CLIPCODE™

Microservices And Containers Touring Library

The specs and tooling that power containers

by Eamon O'Tuathail

Table of Contents

Preface

The Open Container Initiative (OCI) is an industry-wide grouping of companies interested in container technology. Its website is here:

- https://www.opencontainers.org

Its github repository is here:

- https://github.com/opencontainers

It offer two completed specifications:

- The Runtime Specification (runtime-spec)
  https://github.com/opencontainers/runtime-spec
- The Image Specification (image-spec)
  https://github.com/opencontainers/image-spec

The runtime spec "aims to specify the configuration, execution environment, and lifecycle of a container". The image spec defines the "OCI Image, consisting of a 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."

A work-in-progess specification that will become very important is:

- The Distribution Specification
  https://github.com/opencontainers/distribution-spec

It is for distribution of standards-compliant container image (taking on the role of Docker Registry). Docker has donated the Docker Regisry V2 specification to the OCI and this will act as the basis for this new spec.

The OCI also offers a number of open-source projects. The most important project is a command-line interface (CLI) to interacting with a container engine:

- runc
  https://github.com/opencontainers/runc

This repo includes a significant library for programmatic interaction with a container engine (think of runc as a CLI wrapper around libcontainer, but your own apps could use libcontainer directly):

- libcontainer
  https://github.com/opencontainers/runc/tree/master/libcontainer

Runc is an implementation of the runtime specification. Other implementations can also be created based off the same spec. the vast majority of installations using Docker and/or Kubernetes have been using the current or previous versions of runc as the lowest layer of their software stacks. So if you have been using containers up to now, mostly likely you have been using runc under the hood.

The OCI also have a number of other repositories, which you should explore. For example:

- runtime-tools
  https://github.com/opencontainers/runtime-tools
- image-tools
  https://github.com/opencontainers/image-tools

are collections of tools to work with OCI runtime spec and OCI image spec respectively. There is no distribution-tools project (yet).

Two specialist repositories are:

- go-digest (a common digest package)
  https://github.com/opencontainers/go-digest
- selinux
  https://github.com/opencontainers/selinux

A digest is using for hashing and we see it is needed to uniquely identify content blobs. SELinux is for secure Linux and this project is for seucrity inside containers that are running on Linux.

The above specs and runc are based on initial contributions that Docker generously donated to the OCI, which are now being evolved with substantial industry contributions. Docker also participates in these enhancement efforts and their own tooling are regularly updated to reflect further OCI enhancements. It should be noted that though Docker gets most of the publicity surrounding containers, there were many companies substantially involved in the engineering behind containers. For example, Google wrote cgroups [https://en.wikipedia.org/wiki/Cgroups], which is one of the two key foundation stones for containers - the other being namespaces [https://en.wikipedia.org/wiki/Linux_namespaces].

Most container code is written in Go. Documentation for various OCI offerings are on GoDoc here:

- https://godoc.org/?q=opencontainers

Licensing

Any content quoted in this document from Open Container Initiative repositories is copyright by them and subject to their licensing regulations – as found in the LICENSE document in each repo. Up to now, they use the open source Apache License 2.0.

Membership

The members of the OCI are listed here:

- https://www.opencontainers.org/about/members

It is clear that all the leading participants in the container industry are members.

1: Runtime Spec

Overview

The runtime-spec repository contains the specification (description), schema and config objects for a tool or a group of tools to specify a container's configuration, execution environment and lifecycle. The description is provided as a set of markdown (*.md) files. The schema is provided as JSON Schema. The config types are provided as Go source. If you are going to get involved in exploring the internals of container technologies, you are going to have to learn Go.

Specification

Unlike most specifications which are provided as HTML or PDF files, with runtime-spec, markdown (*.md) is used and the spec is delivered as separate files.

