Open Container Initiative Image Format Specification

Open Container Initiative

Image Format Specification

This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration.

The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.

Table of Contents

Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 (Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997).

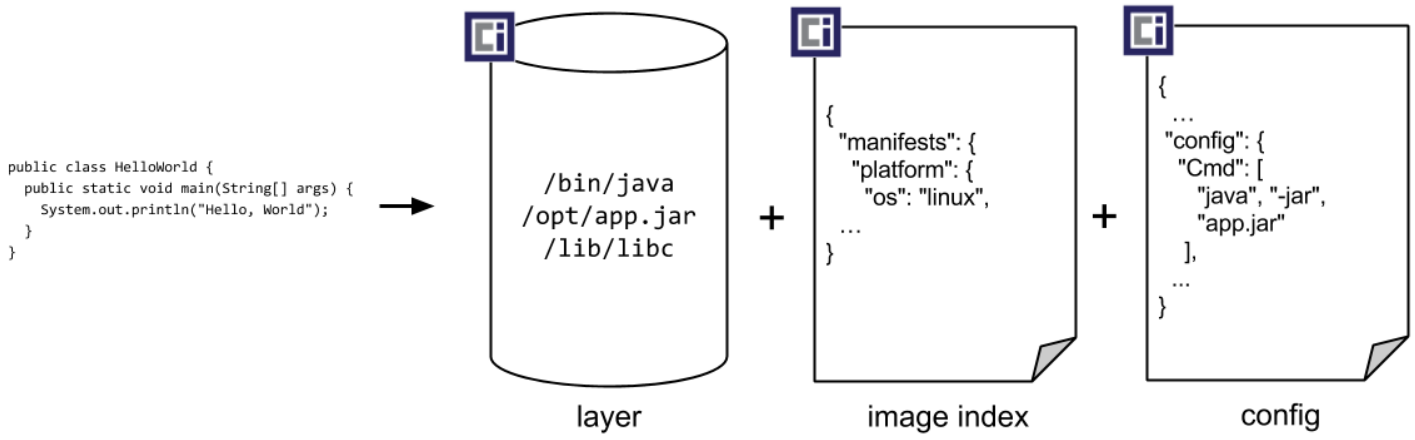
The key words "unspecified", "undefined", and "implementation-defined" are to be interpreted as described in the rationale for the C99 standard.

An implementation is not compliant if it fails to satisfy one or more of the MUST, MUST NOT, REQUIRED, SHALL, or SHALL NOT requirements for the protocols it implements. An implementation is compliant if it satisfies all the MUST, MUST NOT, REQUIRED, SHALL, and SHALL NOT requirements for the protocols it implements.

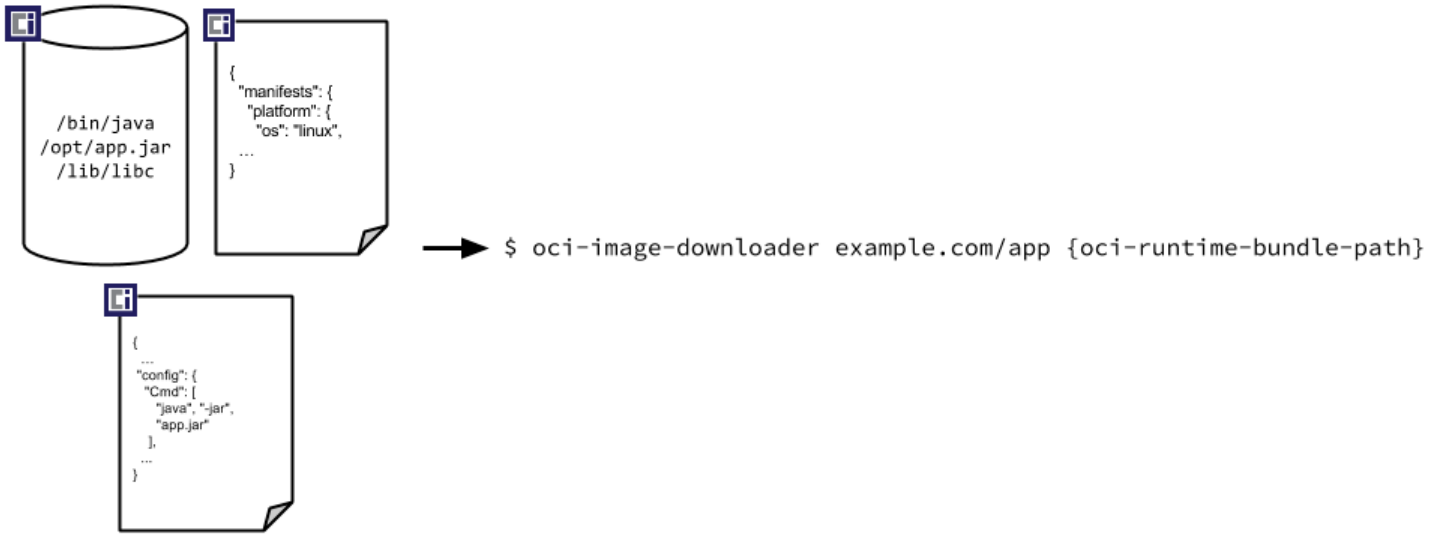
Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final

runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



Once built the OCI Image can then be discovered by name, downloaded, verified by hash, trusted through a signature, and unpacked into an OCI Runtime Bundle.



Understanding the Specification

The OCI Image Media Types document is a starting point to understanding the overall structure of the specification.

The high-level components of the spec include:

- Image Manifest - a document describing the components that make up a container image
- Image Index - an annotated list of manifests
- Image Layout - a filesystem layout representing the contents of an image
- Filesystem Layer - a changeset that describes a container's filesystem
- Image Configuration - a document determining layer ordering and configuration of the image suitable for translation into a [runtime bundle][runtime-spec]
- Conversion - a document describing how this translation should occur
- Artifacts Guidance - a document describing how to use the spec for packaging content other than OCI images
- Descriptor - a reference that describes the type, metadata and content address of referenced content

Future versions of this specification may include the following OPTIONAL features:

- Signatures that are based on signing image content address
- Naming that is federated based on DNS and can be delegated

OCI Image Media Types

The following media types identify the formats described here and their referenced resources:

- `application/vnd.oci.descriptor.v1+json`: Content Descriptor
- `application/vnd.oci.layout.header.v1+json`: OCI Layout
- `application/vnd.oci.image.index.v1+json`: Image Index
- `application/vnd.oci.image.manifest.v1+json`: Image manifest
- `application/vnd.oci.image.config.v1+json`: Image config
- `application/vnd.oci.image.layer.v1.tar`: "Layer", as a tar archive
- `application/vnd.oci.image.layer.v1.tar+gzip`: "Layer", as a tar archive compressed with gzip
- `application/vnd.oci.image.layer.v1.tar+zstd`: "Layer", as a tar archive compressed with zstd
- `application/vnd.oci.empty.v1+json`: Empty for unused descriptors

The following media types identify a "Layer" with distribution restrictions, but are **deprecated** and not recommended for future use:

- `application/vnd.oci.image.layer.nondistributable.v1.tar`: "Layer", as a tar archive
- `application/vnd.oci.image.layer.nondistributable.v1.tar+gzip`: "Layer", as a tar archive with distribution restrictions compressed with gzip
- `application/vnd.oci.image.layer.nondistributable.v1.tar+zstd`: "Layer", as a tar archive with distribution restrictions compressed with zstd

Media Type Conflicts

Blob retrieval methods MAY return media type metadata. For example, a HTTP response might return a manifest with the Content-Type header set to `application/vnd.oci.image.manifest.v1+json`. Implementations MAY also have expectations for the blob's media type and digest (e.g. from a descriptor referencing the blob).

- Implementations that do not have an expected media type for the blob SHOULD respect the returned media type.
- Implementations that have an expected media type which matches the returned media type SHOULD respect the matched media type.
- Implementations that have an expected media type which does not match the returned media type SHOULD:
  - Respect the expected media type if the blob matches the expected digest. Implementations MAY warn about the media type mismatch.
  - Return an error if the blob does not match the expected digest (as recommended for descriptors).
  - Return an error if they do not have an expected digest.

Compatibility Matrix

The OCI Image Specification strives to be backwards and forwards compatible when possible. Breaking compatibility with existing systems creates a burden on users whether they are build systems, distribution systems, container engines, etc. This section shows where the OCI Image Specification is compatible with formats external to the OCI Image and different versions of this specification.

application/vnd.oci.image.index.v1+json

Similar/related schema:

- application/vnd.docker.distribution.manifest.list.v2+json

  - `.annotations`: only present in OCI
  - `.[]manifests.annotations`: only present in OCI
  - `.[]manifests.urls`: only present in OCI

application/vnd.oci.image.manifest.v1+json

Similar/related schema:

- application/vnd.docker.distribution.manifest.v2+json
  - `.annotations`: only present in OCI
  - `.config.annotations`: only present in OCI
  - `.config.urls`: only present in OCI
  - `.[]layers.annotations`: only present in OCI

application/vnd.oci.image.layer.v1.tar+gzip

Interchangeable and fully compatible mime-types:

