Specification Version: *1.0.2*

Open Container Initiative Runtime Specification

The Open Container Initiative develops specifications for standards on Operating System process and application containers.

Abstract

The Open Container Initiative Runtime Specification aims to specify the configuration, execution environment, and lifecycle of a container.

A container's configuration is specified as the `config.json` for the supported platforms and details the fields that enable the creation of a container.
The execution environment is specified to ensure that applications running inside a container have a consistent environment between runtimes along with common actions defined for the container's lifecycle.

Platforms

Platforms defined by this specification are:

- `linux`: runtime.md, config.md, config-linux.md, and runtime-linux.md.
- `solaris`: runtime.md, config.md, and config-solaris.md.
- `windows`: runtime.md, config.md, and config-windows.md.
- `vm`: runtime.md, config.md, and config-vm.md.

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.

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 for a given CPU architecture if it fails to satisfy one or more of the MUST, REQUIRED, or SHALL requirements for the platforms it implements.
An implementation is compliant for a given CPU architecture if it satisfies all the MUST, REQUIRED, and SHALL requirements for the platforms it implements.

The 5 principles of Standard Containers

Define a unit of software delivery called a Standard Container.
The goal of a Standard Container is to encapsulate a software component and all its dependencies in a format that is self-describing and portable, so that any compliant runtime can run it without extra dependencies, regardless of the underlying machine and the contents of the container.

The specification for Standard Containers defines:

1. configuration file formats
2. a set of standard operations
3. an execution environment.

A great analogy for this is the physical shipping container used by the transportation industry.
Shipping containers are a fundamental unit of delivery, they can be lifted, stacked, locked, loaded, unloaded and labelled.
Irrespective of their contents, by standardizing the container itself it allowed for a consistent, more streamlined and efficient set of processes to be defined.
For software Standard Containers offer similar functionality by being the fundamental, standardized, unit of delivery for a software package.

1. Standard operations

Standard Containers define a set of STANDARD OPERATIONS.
They can be created, started, and stopped using standard container tools; copied and snapshotted using standard filesystem tools; and downloaded and uploaded using standard network tools.

2. Content-agnostic

Standard Containers are CONTENT-AGNOSTIC: all standard operations have the same effect regardless of the contents.
They are started in the same way whether they contain a postgres database, a php application with its dependencies and application server, or Java build artifacts.

3. Infrastructure-agnostic

Standard Containers are INFRASTRUCTURE-AGNOSTIC: they can be run in any OCI supported infrastructure.
For example, a standard container can be bundled on a laptop, uploaded to cloud storage, downloaded, run and snapshotted by a build server at a fiber hotel in Virginia, uploaded to 10 staging servers in a home-made private cloud cluster, then sent to 30 production instances across 3 public cloud regions.

4. Designed for automation

Standard Containers are DESIGNED FOR AUTOMATION: because they offer the same standard operations regardless of content and infrastructure, Standard Containers, are extremely well-suited for automation.
In fact, you could say automation is their secret weapon.

Many things that once required time-consuming and error-prone human effort can now be programmed.
Before Standard Containers, by the time a software component ran in production, it had been individually built, configured, bundled, documented, patched, vendored, templated, tweaked and instrumented by 10 different people on 10 different computers.
Builds failed, libraries conflicted, mirrors crashed, post-it notes were lost, logs were misplaced, cluster updates were half-broken.
The process was slow, inefficient and cost a fortune - and was entirely different depending on the language and infrastructure provider.

5. Industrial-grade delivery

Standard Containers make INDUSTRIAL-GRADE DELIVERY of software a reality.
Leveraging all of the properties listed above, Standard Containers are enabling large and small enterprises to streamline and automate their software delivery pipelines.
Whether it is in-house devOps flows, or external customer-based software delivery mechanisms, Standard Containers are changing the way the community thinks about software packaging and delivery.

Filesystem Bundle

Container Format

This section defines a format for encoding a container as a *filesystem bundle* - a set of files organized in a certain way, and containing all the necessary data and metadata for any compliant runtime to perform all standard operations against it.
See also MacOS application bundles for a similar use of the term *bundle*.

The definition of a bundle is only concerned with how a container, and its configuration data, are stored on a local filesystem so that it can be consumed by a compliant runtime.

A Standard Container bundle contains all the information needed to load and run a container.
This includes the following artifacts:

1. `config.json`: contains configuration data.
   This REQUIRED file MUST reside in the root of the bundle directory and MUST be named `config.json`.
   See `config.json` for more details.

2. container's root filesystem: the directory referenced by `root.path`, if that property is set in `config.json`.

When supplied, while these artifacts MUST all be present in a single directory on the local filesystem, that directory itself is not part of the bundle.
In other words, a tar archive of a *bundle* will have these artifacts at the root of the archive, not nested within a top-level directory.

Runtime and Lifecycle

Scope of a Container

The entity using a runtime to create a container MUST be able to use the operations defined in this specification against that same container.
Whether other entities using the same, or other, instance of the runtime can see that container is out of scope of this specification.

State

The state of a container includes the following properties:

- **`ociVersion`** (string, REQUIRED) is version of the Open Container Initiative Runtime Specification with which the state complies.
- **`id`** (string, REQUIRED) is the container's ID.
  This MUST be unique across all containers on this host.
  There is no requirement that it be unique across hosts.
- **`status`** (string, REQUIRED) is the runtime state of the container.
  The value MAY be one of:
  - `creating`: the container is being created (step 2 in the lifecycle)
  - `created`: the runtime has finished the create operation (after step 2 in the lifecycle), and the container process has neither exited nor executed the user-specified program
  - `running`: the container process has executed the user-specified program but has not exited (after step 5 in the lifecycle)
  - `stopped`: the container process has exited (step 7 in the lifecycle)

  Additional values MAY be defined by the runtime, however, they MUST be used to represent new runtime states not defined above.
- **`pid`** (int, REQUIRED when `status` is `created` or `running` on Linux, OPTIONAL on other platforms) is the ID of the container process, as seen by the host.
- **`bundle`** (string, REQUIRED) is the absolute path to the container's bundle directory.
  This is provided so that consumers can find the container's configuration and root filesystem on the host.
- **`annotations`** (map, OPTIONAL) contains the list of annotations associated with the container.
  If no annotations were provided then this property MAY either be absent or an empty map.

The state MAY include additional properties.

When serialized in JSON, the format MUST adhere to the following pattern:

```
{
    "ociVersion": "0.2.0",
    "id": "oci-container1",
    "status": "running",
    "pid": 4422,
    "bundle": "/containers/redis",
    "annotations": {
        "myKey": "myValue"
    }
}
```

See Query State for information on retrieving the state of a container.

Lifecycle

The lifecycle describes the timeline of events that happen from when a container is created to when it ceases to exist.

1. OCI compliant runtime's `create` command is invoked with a reference to the location of the bundle and a unique identifier.
2. The container's runtime environment MUST be created according to the configuration in `config.json`.
   If the runtime is unable to create the environment specified in the `config.json`, it MUST generate an error.
   While the resources requested in the `config.json` MUST be created, the user-specified program (from `process`) MUST NOT be run at this time.
   Any updates to `config.json` after this step MUST NOT affect the container.
3. The `prestart` hooks MUST be invoked by the runtime.
   If any `prestart` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
4. The `createRuntime` hooks MUST be invoked by the runtime.
   If any `createRuntime` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
5. The `createContainer` hooks MUST be invoked by the runtime.
   If any `createContainer` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
6. Runtime's `start` command is invoked with the unique identifier of the container.

7. The `startContainer` hooks MUST be invoked by the runtime.
   If any `startContainer` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
8. The runtime MUST run the user-specified program, as specified by `process`.
9. The `poststart` hooks MUST be invoked by the runtime.
   If any `poststart` hook fails, the runtime MUST log a warning, but the remaining hooks and lifecycle continue as if the hook had succeeded.
10. The container process exits.
    This MAY happen due to erroring out, exiting, crashing or the runtime's `kill` operation being invoked.
11. Runtime's `delete` command is invoked with the unique identifier of the container.
12. The container MUST be destroyed by undoing the steps performed during create phase (step 2).
13. The `poststop` hooks MUST be invoked by the runtime.
    If any `poststop` hook fails, the runtime MUST log a warning, but the remaining hooks and lifecycle continue as if the hook had succeeded.

Errors

In cases where the specified operation generates an error, this specification does not mandate how, or even if, that error is returned or exposed to the user of an implementation.
Unless otherwise stated, generating an error MUST leave the state of the environment as if the operation were never attempted - modulo any possible trivial ancillary changes such as logging.

Warnings

In cases where the specified operation logs a warning, this specification does not mandate how, or even if, that warning is returned or exposed to the user of an implementation.
Unless otherwise stated, logging a warning does not change the flow of the operation; it MUST continue as if the warning had not been logged.

Operations

Unless otherwise stated, runtimes MUST support the following operations.

Note: these operations are not specifying any command-line APIs, and the parameters are inputs for general operations.

Query State

`state <container-id>`

This operation MUST generate an error if it is not provided the ID of a container.
Attempting to query a container that does not exist MUST generate an error.
This operation MUST return the state of a container as specified in the State section.