The Runtime Specification is provided as a set of host-independent pages:

- spec.md (entrypoint to the spec; abstract, links to the other pages)
- principles.md (architectural principles behind containers)
- bundle.md (describes a filessytem bundle, which is how files – e.g. config.json – are arranged and presented to the container)
- runtime.md (describes the state of a container & operations to perform on it)
- config.md (general metadata needed for operations)
- glossary.md (terminology)

and host-dependent pages:

- config-linux.md (Linux-specific configuration)
- config-solaris.md (Solaris-specific configuration)
- config-vm.md (VM-specific configuration)
- config-windows.md (Windows-specific configuration)
- runtime-linux.md (additional information about Linux file descriptors and symbolic links)

There are also some additional Markdown files, such as:

- implementations.md (links to related projects)

As explained in the repo's homepage, the Runtime Specification is aimed at three developer populations:

- Runtime developers – those who intend to develop implementations of this specification (e.g. folks working on runc, rkt or Knowist-ng-toolkit)
- Hook developers – those who wish to extend how the runtime behaves and react to certain events (which can be hooked)
- Bundle Tooling Developers – those who are working on devops tooling to make bundles, which are the content (e.g. configuration) need ded to launch an application.

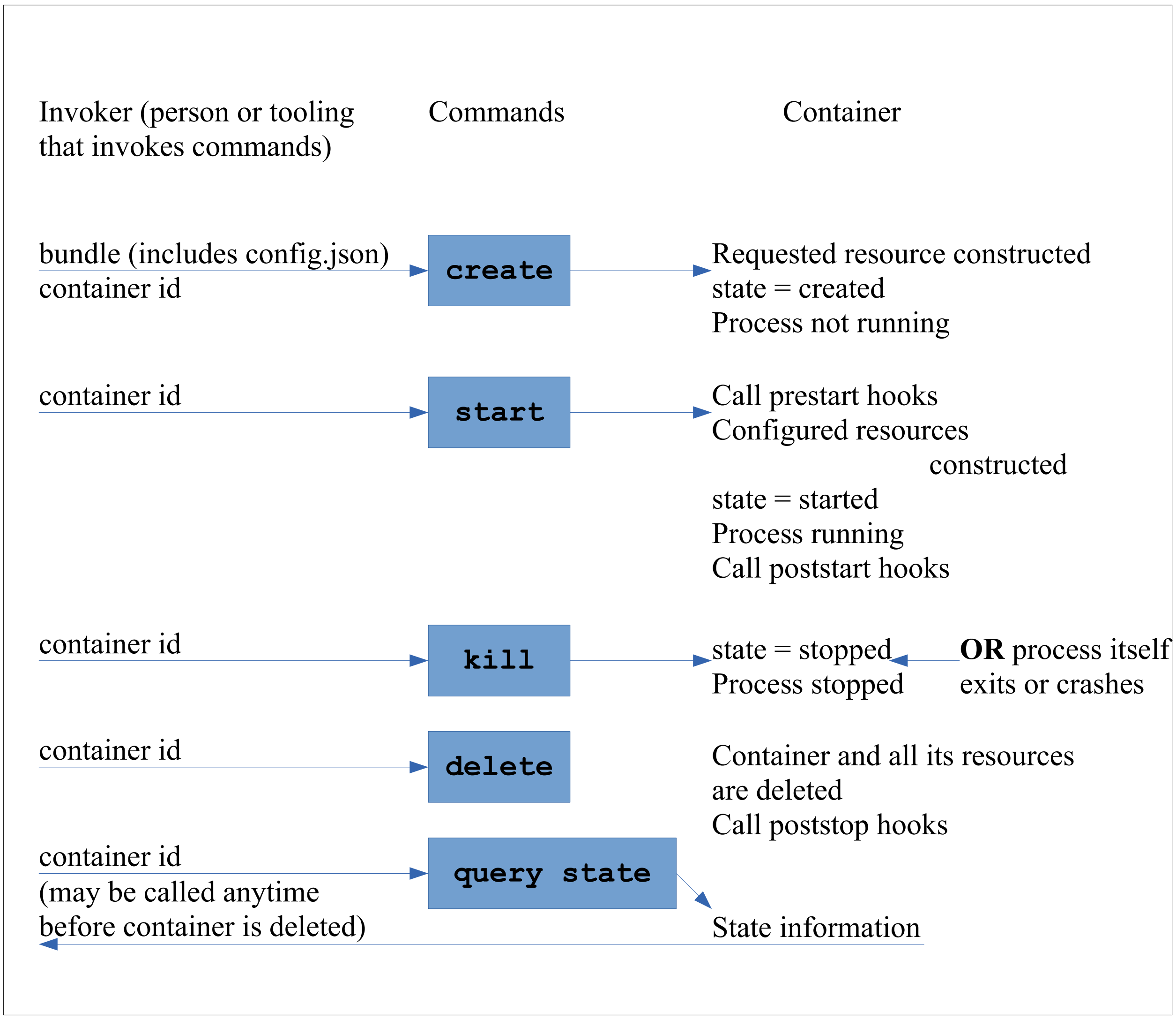
The runtime-spec defines these key constructs:

- Operations
- Lifcycle
- Hooks
- Configuration
- Bundle

Let’s look at each in turn.

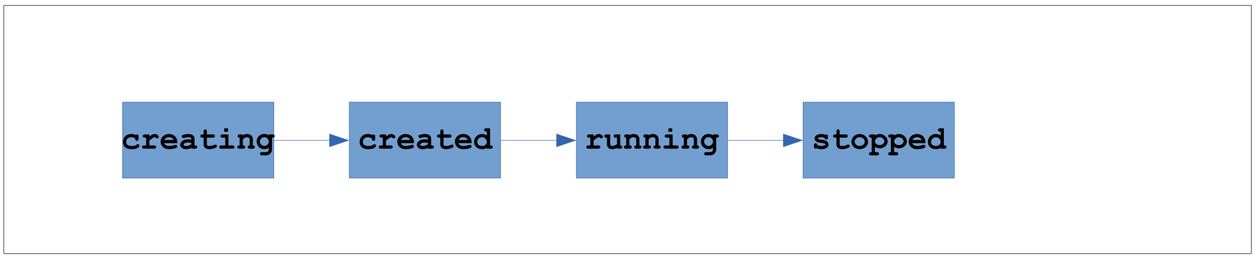
Lifecycle

The container lifecycle is described in 10 steps in the Lifecycle section of runtime.md. The following diagrams illustrates the steps.


Invoker (person or tooling that invokes commands)
Commands
Container
bundle (includes config.json) container id
create
Requested resource constructed state = created Process not running
container id
start
Call prestart hooks Configured resources constructed state = started Process running Call poststart hooks
container id
kill
state = stopped Process stopped
OR process itself exits or crashes
container id
delete
Container and all its resources are deleted Call poststop hooks
container id (may be called anytime before container is deleted)
query state
State information

Coupled with this is the status of a container.

This finite state machine explains status transformations:



Operations

Operations are the actions devops professionals would like to carry out on a container. Implementations can (and most do) add additional operations. The following are the operations defined in this specification:

- query state - what is the state of a container
- create – create a container with a specific id based on an identified bundle
- start – starts running an already created container
- kill – sends a signal to a container to shut it down
- delete – deletes a container and its resources

State information comes in the form of a state struct, that as we have seen has fields for items such as status, id and bundle.

The difference between create and start is that create constructs the container but does not set it running. That is the job of start. The difference between kill and delete is kills stops a running container, whereas delete removes a stopped container completely.

Hooks

Hooks are provided to allow external developer attach code to be executed at certain stages of the lifecycle. If additional tools need to be informed of changes to a container state then hooks are the way to enabled this kind of functionality.

Configuration

Much of the runtime spec is devoted to configuration. Some of this is host-independent and some is host-dependent.