- application/vnd.docker.image.rootfs.diff.tar.gzip

application/vnd.oci.image.config.v1+json

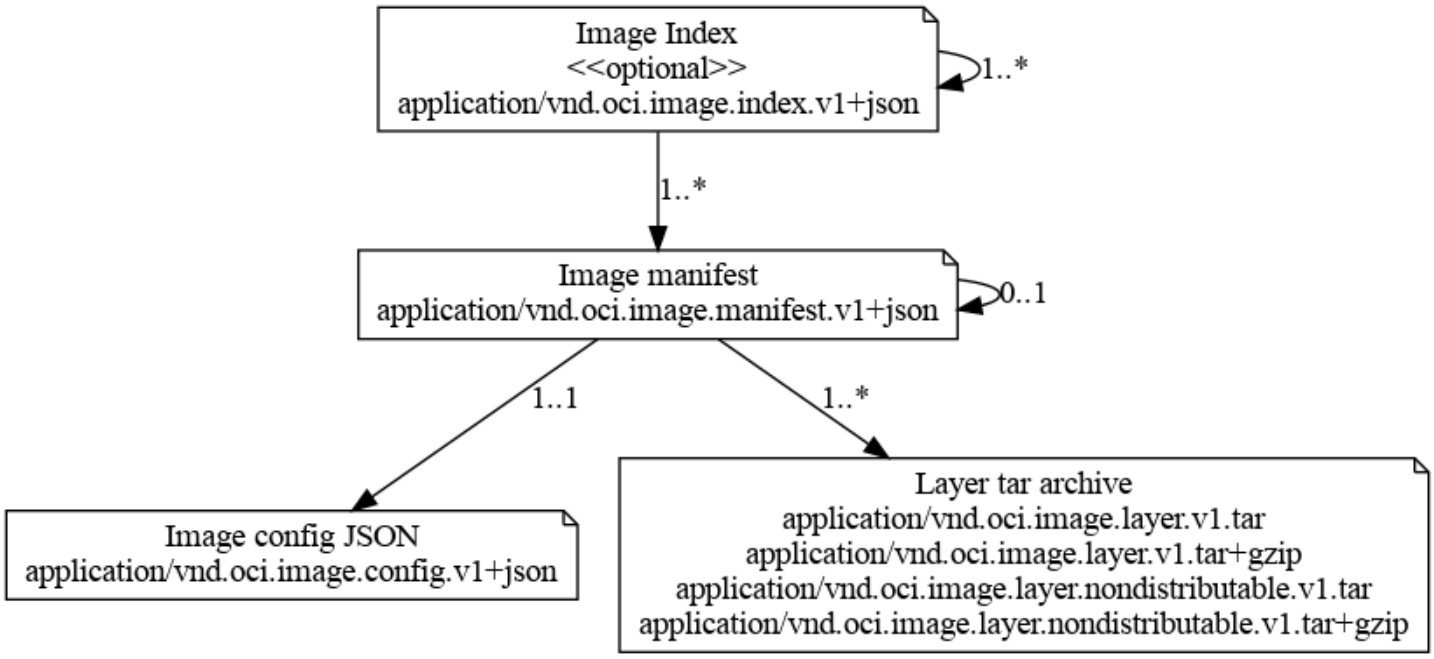
Similar/related schema:

- application/vnd.docker.container.image.v1+json (Docker Image Spec v1.2)
  - `.config.Memory`: only present in Docker, and reserved in OCI
  - `.config.MemorySwap`: only present in Docker, and reserved in OCI
  - `.config.CpuShares`: only present in Docker, and reserved in OCI
  - `.config.Healthcheck`: only present in Docker, and reserved in OCI
- Moby/Docker
  - `.config.ArgsEscaped`: Windows-specific Moby/Docker extension, deprecated in OCI, available for compatibility with older images.

`.config.StopSignal` and `.config.Labels` are accidentally undocumented in Docker Image Spec v1.2, but these fields are not OCI-specific concepts.

Relations

The following figure shows how the above media types reference each other:



Descriptors are used for all references. The image-index being a "fat manifest" references a list of image manifests per target platform. An image manifest references exactly one target configuration and possibly many layers.

OCI Content Descriptors

- An OCI image consists of several different components, arranged in a Merkle Directed Acyclic Graph (DAG).
- References between components in the graph are expressed through *Content Descriptors*.
- A Content Descriptor (or simply *Descriptor*) describes the disposition of the targeted content.
- A Content Descriptor includes the type of the content, a content identifier (*digest*), and the byte-size of the raw content. Optionally, it includes the type of artifact it is describing.
- Descriptors SHOULD be embedded in other formats to securely reference external content.
- Other formats SHOULD use descriptors to securely reference external content.

This section defines the `application/vnd.oci.descriptor.v1+json` media type.

Properties

A descriptor consists of a set of properties encapsulated in key-value fields.

The following fields contain the primary properties that constitute a Descriptor:

- **`mediaType`** *string*

  This REQUIRED property contains the media type of the referenced content. Values MUST comply with RFC 6838, including the naming requirements in its section 4.2.

  The OCI image specification defines several of its own MIME types for resources defined in the specification.

- **`digest`** *string*

  This REQUIRED property is the *digest* of the targeted content, conforming to the requirements outlined in Digests. Retrieved content SHOULD be verified against this digest when consumed via untrusted sources.

- **`size`** *int64*

  This REQUIRED property specifies the size, in bytes, of the raw content. This property exists so that a client will have an expected size for the content before processing. If the length of the retrieved content does not match the specified length, the content SHOULD NOT be trusted.

- `urls` *array of strings*

  This OPTIONAL property specifies a list of URIs from which this object MAY be downloaded. Each entry MUST conform to RFC 3986. Entries SHOULD use the `http` and `https` schemes, as defined in RFC 7230.

- `annotations` *string-string map*

  This OPTIONAL property contains arbitrary metadata for this descriptor. This OPTIONAL property MUST use the annotation rules.

- `data` *string*

  This OPTIONAL property contains an embedded representation of the referenced content. Values MUST conform to the Base 64 encoding, as defined in RFC 4648. The decoded data MUST be identical to the referenced content and SHOULD be verified against the `digest` and `size` fields by content consumers. See Embedded Content for when this is appropriate.

- `artifactType` *string*

  This OPTIONAL property contains the type of an artifact when the descriptor points to an artifact. This is the value of the config descriptor `mediaType` when the descriptor references an image manifest. If defined, the value MUST comply with RFC 6838, including the naming requirements in its section 4.2, and MAY be registered with IANA.

Descriptors pointing to `application/vnd.oci.image.manifest.v1+json` SHOULD include the extended field `platform`, see Image Index Property Descriptions for details.

Reserved

Extended *Descriptor* field additions proposed in other OCI specifications SHOULD first be considered for addition into this specification.

Digests

The *digest* property of a Descriptor acts as a content identifier, enabling content addressability. It uniquely identifies content by taking a collision-resistant hash of the bytes. If the *digest* can be communicated in a secure manner, one can verify content from an insecure source by recalculating the digest independently, ensuring the content has not been modified.

The value of the `digest` property is a string consisting of an *algorithm* portion and an *encoded* portion. The *algorithm* specifies the cryptographic hash function and encoding used for the digest; the *encoded* portion contains the encoded result of the hash function.

A digest string MUST match the following grammar:

```
digest                ::= algorithm ":" encoded
algorithm             ::= algorithm-component (algorithm-separator algorithm-component)*
algorithm-component   ::= [a-z0-9]+
algorithm-separator   ::= [+._-]
encoded               ::= [a-zA-Z0-9=_-]+
```

Note that *algorithm* MAY impose algorithm-specific restriction on the grammar of the *encoded* portion. See also Registered Algorithms.

Some example digest strings include the following:

digest	algorithm	Registered
`sha256:6c3c624b58dbbcd3c0dd82b4c53f04194d1247c6eebdaab7c610cf7d66709b3b`	SHA-256	Yes
`sha512:401b09eab3c013d4ca54922bb802bec8fd5318192b0a75f201d8b372742...`	SHA-512	Yes
`multihash+base58:QmRZxt2b1FVZPNqd8hsiykDL3TdBDeTSPX9Kv46HmX4Gx8`	Multihash	No
`sha256+b64u:LCa0a2j_xo_5m0U8HTBBNBNCLXBkg7-g-YpeiGJm564`	SHA-256 with urlsafe base64	No

Please see Registered Algorithms for a list of registered algorithms.

Implementations SHOULD allow digests with unrecognized algorithms to pass validation if they comply with the above grammar. While `sha256` will only use hex encoded digests, separators in *algorithm* and alphanumerics in *encoded* are included to allow for extensions. As an example, we can parameterize the encoding and algorithm as `multihash+base58:QmRZxt2b1FVZPNqd8hsiykDL3TdBDeTSPX9Kv46HmX4Gx8`, which would be considered valid but unregistered by this specification.

Verification

Before consuming content targeted by a descriptor from untrusted sources, the byte content SHOULD be verified against the digest string. Before calculating the digest, the size of the content SHOULD be verified to reduce hash collision space. Heavy processing before calculating a hash SHOULD be avoided. Implementations MAY employ canonicalization of the underlying content to ensure stable content identifiers.

Digest calculations

A *digest* is calculated by the following pseudo-code, where `H` is the selected hash algorithm, identified by string `<alg>`:

```
let ID(C) = Descriptor.digest
let C = <bytes>
let D = '<alg>:' + Encode(H(C))
let verified = ID(C) == D
```

Above, we define the content identifier as `ID(C)`, extracted from the `Descriptor.digest` field. Content `C` is a string of bytes. Function `H` returns the hash of `C` in bytes and is passed to function `Encode` and prefixed with the algorithm to obtain the digest. The result `verified` is true if `ID(C)` is equal to `D`, confirming that `C` is the content identified by `D`. After verification, the following is true:

```
D == ID(C) == '<alg>:' + Encode(H(C))
```

The *digest* is confirmed as the content identifier by independently calculating the *digest*.

Registered algorithms

While the *algorithm* component of the digest string allows the use of a variety of cryptographic algorithms, compliant implementations SHOULD use SHA-256.

The following algorithm identifiers are currently defined by this specification:

algorithm identifier	algorithm
`sha256`	SHA-256
`sha512`	SHA-512

If a useful algorithm is not included in the above table, it SHOULD be submitted to this specification for registration.

SHA-256

SHA-256 is a collision-resistant hash function, chosen for ubiquity, reasonable size and secure characteristics. Implementations MUST implement SHA-256 digest verification for use in descriptors.