Create

`create <container-id> <path-to-bundle>`

This operation MUST generate an error if it is not provided a path to the bundle and the container ID to associate with the container.
If the ID provided is not unique across all containers within the scope of the runtime, or is not valid in any other way, the implementation MUST generate an error and a new container MUST NOT be created.
This operation MUST create a new container.

All of the properties configured in `config.json` except for `process` MUST be applied.
`process.args` MUST NOT be applied until triggered by the `start` operation.
The remaining `process` properties MAY be applied by this operation.
If the runtime cannot apply a property as specified in the configuration, it MUST generate an error and a new container MUST NOT be created.

The runtime MAY validate `config.json` against this spec, either generically or with respect to the local system capabilities, before creating the container (step 2).
Runtime callers who are interested in pre-create validation can run bundle-validation tools before invoking the create operation.

Any changes made to the `config.json` file after this operation will not have an effect on the container.

Start

`start <container-id>`

This operation MUST generate an error if it is not provided the container ID.
Attempting to `start` a container that is not `created` MUST have no effect on the container and MUST generate an error.
This operation MUST run the user-specified program as specified by `process`.
This operation MUST generate an error if `process` was not set.

Kill

`kill <container-id> <signal>`

This operation MUST generate an error if it is not provided the container ID.
Attempting to send a signal to a container that is neither `created` nor `running` MUST have no effect on the container and MUST generate an error.
This operation MUST send the specified signal to the container process.

Delete

`delete <container-id>`

This operation MUST generate an error if it is not provided the container ID.
Attempting to `delete` a container that is not `stopped` MUST have no effect on the container and MUST generate an error.
Deleting a container MUST delete the resources that were created during the `create` step.
Note that resources associated with the container, but not created by this container, MUST NOT be deleted.
Once a container is deleted its ID MAY be used by a subsequent container.

Hooks

Many of the operations specified in this specification have "hooks" that allow for additional actions to be taken before or after each operation.
See runtime configuration for hooks for more information.

Linux Runtime

File descriptors

By default, only the `stdin`, `stdout` and `stderr` file descriptors are kept open for the application by the runtime.
The runtime MAY pass additional file descriptors to the application to support features such as socket activation.
Some of the file descriptors MAY be redirected to `/dev/null` even though they are open.

Dev symbolic links

While creating the container (step 2 in the lifecycle), runtimes MUST create the following symlinks if the source file exists after processing `mounts`:

Source	Destination
/proc/self/fd	/dev/fd
/proc/self/fd/0	/dev/stdin
/proc/self/fd/1	/dev/stdout
/proc/self/fd/2	/dev/stderr

Configuration

This configuration file contains metadata necessary to implement standard operations against the container.
This includes the process to run, environment variables to inject, sandboxing features to use, etc.

The canonical schema is defined in this document, but there is a JSON Schema in `schema/config-schema.json` and Go bindings in `specs-go/config.go`.
Platform-specific configuration schema are defined in the platform-specific documents linked below.
For properties that are only defined for some platforms, the Go property has a `platform` tag listing those protocols (e.g. `platform:"linux,solaris"`).

Below is a detailed description of each field defined in the configuration format and valid values are specified.
Platform-specific fields are identified as such.
For all platform-specific configuration values, the scope defined below in the Platform-specific configuration section applies.

Specification version

- **`ociVersion`** (string, REQUIRED) MUST be in SemVer v2.0.0 format and specifies the version of the Open Container Initiative Runtime Specification with which the bundle complies.
  The Open Container Initiative Runtime Specification follows semantic versioning and retains forward and backward compatibility within major versions.
  For example, if a configuration is compliant with version 1.1 of this specification, it is compatible with all runtimes that support any 1.1 or later release of this specification, but is not compatible with a runtime that supports 1.0 and not 1.1.

Example

```
"ociVersion": "0.1.0"
```

Root

root (object, OPTIONAL) specifies the container's root filesystem.
On Windows, for Windows Server Containers, this field is REQUIRED.
For Hyper-V Containers, this field MUST NOT be set.

On all other platforms, this field is REQUIRED.

- **path** (string, REQUIRED) Specifies the path to the root filesystem for the container.
  - On Windows, **path** MUST be a volume GUID path.
  - On POSIX platforms, **path** is either an absolute path or a relative path to the bundle.
    For example, with a bundle at **/to/bundle** and a root filesystem at **/to/bundle/rootfs**, the **path** value can be either **/to/bundle/rootfs** or **rootfs**.
    The value SHOULD be the conventional **rootfs**.

  A directory MUST exist at the path declared by the field.
- **readonly** (bool, OPTIONAL) If true then the root filesystem MUST be read-only inside the container, defaults to false.
  - On Windows, this field MUST be omitted or false.

Example (POSIX platforms)

```
"root": {
    "path": "rootfs",
    "readonly": true
}
```

Example (Windows)

```
"root": {
    "path": "\\\\?\\Volume{ec84d99e-3f02-11e7-ac6c-00155d7682cf}\\"
}
```

Mounts

mounts (array of objects, OPTIONAL) specifies additional mounts beyond root.
The runtime MUST mount entries in the listed order.
For Linux, the parameters are as documented in mount(2) system call man page.
For Solaris, the mount entry corresponds to the 'fs' resource in the zonecfg(1M) man page.

- **`destination`** (string, REQUIRED) Destination of mount point: path inside container.
  This value MUST be an absolute path.
  - Windows: one mount destination MUST NOT be nested within another mount (e.g., c:\foo and c:\foo\bar).
  - Solaris: corresponds to "dir" of the fs resource in zonecfg(1M).
- **`source`** (string, OPTIONAL) A device name, but can also be a file or directory name for bind mounts or a dummy.
  Path values for bind mounts are either absolute or relative to the bundle.
  A mount is a bind mount if it has either **`bind`** or **`rbind`** in the options.
  - Windows: a local directory on the filesystem of the container host. UNC paths and mapped drives are not supported.
  - Solaris: corresponds to "special" of the fs resource in zonecfg(1M).
- **`options`** (array of strings, OPTIONAL) Mount options of the filesystem to be used.
  - Linux: supported options are listed in the mount(8) man page.
    Note both filesystem-independent and filesystem-specific options are listed.
  - Solaris: corresponds to "options" of the fs resource in zonecfg(1M).
  - Windows: runtimes MUST support **`ro`**, mounting the filesystem read-only when **`ro`** is given.

Example (Windows)

```
"mounts": [
    {
        "destination": "C:\\folder-inside-container",
        "source": "C:\\folder-on-host",
        "options": ["ro"]
    }
]
```

POSIX-platform Mounts

For POSIX platforms the **`mounts`** structure has the following fields:

- **`type`** (string, OPTIONAL) The type of the filesystem to be mounted.
  - Linux: filesystem types supported by the kernel as listed in */proc/filesystems* (e.g., "minix", "ext2", "ext3", "jfs", "xfs", "reiserfs", "msdos", "proc", "nfs", "iso9660"). For bind mounts (when **`options`** include either **`bind`** or **`rbind`**), the type is a dummy, often "none" (not listed in */proc/filesystems*).

– Solaris: corresponds to “type” of the fs resource in zonecfg(1M).

Example (Linux)

```
"mounts": [
    {
        "destination": "/tmp",
        "type": "tmpfs",
        "source": "tmpfs",
        "options": ["nosuid","strictatime","mode=755","size=65536k"]
    },
    {
        "destination": "/data",
        "type": "none",
        "source": "/volumes/testing",
        "options": ["rbind","rw"]
    }
]
```

Example (Solaris)

```
"mounts": [
    {
        "destination": "/opt/local",
        "type": "lofs",
        "source": "/usr/local",
        "options": ["ro","nodevices"]
    },
    {
        "destination": "/opt/sfw",
        "type": "lofs",
        "source": "/opt/sfw"
    }
]
```

Process

process (object, OPTIONAL) specifies the container process.
This property is REQUIRED when start is called.

- **terminal** (bool, OPTIONAL) specifies whether a terminal is attached to the process, defaults to false.
  As an example, if set to true on Linux a pseudoterminal pair is allocated

for the process and the pseudoterminal slave is duplicated on the process's standard streams.

- **`consoleSize`** (object, OPTIONAL) specifies the console size in characters of the terminal.
  Runtimes MUST ignore `consoleSize` if `terminal` is `false` or unset.
  - **`height`** (uint, REQUIRED)
  - **`width`** (uint, REQUIRED)
- **`cwd`** (string, REQUIRED) is the working directory that will be set for the executable.
  This value MUST be an absolute path.
- **`env`** (array of strings, OPTIONAL) with the same semantics as IEEE Std 1003.1-2008's `environ`.
- **`args`** (array of strings, OPTIONAL) with similar semantics to IEEE Std 1003.1-2008 `execvp`'s *argv*.
  This specification extends the IEEE standard in that at least one entry is REQUIRED (non-Windows), and that entry is used with the same semantics as `execvp`'s *file*. This field is OPTIONAL on Windows, and `commandLine` is REQUIRED if this field is omitted.
- **`commandLine`** (string, OPTIONAL) specifies the full command line to be executed on Windows.
  This is the preferred means of supplying the command line on Windows. If omitted, the runtime will fall back to escaping and concatenating fields from `args` before making the system call into Windows.

