Open Container Initiative Distribution Specification

Open Container Initiative Distribution Specification

Table of Contents

Overview

Introduction

The **Open Container Initiative Distribution Specification** (a.k.a. "OCI Distribution Spec") defines an API protocol to facilitate and standardize the distribution of content.

While OCI Image is the most prominent, the specification is designed to be agnostic of content types. Concepts such as "manifests" and "digests", are currently defined in the Open Container Initiative Image Format Specification (a.k.a. "OCI Image Spec").

To support other artifact types, please see the Open Container Initiative Artifact Authors Guide (a.k.a. "OCI Artifacts").

Historical Context

The spec is based on the specification for the Docker Registry HTTP API V2 protocol apdx-1.

For relevant details and a history leading up to this specification, please see the following issues:

- moby/moby#8093
- moby/moby#9015
- docker/docker-registry#612

Legacy Docker support HTTP headers

Because of the origins this specification, the client MAY encounter Docker-specific headers, such as `Docker-Content-Digest`, or `Docker-Distribution-API-Version`. These headers are OPTIONAL and clients SHOULD NOT depend on them.

Definitions

Several terms are used frequently in this document and warrant basic definitions:

- **Registry**: a service that handles the required APIs defined in this specification
- **Client**: a tool that communicates with Registries
- **Push**: the act of uploading Blobs and Manifests to a Registry
- **Pull**: the act of downloading Blobs and Manifests from a Registry
- **Blob**: the binary form of content that is stored by a Registry, addressable by a Digest
- **Manifest**: a JSON document which defines an Artifact. Manifests are defined under the OCI Image Spec apdx-2
- **Config**: a blob referenced in the Manifest which contains Artifact metadata. Config is defined under the OCI Image Spec apdx-4
- **Artifact**: one conceptual piece of content stored as Blobs with an accompanying Manifest containing a Config
- **Digest**: a unique identifier created from a cryptographic hash of a Blob's content. Digests are defined under the OCI Image Spec apdx-3
- **Tag**: a custom, human-readable Manifest identifier

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).

Use Cases

Artifact Verification

A container engine would like to run verified image named "library/ubuntu", with the tag "latest". The engine contacts the registry, requesting the manifest for "library/ubuntu:latest". An untrusted registry returns a manifest. After each layer is downloaded, the engine verifies the digest of the layer, ensuring that the content matches that specified by the manifest.

Resumable Push

Company X's build servers lose connectivity to a distribution endpoint before completing an artifact layer transfer. After connectivity returns, the build server attempts to re-upload the artifact. The registry notifies the build server that the upload has already been partially attempted. The build server responds by only sending the remaining data to complete the artifact file.

Resumable Pull

Company X is having more connectivity problems but this time in their deployment datacenter. When downloading an artifact, the connection is interrupted before completion. The client keeps the partial data and uses http `Range` requests to avoid downloading repeated data.

Layer Upload De-duplication

Company Y's build system creates two identical layers from build processes A and B. Build process A completes uploading the layer before B. When process B attempts to upload the layer, the registry indicates that its not necessary because the layer is already known.

If process A and B upload the same layer at the same time, both operations will proceed and the first to complete will be stored in the registry. Even in the case where both uploads are accepted, the registry may securely only store one copy of the layer since the computed digests match.

Conformance

For more information on testing for conformance, please see the conformance README

Official Certification

Registry providers can self-certify by submitting conformance results to opencontainers/oci-conformance.

Requirements

Registry conformance applies to the following workflow categories:

1. **Pull** - Clients are able to pull from the registry
2. **Push** - Clients are able to push to the registry
3. **Content Discovery** - Clients are able to list or otherwise query the content stored in the registry
4. **Content Management** - Clients are able to control the full life-cycle of the content stored in the registry

All registries conforming to this specification MUST support, at a minimum, all APIs in the **Pull** category.

Registries SHOULD also support the **Push**, **Content Discovery**, and **Content Management** categories. A registry claiming conformance with one of these specification categories MUST implement all APIs in the claimed category.

In order to test a registry's conformance against these workflow categories, please use the conformance testing tool.

Workflow Categories

Pull

The process of pulling an artifact centers around retrieving two components: the manifest and one or more blobs.

Typically, the first step in pulling an artifact is to retrieve the manifest. However, you MAY retrieve content from the registry in any order.

