bootc

Transactional, in-place operating system updates using OCI/Docker container images. bootc is the key component in a broader mission of bootable containers.

The original Docker container model of using "layers" to model applications has been extremely successful. This project aims to apply the same technique for bootable host systems - using standard OCI/Docker containers as a transport and delivery format for base operating system updates.

The container image includes a Linux kernel (in e.g. `/usr/lib/modules`), which is used to boot. At runtime on a target system, the base userspace is *not* itself running in a container by default. For example, assuming systemd is in use, systemd acts as pid1 as usual - there's no "outer" process.

Status

The CLI and API for bootc are now considered stable. Every existing system can be upgraded in place seamlessly across any future changes.

However, the core underlying code uses the ostree project which has been powering stable operating system updates for many years. The stability here generally refers to the surface APIs, not the underlying logic.

Base images

Many users will be more interested in base (container) images.

Fedora/CentOS

Currently, the Fedora/CentOS bootc project is the most closely aligned upstream project.

For pre-built base images; any Fedora derivative already using `ostree` can be seamlessly converted into using bootc; for example, Fedora CoreOS can be used as a base image; you will want to also `rpm-ostree install bootc` in your image builds currently. There are some overlaps between `bootc` and `ignition` and `zincati` however; see this pull request for more information.

For other derivatives such as the "Atomic desktops", see discussion of relationships which particularly covers interactions with rpm-ostree.

Other

However, bootc itself is not tied to Fedora derivatives; this issue tracks the main blocker for other distributions.

Generic guidance for building images

The bootc project intends to be operating system and distribution independent as possible, similar to its related projects podman and systemd, etc.

The recommendations for creating bootc-compatible images will in general need to be owned by the OS/distribution - in particular the ones who create the default bootc base image(s). However, some guidance is very generic to most Linux systems (and bootc only supports Linux).

Let's however restate a base goal of this project:

> The original Docker container model of using "layers" to model applications has been extremely successful. This project aims to apply the same technique for bootable host systems - using standard OCI/Docker containers as a transport and delivery format for base operating system updates.

Every tool and technique for creating application base images should apply to the host Linux OS as much as possible.

Understanding mutability

When run as a container (particularly as part of a build), bootc-compatible images have all parts of the filesystem (e.g. `/usr` in particular) as fully mutable state, and writing there is encouraged (see below).

When "deployed" to a physical or virtual machine, the container image files are read-only by default; for more, see filesystem.

Installing software

For package management tools like `apt`, `dnf`, `zypper` etc. (generically, `$pkgsystem`) it is very much expected that the pattern of

```
RUN $pkgsystem install somepackage && $pkgsystem clean all
```

type flow Just Works here - the same way as it does "application" container images. This pattern is really how Docker got started.

There's not much special to this that doesn't also apply to application containers; but see below.

Nesting OCI containers in bootc containers

The OCI format uses "whiteouts" represented in the tar stream as special `.wh` files, and typically consumed by the Linux kernel `overlayfs` driver as special `0:0` character devices. Without special work, whiteouts cannot be nested.

Hence, an invocation like

```
RUN podman pull quay.io/exampleimage/someimage
```

will create problems, as the `podman` runtime will create whiteout files inside the container image filesystem itself.

Special care and code changes will need to be made to container runtimes to support such nesting. Some more discussion in this tracker issue.

systemd units

The model that is most popular with the Docker/OCI world is "microservice" style containers with the application as pid 1, isolating the applications from each other and from the host system - as opposed to "system containers" which run an init system like systemd, typically also SSH and often multiple logical "application" components as part of the same container.

The bootc project generally expects systemd as pid 1, and if you embed software in your derived image, the default would then be that that software is initially launched via a systemd unit.

```
RUN dnf -y install postgresql && dnf clean all
```

Would typically also carry a systemd unit, and that service will be launched the same way as it would on a package-based system.

Users and groups

Note that the above `postgresql` today will allocate a user; this leads to the topic of users, groups and SSH keys.