POSIX process

For systems that support POSIX rlimits (for example Linux and Solaris), the `process` object supports the following process-specific properties:

- **`rlimits`** (array of objects, OPTIONAL) allows setting resource limits for the process.
  Each entry has the following structure:
  - **`type`** (string, REQUIRED) the platform resource being limited.
    - Linux: valid values are defined in the `getrlimit(2)` man page, such as `RLIMIT_MSGQUEUE`.
    - Solaris: valid values are defined in the `getrlimit(3)` man page, such as `RLIMIT_CORE`.

    The runtime MUST generate an error for any values which cannot be mapped to a relevant kernel interface.
    For each entry in `rlimits`, a `getrlimit(3)` on `type` MUST succeed.
    For the following properties, `rlim` refers to the status returned by the `getrlimit(3)` call.

  - **soft** (uint64, REQUIRED) the value of the limit enforced for the corresponding resource.
    rlim.rlim_cur MUST match the configured value.
  - **hard** (uint64, REQUIRED) the ceiling for the soft limit that could be set by an unprivileged process.
    rlim.rlim_max MUST match the configured value.
    Only a privileged process (e.g. one with the CAP_SYS_RESOURCE capability) can raise a hard limit.

If rlimits contains duplicated entries with same type, the runtime MUST generate an error.

Linux Process

For Linux-based systems, the process object supports the following process-specific properties.

- **apparmorProfile** (string, OPTIONAL) specifies the name of the AppArmor profile for the process.
  For more information about AppArmor, see AppArmor documentation.
- **capabilities** (object, OPTIONAL) is an object containing arrays that specifies the sets of capabilities for the process.
  Valid values are defined in the capabilities(7) man page, such as CAP_CHOWN.
  Any value which cannot be mapped to a relevant kernel interface MUST cause an error.
  capabilities contains the following properties:
  - **effective** (array of strings, OPTIONAL) the effective field is an array of effective capabilities that are kept for the process.
  - **bounding** (array of strings, OPTIONAL) the bounding field is an array of bounding capabilities that are kept for the process.
  - **inheritable** (array of strings, OPTIONAL) the inheritable field is an array of inheritable capabilities that are kept for the process.
  - **permitted** (array of strings, OPTIONAL) the permitted field is an array of permitted capabilities that are kept for the process.
  - **ambient** (array of strings, OPTIONAL) the ambient field is an array of ambient capabilities that are kept for the process.
- **noNewPrivileges** (bool, OPTIONAL) setting noNewPrivileges to true prevents the process from gaining additional privileges.
  As an example, the no_new_privs article in the kernel documentation has information on how this is achieved using a prctl system call on Linux.

- **`oomScoreAdj`** *(int, OPTIONAL)* adjusts the oom-killer score in `[pid]/oom_score_adj` for the process's `[pid]` in a proc pseudo-filesystem.
  If `oomScoreAdj` is set, the runtime MUST set `oom_score_adj` to the given value.
  If `oomScoreAdj` is not set, the runtime MUST NOT change the value of `oom_score_adj`.

  This is a per-process setting, where as `disableOOMKiller` is scoped for a memory cgroup.
  For more information on how these two settings work together, see the memory cgroup documentation section 10. OOM Contol.

- **`selinuxLabel`** (string, OPTIONAL) specifies the SELinux label for the process.
  For more information about SELinux, see SELinux documentation.

User

The user for the process is a platform-specific structure that allows specific control over which user the process runs as.

POSIX-platform User For POSIX platforms the `user` structure has the following fields:

- **`uid`** (int, REQUIRED) specifies the user ID in the container namespace.
- **`gid`** (int, REQUIRED) specifies the group ID in the container namespace.
- **`umask`** (int, OPTIONAL) specifies the [umask][umask_2] of the user. If unspecified, the umask should not be changed from the calling process' umask.
- **`additionalGids`** (array of ints, OPTIONAL) specifies additional group IDs in the container namespace to be added to the process.

Note: symbolic name for uid and gid, such as uname and gname respectively, are left to upper levels to derive (i.e. `/etc/passwd` parsing, NSS, etc)

Example (Linux)

```
"process": {
    "terminal": true,
    "consoleSize": {
        "height": 25,
        "width": 80
    },
```

```
"user": {
    "uid": 1,
    "gid": 1,
    "umask": 63,
    "additionalGids": [5, 6]
},
"env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "TERM=xterm"
],
"cwd": "/root",
"args": [
    "sh"
],
"apparmorProfile": "acme_secure_profile",
"selinuxLabel": "system_u:system_r:svirt_lxc_net_t:s0:c124,c675",
"noNewPrivileges": true,
"capabilities": {
    "bounding": [
        "CAP_AUDIT_WRITE",
        "CAP_KILL",
        "CAP_NET_BIND_SERVICE"
    ],
   "permitted": [
        "CAP_AUDIT_WRITE",
        "CAP_KILL",
        "CAP_NET_BIND_SERVICE"
    ],
   "inheritable": [
        "CAP_AUDIT_WRITE",
        "CAP_KILL",
        "CAP_NET_BIND_SERVICE"
    ],
    "effective": [
        "CAP_AUDIT_WRITE",
        "CAP_KILL"
    ],
    "ambient": [
        "CAP_NET_BIND_SERVICE"
    ]
},
"rlimits": [
    {
        "type": "RLIMIT_NOFILE",
        "hard": 1024,
        "soft": 1024
```

```
        }
    ]
}
```

Example (Solaris)

```
"process": {
    "terminal": true,
    "consoleSize": {
        "height": 25,
        "width": 80
    },
    "user": {
        "uid": 1,
        "gid": 1,
        "umask": 7,
        "additionalGids": [2, 8]
    },
    "env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "TERM=xterm"
    ],
    "cwd": "/root",
    "args": [
        "/usr/bin/bash"
    ]
}
```

Windows User For Windows based systems the user structure has the following fields:

- **username** (string, OPTIONAL) specifies the user name for the process.

Example (Windows)

```
"process": {
    "terminal": true,
    "user": {
        "username": "containeradministrator"
    },
    "env": [
        "VARIABLE=1"
    ],
```

```
    "cwd": "c:\\foo",
    "args": [
        "someapp.exe",
    ]
}
```

Hostname

- **`hostname`** (string, OPTIONAL) specifies the container's hostname as seen by processes running inside the container.
  On Linux, for example, this will change the hostname in the container UTS namespace.
  Depending on your namespace configuration, the container UTS namespace may be the runtime UTS namespace.

Example

```
"hostname": "mrsdalloway"
```

Platform-specific configuration

- **`linux`** (object, OPTIONAL) Linux-specific configuration.
  This MAY be set if the target platform of this spec is `linux`.
- **`windows`** (object, OPTIONAL) Windows-specific configuration.
  This MUST be set if the target platform of this spec is `windows`.
- **`solaris`** (object, OPTIONAL) Solaris-specific configuration.
  This MAY be set if the target platform of this spec is `solaris`.
- **`vm`** (object, OPTIONAL) Virtual-machine-specific configuration.
  This MAY be set if the target platform and architecture of this spec support hardware virtualization.

Example (Linux)

```
{
    "linux": {
        "namespaces": [
            {
                "type": "pid"
            }
        ]
    }
}
```

POSIX-platform Hooks

For POSIX platforms, the configuration structure supports `hooks` for configuring custom actions related to the lifecycle of the container.

- **`hooks`** (object, OPTIONAL) MAY contain any of the following properties:
  - **`prestart`** (array of objects, OPTIONAL, **DEPRECATED**) is an array of `prestart` hooks.
    - Entries in the array contain the following properties:
      - **`path`** (string, REQUIRED) with similar semantics to IEEE Std 1003.1-2008 `execv`'s *path*.
        This specification extends the IEEE standard in that **`path`** MUST be absolute.
      - **`args`** (array of strings, OPTIONAL) with the same semantics as IEEE Std 1003.1-2008 `execv`'s *argv*.
      - **`env`** (array of strings, OPTIONAL) with the same semantics as IEEE Std 1003.1-2008's `environ`.
      - **`timeout`** (int, OPTIONAL) is the number of seconds before aborting the hook.
        If set, `timeout` MUST be greater than zero.
    - The value of `path` MUST resolve in the runtime namespace.
    - The `prestart` hooks MUST be executed in the runtime namespace.
  - **`createRuntime`** (array of objects, OPTIONAL) is an array of `createRuntime` hooks.
    - Entries in the array contain the following properties (the entries are identical to the entries in the deprecated `prestart` hooks):
      - **`path`** (string, REQUIRED) with similar semantics to IEEE Std 1003.1-2008 `execv`'s *path*.
        This specification extends the IEEE standard in that **`path`** MUST be absolute.
      - **`args`** (array of strings, OPTIONAL) with the same semantics as IEEE Std 1003.1-2008 `execv`'s *argv*.
      - **`env`** (array of strings, OPTIONAL) with the same semantics as IEEE Std 1003.1-2008's `environ`.
      - **`timeout`** (int, OPTIONAL) is the number of seconds before aborting the hook.
        If set, `timeout` MUST be greater than zero.
    - The value of `path` MUST resolve in the runtime namespace.
    - The `createRuntime` hooks MUST be executed in the runtime namespace.
  - **`createContainer`** (array of objects, OPTIONAL) is an array of `createContainer` hooks.