Pulling manifests

To pull a manifest, perform a `GET` request to a URL in the following form: `/v2/<name>/manifests/<reference>` end-3

`<name>` refers to the namespace of the repository. `<reference>` MUST be either (a) the digest of the manifest or (b) a tag name. The `<reference>` MUST NOT be in any other format. Throughout this document, `<name>` MUST match the following regular expression:

`[a-z0-9]+([._-][a-z0-9]+)*(/[a-z0-9]+([._-][a-z0-9]+)*)*`

The client SHOULD include an `Accept` header indicating which manifest content types it supports. In a successful response, the `Content-Type` header will indicate the type of the returned manifest. For more information on the use of `Accept` headers and content negotiation, please see Content Negotiation

A GET request to an existing manifest URL MUST provide the expected manifest, with a response code that MUST be `200 OK`. A successful response SHOULD contain the digest of the uploaded blob in the header `Docker-Content-Digest`.

The `Docker-Content-Digest` header, if present on the response, returns the canonical digest of the uploaded blob which MAY differ from the provided digest. If the digest does differ, it MAY be the case that the hashing algorithms used do not match. See Content Digests apdx-3 for information on how to detect the hashing algorithm in use. Most clients MAY ignore the value, but if it is used, the client MUST verify the value against the uploaded blob data.

If the manifest is not found in the registry, the response code MUST be `404 Not Found`.

Pulling blobs

To pull a blob, perform a `GET` request to a URL in the following form: `/v2/<name>/blobs/<digest>` end-2

`<name>` is the namespace of the repository, and `<digest>` is the blob's digest.

A GET request to an existing blob URL MUST provide the expected blob, with a response code that MUST be `200 OK`. A successful response SHOULD contain the digest of the uploaded blob in the header `Docker-Content-Digest`. If present, the value of this header MUST be a digest matching that of the response body.

If the blob is not found in the registry, the response code MUST be `404 Not Found`.

Checking if content exists in the registry

In order to verify that a repository contains a given manifest or blob, make a `HEAD` request to a URL in the following form:

`/v2/<name>/manifests/<reference>` end-3 (for manifests), or

`/v2/<name>/blobs/<digest>` end-2 (for blobs).

A HEAD request to an existing blob or manifest URL MUST return `200 OK`. A successful response SHOULD contain the digest of the uploaded blob in the header `Docker-Content-Digest`.

If the blob or manifest is not found in the registry, the response code MUST be `404 Not Found`.

Push

Pushing an artifact typically works in the opposite order as a pull: the blobs making up the artifact are uploaded first, and the manifest last. Strictly speaking, content can be uploaded to the registry in any order, but a registry MAY reject a manifest if it references blobs that are not yet uploaded, resulting in a `BLOB_UNKNOWN` error code-1. A useful diagram is provided here.

Pushing blobs

There are two ways to push blobs: chunked or monolithic.

Pushing a blob monolithically

There are two ways to push a blob monolithically:

1. A `POST` request followed by a `PUT` request
2. A single `POST` request

POST then PUT

To push a blob monolithically by using a POST request followed by a PUT request, there are two steps:

1. Obtain a session id (upload URL)
2. Upload the blob to said URL

To obtain a session ID, perform a `POST` request to a URL in the following format:

`/v2/<name>/blobs/uploads/` end-4a

Here, `<name>` refers to the namespace of the repository. Upon success, the response MUST have a code of `202 Accepted`, and MUST include the following header:

```
Location: <location>
```

The `<location>` MUST contain a UUID representing a unique session ID for the upload to follow. The `<location>` does not necessarily need to be provided by the registry itself. In fact, offloading to another server can be a better strategy.

Optionally, the location MAY be absolute (containing the protocol and/or hostname), or it MAY be relative (containing just the URL path). For more information, see RFC 7231.

Once the `<location>` has been obtained, perform the upload proper by making a `PUT` request to the following URL path, and with the following headers and body:

`<location>?digest=<digest>` end-6

```
Content-Length: <length>
Content-Type: application/octet-stream

<upload byte stream>
```

The `<location>` MAY contain critical query parameters. Additionally, it SHOULD match exactly the `<location>` obtained from the `POST` request. It SHOULD NOT be assembled manually by clients except where absolute/relative conversion is necessary.