Configuration

A key aspect of choosing a bootc-based operating system model is that *code* and *configuration* can be strictly "lifecycle bound" together in exactly the same way.

(Today, that's by including the configuration into the base container image; however a future enhancement for bootc will also support dynamically-injected ConfigMaps, similar to kubelet)

You can add configuration files to the same places they're expected by typical package systems on Debian/Fedora/Arch etc. and others - in `/usr` (preferred where possible) or `/etc`. systemd has long advocated and supported a model where `/usr` (e.g. `/usr/lib/systemd/system`) contains content owned by the operating system image.

`/etc` is machine-local state. However, per filesystem.md it's important to note that the underlying OSTree system performs a 3-way merge of `/etc`, so changes you make in the container image to e.g. `/etc/postgresql.conf` will be applied on update, assuming it is not modified locally.

Prefer using drop-in directories

These "locally modified" files can be a source of state drift. The best pattern to use is "drop-in" directories that are merged dynamically by the relevant software. systemd supports this comprehensively; see drop-ins for example in units.

And instead of modifying `/etc/sudoers.conf`, it's best practice to add a file into `/etc/sudoers.d` for example.

Not all software supports this, however; and this is why there is generic support for `/etc`.

Configuration in /usr vs /etc

Some software supports generic configuration both `/usr` and `/etc` - systemd, among others. Because bootc supports *derivation* (the way OCI containers work) - it is supported and encouraged to put configuration files in `/usr` (instead of `/etc`) where possible, because then the state is consistently immutable.

One pattern is to replace a configuration file like `/etc/postgresql.conf` with a symlink to e.g. `/usr/postgres/etc/postgresql.conf` for example, although this can run afoul of SELinux labeling.

Secrets

There is a dedicated document for secrets, which is a special case of configuration.

Handling read-only vs writable locations

The high level pattern for bootc systems is summarized again this way:

- Put read-only data and executables in `/usr`
- Put configuration files in `/usr` (if they're static), or `/etc` if they need to be machine-local
- Put "data" (log files, databases, etc.) underneath `/var`

However, some software installs to `/opt/examplepkg` or another location outside of `/usr`, and may include all three types of data undernath its single toplevel directory. For example, it may write log files to `/opt/examplepkg/logs`. A simple way to handle this is to change the directories that need to be writable to symbolic links to `/var`:

```
RUN apt|dnf install examplepkg && \
    mv /opt/examplepkg/logs /var/log/examplepkg && \
    ln -sr /var/log/examplepkg /opt/examplepkg/logs
```

The Fedora/CentOS bootc puppet example is one instance of this.

Another option is to configure the systemd unit launching the service to do these mounts dynamically via e.g.

```
BindPaths=/var/log/exampleapp:/opt/exampleapp/logs
```

Container runtime vs "bootc runtime"

Fundamentally, `bootc` reuses the OCI image format as a way to transport serialized filesystem trees with included metadata such as a `version` label, etc.

A bootc container operates in two basic modes. First, when invoked by a container run time such as `podman` or `docker` (typically as part of a build process), the bootc container behaves exactly the same as any other container. For example, although there is a kernel embedded in the container image, it is not executed - the host kernel is used. There's no additional mount namespaces, etc. Ultimately, the container runtime is in full control here.

The second, and most important mode of operation is when a bootc container is installed to a physical or virtual machine. Here, bootc is in control; the container runtime used to build is no longer relevant. However, it's *very* important to understand that bootc's role is quite limited:

- On boot, there is code in the initramfs to do a "chroot" equivalent into the target filesystem root
- On upgrade, bootc will fetch new content, but this will not affect the running root

Crucially, besides setting up some mounts, bootc itself does not act as any kind of "container runtime". It does not set up pid or other namespace, does not change cgroups, etc. That remains the role of other code (typically systemd). `bootc` is not a persistent daemon by default; it does not impose any runtime overhead.

Another example of this: While one can add Container configuration metadata, `bootc` generally ignores that at runtime today.