The host independent configuration is described by:

- https://github.com/opencontainers/runtime-spec/blob/master/config.md

Its main fields are:

- ociVersion – version info of the runtime spec that this config complies with
- Root (path and optional readonly flag) – root filesystem for container
- Mounts (destination, source, options) – mounts (in addition to root)
- Process (args, cwd, env, terminal, consoleSize) – the process to execute inside the container, with the first (mandatory) args entry being the actual file to run
- User – the user under which to run the process
- Hostname – the name of the host that the container uses to represent hostname
- Annotations – name-value pairs of additional metadata

Go Types

There are three Go language files in the specs-go sub-directory:

- config.go
- state.go
- version.go

version.go is the simplest so we will start with that. It creates a string variable, `Version`, that is a dotted combination of major.minor.patch version numbers:

```
package specs
import "fmt"
const (
  // VersionMajor is for an API incompatible changes
  VersionMajor = 1
  // VersionMinor is for functionality in a backwards-compatible manner
  VersionMinor = 0
  // VersionPatch is for backwards-compatible bug fixes
  VersionPatch = 1

  // VersionDev indicates development branch. Releases will be empty string.
  VersionDev = "-dev"
)
// Version is the specification version that the package types support.
var Version = fmt.Sprintf("%d.%d.%d%s",
                        VersionMajor, VersionMinor, VersionPatch, VersionDev)
```

The state.go file defines the `State` struct which is used to hold information about the state of the container:

```
package specs

// State holds information about the runtime state of the container.
type State struct {
  // Version is the version of the specification that is supported.
  Version string `json:"ociVersion"`
  // ID is the container ID
  ID string `json:"id"`
  // Status is the runtime status of the container.
  Status string `json:"status"`
  // Pid is the process ID for the container process.
  Pid int `json:"pid,omitempty"`
  // Bundle is the path to the container's bundle directory.
  Bundle string `json:"bundle"`
  // Annotations are key values associated with the container.
  Annotations map[string]string `json:"annotations,omitempty"`
}
```

The large config.go file contains many configuration-related structs, some of which are host-dependent. Read the config.go file to see the full contents of each struct – here we will just list the struct names. The host-independent structs are:

```
type Spec struct
type Process struct
type Box struct
type User struct
type Root struct
type Mount struct
type Hook struct
type Hooks struct
```

The Linux specific structs are:

```
type LinuxCapabilities struct
type Linux struct
type LinuxNamespace struct
type LinuxNamespaceType string
type LinuxIDMapping struct
type POSIXRlimit struct
type LinuxHugepageLimit struct
type LinuxInterfacePriority struct
type linuxBlockIODevice struct
type LinuxWeightDevice struct
type LinuxThrottleDevice struct
type LinuxBlockIO struct
type LinuxMemory struct
type LinuxCPU struct
type LinuxPids struct
type LinuxNetwork struct
type LinuxRdma struct
type LinuxResources struct
type LinuxDevice struct
type LinuxDeviceCgroup struct
type LinuxSeccomp struct
type LinuxSeccompAction string
type LinuxSeccompArg struct
type LinuxSyscall struct
type LinuxIntelRdt struct
```

The Solaris specific structs are:

```
type Solaris struct
type SolarisCappedMemory struct
type SolarisAnet struct
```

The Windows specific structs are:

```
type Windows struct
type WindowsResources struct
type WindowsMemoryResources struct
type WindowsCPUResources struct
type WindowsStorageResources struct
type WindowsNetwork struct
type WindowsHyperV struct
```

The VM specific structs are:

```
type VM struct
type VMHypervisor struct
type VMKernel struct
type VMImage struct
type Arch string
```

There is also a .tool sub-directory and it contains a single file:

- version-doc.go

with this content (slightly abbreviated for here):

```
package main
import ( “github.com/opencontainers/runtime-spec/specs-go" )
var markdownTemplateString = `**Specification Version:** *{{.}}*`
var markdownTemplate =
template.Must(template.New("markdown").Parse(markdownTemplateString))
func main() {
  if err := markdownTemplate.Execute(os.Stdout, specs.Version); err != nil {
    fmt.Fprintln(os.Stderr, err)
  }
}
```

It essentially prints out to stdout the version string defined in the specs-go sub-directory.