When the *algorithm identifier* is `sha256`, the *encoded* portion MUST match `/[a-f0-9]{64}/`. Note that `[A-F]` MUST NOT be used here.

SHA-512

SHA-512 is a collision-resistant hash function which may be more performant than SHA-256 on some CPUs. Implementations MAY implement SHA-512 digest verification for use in descriptors.

When the *algorithm identifier* is `sha512`, the *encoded* portion MUST match `/[a-f0-9]{128}/`. Note that `[A-F]` MUST NOT be used here.

Embedded Content

In many contexts, such as when downloading content over a network, resolving a descriptor to its content has a measurable fixed "roundtrip" latency cost. For large blobs, the fixed cost is usually inconsequential, as the majority of time will be spent actually fetching the content. For very small blobs, the fixed cost can be quite significant.

Implementations MAY choose to embed small pieces of content directly within a descriptor to avoid roundtrips.

Implementations MUST NOT populate the `data` field in situations where doing so would modify existing content identifiers. For example, a registry MUST NOT arbitrarily populate `data` fields within uploaded manifests, as that would modify the content identifier of those manifests. In contrast, a client MAY populate the `data` field before uploading a manifest, because the manifest would not yet have a content identifier in the registry.

Implementations SHOULD consider portability when deciding whether to embed data, as some providers are known to refuse to accept or parse manifests that exceed a certain size.

Examples

The following example describes a *Manifest* with a content identifier of "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392
and a size of 7682 bytes:

```
{
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "size": 7682,
  "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270"
}
```

In the following example, the descriptor indicates that the referenced manifest is retrievable from a particular URL:

```
{
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "size": 7682,
  "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270",
  "urls": [
    "https://example.com/example-manifest"
  ]
}
```

In the following example, the descriptor indicates the type of artifact it is referencing:

```
{
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "size": 123,
  "digest": "sha256:87923725d74f4bfb94c9e86d64170f7521aad8221a5de834851470ca142da630",
  "artifactType": "application/vnd.example.sbom.v1"
}
```

OCI Image Layout Specification

- The OCI Image Layout is the directory structure for OCI content-addressable blobs and location-addressable references (refs).
- This layout MAY be used in a variety of different transport mechanisms: archive formats (e.g. tar, zip), shared filesystem environments (e.g. nfs), or networked file fetching (e.g. http, ftp, rsync).

Given an image layout and a ref, a tool can create an OCI Runtime Specification bundle by:

- Following the ref to find a manifest, possibly via an image index
- Applying the filesystem layers in the specified order
- Converting the image configuration into an OCI Runtime Specification `config.json`

Content

The image layout is as follows:

- `blobs` directory
  - Contains content-addressable blobs
  - A blob has no schema and SHOULD be considered opaque
  - Directory MUST exist and MAY be empty
  - See blobs section
- `oci-layout` file
  - It MUST exist
  - It MUST be a JSON object
  - It MUST contain an `imageLayoutVersion` field
  - See oci-layout file section
  - It MAY include additional fields
- `index.json` file
  - It MUST exist
  - It MUST be an image index JSON object.
  - See index.json section

Example Layout

This is an example image layout:

```
$ cd example.com/app/
$ find . -type f
./index.json
./oci-layout
./blobs/sha256/3588d02542238316759cbf24502f4344ffcc8a60c803870022f335d1390c13b4
./blobs/sha256/4b0bc1c4050b03c95ef2a8e36e25feac42fd31283e8c30b3ee5df6b043155d3c
./blobs/sha256/7968321274dc6b6171697c33df7815310468e694ac5be0ec03ff053bb135e768
```

Blobs are named by their contents:

```
$ shasum -a 256 ./blobs/sha256/afff3924849e458c5ef237db5f89539274d5e609db5db935ed3959c90f1f2d51
afff3924849e458c5ef237db5f89539274d5e609db5db935ed3959c90f1f2d51 ./blobs/sha256/afff3924849e458c5ef237db5f8953
```

Blobs

- Object names in the `blobs` subdirectories are composed of a directory for each hash algorithm, the children of which will contain the actual content.
- The content of `blobs/<alg>/<encoded>` MUST match the digest `<alg>:<encoded>` (referenced per descriptor). For example, the content of `blobs/sha256/da39a3ee5e6b4b0d3255bfef95601890afd80709` MUST match the digest `sha256:da39a3ee5e6b4b0d3255bfef95601890afd80709`.
- The character set of the entry name for `<alg>` and `<encoded>` MUST match the respective grammar elements described in descriptor.
- The blobs directory MAY contain blobs which are not referenced by any of the refs.
- The blobs directory MAY be missing referenced blobs, in which case the missing blobs SHOULD be fulfilled by an external blob store.

Example Blobs

```
$ cat ./blobs/sha256/9b97579de92b1c195b85bb42a11011378ee549b02d7fe9c17bf2a6b35d5cb079 | jq
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:afff3924849e458c5ef237db5f89539274d5e609db5db935ed3959c90f1f2d51",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      }
    },
...

$ cat ./blobs/sha256/afff3924849e458c5ef237db5f89539274d5e609db5db935ed3959c90f1f2d51 | jq
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "size": 7023,
    "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270"
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 32654,
      "digest": "sha256:9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0"
    },
...

$ cat ./blobs/sha256/5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270 | jq
{
  "architecture": "amd64",
  "author": "Alyssa P. Hacker <alyspdev@example.com>",
  "config": {
    "Hostname": "8dfe43d80430",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
```

```
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": null,
    "Image": "sha256:6986ae504bbf843512d680cc959484452034965db15f75ee8bdd1b107f61500b",
...

$ cat ./blobs/sha256/9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0
[gzipped tar stream]
```

oci-layout file

This JSON object serves as a marker for the base of an Open Container Image Layout and to provide the version of the image-layout in use. The `imageLayoutVersion` value will align with the OCI Image Specification version at the time changes to the layout are made, and will pin a given version until changes to the image layout are required. This section defines the `application/vnd.oci.layout.header.v1+json` media type.

oci-layout Example

```
{
    "imageLayoutVersion": "1.0.0"
}
```

index.json file

This REQUIRED file is the entry point for references and descriptors of the image-layout. The image index is a multi-descriptor entry point.

This index provides an established path (`/index.json`) to have an entry point for an image-layout and to discover auxiliary descriptors.

- No semantic restriction is given for the "org.opencontainers.image.ref.name" annotation of descriptors.
- In general the `mediaType` of each descriptor object in the `manifests` field will be either `application/vnd.oci.image.index.v1+` or `application/vnd.oci.image.manifest.v1+json`.
- Future versions of the spec MAY use a different mediatype (i.e. a new versioned format).
- An encountered `mediaType` that is unknown MUST NOT generate an error.

Implementor's Note: A common use case of descriptors with a "org.opencontainers.image.ref.name" annotation is representing a "tag" for a container image. For example, an image may have a tag for different versions or builds of the software. In the wild you often see "tags" like "v1.0.0-vendor.0", "2.0.0-debug", etc. Those tags will often be represented in an image-layout repository with matching "org.opencontainers.image.ref.name" annotations like "v1.0.0-vendor.0", "2.0.0-debug", etc.

Index Example

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.index.v1+json",
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.index.v1+json",
      "size": 7143,
      "digest": "sha256:0228f90e926ba6b96e4f39cf294b2586d38fbb5a1e385c05cd1ee40ea54fe7fd",
```