Labels

A key aspect of OCI is the ability to use standardized (or semi-standardized) labels. The are stored and rendered by `bootc`; especially the `org.opencontainers.image.version` label.

Example ignored runtime metadata, and recommendations

ENTRYPOINT and CMD (OCI: `Entrypoint/Cmd`)

Ignored by bootc.

It's recommended for bootc containers to set `CMD /sbin/init`; but this is not required.

The booted host system will launch from the bootloader, to the kernel+initramfs and real root however it is "physically" configured inside the image. Typically today this is using systemd in both the initramfs and at runtime; but this is up to how you build the image.

ENV (OCI: `Env`)

Ignored by bootc; to configure the global system environment you can change the systemd configuration. (Though this is generally not a good idea; instead it's usually better to change the environment of individual services)

EXPOSE (OCI: `exposedPorts`)

Ignored by bootc; it is agnostic to how the system firewall and network function at runtime.

USER (OCI: `User`)

Ignored by bootc; typically you should configure individual services inside the bootc container to run as unprivileged users instead.

HEALTHCHECK (OCI: *no equivalent*)

This is currently a Docker-specific metadata, and did not make it into the OCI standards. (Note podman healthchecks)

It is important to understand again is that there is no "outer container runtime" when a bootc container is deployed on a host. The system must perform health checking on itself (or have an external system do it).

Relevant links:

- bootc rollback
- CentOS Automotive SIG unattended updates (note that as of right now, greenboot does not yet integrate with bootc)
- https://systemd.io/AUTOMATIC_BOOT_ASSESSMENT/

Kernel

When run as a container, the Linux kernel binary in `/usr/lib/modules/$kver/vmlinuz` is ignored. It is only used when a bootc container is deployed to a physical or virtual machine.

Security properties

When run as a container, the container runtime will by default apply various Linux kernel features such as namespacing to isolate the container processes from other system processes.

None of these isolation properties apply when a bootc system is deployed.

SELinux

For more on the intersection of SELinux and current bootc (OSTree container) images, see bootc images - SELinux.

Users and groups

This is one of the more complex topics. Generally speaking, bootc has nothing to do directly with configuring users or groups; it is a generic OS update/configuration mechanism. (There is currently just one small exception in that `bootc install` has a special case `--root-ssh-authorized-keys` argument, but it's very much optional).

Generic base images

Commonly OS/distribution base images will be generic, i.e. without any configuration. It is *very strongly recommended* to avoid hardcoded passwords and ssh keys with publicly-available private keys (as Vagrant does) in generic images.

Injecting SSH keys via systemd credentials

The systemd project has documentation for credentials which can be used in some environments to inject a root password or SSH authorized_keys. For many cases, this is a best practice.

At the time of this writing this relies on SMBIOS which is mainly configurable in local virtualization environments. (qemu).

Injecting users and SSH keys via cloud-init, etc.

Many IaaS and virtualization systems are oriented towards a "metadata server" (see e.g. AWS instance metadata) that are commonly processed by software such as cloud-init or Ignition or equivalent.

The base image you're using may include such software, or you can install it in your own derived images.

In this model, SSH configuration is managed outside of the bootable image. See e.g. GCP oslogin for an example of this where operating system identities are linked to the underlying Google accounts.

Adding users and credentials via custom logic (container or unit)

Of course, systems like `cloud-init` are not privileged; you can inject any logic you want to manage credentials via e.g. a systemd unit (which may launch a container image) that manages things however you prefer. Commonly, this would be a custom network-hosted source. For example, FreeIPA.

Another example in a Kubernetes-oriented infrastructure would be a container image that fetches desired authentication credentials from a CRD hosted in the API server. (To do things like this it's suggested to reuse the kubelet credentials)

System users and groups (added via packages, etc)

It is common for packages (deb/rpm/etc) to allocate system users or groups as part of e.g `apt|dnf install <server package>` such as Apache or MySQL, and this is often done by directly invoking `useradd` or `groupadd` as part of package pre/post installation scripts.