    - Entries in the array have the same schema as `createRuntime` entries.
    - The value of `path` MUST resolve in the runtime namespace.
    - The `createContainer` hooks MUST be executed in the container namespace.
- `startContainer` (array of objects, OPTIONAL) is an array of `startContainer` hooks.
    - Entries in the array have the same schema as `createRuntime` entries.
    - The value of `path` MUST resolve in the container namespace.
    - The `startContainer` hooks MUST be executed in the container namespace.
- `poststart` (array of objects, OPTIONAL) is an array of `poststart` hooks.
    - Entries in the array have the same schema as `createRuntime` entries.
    - The value of `path` MUST resolve in the runtime namespace.
    - The `poststart` hooks MUST be executed in the runtime namespace.
- `poststop` (array of objects, OPTIONAL) is an array of `poststop` hooks.
    - Entries in the array have the same schema as `createRuntime` entries.
    - The value of `path` MUST resolve in the runtime namespace.
    - The `poststop` hooks MUST be executed in the runtime namespace.

Hooks allow users to specify programs to run before or after various lifecycle events.
Hooks MUST be called in the listed order.
The state of the container MUST be passed to hooks over stdin so that they may do work appropriate to the current state of the container.

Prestart

The `prestart` hooks MUST be called after the `start` operation is called but before the user-specified program command is executed.
On Linux, for example, they are called after the container namespaces are created, so they provide an opportunity to customize the container (e.g. the network namespace could be specified in this hook).

Note: `prestart` hooks were deprecated in favor of `createRuntime`, `createContainer` and `startContainer` hooks, which allow more granular hook control during the create and start phase.

The `prestart` hooks' path MUST resolve in the runtime namespace.
The `prestart` hooks MUST be executed in the runtime namespace.

CreateRuntime Hooks

The `createRuntime` hooks MUST be called as part of the `create` operation after the runtime environment has been created (according to the configuration in config.json) but before the `pivot_root` or any equivalent operation has been executed.

The `createRuntime` hooks' path MUST resolve in the runtime namespace.
The `createRuntime` hooks MUST be executed in the runtime namespace.

On Linux, for example, they are called after the container namespaces are created, so they provide an opportunity to customize the container (e.g. the network namespace could be specified in this hook).

The definition of `createRuntime` hooks is currently underspecified and hooks authors, should only expect from the runtime that the mount namespace have been created and the mount operations performed. Other operations such as cgroups and SELinux/AppArmor labels might not have been performed by the runtime.

Note: `runc` originally implemented `prestart` hooks contrary to the spec, namely as part of the `create` operation (instead of during the `start` operation). This incorrect implementation actually corresponds to `createRuntime` hooks. For runtimes that implement the deprecated `prestart` hooks as `createRuntime` hooks, `createRuntime` hooks MUST be called after the `prestart` hooks.

CreateContainer Hooks

The `createContainer` hooks MUST be called as part of the `create` operation after the runtime environment has been created (according to the configuration in config.json) but before the `pivot_root` or any equivalent operation has been executed.
The `createContainer` hooks MUST be called after the `createRuntime` hooks.

The `createContainer` hooks' path MUST resolve in the runtime namespace.
The `createContainer` hooks MUST be executed in the container namespace.

For example, on Linux this would happen before the `pivot_root` operation is executed but after the mount namespace was created and setup.

The definition of `createContainer` hooks is currently underspecified and hooks authors, should only expect from the runtime that the mount namespace and different mounts will be setup. Other operations such as cgroups and SELinux/AppArmor labels might not have been performed by the runtime.

StartContainer Hooks

The `startContainer` hooks MUST be called before the user-specified process is executed as part of the `start` operation.
This hook can be used to execute some operations in the container, for example running the `ldconfig` binary on linux before the container process is spawned.

The `startContainer` hooks' path MUST resolve in the container namespace.
The `startContainer` hooks MUST be executed in the container namespace.

Poststart

The `poststart` hooks MUST be called after the user-specified process is executed but before the `start` operation returns.
For example, this hook can notify the user that the container process is spawned.

The `poststart` hooks' path MUST resolve in the runtime namespace.
The `poststart` hooks MUST be executed in the runtime namespace.

Poststop

The `poststop` hooks MUST be called after the container is deleted but before the `delete` operation returns.
Cleanup or debugging functions are examples of such a hook.

The `poststop` hooks' path MUST resolve in the runtime namespace.
The `poststop` hooks MUST be executed in the runtime namespace.

Summary

See the below table for a summary of hooks and when they are called:

Name	Namespace	When
`prestart` (Deprecated)	runtime	After the start operation is called but before the user-specified progra
`createRuntime`	runtime	During the create operation, after the runtime environment has been
`createContainer`	container	During the create operation, after the runtime environment has been
`startContainer`	container	After the start operation is called but before the user-specified progra
`poststart`	runtime	After the user-specified process is executed but before the start opera
`poststop`	runtime	After the container is deleted but before the delete operation returns.

Example

```
"hooks": {
    "prestart": [
        {
            "path": "/usr/bin/fix-mounts",
            "args": ["fix-mounts", "arg1", "arg2"],
            "env":  [ "key1=value1"]
        },
        {
            "path": "/usr/bin/setup-network"
        }
    ],
    "createRuntime": [
        {
            "path": "/usr/bin/fix-mounts",
            "args": ["fix-mounts", "arg1", "arg2"],
            "env":  [ "key1=value1"]
        },
        {
            "path": "/usr/bin/setup-network"
        }
    ],
    "createContainer": [
        {
            "path": "/usr/bin/mount-hook",
            "args": ["-mount", "arg1", "arg2"],
            "env":  [ "key1=value1"]
        }
    ],
    "startContainer": [
        {
            "path": "/usr/bin/refresh-ldcache"
        }
    ],
    "poststart": [
        {
            "path": "/usr/bin/notify-start",
            "timeout": 5
        }
    ],
    "poststop": [
        {
            "path": "/usr/sbin/cleanup.sh",
            "args": ["cleanup.sh", "-f"]
        }
```

```
    ]
}
```

Annotations

annotations (object, OPTIONAL) contains arbitrary metadata for the container.
This information MAY be structured or unstructured.
Annotations MUST be a key-value map.
If there are no annotations then this property MAY either be absent or an empty map.

Keys MUST be strings.
Keys MUST NOT be an empty string.
Keys SHOULD be named using a reverse domain notation - e.g. `com.example.myKey`.
Keys using the `org.opencontainers` namespace are reserved and MUST NOT be used by subsequent specifications.
Runtimes MUST handle unknown annotation keys like any other unknown property.

Values MUST be strings.
Values MAY be an empty string.

```
"annotations": {
    "com.example.gpu-cores": "2"
}
```

Extensibility

Runtimes MAY log unknown properties but MUST otherwise ignore them.
That includes not generating errors if they encounter an unknown property.

Valid values

Runtimes MUST generate an error when invalid or unsupported values are encountered.
Unless support for a valid value is explicitly required, runtimes MAY choose which subset of the valid values it will support.