JSON Schema

JSON Schema is to JSON what XML Schema is to XML. JSON Schema precisely describes what is considered valid layout for a particular usage of JSON. To learn more about JSON Schema, view:

- http://json-schema.org

In the Open Containers’ runtime-spec repository, the schema sub-directory contains JSON Schema for JSON used elsewhere in the runtime spec. The state-schema.json file describes the state of a container. Its main fields are:

- ociVersion
- id
- status (one of creating, created, running, stopped)
- pid
- bundle
- annotations

There is a set of defs-XYZ.json files that describe definitions used elsewhere.

As an example, lets look at `Mount`, which is used to mount file systems:

```
"Mount": {
            "type": "object",
            "properties": {
                "source": {
                    "$ref": "#/definitions/FilePath"
                },
                "destination": {
                    "$ref": "#/definitions/FilePath"
                },
                "options": {
                    "$ref": "#/definitions/ArrayOfStrings"
                },
                "type": {
                    "type": "string"
                }
            },
            "required": [
                "destination"
            ]
        },
```

There is a set of config-XYZ.json files used to describe the configurations. The config-schema.json file is host neutral and the other config files are host-dependent. Example content from config-schema.json is:

```
"mounts": {
            "type": "array",
            "items": {
                "$ref": "defs.json#/definitions/Mount"
            } },
```

Note the way it references into the definitions file for the contents of `item`.

2: Image Spec

Overview

The image-spec repository contains the specification (description), schema and types for an image format. It is expected that a container implementation will be able to download an image that complies with this spec from an image repository/registry, unpack it, and then its runtime engine (that complies with the runtime-spec) uses that as its filesystem bundle to run a container.

The description is provided as a set of markdown (*.md) files. The schema is provided as JSON Schema. The types are provided as Go source.

Specification

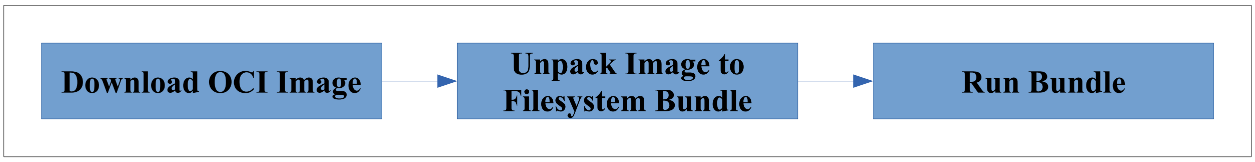
The Image Specification is provided as a set of pages in markdown:

- spec.md (entrypoint to the spec; good overview, links to the other pages)
- media-types.md (media types for formats)
- descriptor.md (content descriptors are used to reference other components)
- image-layout.md (directory structure for image)
- manifest.md (configuration and layering information)
- image-index.md (index of manifests)
- layer.md (layers for a filesystem and its changes)
- config.md (configuration)
- annotations.md (additional metadata)
- conversion.md (how to convert blobs)
- considerations.md (additional considerations)

Spec.md starts with an accurate description of what image-spec is:

> *This specification defines an OCI Image, consisting of a 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.*

An OCI implementation needs to perform three operations:



To have sufficient info to run, the bundle needs the command to run in the container's process, its environment, additional arguments and the filesystem to present to it.