With the `shadow-utils` implementation of `useradd` and the default glibc `files` this will result in changes to the traditional `/etc/passwd` and `/etc/shadow` files as part of the container build.

System drift from local /etc/passwd modifications

When the system is initially installed, the `/etc/passwd` in the container image will be applied and contain desired users.

By default (without `etc = transient`, see below), the `/etc` directory is machine-local persistent state. If subsequently `/etc/passwd` is modified local to the machine (as is common for e.g. setting a root password) then any new changes in the container image (such as users from new packages) *will not appear* on subsequent updates by default (they will be in `/usr/etc/passwd` instead - the default image version).

The general best fix for this is to use `systemd-sysusers` instead of allocating a user/group at build time at all.

Using systemd-sysusers

See systemd-sysusers. For example in your derived build:

```
COPY mycustom-user.conf /usr/lib/sysusers.d
```

A key aspect of how this works is that `sysusers` will make changes to the traditional `/etc/passwd` file as necessary on boot instead of at build time. If `/etc` is persistent, this can avoid uid/gid drift (but in the general case it does mean that uid/gid allocation can depend on how a specific machine was upgraded over time).

Note that the default `sysusers` design is that users are allocated on the client side (per machine). Avoid having non-root owned files managed by `sysusers` inside your image, especially underneath `/usr`. With the exception of `setuid` or `setgid` binaries (which should also be strongly avoided), there is generally no valid reason for having non-root owned files in `/usr` or other runtime-immutable directories.

User and group home directories and `/var`

For systems configured with persistent `/home` → `/var/home`, any changes to `/var` made in the container image after initial installation *will not be applied on subsequent updates*. If for example you inject `/var/home/someuser/.ssh/authorized_keys` into a container build, existing systems will *not* get the updated authorized keys file.

Using DynamicUser=yes for systemd units

For "system" users it's strongly recommended to use systemd DynamicUser=yes where possible.

This is significantly better than the pattern of allocating users/groups at "package install time" (e.g. Fedora package user/group guidelines) because it avoids potential UID/GID drift (see below).

Using systemd JSON user records

See JSON user records. Unlike `sysusers`, the canonical state for these live in `/usr` - if a subsequent image drops a user record, then it will also vanish from the system - unlike `sysusers.d`.

nss-altfiles

The nss-altfiles project (long) predates systemd JSON user records. It aims to help split

"system" users into `/usr/lib/passwd` and `/usr/lib/group`. It's very important to understand that this aligns with the way the OSTree project handles the "3 way merge" for `/etc` as it relates to `/etc/passwd`. Currently, if the `/etc/passwd` file is modified in any way on the local system, then subsequent changes to `/etc/passwd` in the container image *will not be applied*.

Some base images may have `nss-altfiles` enabled by default; this is currently the case for base images built by rpm-ostree.

Commonly, base images will have some "system" users pre-allocated and managed via this file again to avoid uid/gid drift.

In a derived container build, you can also append users to `/usr/lib/passwd` for example. (At the time of this writing there is no command line to do so though).

Typically it is more preferable to use `sysusers.d` or `DynamicUser=yes`.

Machine-local state for users

At this point, it is important to understand the filesystem layout - the default is up to the base image.

The default Linux concept of a user has data stored in both `/etc` (`/etc/passwd`, `/etc/shadow` and groups) and `/home`. The choice for how these work is up to the base image, but a common default for generic base images is to have both be machine-local persistent state. In this model `/home` would be a symlink to `/var/home/someuser`.

Injecting users and SSH keys via at system provisioning time

For base images where `/etc` and `/var` are configured to persist by default, it will then be generally supported to inject users via "installers" such as Anaconda (interactively or via kickstart) or any others.

Typically generic installers such as this are designed for "one time bootstrap" and again then the configuration becomes mutable machine-local state that can be changed "day 2" via some other mechanism.