```
      "annotations": {
        "org.opencontainers.image.ref.name": "stable-release"
      }
    },
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab815ad7fc331f",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      },
      "annotations": {
        "org.opencontainers.image.ref.name": "v1.0"
      }
    },
    {
      "mediaType": "application/xml",
      "size": 7143,
      "digest": "sha256:b3d63d132d21c3ff4c35a061adf23cf43da8ae054247e32faa95494d904a007e",
      "annotations": {
        "org.freedesktop.specifications.metainfo.version": "1.0",
        "org.freedesktop.specifications.metainfo.type": "AppStream"
      }
    }
  ],
  "annotations": {
    "com.example.index.revision": "r124356"
  }
}
```

This illustrates an index that provides two named references and an auxiliary mediatype for this image layout.

The first named reference (`stable-release`) points to another index that might contain multiple references with distinct platforms and annotations. Note that the `org.opencontainers.image.ref.name` annotation SHOULD only be considered valid when on descriptors on `index.json`.

The second named reference (`v1.0`) points to a manifest that is specific to the linux/ppc64le platform.

OCI Image Manifest Specification

There are three main goals of the Image Manifest Specification. The first goal is content-addressable images, by supporting an image model where the image's configuration can be hashed to generate a unique ID for the image and its components. The second goal is to allow multi-architecture images, through a "fat manifest" which references image manifests for platform-specific versions of an image. In OCI, this is codified in an image index. The third goal is to be translatable to the OCI Runtime Specification.

This section defines the `application/vnd.oci.image.manifest.v1+json` media type. For the media type(s) that this is compatible with see the matrix.

Image Manifest

Unlike the image index, which contains information about a set of images that can span a variety of architectures and operating systems, an image manifest provides a configuration and set of layers for a single container image for a specific architecture and operating system.

Image Manifest Property Descriptions

- `schemaVersion` *int*

  This REQUIRED property specifies the image manifest schema version. For this version of the specification, this MUST be `2` to ensure backward compatibility with older versions of Docker. The value of this field will not change. This field MAY be removed in a future version of the specification.

- `mediaType` *string*

  This property SHOULD be used and remain compatible with earlier versions of this specification and with other similar external formats. When used, this field MUST contain the media type `application/vnd.oci.image.manifest.v1+json`. This field usage differs from the descriptor use of `mediaType`.

- `artifactType` *string*

  This OPTIONAL property contains the type of an artifact when the manifest is used for an artifact. This MUST be set when `config.mediaType` is set to the empty value. If defined, the value MUST comply with RFC 6838, including the naming requirements in its section 4.2, and MAY be registered with IANA. Implementations storing or copying image manifests MUST NOT error on encountering an `artifactType` that is unknown to the implementation.

- `config` *descriptor*

  This REQUIRED property references a configuration object for a container, by digest. Beyond the descriptor requirements, the value has the following additional restrictions:

  - `mediaType` *string*

    This descriptor property has additional restrictions for `config`.

    Implementations MUST NOT attempt to parse the referenced content if this media type is unknown and instead consider the referenced content as arbitrary binary data (e.g.: as `application/octet-stream`).

    Implementations storing or copying image manifests MUST NOT error on encountering a value that is unknown to the implementation.

    Implementations MUST support at least the following media types:

    * `application/vnd.oci.image.config.v1+json`

    Manifests for container images concerned with portability SHOULD use one of the above media types. Manifests for artifacts concerned with portability SHOULD use `config.mediaType` as described in Guidelines for Artifact Usage.

    If the manifest uses a different media type than the above, it MUST comply with RFC 6838, including the naming requirements in its section 4.2, and MAY be registered with IANA.

  To set an effectively null or empty config and maintain portability see the guidance for an empty descriptor below, and `DescriptorEmptyJSON` of the reference code.

- `layers` *array of objects*

  Each item in the array MUST be a descriptor. For portability, `layers` SHOULD have at least one entry. See the guidance for an empty descriptor below, and `DescriptorEmptyJSON` of the reference code.

  When the `config.mediaType` is set to `application/vnd.oci.image.config.v1+json`, the following additional restrictions apply:

  - The array MUST have the base layer at index 0.
  - Subsequent layers MUST then follow in stack order (i.e. from `layers[0]` to `layers[len(layers)-1]`).
  - The final filesystem layout MUST match the result of applying the layers to an empty directory.
  - The ownership, mode, and other attributes of the initial empty directory are unspecified.

  Beyond the descriptor requirements, the value has the following additional restrictions:

  - `mediaType` *string*

    This descriptor property has additional restrictions for `layers[]`. Implementations MUST support at least the following media types:

    * `application/vnd.oci.image.layer.v1.tar`
    * `application/vnd.oci.image.layer.v1.tar+gzip`

* `application/vnd.oci.image.layer.nondistributable.v1.tar`
* `application/vnd.oci.image.layer.nondistributable.v1.tar+gzip`

Manifests concerned with portability SHOULD use one of the above media types. Implementations storing or copying image manifests MUST NOT error on encountering a `mediaType` that is unknown to the implementation.

Entries in this field will frequently use the `+gzip` types.

If the manifest uses a different media type than the above, it MUST comply with RFC 6838, including the naming requirements in its section 4.2, and MAY be registered with IANA.

See Guidelines for Artifact Usage for other uses of the `layers`.

- **`subject`** *descriptor*

  This OPTIONAL property specifies a descriptor of another manifest. This value, used by the `referrers` API, indicates a relationship to the specified manifest.

- **`annotations`** *string-string map*

  This OPTIONAL property contains arbitrary metadata for the image manifest. This OPTIONAL property MUST use the annotation rules.

  See Pre-Defined Annotation Keys.

Example Image Manifest

Example showing an image manifest:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "digest": "sha256:b5b2b2c507a0944348e0303114d8d93aaaa081732b86451d9bce1f432a537bc7",
    "size": 7023
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0",
      "size": 32654
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:3c3a4604a545cdc127456d94e421cd355bca5b528f4a9c1905b15da2eb4a4c6b",
      "size": 16724
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:ec4b8955958665577945c89419d1af06b5f7636b4ac3da7f12184802ad867736",
      "size": 73109
    }
  ],
  "subject": {
    "mediaType": "application/vnd.oci.image.manifest.v1+json",
    "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270",
    "size": 7682
  },
  "annotations": {
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

Guidance for an Empty Descriptor

Implementers note: The following is considered GUIDANCE for portability.

Parts of the spec necessitate including a descriptor to a blob where some implementations of artifacts do not have associated content. While an empty blob (`size` of 0) may be preferable, practice has shown that not to be ubiquitously supported. The media type `application/vnd.oci.empty.v1+json` (`MediaTypeEmptyJSON`) has been specified for a descriptor that has no content for the implementation. The blob payload is the most minimal content that is still a valid JSON object: `{}` (`size` of 2). The blob digest of `{}` is `sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a`. The data field is optional, and if included is the base64 encoding of `{}`: `e30=`.

The resulting descriptor shown here is also defined in reference code as `DescriptorEmptyJSON`:

```
{
  "mediaType": "application/vnd.oci.empty.v1+json",
  "digest": "sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a",
  "size": 2,
  "data": "e30="
}
```

Guidelines for Artifact Usage

Content other than OCI container images MAY be packaged using the image manifest. When this is done, the `config.mediaType` value MUST be set to a value specific to the artifact type or the empty value. If the `config.mediaType` is set to the empty value, the `artifactType` MUST be defined. If the artifact does not need layers, a single layer SHOULD be included with a non-zero size. The suggested content for an unused `layers` array is the empty descriptor.

The design of the artifact depends on what content is being packaged with the artifact. The decision tree below and the associated examples MAY be used to design new artifacts:

1. Does the artifact consist of at least one file or blob? If yes, continue to 2. If no, specify the `artifactType`, and set the `config` and a single `layers` element to the empty descriptor value. Here is an example of this with annotations included:

   ```
   {
     "schemaVersion": 2,
     "mediaType": "application/vnd.oci.image.manifest.v1+json",
     "artifactType": "application/vnd.example+type",
     "config": {
       "mediaType": "application/vnd.oci.empty.v1+json",
       "digest": "sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a",
       "size": 2
     },
     "layers": [
       {
         "mediaType": "application/vnd.oci.empty.v1+json",
         "digest": "sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a",
         "size": 2
       }
     ],
     "annotations": {
       "oci.opencontainers.image.created": "2023-01-02T03:04:05Z",
       "com.example.data": "payload"
     }
   }
   ```

2. Does the artifact have additional JSON formatted metadata as configuration? If yes, continue to 3. If no, specify the `artifactType`, include the artifact in the `layers`, and set `config` to the empty descriptor value. Here is an example of this with a single layer:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "artifactType": "application/vnd.example+type",
  "config": {
    "mediaType": "application/vnd.oci.empty.v1+json",
    "digest": "sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a",
    "size": 2
  },
  "layers": [
    {
      "mediaType": "application/vnd.example+type",
      "digest": "sha256:e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317",
      "size": 1234
    }
  ]
}
```

3. For artifacts with a config blob, specify the `artifactType` to a common value for your artifact tooling, specify the `config` with the metadata for this artifact, and include the artifact in the `layers`. Here is an example of this:

OCI Image Index Specification

The image index is a higher-level manifest which points to specific image manifests, ideal for one or more platforms. While the use of an image index is OPTIONAL for image providers, image consumers SHOULD be prepared to process them.

This section defines the `application/vnd.oci.image.index.v1+json` media type.

For the media type(s) that this document is compatible with, see the matrix.

Image Index Property Descriptions

- **`schemaVersion`** *int*

  This REQUIRED property specifies the image manifest schema version. For this version of the specification, this MUST be `2` to ensure backward compatibility with older versions of Docker. The value of this field will not change. This field MAY be removed in a future version of the specification.

- **`mediaType`** *string*

  This property SHOULD be used and remain compatible with earlier versions of this specification and with other similar external formats. When used, this field MUST contain the media type `application/vnd.oci.image.index.v1+json`. This field usage differs from the descriptor use of `mediaType`.

- **`artifactType`** *string*

  This OPTIONAL property contains the type of an artifact when the manifest is used for an artifact. If defined, the value MUST comply with RFC 6838, including the naming requirements in its section 4.2, and MAY be registered with IANA.

- **`manifests`** *array of objects*

  This REQUIRED property contains a list of manifests for specific platforms. While this property MUST be present, the size of the array MAY be zero.

  Each object in `manifests` includes a set of descriptor properties with the following additional properties and restrictions:

  - **`mediaType`** *string*

    This descriptor property has additional restrictions for `manifests`. Implementations MUST support at least the following media types:

    * `application/vnd.oci.image.manifest.v1+json`

    Also, implementations SHOULD support the following media types:

    * `application/vnd.oci.image.index.v1+json` (nested index)

    Image indexes concerned with portability SHOULD use one of the above media types. Future versions of the spec MAY use a different mediatype (i.e. a new versioned format). An encountered `mediaType` that is unknown to the implementation MUST NOT generate an error.

  - `platform` *object*

    This OPTIONAL property describes the minimum runtime requirements of the image. This property SHOULD be present if its target is platform-specific.

    * `architecture` *string*
      This REQUIRED property specifies the CPU architecture. Image indexes SHOULD use, and implementations SHOULD understand, values listed in the Go Language document for `GOARCH`.
    * `os` *string*
      This REQUIRED property specifies the operating system. Image indexes SHOULD use, and implementations SHOULD understand, values listed in the Go Language document for `GOOS`.
    * `os.version` *string*
      This OPTIONAL property specifies the version of the operating system targeted by the referenced blob. Implementations MAY refuse to use manifests where `os.version` is not known to work with the host OS version. Valid values are implementation-defined. e.g. `10.0.14393.1066` on `windows`.
    * `os.features` *array of strings*
      This OPTIONAL property specifies an array of strings, each specifying a mandatory OS feature. When `os` is `windows`, image indexes SHOULD use, and implementations SHOULD understand the following values:
      · `win32k`: image requires `win32k.sys` on the host (Note: `win32k.sys` is missing on Nano Server)

      When `os` is not `windows`, values are implementation-defined and SHOULD be submitted to this specification for standardization.
    * `variant` *string*
      This OPTIONAL property specifies the variant of the CPU. Image indexes SHOULD use, and implementations SHOULD understand, `variant` values listed in the Platform Variants table.
    * `features` *array of strings*
      This property is RESERVED for future versions of the specification.