Here, `<digest>` is the digest of the blob being uploaded, and `<length>` is its size in bytes.

Upon successful completion of the request, the response MUST have code `201 Created` and MUST have the following header:

```
Location: <blob-location>
```

With `<blob-location>` being a pullable blob URL.

Single POST

Registries MAY support pushing blobs using a single POST request.

To push a blob monolithically by using a single POST request, perform a `POST` request to a URL in the following form, and with the following headers and body:

`/v2/<name>/blobs/uploads/?digest=<digest>` end-4b

```
Content-Length: <length>
Content-Type: application/octet-stream

<upload byte stream>
```

Here, `<name>` is the repository's namespace, `<digest>` is the blob's digest, and `<length>` is the size (in bytes) of the blob.

The `Content-Length` header MUST match the blob's actual content length. Likewise, the `<digest>` MUST match the blob's digest.

Registries that do not support single request monolithic uploads SHOULD return a `202 Accepted` status code and `Location` header and clients SHOULD proceed with a subsequent PUT request, as described by the POST then PUT upload method.

Successful completion of the request MUST return a `201 Created` and MUST include the following header:

```
Location: <blob-location>
```

Here, `<blob-location>` is a pullable blob URL. This location does not necessarily have to be served by your register, for example, in the case of a signed URL from some cloud storage provider that your registry generates.

Pushing a blob in chunks

A chunked blob upload is accomplished in three phases:

1. Obtain a session ID (upload URL) (`POST`)
2. Upload the chunks (`PATCH`)
3. Close the session (`PUT`)

For information on obtaining a session ID, reference the above section on pushing a blob monolithically via the `POST`/`PUT` method. The process remains unchanged for chunked upload, except that the post request MUST include the following header:

```
Content-Length: 0
```

Please reference the above section for restrictions on the `<location>`.

To upload a chunk, issue a `PATCH` request to a URL path in the following format, and with the following headers and body:

URL path: `<location>` end-5

```
Content-Type: application/octet-stream
Content-Range: <range>
Content-Length: <length>
```

```
<upload byte stream of chunk>
```

The `<location>` refers to the URL obtained from the preceding `POST` request.

The `<range>` refers to the byte range of the chunk, and MUST be inclusive on both ends. The first chunk's range MUST begin with `0`. It MUST match the following regular expression:

```
^[0-9]+-[0-9]+$
```

The `<length>` is the content-length, in bytes, of the current chunk.

Each successful chunk upload MUST have a `202 Accepted` response code, and MUST have the following header:

```
Location <location>
```

Each consecutive chunk upload SHOULD use the `<location>` provided in the response to the previous chunk upload.

Chunks MUST be uploaded in order, with the first byte of a chunk being the last chunk's `<end-of-range>` plus one. If a chunk is uploaded out of order, the registry MUST respond with a `416 Requested Range Not Satisfiable` code.

The final chunk MAY be uploaded using a `PATCH` request or it MAY be uploaded in the closing `PUT` request. Regardless of how the final chunk is uploaded, the session MUST be closed with a `PUT` request.

To close the session, issue a `PUT` request to a url in the following format, and with the following headers (and optional body, depending on whether or not the final chunk was uploaded already via a `PATCH` request):

`<location>?digest=<digest>`

```
Content-Length: <length of chunk, if present>
Content-Range: <range of chunk, if present>
Content-Type: application/octet-stream <if chunk provided>
```

```
OPTIONAL: <final chunk byte stream>
```

The closing `PUT` request MUST include the `<digest>` of the whole blob (not the final chunk) as a query parameter.

The response to a successful closing of the session MUST be `201 Created`, and MUST contain the following header:

```
Location: <blob-location>
```

Here, `<blob-location>` is a pullable blob URL.

Mounting a blob from another repository

If a necessary blob exists already in another repository, it can be mounted into a different repository via a `POST` request in the following format:

`/v2/<name>/blobs/uploads/?mount=<digest>&from=<other_name>` end-11.

In this case, `<name>` is the namespace to which the blob will be mounted. `<digest>` is the digest of the blob to mount, and `<other_name>` is the namespace from which the blob should be mounted. This step is usually taken in place of the previously-described `POST` request to `/v2/<name>/blobs/uploads/` end-4a (which is used to initiate an upload session).