An OCI Image contains at least three and possible four pieces of information. The three mandatory items are:

- The manifests
- Filesystem layer
- Configuration

The optionally fourth item is:

- image index

Go Types

The specs-go sub-directory contains two Go language files:

- version.go – a versionMajor.versionMinor.patch.dev version string
- versioned.go – to be used with unknown schema versions are detected

and one sub-directory:

- v1 – Types to work with v1 (i.e. the current) version of the image format

version.go has the same content as version.go in runtime-spec. versioned.go is as follows:

```
// Versioned provides a struct with the manifest schemaVersion and mediaType.
// Incoming content with unknown schema version can be decoded against this
// struct to check the version.
type Versioned struct {
  // SchemaVersion is the image manifest schema that this image follows
  SchemaVersion int `json:"schemaVersion"`
}
```

The v1 sub-directory contains these go files:

- mediatype.go – string consts for the various media types
- index.go – a type containing an array of manifests and an array of annotations
- manifest.go – a type containing a config descriptor, an array of layer descriptors and an array of annotations
- layout.go – image layout definition
- descriptor.go – the target for the content
- config.go – image configuration
- annotations.go – Predefined annotations

Let’s start by explore mediatypes.go. This lists media types needed for various fragments:

```
const (
  // MediaTypeDescriptor specifies the media type for a content descriptor.
  MediaTypeDescriptor = "application/vnd.oci.descriptor.v1+json"

  // MediaTypeLayoutHeader specifies the media type for the oci-layout.
  MediaTypeLayoutHeader = "application/vnd.oci.layout.header.v1+json"

  // MediaTypeImageManifest specifies the media type for an image manifest.
  MediaTypeImageManifest = "application/vnd.oci.image.manifest.v1+json"

  // MediaTypeImageIndex specifies the media type for an image index.
  MediaTypeImageIndex = "application/vnd.oci.image.index.v1+json"
  ..
  // MediaTypeImageConfig specifies the media type for the image config.
  MediaTypeImageConfig = "application/vnd.oci.image.config.v1+json"
)
```

The five fragments described are for:

- Image Configuration
- Image Index
- Image Manifest
- Layout Header
- Descriptor

Additional important fragments are for layering. These can be distributable or non-distributable, and can be just tar or tar-and-gzipped:

```
const (
  ..
  // MediaTypeImageLayer is the media type used for layers referenced by
  // the manifest.
  MediaTypeImageLayer = "application/vnd.oci.image.layer.v1.tar"

  // MediaTypeImageLayerGzip is the media type used for gzipped layers
  // referenced by the manifest.
  MediaTypeImageLayerGzip = "application/vnd.oci.image.layer.v1.tar+gzip"

  // MediaTypeImageLayerNonDistributable is the media type for layers
  // referenced by the manifest but with distribution restrictions.
  MediaTypeImageLayerNonDistributable =
    "application/vnd.oci.image.layer.nondistributable.v1.tar"

  // MediaTypeImageLayerNonDistributableGzip is the media type for
  // gzipped layers referenced by the manifest but with distribution
  // restrictions.
  MediaTypeImageLayerNonDistributableGzip =
    "application/vnd.oci.image.layer.nondistributable.v1.tar+gzip"
  ..
)
```

The index.go file defines the Index type:

```
package v1

import "github.com/opencontainers/image-spec/specs-go"

// Index references manifests for various platforms.
// This structure provides `application/vnd.oci.image.index.v1+json`
mediatype when marshalled to JSON.
type Index struct {
  specs.Versioned

  // Manifests references platform specific manifests.
  Manifests []Descriptor `json:"manifests"`

  // Annotations contains arbitrary metadata for the image index.
  Annotations map[string]string `json:"annotations,omitempty"`
}
```

Note it imports specs.Versioned from the specs-go file.