The simple case is a user with a password - typically the installer helps set the initial password, but to change it there is a different in-system tool (such as `passwd` or a GUI as

part of Cockpit, GNOME/KDE/etc).

It is intended that these flows work equivalently in a bootc-compatible system, to support users directly installing "generic" base images, without requiring changes to the tools above.

Transient home directories

Many operating system deployments will want to minimize persistent, mutable and executable state - and user home directories are that

But it is also valid to default to having e.g. `/home` be a `tmpfs` to ensure user data is cleaned up across reboots (and this pairs particularly well with a transient `/etc` as well):

In order to set up the user's home directory to e.g. inject SSH `authorized_keys` or other files, a good approach is to use systemd `tmpfiles.d` snippets:

```
f~ /home/someuser/.ssh/authorized_keys 600 someuser someuser - <base64
encoded data>
```

which can be embedded in the image as `/usr/lib/tmpfiles.d/someuser-keys.conf`.

Or a service embedded in the image can fetch keys from the network and write them; this is the pattern used by cloud-init and afterburn.

UID/GID drift

Any invocation of `useradd` or `groupadd` that does not allocate a *fixed* UID/GID may be subject to "drift" in subsequent rebuilds by default.

One possibility is to explicitly force these user/group allocations into a static state, via `systemd-sysusers` (per above) or explicitly adding the users with static IDs *before* a dpkg/RPM installation script operates on it:

```
RUN <<EORUN
set -xeuo pipefail
groupadd -g 10044 mycustom-group
useradd -u 10044 -g 10044 -d /dev/null -M mycustom-user
dnf install -y mycustom-package.rpm
bootc container lint
EORUN
```

Ultimately the `/etc/passwd` and similar files are a mapping between names and numeric identifiers. A problem then becomes when this mapping is dynamic and mixed with "stateless" container image builds.

For example today the CentOS Stream 9 `postgresql` package allocates a static uid of `26`.

This means that

```
RUN dnf -y install postgresql
```

will always result in a change to `/etc/passwd` that allocates uid 26 and data in `/var/lib/postgres` will always be owned by that UID.

However in contrast, the cockpit project allocates a floating cockpit-ws user.

This means that each container image build (without additional work, unlike the example at the beginning of this section),may (due to RPM installation ordering or other reasons) result in the uid changing.

This can be a problem if that user maintains persistent state. Such cases are best handled by being converted to use `sysusers.d` (see Fedora change) - or again even better, using `DynamicUser=yes` (see above).

tmpfiles.d use for setting ownership

Systemd's tmpfiles.d provides a way to define files and directories in a way that will be processes at startup as needed. One way to work around SELinux security context and user or group ownership of a directory or file can be by using the z or Z directives.

These directives will adjust the access mode, user and group ownership and the SELinux security context as stated on the doc linked above.

For example, if we need `/var/lib/my_file.conf` to be part of the `tss` group but owned by `root` we could create a tmpfiles.d entry with:

```
+z /var/lib/my_file 0640 root tss -
```

Kernel arguments

The default bootc model uses "type 1" bootloader config files stored in `/boot/loader/entries`, which define arguments provided to the Linux kernel.

The set of kernel arguments can be machine-specific state, but can also be managed via container updates.

The bootloader entries are currently written by the OSTree backend.

More on Linux kernel arguments: https://docs.kernel.org/admin-guide/kernel-parameters.html

/usr/lib/bootc/kargs.d

Many bootc use cases will use generic "OS/distribution" kernels. In order to support injecting kernel arguments, bootc supports a small custom config file format in `/usr/lib/bootc/kargs.d` in TOML format, that have the following form:

```
# /usr/lib/bootc/kargs.d/10-example.toml
kargs = ["mitigations=auto,nosmt"]
```

There is also support for making these kernel arguments architecture specific via the `match-architectures` key:

```
# /usr/lib/bootc/kargs.d/00-console.toml
kargs = ["console=ttyS0,115200n8"]
match-architectures = ["x86_64"]
```

NOTE: The architecture matching here accepts values defined by the Rust standard library (using the architecture of the `bootc` binary itself).