Configuration Schema Example

Here is a full example `config.json` for reference.

```
{
    "ociVersion": "1.0.1",
    "process": {
        "terminal": true,
        "user": {
            "uid": 1,
            "gid": 1,
            "additionalGids": [
                5,
                6
            ]
        },
        "args": [
            "sh"
        ],
        "env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
            "TERM=xterm"
        ],
        "cwd": "/",
        "capabilities": {
            "bounding": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ],
            "permitted": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ],
            "inheritable": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL",
                "CAP_NET_BIND_SERVICE"
            ],
            "effective": [
                "CAP_AUDIT_WRITE",
                "CAP_KILL"
            ],
            "ambient": [
                "CAP_NET_BIND_SERVICE"
            ]
        },
        "rlimits": [
            {
```

```
            "type": "RLIMIT_CORE",
            "hard": 1024,
            "soft": 1024
        },
        {
            "type": "RLIMIT_NOFILE",
            "hard": 1024,
            "soft": 1024
        }
    ],
    "apparmorProfile": "acme_secure_profile",
    "oomScoreAdj": 100,
    "selinuxLabel": "system_u:system_r:svirt_lxc_net_t:s0:c124,c675",
    "noNewPrivileges": true
},
"root": {
    "path": "rootfs",
    "readonly": true
},
"hostname": "slartibartfast",
"mounts": [
    {
        "destination": "/proc",
        "type": "proc",
        "source": "proc"
    },
    {
        "destination": "/dev",
        "type": "tmpfs",
        "source": "tmpfs",
        "options": [
            "nosuid",
            "strictatime",
            "mode=755",
            "size=65536k"
        ]
    },
    {
        "destination": "/dev/pts",
        "type": "devpts",
        "source": "devpts",
        "options": [
            "nosuid",
            "noexec",
            "newinstance",
            "ptmxmode=0666",
```

```
            "mode=0620",
            "gid=5"
        ]
    },
    {
        "destination": "/dev/shm",
        "type": "tmpfs",
        "source": "shm",
        "options": [
            "nosuid",
            "noexec",
            "nodev",
            "mode=1777",
            "size=65536k"
        ]
    },
    {
        "destination": "/dev/mqueue",
        "type": "mqueue",
        "source": "mqueue",
        "options": [
            "nosuid",
            "noexec",
            "nodev"
        ]
    },
    {
        "destination": "/sys",
        "type": "sysfs",
        "source": "sysfs",
        "options": [
            "nosuid",
            "noexec",
            "nodev"
        ]
    },
    {
        "destination": "/sys/fs/cgroup",
        "type": "cgroup",
        "source": "cgroup",
        "options": [
            "nosuid",
            "noexec",
            "nodev",
            "relatime",
            "ro"
```

```
            ]
        }
    ],
    "hooks": {
        "prestart": [
            {
                "path": "/usr/bin/fix-mounts",
                "args": [
                    "fix-mounts",
                    "arg1",
                    "arg2"
                ],
                "env": [
                    "key1=value1"
                ]
            },
            {
                "path": "/usr/bin/setup-network"
            }
        ],
        "poststart": [
            {
                "path": "/usr/bin/notify-start",
                "timeout": 5
            }
        ],
        "poststop": [
            {
                "path": "/usr/sbin/cleanup.sh",
                "args": [
                    "cleanup.sh",
                    "-f"
                ]
            }
        ]
    },
    "linux": {
        "devices": [
            {
                "path": "/dev/fuse",
                "type": "c",
                "major": 10,
                "minor": 229,
                "fileMode": 438,
                "uid": 0,
                "gid": 0
```

```
        },
        {
            "path": "/dev/sda",
            "type": "b",
            "major": 8,
            "minor": 0,
            "fileMode": 432,
            "uid": 0,
            "gid": 0
        }
    ],
    "uidMappings": [
        {
            "containerID": 0,
            "hostID": 1000,
            "size": 32000
        }
    ],
    "gidMappings": [
        {
            "containerID": 0,
            "hostID": 1000,
            "size": 32000
        }
    ],
    "sysctl": {
        "net.ipv4.ip_forward": "1",
        "net.core.somaxconn": "256"
    },
    "cgroupsPath": "/myRuntime/myContainer",
    "resources": {
        "network": {
            "classID": 1048577,
            "priorities": [
                {
                    "name": "eth0",
                    "priority": 500
                },
                {
                    "name": "eth1",
                    "priority": 1000
                }
            ]
        },
        "pids": {
            "limit": 32771
```

```
},
"hugepageLimits": [
    {
        "pageSize": "2MB",
        "limit": 9223372036854772000
    },
    {
        "pageSize": "64KB",
        "limit": 1000000
    }
],
"memory": {
    "limit": 536870912,
    "reservation": 536870912,
    "swap": 536870912,
    "kernel": -1,
    "kernelTCP": -1,
    "swappiness": 0,
    "disableOOMKiller": false
},
"cpu": {
    "shares": 1024,
    "quota": 1000000,
    "period": 500000,
    "realtimeRuntime": 950000,
    "realtimePeriod": 1000000,
    "cpus": "2-3",
    "mems": "0-7"
},
"devices": [
    {
        "allow": false,
        "access": "rwm"
    },
    {
        "allow": true,
        "type": "c",
        "major": 10,
        "minor": 229,
        "access": "rw"
    },
    {
        "allow": true,
        "type": "b",
        "major": 8,
        "minor": 0,
```

```
            "access": "r"
        }
    ],
    "blockIO": {
        "weight": 10,
        "leafWeight": 10,
        "weightDevice": [
            {
                "major": 8,
                "minor": 0,
                "weight": 500,
                "leafWeight": 300
            },
            {
                "major": 8,
                "minor": 16,
                "weight": 500
            }
        ],
        "throttleReadBpsDevice": [
            {
                "major": 8,
                "minor": 0,
                "rate": 600
            }
        ],
        "throttleWriteIOPSDevice": [
            {
                "major": 8,
                "minor": 16,
                "rate": 300
            }
        ]
    }
},
"rootfsPropagation": "slave",
"seccomp": {
    "defaultAction": "SCMP_ACT_ALLOW",
    "architectures": [
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "names": [
                "getcwd",
```

```
                    "chmod"
                ],
                "action": "SCMP_ACT_ERRNO"
            }
        ]
    },
    "namespaces": [
        {
            "type": "pid"
        },
        {
            "type": "network"
        },
        {
            "type": "ipc"
        },
        {
            "type": "uts"
        },
        {
            "type": "mount"
        },
        {
            "type": "user"
        },
        {
            "type": "cgroup"
        }
    ],
    "maskedPaths": [
        "/proc/kcore",
        "/proc/latency_stats",
        "/proc/timer_stats",
        "/proc/sched_debug"
    ],
    "readonlyPaths": [
        "/proc/asound",
        "/proc/bus",
        "/proc/fs",
        "/proc/irq",
        "/proc/sys",
        "/proc/sysrq-trigger"
    ],
    "mountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c715,c811"
},
"annotations": {
```

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

Linux Container Configuration

This document describes the schema for the Linux-specific section of the container configuration.
The Linux container specification uses various kernel features like namespaces, cgroups, capabilities, LSM, and filesystem jails to fulfill the spec.

Default Filesystems

The Linux ABI includes both syscalls and several special file paths.
Applications expecting a Linux environment will very likely expect these file paths to be set up correctly.

The following filesystems SHOULD be made available in each container's filesystem:

Path	Type
/proc	proc
/sys	sysfs
/dev/pts	devpts
/dev/shm	tmpfs

Namespaces

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.
Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.
For more information, see the namespaces(7) man page.

Namespaces are specified as an array of entries inside the `namespaces` root field.
The following parameters can be specified to set up namespaces:

- **`type`** *(string, REQUIRED)* - namespace type. The following namespace types SHOULD be supported:

  - **pid** processes inside the container will only be able to see other processes inside the same container or inside the same pid namespace.
  - **network** the container will have its own network stack.
  - **mount** the container will have an isolated mount table.
  - **ipc** processes inside the container will only be able to communicate to other processes inside the same container via system level IPC.
  - **uts** the container will be able to have its own hostname and domain name.
  - **user** the container will be able to remap user and group IDs from the host to local users and groups within the container.
  - **cgroup** the container will have an isolated view of the cgroup hierarchy.

- **path** *(string, OPTIONAL)* - namespace file.
  This value MUST be an absolute path in the runtime mount namespace.
  The runtime MUST place the container process in the namespace associated with that **path**.
  The runtime MUST generate an error if **path** is not associated with a namespace of type **type**.

  If **path** is not specified, the runtime MUST create a new container namespace of type **type**.

If a namespace type is not specified in the **namespaces** array, the container MUST inherit the runtime namespace of that type.
If a **namespaces** field contains duplicated namespaces with same **type**, the runtime MUST generate an error.

Example

```
"namespaces": [
    {
        "type": "pid",
        "path": "/proc/1234/ns/pid"
    },
    {
        "type": "network",
        "path": "/var/run/netns/neta"
    },
    {
        "type": "mount"
    },
    {
        "type": "ipc"
    },
    {
```

```
        "type": "uts"
    },
    {
        "type": "user"
    },
    {
        "type": "cgroup"
    }
]
```

User namespace mappings

uidMappings (array of objects, OPTIONAL) describes the user namespace uid mappings from the host to the container.
gidMappings (array of objects, OPTIONAL) describes the user namespace gid mappings from the host to the container.

Each entry has the following structure:

- **containerID** *(uint32, REQUIRED)* - is the starting uid/gid in the container.
- **hostID** *(uint32, REQUIRED)* - is the starting uid/gid on the host to be mapped to *containerID*.
- **size** *(uint32, REQUIRED)* - is the number of ids to be mapped.

The runtime SHOULD NOT modify the ownership of referenced filesystems to realize the mapping.
Note that the number of mapping entries MAY be limited by the kernel.

Example

```
"uidMappings": [
    {
        "containerID": 0,
        "hostID": 1000,
        "size": 32000
    }
],
"gidMappings": [
    {
        "containerID": 0,
        "hostID": 1000,
        "size": 32000
    }
]
```

Devices

`devices` (array of objects, OPTIONAL) lists devices that MUST be available in the container.
The runtime MAY supply them however it likes (with **`mknod`**, by bind mounting from the runtime mount namespace, using symlinks, etc.).

Each entry has the following structure:

- **`type`** *(string, REQUIRED)* - type of device: `c`, `b`, `u` or `p`.
  More info in mknod(1).
- **`path`** *(string, REQUIRED)* - full path to device inside container.
  If a file already exists at `path` that does not match the requested device, the runtime MUST generate an error.
- **`major, minor`** *(int64, REQUIRED unless `type` is `p`)* - major, minor numbers for the device.
- **`fileMode`** *(uint32, OPTIONAL)* - file mode for the device.
  You can also control access to devices with cgroups.
- **`uid`** *(uint32, OPTIONAL)* - id of device owner in the container namespace.
- **`gid`** *(uint32, OPTIONAL)* - id of device group in the container namespace.