  If multiple manifests match a client or runtime's requirements, the first matching entry SHOULD be used.

- `subject` *descriptor*

  This OPTIONAL property specifies a descriptor of another manifest. This value, used by the `referrers` API, indicates a relationship to the specified manifest.

- `annotations` *string-string map*

  This OPTIONAL property contains arbitrary metadata for the image index. This OPTIONAL property MUST use the annotation rules.

  See Pre-Defined Annotation Keys.

Platform Variants

When the variant of the CPU is not listed in the table, values are implementation-defined and SHOULD be submitted to this specification for standardization.

ISA/ABI	`architecture`	`variant`
ARM 32-bit, v6	`arm`	`v6`
ARM 32-bit, v7	`arm`	`v7`
ARM 32-bit, v8	`arm`	`v8`
ARM 64-bit, v8	`arm64`	`v8`

Example Image Index

Example showing a simple image index pointing to image manifests for two platforms:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.index.v1+json",
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab815ad7fc331f",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7682,
      "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ],
  "annotations": {
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

Example Image Index with multiple media types

Example showing an image index pointing to manifests with multiple media types:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.index.v1+json",
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab815ad7fc331f",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.oci.image.index.v1+json",
      "size": 7682,
      "digest": "sha256:601570aaff1b68a61eb9c85b8beca1644e698003e0cdb5bce960f193d265a8b7"
    }
  ],
  "annotations": {
```

```
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

Image Layer Filesystem Changeset

This document describes how to serialize a filesystem and filesystem changes like removed files into a blob called a layer. One or more layers are applied on top of each other to create a complete filesystem. This document will use a concrete example to illustrate how to create and consume these filesystem layers.

This section defines the `application/vnd.oci.image.layer.v1.tar`, `application/vnd.oci.image.layer.v1.tar+gzip`, `application/vnd.oci.image.layer.v1.tar+zstd`, `application/vnd.oci.image.layer.nondistributable.v1.tar`, `application/vnd.oci.image.layer.nondistributable.v1.tar+gzip`, and `application/vnd.oci.image.layer.nondistributab` media types.

`+gzip` Media Types

- The media type `application/vnd.oci.image.layer.v1.tar+gzip` represents an `application/vnd.oci.image.layer.v1.tar` payload which has been compressed with gzip.
- The media type `application/vnd.oci.image.layer.nondistributable.v1.tar+gzip` represents an `application/vnd.oci.i` payload which has been compressed with gzip.

`+zstd` Media Types

- The media type `application/vnd.oci.image.layer.v1.tar+zstd` represents an `application/vnd.oci.image.layer.v1.tar` payload which has been compressed with zstd.
- The media type `application/vnd.oci.image.layer.nondistributable.v1.tar+zstd` represents an `application/vnd.oci.i` payload which has been compressed with zstd.

Distributable Format

- Layer Changesets for the media type `application/vnd.oci.image.layer.v1.tar` MUST be packaged in tar archive.
- Layer Changesets for the media type `application/vnd.oci.image.layer.v1.tar` MUST NOT include duplicate entries for file paths in the resulting tar archive.

Change Types

Types of changes that can occur in a changeset are:

- Additions
- Modifications
- Removals

Additions and Modifications are represented the same in the changeset tar archive.

Removals are represented using ”whiteout” file entries (See Representing Changes).

File Types

Throughout this document section, the use of word ”files” or ”entries” includes the following, where supported:

- regular files

- directories
- sockets
- symbolic links
- block devices
- character devices
- FIFOs

File Attributes

Where supported, MUST include file attributes for Additions and Modifications include:

- Modification Time (`mtime`)
- User ID (`uid`)
  - User Name (`uname`) *secondary to* ***`uid`***
- Group ID (`gid`)
  - Group Name (`gname`) *secondary to* ***`gid`***
- Mode (`mode`)
- Extended Attributes (`xattrs`)
- Symlink reference (`linkname` + symbolic link type)
- Hardlink reference (`linkname`)

Sparse files SHOULD NOT be used because they lack consistent support across tar implementations.

Hardlinks

- Hardlinks are a POSIX concept for having one or more directory entries for the same file on the same device.
- Not all filesystems support hardlinks (e.g. FAT).
- Hardlinks are possible with all file types except `directories`.
- Non-directory files are considered "hardlinked" when their link count is greater than 1.
- Hardlinked files are on a same device (i.e. comparing Major:Minor pair) and have the same inode.
- The corresponding files that share the link with the > 1 linkcount may be outside the directory that the changeset is being produced from, in which case the `linkname` is not recorded in the changeset.
- Hardlinks are stored in a tar archive with type of a `1` char, per the GNU Basic Tar Format and libarchive tar(5).
- While approaches to deriving new or changed hardlinks may vary, a possible approach is:

```
SET LinkMap to map[< Major:Minor String >]map[< inode integer >]< path string >
SET LinkNames to map[< src path string >]< dest path string >
FOR each path in root path
  IF path type is directory
    CONTINUE
  ENDIF
  SET filestat to stat(path)
  IF filestat num of links == 1
    CONTINUE
  ENDIF
  IF LinkMap[filestat device][filestat inode] is not empty
    SET LinkNames[path] to LinkMap[filestat device][filestat inode]
  ELSE
    SET LinkMap[filestat device][filestat inode] to path
  ENDIF
END FOR
```

With this approach, the link map and links names of a directory could be compared against that of another directory to derive additions and changes to hardlinks.

Platform-specific attributes

Implementations on Windows MUST support these additional attributes, encoded in PAX vendor extensions as follows:

- Windows file attributes (`MSWINDOWS.fileattr`)
- Security descriptor (`MSWINDOWS.rawsd`): base64-encoded self-relative binary security descriptor
- Mount points (`MSWINDOWS.mountpoint`): if present on a directory symbolic link, then the link should be created as a directory junction
- Creation time (`LIBARCHIVE.creationtime`)

Creating

Initial Root Filesystem

The initial root filesystem is the base or parent layer.

For this example, an image root filesystem has an initial state as an empty directory. The name of the directory is not relevant to the layer itself, only for the purpose of producing comparisons.

Here is an initial empty directory structure for a changeset, with a unique directory name `rootfs-c9d-v1`.

```
rootfs-c9d-v1/
```

Populate Initial Filesystem

Files and directories are then created:

```
rootfs-c9d-v1/
  etc/
    my-app-config
  bin/
    my-app-binary
    my-app-tools
```

The `rootfs-c9d-v1` directory is then created as a plain tar archive with relative path to `rootfs-c9d-v1`. Entries for the following files:

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

Populate a Comparison Filesystem

Create a new directory and initialize it with a copy or snapshot of the prior root filesystem. Example commands that can preserve file attributes to make this copy are:

- cp(1): `cp -a rootfs-c9d-v1/ rootfs-c9d-v1.s1/`
- rsync(1): `rsync -aHAX rootfs-c9d-v1/ rootfs-c9d-v1.s1/`
- tar(1): `mkdir rootfs-c9d-v1.s1 && tar --acls --xattrs -C rootfs-c9d-v1/ -c . | tar -C rootfs-c9d-v1.s1/ --acls --xattrs -x` (including `--selinux` where supported)

Any changes to the snapshot MUST NOT change or affect the directory it was copied from.

For example `rootfs-c9d-v1.s1` is an identical snapshot of `rootfs-c9d-v1`. In this way `rootfs-c9d-v1.s1` is prepared for updates and alterations.

Implementor's Note: *a copy-on-write or union filesystem can efficiently make directory snapshots*

Initial layout of the snapshot:

```
rootfs-c9d-v1.s1/
  etc/
    my-app-config
  bin/
    my-app-binary
    my-app-tools
```

See Change Types for more details on changes.

For example, add a directory at `/etc/my-app.d` containing a default config file, removing the existing config file. Also a change (in attribute or file content) to `./bin/my-app-tools` binary to handle the config layout change.

Following these changes, the representation of the `rootfs-c9d-v1.s1` directory:

```
rootfs-c9d-v1.s1/
  etc/
    my-app.d/
      default.cfg
  bin/
    my-app-binary
    my-app-tools
```

Determining Changes

When two directories are compared, the relative root is the top-level directory. The directories are compared, looking for files that have been added, modified, or removed.

For this example, `rootfs-c9d-v1/` and `rootfs-c9d-v1.s1/` are recursively compared, each as relative root path.

The following changeset is found:

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

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.

Representing Changes

A tar archive is then created which contains *only* this changeset:

- Added and modified files and directories in their entirety
- Deleted files or directories marked with a whiteout file

The resulting tar archive for `rootfs-c9d-v1.s1` has the following entries:

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

To signify that the resource `./etc/my-app-config` MUST be removed when the changeset is applied, the basename of the entry is prefixed with `.wh.`.

Applying Changesets

- Layer Changesets of media type `application/vnd.oci.image.layer.v1.tar` are *applied*, rather than simply extracted as tar archives.
- Applying a layer changeset requires special consideration for the whiteout files.
- In the absence of any whiteout files in a layer changeset, the archive is extracted like a regular tar archive.

Changeset over existing files

This section specifies applying an entry from a layer changeset if the target path already exists.

If the entry and the existing path are both directories, then the existing path's attributes MUST be replaced by those of the entry in the changeset. In all other cases, the implementation MUST do the semantic equivalent of the following:

- removing the file path (e.g. `unlink(2)` on Linux systems)
- recreating the file path, based on the contents and attributes of the changeset entry

Whiteouts

- A whiteout file is an empty file with a special filename that signifies a path should be deleted.
- A whiteout filename consists of the prefix `.wh.` plus the basename of the path to be deleted.
- As files prefixed with `.wh.` are special whiteout markers, it is not possible to create a filesystem which has a file or directory with a name beginning with `.wh.`.
- Once a whiteout is applied, the whiteout itself MUST also be hidden.
- Whiteout files MUST only apply to resources in lower/parent layers.
- Files that are present in the same layer as a whiteout file can only be hidden by whiteout files in subsequent layers.