In some cases for Linux, this matches the value of `uname -m`, but definitely not all. For example, on Fedora derivatives there is `ppc64le`, but in Rust only `powerpc64`. A common discrepancy is that Debian derivatives use `amd64`, whereas Rust (and Fedora derivatives) use `x86_64`.

Changing kernel arguments post-install via kargs.d

Changes to `kargs.d` files included in a container build are honored post-install; the difference between the set of kernel arguments is applied to the current bootloader configuration. This will preserve any machine-local kernel arguments.

Kernel arguments injected at installation time

The `bootc install` flow supports a `--karg` to provide install-time kernel arguments. These become machine-local state.

Higher level install tools (ideally at least using `bootc install to-filesystem` can inject kernel arguments this way) too; for example, the Anaconda installer has a `bootloader` verb which ultimately uses an API similar to this.

Post-install, it is supported for any tool to edit the `/boot/loader/entries` files, which are in a standardized format.

Typically, `/boot` is mounted read-only to limit the set of tools which write to this filesystem. It is not "physically" read-only by default. One approach to edit them is to run a tool under a new mount namespace, e.g.

```
unshare -m
mount -o remount,rw /boot
# tool to edit /boot/loader/entries
```

At the current time, `bootc` does not itself offer an API to manipulate kernel arguments maintained per-machine.

Other projects such as `rpm-ostree` do, via e.g. `rpm-ostree kargs`, which is just a frontend for editing the bootloader configuration files. Note an important detail is that `rpm-ostree kargs` always creates a new deployment.

`rpm-ostree kargs` and bootc will interoperate as they both use the ostree backend today, and any kernel arguments changed via that mechanism will persist across upgrades.

It is currently undefined behavior to remove kernel arguments locally that are included in the base image via `/usr/lib/bootc/kargs.d`.

Injecting default arguments into custom kernels

The Linux kernel supports building in arguments into the kernel binary, at the time of this writing via the `config CMDLINE` build option. If you are building a custom kernel, then it often makes sense to use this instead of `/usr/lib/bootc/kargs.d` for example.

Secrets (e.g. container pull secrets)

To have `bootc` fetch updates from registry which requires authentication, you must include a pull secret in one of `/etc/ostree/auth.json`, `/run/ostree/auth.json` or `/usr/lib/ostree/auth.json`.

The path to the authentication file differs from that used by e.g. `podman` by default as some of the file paths used there are not appropriate for system services (e.g. reading the `/root` home directory).

Regardless, injecting this data is a good example of a generic "secret". The bootc project does not currently include one single opinionated mechanism for secrets.

Synchronizing the bootc and podman credentials

See the containers-auth.json man page. In many cases, you will want to keep both the bootc and podman/skopeo credentials in sync. One pattern is to symlink the two via e.g. a systemd `tmpfiles.d` fragment.

If you have a process invoking `podman login` (which by default writes to an ephemeral `$XDG_RUNTIME_DIR/containers/auth.json`) you can then `ln -s /run/user/0/containers/auth.json /run/ostree/auth.json`.

Performing an explicit login

If you have automation (or manual processes) performing a login, you can pass `--authfile` to set the bootc authfile explicitly; for example

```
echo <somepassword> | podman login --authfile /run/ostree/auth.json -u someuser --password-stdin
```

This pattern of using the ephemeral location in `/run` can work well when the credentials are derived on system start from an external system. For example, `aws ecr get-login-password --region region` as suggested by this document.

You can also use the machine-local persistent location `/etc/ostree/auth.json` via this method.

Using a credential helper

In order to use a credential helper as configured in `registries.conf` such as `credential-helpers = ["ecr-login"]`, you must currently also write a "no-op" authentication file with the contents `{}` (i.e. an empty JSON object, not an empty file) into the pull secret location.

Embedding in container build

This was mentioned above; you can include secrets in the container image if the registry server is suitably protected.

