed in the JSON metadata along with the execution parameters.

To make changes to the filesystem of this container image, create a new directory named with a new ID, such as `f60c56784b83`, and initialize it with a snapshot of the parent image's root filesystem, so that the directory is identical to that of `c3167915dc9d`. NOTE: a copy-on-write or union filesystem can make this very efficient:

```
f60c56784b83/
    etc/
        my-app-config
    bin/
        my-app-binary
        my-app-tools
```

This example change is going to add a configuration directory at `/etc/my-app.d` which contains a default config file. There's also a change to the `my-app-tools` binary to handle the config layout change. The `f60c56784b83` directory then looks like this:

```
f60c56784b83/
    etc/
        my-app.d/
            default.cfg
    bin/
        my-app-binary
        my-app-tools
```

This reflects the removal of `/etc/my-app-config` and creation of a file and directory at `/etc/my-app.d/default.cfg`. `/bin/my-app-tools` has also been replaced with an updated version. Before committing this directory to a changeset, because it has a parent image, it is first compared with the directory tree of the parent snapshot, `f60c56784b83`, looking for files and directories that have been added, modified, or removed. The following changeset is found:

```
Added:      /etc/my-app.d/default.cfg
Modified:   /bin/my-app-tools
Deleted:    /etc/my-app-config
```

A Tar Archive is then created which contains *only* this changeset: The added and modified files and directories in their entirety, and for each deleted item an entry for an empty file at the same location but with the basename of the deleted file or directory prefixed with `.wh.`. The filenames prefixed with `.wh.` are known as "whiteout" files. NOTE: For this reason, it is not possible to create an image root filesystem which contains a file or directory with a name beginning with `.wh.`. The resulting Tar archive for `f60c56784b83` has the following entries:

```
/etc/my-app.d/default.cfg
/bin/my-app-tools
/etc/.wh.my-app-config
```

Any given image is likely to be composed of several of these Image Filesystem Changeset tar archives.

Combined Image JSON + Filesystem Changeset Format

There is also a format for a single archive which contains complete information about an image, including:

- repository names/tags

- all image layer JSON files
- all tar archives of each layer filesystem changesets

For example, here's what the full archive of `library/busybox` is (displayed in `tree` format):

```
.
├── 5785b62b697b99a5af6cd5d0aabc804d5748abbb6d3d07da5d1d3795f2dcc83e
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── a7b8b41220991bfc754d7ad445ad27b7f272ab8b4a2c175b9512b97471d02a8a
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── a936027c5ca8bf8f517923169a233e391cbb38469a75de8383b5228dc2d26ceb
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── f60c56784b832dd990022afc120b8136ab3da9528094752ae13fe63a2d28dc8c
│   ├── VERSION
│   ├── json
│   └── layer.tar
└── repositories
```

There are one or more directories named with the ID for each layer in a full image. Each of these directories contains 3 files:

- `VERSION` - The schema version of the `json` file
- `json` - The JSON metadata for an image layer
- `layer.tar` - The Tar archive of the filesystem changeset for an image layer.

The content of the `VERSION` files is simply the semantic version of the JSON metadata schema:

```
1.0
```

And the `repositories` file is another JSON file which describes names/tags:

```
{
    "busybox":{

"latest":"5785b62b697b99a5af6cd5d0aabc804d5748abbb6d3d07da5d1d3795f2dcc83e"
    }
}
```

Every key in this object is the name of a repository, and maps to a collection of tag suffixes. Each tag maps to the ID of the image represented by that tag.

Loading an Image Filesystem Changeset

Unpacking a bundle of image layer JSON files and their corresponding filesystem changesets can be done using a series of steps:

1. Follow the parent IDs of image layers to find the root ancestor (an image with no parent ID specified).

2. For every image layer, in order from root ancestor and descending down, extract the contents of that layer's filesystem changeset archive into a directory which will be used as the root of a container filesystem.

    - Extract all contents of each archive.
    - Walk the directory tree once more, removing any files with the prefix `.wh.` and the corresponding file or directory named without this prefix.

Implementations

This specification is an admittedly imperfect description of an imperfectly-understood problem. The Docker project is, in turn, an attempt to implement this specification. Our goal and our execution toward it will evolve over time, but our primary concern in this specification and in our implementation is compatibility and interoperability.