The following is a base layer with several resources:

```
a/
a/b/
a/b/c/
a/b/c/bar
```

When the next layer is created, the original `a/b` directory is deleted and recreated with `a/b/c/foo`:

```
a/
a/.wh..wh..opq
a/b/
a/b/c/
a/b/c/foo
```

When processing the second layer, `a/.wh..wh..opq` is applied first, before creating the new version of `a/b`, regardless of the ordering in which the whiteout file was encountered. For example, the following layer is equivalent to the layer above:

```
a/
a/b/
a/b/c/
a/b/c/foo
a/.wh..wh..opq
```

Implementations SHOULD generate layers such that the whiteout files appear before sibling directory entries.

Opaque Whiteout

- In addition to expressing that a single entry should be removed from a lower layer, layers may remove all of the children using an opaque whiteout entry.
- An opaque whiteout entry is a file with the name `.wh..wh..opq` indicating that all siblings are hidden in the lower layer.

Let's take the following base layer as an example:

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

If all children of `bin/` are removed, the next layer would have the following:

```
bin/
  .wh..wh..opq
```

This is called *opaque whiteout* format. An *opaque whiteout* file hides *all* children of the `bin/` including sub-directories and all descendants. Using *explicit whiteout* files, this would be equivalent to the following:

```
bin/
  .wh.my-app-binary
  .wh.my-app-tools
  .wh.tools
```

In this case, a unique whiteout file is generated for each entry. If there were more children of `bin/` in the base layer, there would be an entry for each. Note that this opaque file will apply to *all* children, including sub-directories, other resources and all descendants.

Implementations SHOULD generate layers using *explicit whiteout* files, but MUST accept both.

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

Non-Distributable Layers

> **NOTE**: Non-distributable layers are deprecated, and not recommended for future use. Implementations SHOULD NOT produce new non-distributable layers.

Due to legal requirements, certain layers may not be regularly distributable. Such "non-distributable" layers are typically downloaded directly from a distributor but never uploaded.

Non-distributable layers SHOULD be tagged with an alternative mediatype of `application/vnd.oci.image.layer.nondistributab`
Implementations SHOULD NOT upload layers tagged with this media type; however, such a media type SHOULD NOT affect whether an implementation downloads the layer.

Descriptors referencing non-distributable layers MAY include `urls` for downloading these layers directly; however, the presence of the `urls` field SHOULD NOT be used to determine whether or not a layer is non-distributable.

OCI Image Configuration

An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers.

This section defines the `application/vnd.oci.image.config.v1+json` media type.

Terminology

This specification uses the following terms:

Layer

- Image filesystems are composed of *layers*.
- Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
- Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular 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 JSON

- Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
- The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
- This JSON is considered to be immutable, because changing it would change the computed ImageID.
- Changing it means creating a new derived image, instead of changing the existing image.

Layer DiffID

A layer DiffID is the digest over the layer's uncompressed tar archive and serialized in the descriptor digest format, e.g., `sha256:a9561eb1b190625c9adb5a9513e72c4dedafc1cb2d4c5236c9a6957ec7dfd5a9`. Layers SHOULD be packed and unpacked reproducibly to avoid changing the layer DiffID, for example by using tar-split to save the tar headers.

NOTE: Do not confuse DiffIDs with layer digests, often referenced in the manifest, which are digests over compressed or uncompressed content.

Layer ChainID

For convenience, it is sometimes useful to refer to a stack of layers with a single identifier. While a layer's `DiffID` identifies a single changeset, the `ChainID` identifies the subsequent application of those changesets. This ensures that we have handles referring to both the layer itself, as well as the result of the application of a series of changesets. Use in combination with `rootfs.diff_ids` while applying layers to a root filesystem to uniquely and safely identify the result.

Definition

The `ChainID` of an applied set of layers is defined with the following recursion:

```
ChainID(L ) =  DiffID(L )
ChainID(L |...|L  |L ) = Digest(ChainID(L |...|L  ) + " " + DiffID(L ))
```

For this, we define the binary `|` operation to be the result of applying the right operand to the left operand. For example, given base layer `A` and a changeset `B`, we refer to the result of applying `B` to `A` as `A|B`.

Above, we define the `ChainID` for a single layer (`L`) as equivalent to the `DiffID` for that layer. Otherwise, the `ChainID` for a set of applied layers (`L|...|L|L`) is defined as the recursion `Digest(ChainID(L|...|L) + " " + DiffID(L))`.

Explanation

Let's say we have layers A, B, C, ordered from bottom to top, where A is the base and C is the top. Defining `|` as a binary application operator, the root filesystem may be `A|B|C`. While it is implied that `C` is only useful when applied to `A|B`, the identifier `C` is insufficient to identify this result, as we'd have the equality `C = A|B|C`, which isn't true.

The main issue is when we have two definitions of `C`, `C = C` and `C = A|B|C`. If this is true (with some handwaving), `C = x|C` where `x = any application`. This means that if an attacker can define `x`, relying on `C` provides no guarantee that the layers were applied in any order.

The `ChainID` addresses this problem by being defined as a compound hash. **We differentiate the changeset `C`, from the order-dependent application `A|B|C` by saying that the resulting rootfs is identified by ChainID(A|B|C), which can be calculated by `ImageConfig.rootfs`.**

Let's expand the definition of `ChainID(A|B|C)` to explore its internal structure:

```
ChainID(A) = DiffID(A)
ChainID(A|B) = Digest(ChainID(A) + " " + DiffID(B))
ChainID(A|B|C) = Digest(ChainID(A|B) + " " + DiffID(C))
```

We can replace each definition and reduce to a single equality:

```
ChainID(A|B|C) = Digest(Digest(DiffID(A) + " " + DiffID(B)) + " " + DiffID(C))
```

Hopefully, the above is illustrative of the *actual* contents of the `ChainID`. Most importantly, we can easily see that `ChainID(C) != ChainID(A|B|C)`, otherwise, `ChainID(C) = DiffID(C)`, which is the base case, could not be true.

ImageID

Each image's ID is given by the SHA256 hash of its configuration JSON. It is represented as a hexadecimal encoding of 256 bits, e.g., `sha256:a9561eb1b190625c9adb5a9513e72c4dedafc1cb2d4c5236c9a6957ec7dfd5a9`. Since the configuration JSON that gets hashed references hashes of each layer in the image, this formulation of the ImageID makes images content-addressable.

Properties

Note: Any OPTIONAL field MAY also be set to null, which is equivalent to being absent.

- **created** *string*, OPTIONAL

  An combined date and time at which the image was created, formatted as defined by RFC 3339, section 5.6.

- **author** *string*, OPTIONAL

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

- **architecture** *string*, REQUIRED

  The CPU architecture which the binaries in this image are built to run on. Configurations SHOULD use, and implementations SHOULD understand, values listed in the Go Language document for `GOARCH`.

- **os** *string*, REQUIRED

  The name of the operating system which the image is built to run on. Configurations SHOULD use, and implementations SHOULD understand, values listed in the Go Language document for `GOOS`.

- **os.version** *string*, OPTIONAL

  This OPTIONAL property specifies the version of the operating system targeted by the referenced blob. Implementations MAY refuse to use manifests where `os.version` is not known to work with the host OS version. Valid values are implementation-defined. e.g. `10.0.14393.1066` on `windows`.

- **os.features** *array of strings*, OPTIONAL

  This OPTIONAL property specifies an array of strings, each specifying a mandatory OS feature. When `os` is `windows`, image indexes SHOULD use, and implementations SHOULD understand the following values:

  - `win32k`: image requires `win32k.sys` on the host (Note: `win32k.sys` is missing on Nano Server)

- **variant** *string*, OPTIONAL

  The variant of the specified CPU architecture. Configurations SHOULD use, and implementations SHOULD understand, `variant` values listed in the Platform Variants table.

- **config** *object*, OPTIONAL

  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.

  - **User** *string*, OPTIONAL

    The username or UID which is a platform-specific structure that allows specific control over which user the process run as. This acts as a default value to use when the value is not specified when creating a container. For Linux based systems, 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` and `/etc/group` from the container are applied. If `group`/`gid` is specified, supplementary groups from the container are ignored.

  - **ExposedPorts** *object*, OPTIONAL

    A set of ports to expose from a container running this image. 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. **NOTE:** 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.

  - **Env** *array of strings*, OPTIONAL

    Entries are in the format of `VARNAME=VARVALUE`. These values act as defaults and are merged with any specified when creating a container.

  - **Entrypoint** *array of strings*, OPTIONAL

    A list of arguments to use as the command to execute when the container starts. These values act as defaults and may be replaced by an entrypoint specified when creating a container.

  - **Cmd** *array of strings*, OPTIONAL

    Default arguments to the entrypoint of the container. These values act as defaults and may be replaced by 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** *object*, OPTIONAL

    A set of directories describing where the process is likely to write data specific to a container instance. **NOTE:** 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.

  - **WorkingDir** *string*, OPTIONAL

    Sets the current working directory of the entrypoint process in the container. This value acts as a default and may be replaced by a working directory specified when creating a container.

  - **Labels** *object*, OPTIONAL

    The field contains arbitrary metadata for the container. This property MUST use the annotation rules.

  - **StopSignal** *string*, OPTIONAL

    The field contains the system call signal that will be sent to the container to exit. The signal can be a signal name in the format `SIGNAME`, for instance `SIGKILL` or `SIGRTMIN+3`.