In some cases, embedding only "bootstrap" secrets into the container image is a viable pattern, especially alongside a mechanism for having a machine authenticate to a cluster. In this pattern, a provisioning tool (whether run as part of the host system or a container image) uses the bootstrap secret to lay down and keep updated other secrets (for example, SSH keys, certificates).

Via cloud metadata

Most production IaaS systems support a "metadata server" or equivalent which can securely host secrets - particularly "bootstrap secrets". Your container image can include tooling such as `cloud-init` or `ignition` which fetches these secrets.

Embedded in disk images

Another pattern is to embed bootstrap secrets only in disk images. For example, when generating a cloud disk image (AMI, OpenStack glance image, etc.) from an input container image, the disk image can contain secrets that are effectively machine-local

state. Rotating them would require an additional management tool, or refreshing disk images.

Injected via baremetal installers

It is common for installer tools to support injecting configuration which can commonly cover secrets like this.

Injecting secrets via systemd credentials

The systemd project has documentation for credentials which applies in some deployment methodologies.

Management services

When running a fleet of systems, it is common to use a central management service. Commonly, these services provide a client to be installed on each system which connects to the central service. Often, the management service requires the client to perform a one time registration.

The following example shows how to install the client into a bootc image and run it at startup to register the system. This example assumes the management-client handles future connections to the server, e.g. via a cron job or a separate systemd service. This example could be modified to create a persistent systemd service if that is required. The Containerfile is not optimized in order to more clarly explain each step, e.g. it's generally better to invoke RUN a single time to avoid creating multiple layers in the image.

```
FROM <bootc base image>

# Typically when using a management service, it will determine when to
upgrade the system.
# So, disable bootc-fetch-apply-updates.timer if it is included in the
base image.
RUN systemctl disable bootc-fetch-apply-updates.timer

# Install the client from dnf, or some other method that applies for
your client
RUN dnf install management-client -y && dnf clean all

# Bake the credentials for the management service into the image
ARG activation_key=

# The existence of .run_next_boot acts as a flag to determine if the
# registration is required to run when booting
RUN touch /etc/management-client/.run_next_boot

COPY <<"EOT" /usr/lib/systemd/system/management-client.service
[Unit]
Description=Run management client at boot
After=network-online.target
ConditionPathExists=/etc/management-client/.run_client_next_boot

[Service]
Type=oneshot
EnvironmentFile=/etc/management-client/.credentials
ExecStart=/usr/bin/management-client register --activation-key
${CLIENT_ACTIVATION_KEY}
ExecStartPre=/bin/rm -f /etc/management-client/.run_next_boot
ExecStop=/bin/rm -f /etc/management-client/.credentials

[Install]
WantedBy=multi-user.target
EOT

# Link the service to run at startup
RUN ln -s /usr/lib/systemd/system/management-client.service /usr/lib/
systemd/system/multi-user.target.wants/management-client.service

# Store the credentials in a file to be used by the systemd service
RUN echo -e "CLIENT_ACTIVATION_KEY=${activation_key}" > /etc/management-
client/.credentials

# Set the flag to enable the service to run one time
# The systemd service will remove this file after the registration
completes the first time
RUN touch /etc/management-client/.run_next_boot
```

Managing upgrades

Right now, bootc is a quite simple tool that is designed to do just a few things well. One of those is transactionally fetching new operating system updates from a registry and booting into them, while supporting rollback.

The `bootc upgrade` verb

This will query the container image source and queue an updated container image for the next boot.

This is backed today by ostree, implementing an A/B style upgrade system. Changes to the base image are staged, and the running system is not changed by default.

Use `bootc upgrade --apply` to auto-apply if there are queued changes.

Staged updates with `--download-only`

The `--download-only` flag allows you to prepare updates without automatically applying them on the next reboot:

```
bootc upgrade --download-only
```

This will pull the new container image from the container image source and create a staged deployment in download-only mode. The deployment will not be applied on shutdown or reboot until you explicitly apply it.