The same `type`, `major` and `minor` SHOULD NOT be used for multiple devices.

Example

```
"devices": [
    {
        "path": "/dev/fuse",
        "type": "c",
        "major": 10,
        "minor": 229,
        "fileMode": 438,
        "uid": 0,
        "gid": 0
    },
    {
        "path": "/dev/sda",
        "type": "b",
        "major": 8,
        "minor": 0,
        "fileMode": 432,
        "uid": 0,
        "gid": 0
    }
]
```

Default Devices

In addition to any devices configured with this setting, the runtime MUST also supply:

- `/dev/null`
- `/dev/zero`
- `/dev/full`
- `/dev/random`
- `/dev/urandom`
- `/dev/tty`
- `/dev/console` is set up if `terminal` is enabled in the config by bind mounting the pseudoterminal slave to `/dev/console`.
- `/dev/ptmx`.
  A bind-mount or symlink of the container's `/dev/pts/ptmx`.

Control groups

Also known as cgroups, they are used to restrict resource usage for a container and handle device access.
cgroups provide controls (through controllers) to restrict cpu, memory, IO, pids, network and RDMA resources for the container.
For more information, see the kernel cgroups documentation.

Cgroups Path

`cgroupsPath` (string, OPTIONAL) path to the cgroups.
It can be used to either control the cgroups hierarchy for containers or to run a new process in an existing container.

The value of `cgroupsPath` MUST be either an absolute path or a relative path.

- In the case of an absolute path (starting with `/`), the runtime MUST take the path to be relative to the cgroups mount point.
- In the case of a relative path (not starting with `/`), the runtime MAY interpret the path relative to a runtime-determined location in the cgroups hierarchy.

If the value is specified, the runtime MUST consistently attach to the same place in the cgroups hierarchy given the same value of `cgroupsPath`.
If the value is not specified, the runtime MAY define the default cgroups path.
Runtimes MAY consider certain `cgroupsPath` values to be invalid, and MUST generate an error if this is the case.

Implementations of the Spec can choose to name cgroups in any manner.
The Spec does not include naming schema for cgroups.
The Spec does not support per-controller paths for the reasons discussed in the cgroupv2 documentation.
The cgroups will be created if they don't exist.

You can configure a container's cgroups via the `resources` field of the Linux configuration.
Do not specify `resources` unless limits have to be updated.
For example, to run a new process in an existing container without updating limits, `resources` need not be specified.

Runtimes MAY attach the container process to additional cgroup controllers beyond those necessary to fulfill the `resources` settings.

Example

```
"cgroupsPath": "/myRuntime/myContainer",
"resources": {
    "memory": {
    "limit": 100000,
    "reservation": 200000
    },
    "devices": [
        {
            "allow": false,
            "access": "rwm"
        }
    ]
}
```

Device whitelist

`devices` (array of objects, OPTIONAL) configures the device whitelist.
The runtime MUST apply entries in the listed order.

Each entry has the following structure:

- `allow` *(boolean, REQUIRED)* - whether the entry is allowed or denied.
- `type` *(string, OPTIONAL)* - type of device: `a` (all), `c` (char), or `b` (block). Unset values mean "all", mapping to `a`.
- `major, minor` *(int64, OPTIONAL)* - major, minor numbers for the device. Unset values mean "all", mapping to `*` in the filesystem API.
- `access` *(string, OPTIONAL)* - cgroup permissions for device. A composition of `r` (read), `w` (write), and `m` (mknod).

Example

```
"devices": [
    {
        "allow": false,
        "access": "rwm"
    },
    {
        "allow": true,
        "type": "c",
        "major": 10,
        "minor": 229,
        "access": "rw"
    },
    {
        "allow": true,
        "type": "b",
        "major": 8,
        "minor": 0,
        "access": "r"
    }
]
```

Memory

`memory` (object, OPTIONAL) represents the cgroup subsystem **`memory`** and it's used to set limits on the container's memory usage.
For more information, see the kernel cgroups documentation about memory.

Values for memory specify the limit in bytes, or **`-1`** for unlimited memory.

- **`limit`** *(int64, OPTIONAL)* - sets limit of memory usage
- **`reservation`** *(int64, OPTIONAL)* - sets soft limit of memory usage
- **`swap`** *(int64, OPTIONAL)* - sets limit of memory+Swap usage
- **`kernel`** *(int64, OPTIONAL)* - sets hard limit for kernel memory
- **`kernelTCP`** *(int64, OPTIONAL)* - sets hard limit for kernel TCP buffer memory

The following properties do not specify memory limits, but are covered by the **`memory`** controller:

- **`swappiness`** *(uint64, OPTIONAL)* - sets swappiness parameter of vmscan (See sysctl's vm.swappiness)
  The values are from 0 to 100. Higher means more swappy.

- **disableOOMKiller** *(bool, OPTIONAL)* - enables or disables the OOM killer.
  If enabled (**false**), tasks that attempt to consume more memory than they are allowed are immediately killed by the OOM killer.
  The OOM killer is enabled by default in every cgroup using the **memory** subsystem.
  To disable it, specify a value of **true**.
- **useHierarchy** *(bool, OPTIONAL)* - enables or disables hierarchical memory accounting.
  If enabled (**true**), child cgroups will share the memory limits of this cgroup.

Example

```
"memory": {
    "limit": 536870912,
    "reservation": 536870912,
    "swap": 536870912,
    "kernel": -1,
    "kernelTCP": -1,
    "swappiness": 0,
    "disableOOMKiller": false
}
```

CPU

cpu (object, OPTIONAL) represents the cgroup subsystems **cpu** and **cpusets**. For more information, see the kernel cgroups documentation about cpusets.

The following parameters can be specified to set up the controller:

- **shares** *(uint64, OPTIONAL)* - specifies a relative share of CPU time available to the tasks in a cgroup
- **quota** *(int64, OPTIONAL)* - specifies the total amount of time in microseconds for which all tasks in a cgroup can run during one period (as defined by **period** below)
- **period** *(uint64, OPTIONAL)* - specifies a period of time in microseconds for how regularly a cgroup's access to CPU resources should be reallocated (CFS scheduler only)
- **realtimeRuntime** *(int64, OPTIONAL)* - specifies a period of time in microseconds for the longest continuous period in which the tasks in a cgroup have access to CPU resources
- **realtimePeriod** *(uint64, OPTIONAL)* - same as **period** but applies to realtime scheduler only
- **cpus** *(string, OPTIONAL)* - list of CPUs the container will run in

- **mems** *(string, OPTIONAL)* - list of Memory Nodes the container will run in

Example

```
"cpu": {
    "shares": 1024,
    "quota": 1000000,
    "period": 500000,
    "realtimeRuntime": 950000,
    "realtimePeriod": 1000000,
    "cpus": "2-3",
    "mems": "0-7"
}
```

Block IO

blockIO (object, OPTIONAL) represents the cgroup subsystem **blkio** which implements the block IO controller.
For more information, see the kernel cgroups documentation about blkio.

The following parameters can be specified to set up the controller:

- **weight** *(uint16, OPTIONAL)* - specifies per-cgroup weight. This is default weight of the group on all devices until and unless overridden by per-device rules.
- **leafWeight** *(uint16, OPTIONAL)* - equivalents of **weight** for the purpose of deciding how much weight tasks in the given cgroup has while competing with the cgroup's child cgroups.
- **weightDevice** *(array of objects, OPTIONAL)* - an array of per-device bandwidth weights.
  Each entry has the following structure:
  - **major, minor** *(int64, REQUIRED)* - major, minor numbers for device.
    For more information, see the mknod(1) man page.
  - **weight** *(uint16, OPTIONAL)* - bandwidth weight for the device.
  - **leafWeight** *(uint16, OPTIONAL)* - bandwidth weight for the device while competing with the cgroup's child cgroups, CFQ scheduler only

  You MUST specify at least one of **weight** or **leafWeight** in a given entry, and MAY specify both.
- **throttleReadBpsDevice**, **throttleWriteBpsDevice** *(array of objects, OPTIONAL)* - an array of per-device bandwidth rate limits.
  Each entry has the following structure:

  - **major, minor** *(int64, REQUIRED)* - major, minor numbers for device.
    For more information, see the mknod(1) man page.
  - **rate** *(uint64, REQUIRED)* - bandwidth rate limit in bytes per second for the device

- **throttleReadIOPSDevice**, **throttleWriteIOPSDevice** *(array of objects, OPTIONAL)* - an array of per-device IO rate limits.
  Each entry has the following structure:
  - **major, minor** *(int64, REQUIRED)* - major, minor numbers for device.
    For more information, see the mknod(1) man page.
  - **rate** *(uint64, REQUIRED)* - IO rate limit for the device

Example

```
"blockIO": {
    "weight": 10,
    "leafWeight": 10,
    "weightDevice": [
        {
            "major": 8,
            "minor": 0,
            "weight": 500,
            "leafWeight": 300
        },
        {
            "major": 8,
            "minor": 16,
            "weight": 500
        }
    ],
    "throttleReadBpsDevice": [
        {
            "major": 8,
            "minor": 0,
            "rate": 600
        }
    ],
    "throttleWriteIOPSDevice": [
        {
            "major": 8,
            "minor": 16,
            "rate": 300
        }
```

```
    ]
}
```

Huge page limits

`hugepageLimits` (array of objects, OPTIONAL) represents the `hugetlb` controller which allows to limit the
HugeTLB usage per control group and enforces the controller limit during page fault.
For more information, see the kernel cgroups documentation about HugeTLB.