  - **ArgsEscaped** *boolean*, OPTIONAL

    `[Deprecated]` - This field is present only for legacy compatibility with Docker and should not be used by new image builders. It is used by Docker for Windows images to indicate that the `Entrypoint` or `Cmd` or both, contains only a single element array, that is a pre-escaped, and combined into a single string `CommandLine`. If `true` the value in `Entrypoint` or `Cmd` should be used as-is to avoid double escaping. Note, the exact behavior of `ArgsEscaped` is complex and subject to implementation details in Moby project.
  - **Memory** *integer*, OPTIONAL

    This property is *reserved* for use, to maintain compatibility.
  - **MemorySwap** *integer*, OPTIONAL

    This property is *reserved* for use, to maintain compatibility.
  - **CpuShares** *integer*, OPTIONAL

    This property is *reserved* for use, to maintain compatibility.
  - **Healthcheck** *object*, OPTIONAL

    This property is *reserved* for use, to maintain compatibility.

- **rootfs** *object*, REQUIRED

  The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash.

  - **type** *string*, REQUIRED

    MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image.
  - **diff_ids** *array of strings*, REQUIRED

    An array of layer content hashes (`DiffIDs`), in order from first to last.

- **history** *array of objects*, OPTIONAL

  Describes the history of each layer. The array is ordered from first to last. The object has the following fields:

  - **created** *string*, OPTIONAL

    A combined date and time at which the layer was created, formatted as defined by RFC 3339, section 5.6.
  - **author** *string*, OPTIONAL

    The author of the build point.
  - **created_by** *string*, OPTIONAL

    The command which created the layer.
  - **comment** *string*, OPTIONAL

    A custom message set when creating the layer.
  - **empty_layer** *boolean*, OPTIONAL

    This field is used to mark if the history item created a filesystem diff. It is set to true if this history item doesn't correspond to an actual layer in the rootfs section (for example, Dockerfile's ENV command results in no change to the filesystem).

Any extra fields in the Image JSON struct are considered implementation specific and MUST NOT generate an error by any implementations which are unable to interpret them.

Whitespace is OPTIONAL and implementations MAY have compact JSON with no whitespace.

Example

Here is an example image configuration JSON document:

```
{
    "created": "2015-10-31T22:22:56.015925234Z",
    "author": "Alyssa P. Hacker <alyspdev@example.com>",
    "architecture": "amd64",
    "os": "linux",
    "config": {
        "User": "alice",
        "ExposedPorts": {
            "8080/tcp": {}
        },
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
            "FOO=oci_is_a",
            "BAR=well_written_spec"
        ],
        "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",
        "Labels": {
            "com.example.project.git.url": "https://example.com/project.git",
            "com.example.project.git.commit": "45a939b2999782a3f005621a8d0f29aa387e1d6b"
        }
    },
    "rootfs": {
      "diff_ids": [
        "sha256:c6f988f4874bb0add23a778f753c65efe992244e148a1d2ec2a8b664fb66bbd1",
        "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef"
      ],
      "type": "layers"
    },
    "history": [
      {
        "created": "2015-10-31T22:22:54.690851953Z",
        "created_by": "/bin/sh -c #(nop) ADD file:a3bc1e842b69636f9df5256c49c5374fb4eef1e281fe3f282c65fb853ee1
      },
      {
        "created": "2015-10-31T22:22:55.613815829Z",
        "created_by": "/bin/sh -c #(nop) CMD [\"sh\"]",
        "empty_layer": true
      },
      {
        "created": "2015-10-31T22:22:56.329850019Z",
        "created_by": "/bin/sh -c apk add curl"
      }
    ]
}
```

Annotations

Several components of the specification, like Image Manifests and Descriptors, feature an optional annotations property, whose format is common and defined in this section.

This property contains arbitrary metadata.

Rules

- Annotations MUST be a key-value map where both the key and value MUST be strings.
- While the value MUST be present, it MAY be an empty string.
- Keys MUST be unique within this map, and best practice is to namespace the keys.
- Keys SHOULD be named using a reverse domain notation - e.g. `com.example.myKey`.
- The prefix `org.opencontainers` is reserved for keys defined in Open Container Initiative (OCI) specifications and MUST NOT be used by other specifications and extensions.
- Keys using the `org.opencontainers.image` namespace are reserved for use in the OCI Image Specification and MUST NOT be used by other specifications and extensions, including other OCI specifications.
- If there are no annotations then this property MUST either be absent or be an empty map.
- Consumers MUST NOT generate an error if they encounter an unknown annotation key.

Pre-Defined Annotation Keys

This specification defines the following annotation keys, intended for but not limited to image index, image manifest, and descriptor authors.

- **org.opencontainers.image.created** date and time on which the image was built, conforming to RFC 3339.
- **org.opencontainers.image.authors** contact details of the people or organization responsible for the image (freeform string)
- **org.opencontainers.image.url** URL to find more information on the image (string)
- **org.opencontainers.image.documentation** URL to get documentation on the image (string)
- **org.opencontainers.image.source** URL to get source code for building the image (string)
- **org.opencontainers.image.version** version of the packaged software
  - The version MAY match a label or tag in the source code repository
  - version MAY be Semantic versioning-compatible
- **org.opencontainers.image.revision** Source control revision identifier for the packaged software.
- **org.opencontainers.image.vendor** Name of the distributing entity, organization or individual.
- **org.opencontainers.image.licenses** License(s) under which contained software is distributed as an SPDX License Expression.
- **org.opencontainers.image.ref.name** Name of the reference for a target (string).
  - SHOULD only be considered valid when on descriptors on `index.json` within image layout.
  - Character set of the value SHOULD conform to alphanum of `A-Za-z0-9` and separator set of `-._:@/+`
  - The reference must match the following grammar:

    ```
    ref       ::= component ("/" component)*
    component ::= alphanum (separator alphanum)*
    alphanum  ::= [A-Za-z0-9]+
    separator ::= [-._:@+] | "--"
    ```

- **org.opencontainers.image.title** Human-readable title of the image (string)

- **org.opencontainers.image.description** Human-readable description of the software packaged in the image (string)
- **org.opencontainers.image.base.digest** Digest of the image this image is based on (string)
  - This SHOULD be the immediate image sharing zero-indexed layers with the image, such as from a Dockerfile `FROM` statement.
  - This SHOULD NOT reference any other images used to generate the contents of the image (e.g., multi-stage Dockerfile builds).
- **org.opencontainers.image.base.name** Image reference of the image this image is based on (string)
  - This SHOULD be image references in the format defined by distribution/distribution.
  - This SHOULD be a fully qualified reference name, without any assumed default registry. (e.g., `registry.example.com/my-` instead of `my-org/my-image:tag`).
  - This SHOULD be the immediate image sharing zero-indexed layers with the image, such as from a Dockerfile `FROM` statement.
  - This SHOULD NOT reference any other images used to generate the contents of the image (e.g., multi-stage Dockerfile builds).
  - If the `image.base.name` annotation is specified, the `image.base.digest` annotation SHOULD be the digest of the manifest referenced by the `image.ref.name` annotation.

Back-compatibility with Label Schema

Label Schema defined a number of conventional labels for container images, and these are now superceded by annotations with keys starting **org.opencontainers.image**.

While users are encouraged to use the **org.opencontainers.image** keys, tools MAY choose to support compatible annotations using the **org.label-schema** prefix as follows.

`org.opencontainers.image` prefix	`org.label-schema` prefix	Compatibility notes
`created`	`build-date`	Compatible
`url`	`url`	Compatible
`source`	`vcs-url`	Compatible
`version`	`version`	Compatible
`revision`	`vcs-ref`	Compatible
`vendor`	`vendor`	Compatible
`title`	`name`	Compatible
`description`	`description`	Compatible
`documentation`	`usage`	Value is compatible if the documentation is located by a URL
`authors`		No equivalent in Label Schema
`licenses`		No equivalent in Label Schema
`ref.name`		No equivalent in Label Schema
	`schema-version`	No equivalent in the OCI Image Spec
	`docker.*`, `rkt.*`	No equivalent in the OCI Image Spec

Conversion to OCI Runtime Configuration

When extracting an OCI Image into an OCI Runtime bundle, two orthogonal components of the extraction are relevant:

1. Extraction of the root filesystem from the set of filesystem layers.
2. Conversion of the image configuration blob to an OCI Runtime configuration blob.

This section defines how to convert an `application/vnd.oci.image.config.v1+json` blob to an OCI runtime configuration blob (the latter component of extraction). The former component of extraction is defined elsewhere and is orthogonal to configuration of a runtime bundle. The values of runtime configuration properties not specified by this document are implementation-defined.

A converter MUST rely on the OCI image configuration to build the OCI runtime configuration as described by this document; this will create the "default generated runtime configuration".

The "default generated runtime configuration" MAY be overridden or combined with externally provided inputs from the caller. In addition, a converter MAY have its own implementation-defined defaults and extensions which MAY be combined with the "default generated runtime configuration". The restrictions in this document refer only to combining implementation-defined defaults with the "default generated runtime configuration". Externally provided inputs are considered to be a modification of the `application/vnd.oci.image.config.v1+json` used as a source, and such modifications have no restrictions.

For example, externally provided inputs MAY cause an environment variable to be added, removed or changed. However an implementation-defined default SHOULD NOT result in an environment variable being removed or changed.

Verbatim Fields

Certain image configuration fields have an identical counterpart in the runtime configuration. Some of these are purely annotation-based fields, and have been extracted into a separate subsection. A compliant configuration converter MUST extract the following fields verbatim to the corresponding field in the generated runtime configuration:

Image Field	Runtime Field	Notes
`Config.WorkingDir`	`process.cwd`	
`Config.Env`	`process.env`	1
`Config.Entrypoint`	`process.args`	2
`Config.Cmd`	`process.args`	2

1. The converter MAY add additional entries to `process.env` but it SHOULD NOT add entries that have variable names present in `Config.Env`.
2. If both `Config.Entrypoint` and `Config.Cmd` are specified, the converter MUST append the value of `Config.Cmd` to the value of `Config.Entrypoint` and set `process.args` to that combined value.