The response to a successful mount MUST be `201 Created`, and MUST contain the following header:

```
Location: <blob-location>
```

The Location header will contain the registry URL to access the accepted layer file. The Docker-Content-Digest header returns the canonical digest of the uploaded blob which MAY differ from the provided digest. Most clients MAY ignore the value but if it is used, the client SHOULD verify the value against the uploaded blob data.

Alternatively, if a registry does not support cross-repository mounting or is unable to mount the requested blob, it SHOULD return a `202`. This indicates that the upload session has begun and that the client MAY proceed with the upload.

Pushing Manifests

To push a manifest, perform a `PUT` request to a path in the following format, and with the following headers and body: `/v2/<name>/manifests/<reference>` end-7

```
Content-Type: application/vnd.oci.image.manifest.v1+json
```

```
<manifest byte stream>
```

`<name>` is the namespace of the repository, and the `<reference>` MUST be either a) a digest or b) a tag.

The uploaded manifest MUST reference any blobs that make up the artifact. However, the list of blobs MAY be empty. Upon a successful upload, the registry MUST return response code `201 Created`, and MUST have the following header:

```
Location: <location>
```

The `<location>` is a pullable manifest URL.

An attempt to pull a nonexistent repository MUST return response code `404 Not Found`

Content Discovery

Currently, the only functionality provided by this workflow is the ability to discover tags.

To fetch the list of tags, perform a `GET` request to a path in the following format: `/v2/<name>/tags/list` end-8a

`<name>` is the namespace of the repository. Assuming a repository is found, this request MUST return a `200 OK` response code. The list of tags MAY be empty if there are no tags on the repository. If the list is not empty, the tags MUST be in lexical order (i.e. case-insensitive alphanumeric order).

Upon success, the response MUST be a json body in the following format:

```
{
  "name": "<name>",
  "tags": [
    "<tag1>",
    "<tag2>",
    "<tag3>"
  ]
}
```

`<name>` is the namespace of the repository, and `<tag1>`, `<tag2>`, and `<tag3>` are each tags on the repository.

In addition to fetching the whole list of tags, a subset of the tags can be fetched by providing the `n` query parameter. In this case, the path will look like the following: `/v2/<name>/tags/list?n=<int>` end-8b

`<name>` is the namespace of the repository, and `<int>` is an integer specifying the number of tags requested. The response to such a request MAY return fewer than `<int>` results, but only when the total number of tags attached to the repository is less than `<int>`. Otherwise, the response MUST include `<int>` results. When `n` is zero, this endpoint MUST return an empty list, and MUST NOT include a `Link` header. Without the `last` query parameter (described next), the list returned will start at the beginning of the list and include `<int>` results. As above, the tags MUST be in lexical order.

The `last` query parameter provides further means for limiting the number of tags. It is usually used in combination with the `n` parameter: `/v2/<name>/tags/list?n=<int>&last=<tagname>` end-8b

`<name>` is the namespace of the repository, `<int>` is the number of tags requested, and `<tagname>` is the *value* of the last tag. `<tagname>` MUST NOT be a numerical index, but rather it MUST be a proper tag. A request of this sort will return up to `<int>` tags, beginning non-inclusively with `<tagname>`. That is to say, `<tagname>` will not be included in the results, but up to `<int>` tags *after* `<tagname>` will be returned. The tags MUST be in lexical order.

When using the `last` query parameter, the `n` parameter is OPTIONAL.

Content Management

Content management refers to the deletion of blobs, tags, and manifests. Registries MAY implement deletion or they MAY disable it. Similarly, a registry MAY implement tag deletion, while others MAY allow deletion only by manifest.

Deleting tags

`<name>` is the namespace of the repository, and `<tag>` is the name of the tag to be deleted. Upon success, the registry MUST respond with a `202 Accepted` code. If tag deletion is disabled, the registry MUST respond with either a `400 Bad Request` or a `405 Method Not Allowed`.

To delete a tag, perform a `DELETE` request to a path in the following format: `/v2/<name>/manifests/<tag>` end-9

Deleting Manifests

To delete a manifest, perform a `DELETE` request to a path in the following format: `/v2/<name>/manifests/<digest>` end-9

`<name>` is the namespace of the repository, and `<digest>` is the digest of the manifest to be deleted. Upon success, the registry MUST respond with a `202 Accepted` code. If the repository does not exist, the response MUST return `404 Not Found`.