manifest.go contains this definition:

```
type Manifest struct {
  specs.Versioned
```

```
    // Config references a configuration object for a container, by digest.
    // The referenced configuration object is a JSON blob that the runtime
    // uses to set up the container.
    Config Descriptor `json:"config"`

    // Layers is an indexed list of layers referenced by the manifest.
    Layers []Descriptor `json:"layers"`

    // Annotations contains arbitrary metadata for the image manifest.
    Annotations map[string]string `json:"annotations,omitempty"`
  }
```

layout.go contains two consts and one type:

```
  const (
    // ImageLayoutFile is the file name of oci image layout file
    ImageLayoutFile = "oci-layout"
    // ImageLayoutVersion is the version of ImageLayout
    ImageLayoutVersion = "1.0.0"
  )

  // ImageLayout is the structure in the "oci-layout" file, found in the root
  // of an OCI Image-layout directory.
  type ImageLayout struct {
    Version string `json:"imageLayoutVersion"`
  }
```

descriptor.go uses to go-digest package and defines the `Descriptor` and `Platform` types. The go-digest package (described in a later package) is a way to calculate digest information for a blob. It is a form of hashing to ensure we have the expected bytestream. It is imported by descriptor.go with this line:

```
  import digest "github.com/opencontainers/go-digest"
```

and used later in the `Descriptor` definition:

```
  type Descriptor struct {
    ..
    // Digest is the digest of the targeted content.
    Digest digest.Digest `json:"digest"`
    ..
  }
```

The Platform struct identifies the target platform for the image:

```
  // Platform describes the platform which the image in the manifest runs on.
  type Platform struct {
    // Architecture field specifies the CPU architecture, for example
    // `amd64` or `ppc64`.
    Architecture string `json:"architecture"`

    // OS specifies the operating system, for example `linux` or `windows`.
    OS string `json:"os"`

    // OSVersion is an optional field specifying the operating system
    // version, for example on Windows `10.0.14393.1066`.
    OSVersion string `json:"os.version,omitempty"`

    // OSFeatures is an optional field specifying an array of strings,
    // each listing a required OS feature (for example on Windows `win32k`).
    OSFeatures []string `json:"os.features,omitempty"`
    // Variant is an optional field specifying a variant of the CPU, for
    // example `v7` to specify ARMv7 when architecture is `arm`.
    Variant string `json:"variant,omitempty"`
}
```

Then the `Descriptor` struct provides a description for the target:

```
// Descriptor describes the disposition of targeted content.
// This structure provides `application/vnd.oci.descriptor.v1+json` mediatype
// when marshalled to JSON.
type Descriptor struct {
    // MediaType is the media type of the object this schema refers to.
    MediaType string `json:"mediaType,omitempty"`

    // Digest is the digest of the targeted content.
    Digest digest.Digest `json:"digest"`

    // Size specifies the size in bytes of the blob.
    Size int64 `json:"size"`

    // URLs specifies a list of URLs from which this object MAY be downloaded
    URLs []string `json:"urls,omitempty"`

    // Annotations contains arbitrary metadata relating to
    // the targeted content.
    Annotations map[string]string `json:"annotations,omitempty"`

    // Platform describes the platform which the image in the manifest runs on.
    // This should only be used when referring to a manifest.
    Platform *Platform `json:"platform,omitempty"`
}
```

annotations.go provides annotations for specific kinds of metadata. It is a long list – here we provide a sampling:

```
const (
    // AnnotationCreated is the annotation key for the date and time on
    // which the image was built (date-time string as defined by RFC 3339).
    AnnotationCreated = "org.opencontainers.image.created"

    // AnnotationAuthors is the annotation key for the contact details of
    // the people or organization responsible for the image (freeform string).
    AnnotationAuthors = "org.opencontainers.image.authors"

    // AnnotationURL is the annotation key for the URL to find more
    // information on the image.
    AnnotationURL = "org.opencontainers.image.url"
)
```

Finally, config.go provides It starts by importing Go's standard time package and OCI's go-digest package.

```
import (
    "time"

    digest "github.com/opencontainers/go-digest"
)
```