Each entry has the following structure:

- `pageSize` *(string, REQUIRED)* - hugepage size
  The value has the format `<size><unit-prefix>B` (64KB, 2MB, 1GB), and must match the `<hugepagesize>` of the
  corresponding control file found in `/sys/fs/cgroup/hugetlb/hugetlb.<hugepagesize>.limit_in_byte`
  Values of `<unit-prefix>` are intended to be parsed using base 1024 ("1KB" = 1024, "1MB" = 1048576, etc).
- `limit` *(uint64, REQUIRED)* - limit in bytes of *hugepagesize* HugeTLB usage

Example

```
"hugepageLimits": [
    {
        "pageSize": "2MB",
        "limit": 209715200
    },
    {
        "pageSize": "64KB",
        "limit": 1000000
    }
]
```

Network

`network` (object, OPTIONAL) represents the cgroup subsystems `net_cls` and `net_prio`.
For more information, see the kernel cgroups documentations about net_cls cgroup and net_prio cgroup.

The following parameters can be specified to set up the controller:

- **`classID`** *(uint32, OPTIONAL)* - is the network class identifier the cgroup's network packets will be tagged with
- **`priorities`** *(array of objects, OPTIONAL)* - specifies a list of objects of the priorities assigned to traffic originating from processes in the group and egressing the system on various interfaces.
  The following parameters can be specified per-priority:
  - **`name`** *(string, REQUIRED)* - interface name in runtime network namespace
  - **`priority`** *(uint32, REQUIRED)* - priority applied to the interface

Example

```
"network": {
    "classID": 1048577,
    "priorities": [
        {
            "name": "eth0",
            "priority": 500
        },
        {
            "name": "eth1",
            "priority": 1000
        }
    ]
}
```

PIDs

`pids` (object, OPTIONAL) represents the cgroup subsystem **`pids`**.
For more information, see the kernel cgroups documentation about pids.

The following parameters can be specified to set up the controller:

- **`limit`** *(int64, REQUIRED)* - specifies the maximum number of tasks in the cgroup

Example

```
"pids": {
    "limit": 32771
}
```

RDMA

`rdma` (object, OPTIONAL) represents the cgroup subsystem `rdma`.
For more information, see the kernel cgroups documentation about rdma.

The name of the device to limit is the entry key.
Entry values are objects with the following properties:

- `hcaHandles` *(uint32, OPTIONAL)* - specifies the maximum number of hca_handles in the cgroup
- `hcaObjects` *(uint32, OPTIONAL)* - specifies the maximum number of hca_objects in the cgroup

You MUST specify at least one of the `hcaHandles` or `hcaObjects` in a given entry, and MAY specify both.

Example

```
"rdma": {
    "mlx5_1": {
        "hcaHandles": 3,
        "hcaObjects": 10000
    },
    "mlx4_0": {
        "hcaObjects": 1000
    },
    "rxe3": {
        "hcaObjects": 10000
    }
}
```

IntelRdt

`intelRdt` (object, OPTIONAL) represents the Intel Resource Director Technology.
If `intelRdt` is set, the runtime MUST write the container process ID to the `tasks` file in a proper sub-directory in a mounted `resctrl` pseudo-filesystem. That sub-directory name is specified by `closID` parameter.
If no mounted `resctrl` pseudo-filesystem is available in the runtime mount namespace, the runtime MUST generate an error.

If `intelRdt` is not set, the runtime MUST NOT manipulate any `resctrl` pseudo-filesystems.

The following parameters can be specified for the container:

- `closID` *(string, OPTIONAL)* - specifies the identity for RDT Class of Service (CLOS).
  If `closID` is set, runtimes MUST create `closID` directory in a mounted `resctrl` pseudo-filesystem if it doesn't exist. If not set, runtimes MUST use the container ID from `start` and create the `<container-id>` directory.
- `l3CacheSchema` *(string, OPTIONAL)* - specifies the schema for L3 cache id and capacity bitmask (CBM).
  The value SHOULD start with `L3:` and SHOULD NOT contain newlines.
- `memBwSchema` *(string, OPTIONAL)* - specifies the schema of memory bandwidth per L3 cache id.
  - The value MUST start with `MB:` and MUST NOT contain newlines.
  - If both `l3CacheSchema` and `memBwSchema` are set, runtimes MUST write the combined value to the `schemata` file in that sub-directory discussed in `closID`.
  - If `l3CacheSchema` contains a line beginning with `MB:`, the value written to `schemata` file MUST be the non-`MB:` line(s) from `l3CacheSchema` and the line from `memBWSchema`.
  - If either `l3CacheSchema` or `memBwSchema` is set, runtimes MUST write the value to the `schemata` file in the that sub-directory discussed in `closID`.
  - If neither `l3CacheSchema` nor `memBwSchema` is set, runtimes MUST NOT write to `schemata` files in any `resctrl` pseudo-filesystems.
  - If `closID` is set, `l3CacheSchema` and/or `memBwSchema` is set, runtimes MUST compare `l3CacheSchema` and/or `memBwSchema` value with `schemata` file, and generate an error if doesn't match.
  - If `closID` is set, and neither of `l3CacheSchema` and `memBwSchema` are set, runtime MUST check if corresponding pre-configured directory `closID` is present in mounted `resctrl`. If such pre-configured directory `closID` exists, runtime MUST assign container to this `closID` and generate an error if directory does not exist.

Example

Consider a two-socket machine with two L3 caches where the default CBM is 0x7ff and the max CBM length is 11 bits,
and minimum memory bandwidth of 10% with a memory bandwidth granularity of 10%.

Tasks inside the container only have access to the "upper" 7/11 of L3 cache on socket 0 and the "lower" 5/11 L3 cache on socket 1,
and may use a maximum memory bandwidth of 20% on socket 0 and 70% on socket 1.

```
"linux": {
    "intelRdt": {
        "closID": "guaranteed_group",
        "l3CacheSchema": "L3:0=7f0;1=1f",
        "memBwSchema": "MB:0=20;1=70"
    }
}
```

Sysctl

sysctl (object, OPTIONAL) allows kernel parameters to be modified at runtime for the container.
For more information, see the sysctl(8) man page.

Example

```
"sysctl": {
    "net.ipv4.ip_forward": "1",
    "net.core.somaxconn": "256"
}
```

Seccomp

Seccomp provides application sandboxing mechanism in the Linux kernel. Seccomp configuration allows one to configure actions to take for matched syscalls and furthermore also allows matching on values passed as arguments to syscalls. For more information about Seccomp, see Seccomp kernel documentation. The actions, architectures, and operators are strings that match the definitions in seccomp.h from libseccomp and are translated to corresponding values.

seccomp (object, OPTIONAL)

The following parameters can be specified to set up seccomp:

- **defaultAction** *(string, REQUIRED)* - the default action for seccomp. Allowed values are the same as **syscalls[].action**.
- **architectures** *(array of strings, OPTIONAL)* - the architecture used for system calls.
  A valid list of constants as of libseccomp v2.3.2 is shown below.
  - SCMP_ARCH_X86
  - SCMP_ARCH_X86_64
  - SCMP_ARCH_X32

  - SCMP_ARCH_ARM
  - SCMP_ARCH_AARCH64
  - SCMP_ARCH_MIPS
  - SCMP_ARCH_MIPS64
  - SCMP_ARCH_MIPS64N32
  - SCMP_ARCH_MIPSEL
  - SCMP_ARCH_MIPSEL64
  - SCMP_ARCH_MIPSEL64N32
  - SCMP_ARCH_PPC
  - SCMP_ARCH_PPC64
  - SCMP_ARCH_PPC64LE
  - SCMP_ARCH_S390
  - SCMP_ARCH_S390X
  - SCMP_ARCH_PARISC
  - SCMP_ARCH_PARISC64

- **flags** *(array of strings, OPTIONAL)* - list of flags to use with seccomp(2).
  A valid list of constants is shown below.

  - SECCOMP_FILTER_FLAG_TSYNC
  - SECCOMP_FILTER_FLAG_LOG
  - SECCOMP_FILTER_FLAG_SPEC_ALLOW

- **syscalls** *(array of objects, OPTIONAL)* - match a syscall in seccomp.
  While this property is OPTIONAL, some values of **defaultAction** are not useful without **syscalls** entries.
  For example, if **defaultAction** is SCMP_ACT_KILL and **syscalls** is empty or unset, the kernel will kill the container process on its first syscall.
  Each entry has the following structure:

  - **names** *(array of strings, REQUIRED)* - the names of the syscalls. **names** MUST contain at least one entry.
  - **action** *(string, REQUIRED)* - the action for seccomp rules.
    A valid list of constants as of libseccomp v2.4.0 is shown below.
    * SCMP_ACT_KILL
    * SCMP_ACT_TRAP
    * SCMP_ACT_ERRNO
    * SCMP_ACT_TRACE
    * SCMP_ACT_ALLOW
    * SCMP_ACT_LOG
  - **args** *(array of objects, OPTIONAL)* - the specific syscall in seccomp.
    Each entry has the following structure:
    * **index** *(uint, REQUIRED)* - the index for syscall arguments in seccomp.