Annotation Fields

These fields all affect the `annotations` of the runtime configuration, and are thus subject to precedence.

Image Field	Runtime Field	Notes
`os`	`annotations`	1,2
`architecture`	`annotations`	1,3
`variant`	`annotations`	1,4
`os.version`	`annotations`	1,5
`os.features`	`annotations`	1,6
`author`	`annotations`	1,7
`created`	`annotations`	1,8
`Config.Labels`	`annotations`	
`Config.StopSignal`	`annotations`	1,9

1. If a user has explicitly specified this annotation with `Config.Labels`, then the value specified in this field takes lower precedence and the converter MUST instead use the value from `Config.Labels`.
2. The value of this field MUST be set as the value of `org.opencontainers.image.os` in `annotations`.
3. The value of this field MUST be set as the value of `org.opencontainers.image.architecture` in `annotations`.
4. The value of this field MUST be set as the value of `org.opencontainers.image.variant` in `annotations`.
5. The value of this field MUST be set as the value of `org.opencontainers.image.os.version` in `annotations`.
6. The value of this field MUST be set as the value of `org.opencontainers.image.os.features` in `annotations`.
7. The value of this field MUST be set as the value of `org.opencontainers.image.author` in `annotations`.
8. The value of this field MUST be set as the value of `org.opencontainers.image.created` in `annotations`.
9. The value of this field MUST be set as the value of `org.opencontainers.image.stopSignal` in `annotations`.

Parsed Fields

Certain image configuration fields have a counterpart that must first be translated. A compliant configuration converter SHOULD parse all of these fields and set the corresponding fields in the generated runtime configuration:

Image Field	Runtime Field
`Config.User`	`process.user.*`

The method of parsing the above image fields are described in the following sections.

`Config.User`

If the values of `user` or `group` in `Config.User` are numeric (`uid` or `gid`) then the values MUST be copied verbatim to `process.user.uid` and `process.user.gid` respectively. If the values of `user` or `group` in `Config.User` are not numeric (`user` or `group`) then a converter SHOULD resolve the user information using a method appropriate for the container's context. For Unix-like systems, this MAY involve resolution through NSS or parsing `/etc/passwd` from the extracted container's root filesystem to determine the values of `process.user.uid` and `process.user.gid`.

In addition, a converter SHOULD set the value of `process.user.additionalGids` to a value corresponding to the user in the container's context described by `Config.User`. For Unix-like systems, this MAY involve resolution through NSS or parsing `/etc/group` and determining the group memberships of the user specified in `process.user.uid`. The converter SHOULD NOT modify `process.user.additionalGids` if the value of `user` in `Config.User` is numeric or if `Config.User` specifies a group.

If `Config.User` is not defined, the converted `process.user` value is implementation-defined. If `Config.User` does not correspond to a user in the container's context, the converter MUST return an error.

Optional Fields

Certain image configuration fields are not applicable to all conversion use cases, and thus are optional for configuration converters to implement. A compliant configuration converter SHOULD provide a way for users to extract these fields into the generated runtime configuration:

Image Field	Runtime Field	Notes
`Config.ExposedPorts`	`annotations`	1
`Config.Volumes`	`mounts`	2

1. The runtime configuration does not have a corresponding field for this image field. However, converters SHOULD set the `org.opencontainers.image.exposedPorts` annotation.
2. Implementations SHOULD provide mounts for these locations such that application data is not written to the container's root filesystem. If a converter implements conversion for this field using mountpoints, it SHOULD set the `destination` of the mountpoint to the value specified in `Config.Volumes`. An implementation MAY seed the contents of the mount with data in the image at the same location. If a *new* image is created from a container based on the image described by this configuration, data in these paths SHOULD NOT be included in the *new* image. The other `mounts` fields are platform and context dependent, and thus are implementation-defined. Note that the implementation of `Config.Volumes` need not use mountpoints, as it is effectively a mask of the filesystem.

`Config.ExposedPorts`

The OCI runtime configuration does not provide a way of expressing the concept of "container exposed ports". However, converters SHOULD set the **org.opencontainers.image.exposedPorts** annotation, unless doing so will cause a conflict.

org.opencontainers.image.exposedPorts is the list of values that correspond to the keys defined for `Config.ExposedPorts` (string, comma-separated values).

Annotations

There are three ways of annotating an OCI image in this specification:

1. `Config.Labels` in the configuration of the image.
2. `annotations` in the manifest of the image.
3. `annotations` in the image index of the image.

In addition, there are also implicit annotations that are defined by this section which are determined from the values of the image configuration. A converter SHOULD NOT attempt to extract annotations from manifests or image indices. If there is a conflict (same key but different value) between an implicit annotation (or annotation in manifests or image indices) and an explicitly specified annotation in `Config.Labels`, the value specified in `Config.Labels` MUST take precedence.

A converter MAY add annotations which have keys not specified in the image. A converter MUST NOT modify the values of annotations specified in the image.

Considerations

Extensibility

Implementations storing or copying content MUST NOT modify or alter the content in a way that would change the digest of the content. Examples of these implementations include:

- A registry implementing the distribution specification, including local registries, caching proxies
- An application which copies content to disk or between registries

Implementations processing content SHOULD NOT generate an error if they encounter an unknown property in a known media type. Examples of these implementations include:

- A [runtime implementing the runtime specification][runtime-spec]
- An implementation using OCI to retrieve and utilize artifacts, e.g.: a WASM runtime

Canonicalization

- OCI Images are content-addressable. See descriptors for more.
- One benefit of content-addressable storage is easy deduplication.
- Many images might depend on a particular layer, but there will only be one blob in the store.
- With a different serialization, that same semantic layer would have a different hash, and if both versions of the layer are referenced there will be two blobs with the same semantic content.
- To allow efficient storage, implementations serializing content for blobs SHOULD use a canonical serialization.
- This increases the chance that different implementations can push the same semantic content to the store without creating redundant blobs.

JSON

JSON content SHOULD be serialized as canonical JSON. Of the OCI Image Format Specification media types, all the types ending in `+json` contain JSON content. Implementations:

- Go: github.com/docker/go, which claims to implement canonical JSON except for Unicode normalization.

EBNF

For field formats described in this specification, we use a limited subset of Extended Backus-Naur Form, similar to that used by the [XML specification][xmlebnf]. Grammars present in the OCI specification are regular and can be converted to a single regular expressions. However, regular expressions are avoided to limit ambiguity between regular expression syntax. By defining a subset of EBNF used here, the possibility of variation, misunderstanding or ambiguities from linking to a larger specification can be avoided.

Grammars are made up of rules in the following form:

```
symbol ::= expression
```

We can say we have the production identified by symbol if the input is matched by the expression. Whitespace is completely ignored in rule definitions.

Expressions

The simplest expression is the literal, surrounded by quotes:

```
literal ::= "matchthis"
```

The above expression defines a symbol, ”literal”, that matches the exact input of ”matchthis”. Character classes are delineated by brackets (`[]`), describing either a set, range or multiple range of characters:

```
set := [abc]
range := [A-Z]
```

The above symbol ”set” would match one character of either ”a”, ”b” or ”c”. The symbol ”range” would match any character, ”A” to ”Z”, inclusive. Currently, only matching for 7-bit ascii literals and character classes is defined, as that is all that is required by this specification. Multiple character ranges and explicit characters can be specified in a single character classes, as follows:

```
multipleranges := [a-zA-Z=-]
```

The above matches the characters in the range `A` to `Z`, `a` to `z` and the individual characters `-` and `=`.

Expressions can be made up of one or more expressions, such that one must be followed by the other. This is known as an implicit concatenation operator. For example, to satisfy the following rule, both `A` and `B` must be matched to satisfy the rule:

```
symbol ::= A B
```

Each expression must be matched once and only once, `A` followed by `B`. To support the description of repetition and optional match criteria, the postfix operators `*` and `+` are defined. `*` indicates that the preceding expression can be matched zero or more times. `+` indicates that the preceding expression must be matched one or more times. These appear in the following form:

```
zeroormore ::= expression*
oneormore ::= expression+
```

Parentheses are used to group expressions into a larger expression:

```
group ::= (A B)
```

Like simpler expressions above, operators can be applied to groups, as well. To allow for alternates, we also define the infix operator `|`.

```
oneof ::= A | B
```

The above indicates that the expression should match one of the expressions, `A` or `B`.

Precedence

The operator precedence is in the following order:

- Terminals (literals and character classes)
- Grouping `()`
- Unary operators `+*`
- Concatenation
- Alternates `|`

The precedence can be better described using grouping to show equivalents. Concatenation has higher precedence than alternates, such `A B | C D` is equivalent to `(A B) | (C D)`. Unary operators have higher precedence than alternates and concatenation, such that `A+ | B+` is equivalent to `(A+) | (B+)`.

Examples

The following combines the previous definitions to match a simple, relative path name, describing the individual components:

```
path      ::= component ("/" component)*
component ::= [a-z]+
```

The production "component" is one or more lowercase letters. A "path" is then at least one component, possibly followed by zero or more slash-component pairs. The above can be converted into the following regular expression:

```
[a-z]+(?:/[a-z]+)*
```

[runtime-

OCI Image Implementations

Projects or Companies currently adopting the OCI Image Specification

- projectatomic/skopeo
- Amazon Elastic Container Registry (ECR) (announcement)
- Azure Container Registry (ACR)
- openSUSE/umoci
- cloudfoundry/grootfs (source)
- Mesos plans (design doc)
- Docker
  - docker/docker (`docker save/load` WIP)
  - distribution/distribution (registry PR)
- containerd/containerd
- Containers
  - containers/build
  - containers/image
  - containers/oci-spec-rs
  - containers/libocispec
- krustlet/oci-distribution
- coreos/rkt
- box-builder/box
- coolljt0725/docker2oci
- regclient/regclient

(to add your project please open a pull-request)