It defines the `ImageConfig` struct:

```
// ImageConfig defines the execution parameters which should be used
// as a base when running a container using an image.
type ImageConfig struct {
  // User defines the username or UID which the process in the
  // container should run as.
  User string `json:"User,omitempty"`

  // ExposedPorts a set of ports to expose from a container
  // running this image.
  ExposedPorts map[string]struct{} `json:"ExposedPorts,omitempty"`

  // Env is a list of environment variables to be used in a container.
  Env []string `json:"Env,omitempty"`

  // Entrypoint defines a list of arguments to use as the command to
  // execute when the container starts.
  Entrypoint []string `json:"Entrypoint,omitempty"`

  // Cmd defines the default arguments to the entrypoint of the container.
  Cmd []string `json:"Cmd,omitempty"`

  // Volumes is a set of directories describing where the process is
  // likely write data specific to a container instance.
  Volumes map[string]struct{} `json:"Volumes,omitempty"`

  // WorkingDir sets the current working directory of the entrypoint process
  // in the container.
  WorkingDir string `json:"WorkingDir,omitempty"`

  // Labels contains arbitrary metadata for the container.
  Labels map[string]string `json:"Labels,omitempty"`

  // StopSignal contains the system call signal that will be sent
  // to the container to exit.
  StopSignal string `json:"StopSignal,omitempty"`
}
```

The RootFS struct is also defined with two fields:

```
// RootFS describes a layer content addresses
type RootFS struct {
  // Type is the type of the rootfs.
  Type string `json:"type"`

  // DiffIDs is an array of layer content hashes (DiffIDs), in order
  // from bottom-most to top-most.
  DiffIDs []digest.Digest `json:"diff_ids"`
}
```

The History struct is defined as:

```
// History describes the history of a layer.
type History struct {
  // Created is the combined date and time at which the layer was
  // created, formatted as defined by RFC 3339, section 5.6.
  Created *time.Time `json:"created,omitempty"`

  // CreatedBy is the command which created the layer.
  CreatedBy string `json:"created_by,omitempty"`

  // Author is the author of the build point.
```

```
    Author string `json:"author,omitempty"`

    // Comment is a custom message set when creating the layer.
    Comment string `json:"comment,omitempty"`

    // EmptyLayer is used to mark if the history item created a
    // filesystem diff.
    EmptyLayer bool `json:"empty_layer,omitempty"`
  }
```

The Image struct is defined as:

```
  // Image is the JSON structure which describes some basic information
  // about the image. This provides the
  // `application/vnd.oci.image.config.v1+json` mediatype when marshalled
  // to JSON.
  type Image struct {
    // Created is the combined date and time at which the image was
    // created, formatted as defined by RFC 3339, section 5.6.
    Created *time.Time `json:"created,omitempty"`

    // Author defines the name and/or email address of the person or entity
    // which created and is responsible for maintaining the image.
    Author string `json:"author,omitempty"`

    // Architecture is the CPU architecture which the binaries in this image
    // are built to run on.
    Architecture string `json:"architecture"`

    // OS is the name of the operating system which the image is built
    // to run on.
    OS string `json:"os"`

    // Config defines the execution parameters which should be used as a
    // base when running a container using the image.
    Config ImageConfig `json:"config,omitempty"`

    // RootFS references the layer content addresses used by the image.
    RootFS RootFS `json:"rootfs"`

    // History describes the history of each layer.
    History []History `json:"history,omitempty"`
  }
```

JSON Schema

The schema sub-directory contains a number of JSON schema files and also Go files for intra-blob validation.

The schema files are:

- defs.json – definition of standard types (int16, stringPointer, mapStringString) used elsewhere in this spec
- defs-descriptor.json – definitions related to descriptors
- image-index-schema.json – The Image Index
- image-manifest-schema.json – The Image Manifest
- image-layout-schema.json – The Image Layout
- config-schema.json – The Configuration

- content-descriptor.json - Descriptors

The intra-blob validation files are:

- doc.go
- error.go
- fs.go
- gen.go
- loader.go
- schema.go
- validator.go