    - **value** *(uint64, REQUIRED)* - the value for syscall arguments in seccomp.
    - **valueTwo** *(uint64, OPTIONAL)* - the value for syscall arguments in seccomp.
    - **op** *(string, REQUIRED)* - the operator for syscall arguments in seccomp.
      A valid list of constants as of libseccomp v2.3.2 is shown below.
        - SCMP_CMP_NE
        - SCMP_CMP_LT
        - SCMP_CMP_LE
        - SCMP_CMP_EQ
        - SCMP_CMP_GE
        - SCMP_CMP_GT
        - SCMP_CMP_MASKED_EQ

Example

```
"seccomp": {
    "defaultAction": "SCMP_ACT_ALLOW",
    "architectures": [
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "names": [
                "getcwd",
                "chmod"
            ],
            "action": "SCMP_ACT_ERRNO"
        }
    ]
}
```

Rootfs Mount Propagation

rootfsPropagation (string, OPTIONAL) sets the rootfs's mount propagation. Its value is either **shared**, **slave**, **private** or **unbindable**.
It's worth noting that a peer group is defined as a group of VFS mounts that propagate events to each other.
A nested container is defined as a container launched inside an existing container.

- **shared**: the rootfs mount belongs to a new peer group.
  This means that further mounts (e.g. nested containers) will also belong

to that peer group and will propagate events to the rootfs.
Note this does not mean that it's shared with the host.

- **`slave`**: the rootfs mount receives propagation events from the host (e.g. if something is mounted on the host it will also appear in the container) but not the other way around.
- **`private`**: the rootfs mount doesn't receive mount propagation events from the host and further mounts in nested containers will be isolated from the host and from the rootfs (even if the nested container `rootfsPropagation` option is shared).
- **`unbindable`**: the rootfs mount is a private mount that cannot be bind-mounted.

The Shared Subtrees article in the kernel documentation has more information about mount propagation.

Example

```
"rootfsPropagation": "slave",
```

Masked Paths

`maskedPaths` (array of strings, OPTIONAL) will mask over the provided paths inside the container so that they cannot be read.
The values MUST be absolute paths in the container namespace.

Example

```
"maskedPaths": [
    "/proc/kcore"
]
```

Readonly Paths

`readonlyPaths` (array of strings, OPTIONAL) will set the provided paths as readonly inside the container.
The values MUST be absolute paths in the container namespace.

Example

```
"readonlyPaths": [
    "/proc/sys"
]
```

Mount Label

`mountLabel` (string, OPTIONAL) will set the Selinux context for the mounts in the container.

Example

```
"mountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c715,c811"
```

Personality

`personality` (object, OPTIONAL) sets the Linux execution personality. For more information
see the personality syscall documentation. As most of the options are
obsolete and rarely used, and some reduce security, the currently supported set is a small
subset of the available options.

- **`domain`** *(string, REQUIRED)* - the execution domain.
  The valid list of constants is shown below. `LINUX32` will set the `uname` system call to show
  a 32 bit CPU type, such as `i686`.
  - `LINUX`
  - `LINUX32`
- **`flags`** *(array of strings, OPTIONAL)* - the additional flags to apply.
  Currently no flag values are supported.

Solaris Application Container Configuration

Solaris application containers can be configured using the following properties, all of the below properties have mappings to properties specified under zonecfg(1M) man page, except milestone.

milestone

The SMF(Service Management Facility) FMRI which should go to "online" state before we start the desired process within the container.

`milestone` *(string, OPTIONAL)*

Example

```
"milestone": "svc:/milestone/container:default"
```

limitpriv

The maximum set of privileges any process in this container can obtain.
The property should consist of a comma-separated privilege set specification as described in priv_str_to_set(3C) man page for the respective release of Solaris.

limitpriv *(string, OPTIONAL)*

Example

```
"limitpriv": "default"
```

maxShmMemory

The maximum amount of shared memory allowed for this application container.
A scale (K, M, G, T) can be applied to the value for each of these numbers (for example, 1M is one megabyte).
Mapped to **max-shm-memory** in zonecfg(1M) man page.

maxShmMemory *(string, OPTIONAL)*

Example

```
"maxShmMemory": "512m"
```

cappedCPU

Sets a limit on the amount of CPU time that can be used by a container.
The unit used translates to the percentage of a single CPU that can be used by all user threads in a container, expressed as a fraction (for example, .75) or a mixed number (whole number and fraction, for example, 1.25).
An ncpu value of 1 means 100% of a CPU, a value of 1.25 means 125%, .75 mean 75%, and so forth.
When projects within a capped container have their own caps, the minimum value takes precedence.
cappedCPU is mapped to **capped-cpu** in zonecfg(1M) man page.

- **ncpus** *(string, OPTIONAL)*

Example

```
"cappedCPU": {
    "ncpus": "8"
}
```

cappedMemory

The physical and swap caps on the memory that can be used by this application container.
A scale (K, M, G, T) can be applied to the value for each of these numbers (for example, 1M is one megabyte).
cappedMemory is mapped to `capped-memory` in zonecfg(1M) man page.

- **`physical`** *(string, OPTIONAL)*
- **`swap`** *(string, OPTIONAL)*

Example

```
"cappedMemory": {
    "physical": "512m",
    "swap": "512m"
}
```

Network

Automatic Network (anet)

anet is specified as an array that is used to set up networking for Solaris application containers.
The anet resource represents the automatic creation of a network resource for an application container.
The zones administration daemon, zoneadmd, is the primary process for managing the container's virtual platform.
One of the daemon's responsibilities is creation and teardown of the networks for the container.
For more information on the daemon see the zoneadmd(1M) man page.
When such a container is started, a temporary VNIC(Virtual NIC) is automatically created for the container.
The VNIC is deleted when the container is torn down.
The following properties can be used to set up automatic networks.
For additional information on properties, check the zonecfg(1M) man page for the respective release of Solaris.

- **`linkname`** *(string, OPTIONAL)* Specify a name for the automatically created VNIC datalink.
- **`lowerLink`** *(string, OPTIONAL)* Specify the link over which the VNIC will be created.
  Mapped to `lower-link` in the zonecfg(1M) man page.
- **`allowedAddress`** *(string, OPTIONAL)* The set of IP addresses that the container can use might be constrained by specifying the `allowedAddress` property.
  If `allowedAddress` has not been specified, then they can use any IP address on the associated physical interface for the network resource.
  Otherwise, when `allowedAddress` is specified, the container cannot use IP addresses that are not in the `allowedAddress` list for the physical address.
  Mapped to `allowed-address` in the zonecfg(1M) man page.
- **`configureAllowedAddress`** *(string, OPTIONAL)* If `configureAllowedAddress` is set to true, the addresses specified by `allowedAddress` are automatically configured on the interface each time the container starts.
  When it is set to false, the `allowedAddress` will not be configured on container start.
  Mapped to `configure-allowed-address` in the zonecfg(1M) man page.
- **`defrouter`** *(string, OPTIONAL)* The value for the OPTIONAL default router.
- **`macAddress`** *(string, OPTIONAL)* Set the VNIC's MAC addresses based on the specified value or keyword.
  If not a keyword, it is interpreted as a unicast MAC address.
  For a list of the supported keywords please refer to the zonecfg(1M) man page of the respective Solaris release.
  Mapped to `mac-address` in the zonecfg(1M) man page.
- **`linkProtection`** *(string, OPTIONAL)* Enables one or more types of link protection using comma-separated values.
  See the protection property in dladm(8) for supported values in respective release of Solaris.
  Mapped to `link-protection` in the zonecfg(1M) man page.

Example

```
"anet": [
    {
        "allowedAddress": "172.17.0.2/16",
        "configureAllowedAddress": "true",
        "defrouter": "172.17.0.1/16",
        "linkProtection": "mac-nospoof, ip-nospoof",
        "linkname": "net0",
        "lowerLink": "net2",
        "macAddress": "02:42:f8:52:c7:16"
```

```
    }
]
```

Glossary

Bundle

A directory structure that is written ahead of time, distributed, and used to seed the runtime for creating a container and launching a process within it.

Configuration

The `config.json` file in a bundle which defines the intended container and container process.

Container

An environment for executing processes with configurable isolation and resource limitations.
For example, namespaces, resource limits, and mounts are all part of the container environment.

Container namespace

On Linux,the namespaces in which the configured process executes.

JSON

All configuration JSON MUST be encoded in UTF-8.
JSON objects MUST NOT include duplicate names.
The order of entries in JSON objects is not significant.

Runtime

An implementation of this specification.
It reads the configuration files from a bundle, uses that information to create a container, launches a process inside the container, and performs other lifecycle actions.

Runtime namespace

On Linux, the namespaces from which new container namespaces are created and from which some configured resources are accessed.