Deleting Blobs

To delete a blob, perform a `DELETE` request to a path in the following format: `/v2/<name>/blobs/<digest>` end-10

`<name>` is the namespace of the repository, and `<digest>` is the digest of the blob to be deleted. Upon success, the registry MUST respond with code `202 Accepted`. If the blob is not found, a `404 Not Found` code MUST be returned.

API

The API operates over HTTP. Below is a summary of the endpoints used by the API.

Determining Support

To check whether or not the registry implements this specification, perform a `GET` request to the following endpoint: `/v2/` end-1.

If the response is `200 OK`, then the registry implements this specification.

This endpoint MAY be used for authentication/authorization purposes, but this is out of the purview of this specification.

Endpoints

ID	Method	API Endpoint	Success	Failure
end-1	`GET`	`/v2/`	200	404/401
end-2	`GET` / `HEAD`	`/v2/<name>/blobs/<digest>`	200	404
end-3	`GET` / `HEAD`	`/v2/<name>/manifests/<reference>`	200	404
end-4a	`POST`	`/v2/<name>/blobs/uploads/`	202	404
end-4b	`POST`	`/v2/<name>/blobs/uploads/?digest=<digest>`	201/202	404/400
end-5	`PATCH`	`/v2/<name>/blobs/uploads/<reference>`	202	404/416
end-6	`PUT`	`/v2/<name>/blobs/uploads/<reference>?digest=<digest>`	201	404/400
end-7	`PUT`	`/v2/<name>/manifests/<reference>`	201	404
end-8a	`GET`	`/v2/<name>/tags/list`	200	404
end-8b	`GET`	`/v2/<name>/tags/list?n=<integer>&last=<integer>`	200	404
end-9	`DELETE`	`/v2/<name>/manifests/<reference>`	202	404/400/405
end-10	`DELETE`	`/v2/<name>/blobs/<digest>`	202	404/405
end-11	`POST`	`/v2/<name>/blobs/uploads/?mount=<digest>&from=<other_name>`	201	404

Error Codes

A `4XX` response code from the registry MAY return a body in any format. If the response body is in JSON format, it MUST have the following format:

```
{
    "errors": [
        {
            "code": "<error identifier, see below>",
            "message": "<message describing condition>",
            "detail": "<unstructured>"
        },
        ...
    ]
}
```

The `code` field MUST be a unique identifier, containing only uppercase alphabetic characters and underscores. The `message` field is OPTIONAL, and if present, it SHOULD be a human readable string or MAY be empty. The `detail` field is OPTIONAL and MAY contain arbitrary JSON data providing information the client can use to resolve the issue.

The `code` field MUST be one of the following:

ID	Code	Description
code-1	`BLOB_UNKNOWN`	blob unknown to registry
code-2	`BLOB_UPLOAD_INVALID`	blob upload invalid
code-3	`BLOB_UPLOAD_UNKNOWN`	blob upload unknown to registry
code-4	`DIGEST_INVALID`	provided digest did not match uploaded content
code-5	`MANIFEST_BLOB_UNKNOWN`	blob unknown to registry
code-6	`MANIFEST_INVALID`	manifest invalid
code-7	`MANIFEST_UNKNOWN`	manifest unknown
code-8	`NAME_INVALID`	invalid repository name
code-9	`NAME_UNKNOWN`	repository name not known to registry
code-10	`SIZE_INVALID`	provided length did not match content length
code-12	`UNAUTHORIZED`	authentication required
code-13	`DENIED`	requested access to the resource is denied
code-14	`UNSUPPORTED`	the operation is unsupported
code-15	`TOOMANYREQUESTS`	too many requests

Appendix

The following is a list of documents referenced in this spec:

ID	Title	Description
apdx-1	Docker Registry HTTP API V2	The original document upon which this spec was based
apdx-1	Details	Historical document describing original API endpoints and requests in detail
apdx-2	OCI Image Spec - manifests	Description of manifests, defined by the OCI Image Spec
apdx-3	OCI Image Spec - digests	Description of digests, defined by the OCI Image Spec
apdx-4	OCI Image Spec - config	Description of configs, defined by the OCI Image Spec