Checking download-only status

To see whether a staged deployment is in download-only mode, use:

```
bootc status --verbose
```

In the output, you'll see `Download-only: yes` for deployments in download-only mode or `Download-only: no` for deployments that will apply automatically. This status is only shown in verbose mode.

Applying download-only updates

There are three ways to apply a staged update that is in download-only mode:

Option 1: Apply the staged update without checking for newer updates

```
bootc upgrade --from-downloaded
```

This unlocks the staged deployment for automatic application on the next shutdown or reboot, without fetching updates from the container image source. This is useful when you want to apply the already-downloaded update at a scheduled time.

Option 2: Apply the staged update and reboot immediately

```
bootc upgrade --from-downloaded --apply
```

This unlocks the staged deployment and immediately reboots into it, without checking for newer updates.

Option 3: Check for newer updates and apply

```
bootc upgrade
```

Running `bootc upgrade` without flags will pull from the container image source to check for updates. If the staged deployment matches the latest available update, it will be unlocked. If a newer update is available, the staged deployment will be replaced with the newer version.

Checking for updates without side effects

To check if updates are available without modifying the download-only state:

```
bootc upgrade --check
```

This only downloads updated metadata without changing the download-only state.

Example workflow

A typical workflow for controlled updates:

```
# 1. Download the update in download-only mode
bootc upgrade --download-only

# 2. Verify the staged deployment
bootc status --verbose
# Output shows: Download-only: yes

# 3. Test or wait for maintenance window...

# 4. Apply the update (choose one):
# Option A: Apply staged update without fetching from image source
bootc upgrade --from-downloaded

# Option B: Apply staged update and reboot immediately (without fetching
from image source)
bootc upgrade --from-downloaded --apply

# Option C: Check for newer updates first, then apply
bootc upgrade
```

Important notes:

- **Image source check difference**: `bootc upgrade --from-downloaded` does NOT fetch from the container image source to check for newer updates, while `bootc upgrade` always does. Use `--from-downloaded` when you want to apply the specific version you already downloaded, regardless of whether newer updates are available.
- If you reboot before applying a download-only update, the system will boot into the current deployment and the staged deployment will be discarded. However, the downloaded image data remains cached, so re-running `bootc upgrade --download-only` will be fast and won't re-download the container image.
- If you switch to a different image (using `bootc switch` or `bootc upgrade` to a different image), the new staged deployment will replace the previous download-only deployment, and the previously cached image will become eligible for garbage collection.

There is also an opinionated `bootc-fetch-apply-updates.timer` and corresponding service available in upstream for operating systems and distributions to enable.

Man page: bootc-upgrade.

Changing the container image source

Another useful pattern to implement can be to use a management agent to invoke `bootc switch` (or declaratively via `bootc edit`) to implement e.g. blue/green deployments, where some hosts are rolled onto a new image independently of others.

```
bootc switch quay.io/examplecorp/os-prod-blue:latest
```

`bootc switch` has the same effect as `bootc upgrade`; there is no semantic difference between the two other than changing the container image being tracked.

This will preserve existing state in `/etc` and `/var` - for example, host SSH keys and home directories.

Man page: bootc-switch.

Rollback

There is a `bootc rollback` verb, and associated declarative interface accessible to tools via `bootc edit`. This will swap the bootloader ordering to the previous boot entry.

Man page: bootc-rollback.

Accessing registries and disconnected updates

The `bootc` project uses the containers/image library to fetch container images (the same used by `podman`) which means it honors almost all the same configuration options in `/etc/containers`.

Insecure registries

Container clients such as `podman pull` and `docker pull` have a `--tls-verify=false` flag which says to disable TLS verification when accessing the registry. `bootc` has no such option. Instead, you can globally configure the option to disable TLS verification when accessing a specific registry via the `/etc/containers/registries.conf.d` configuration mechanism, for example:

```
# /etc/containers/registries.conf.d/local-registry.conf
[[registry]]
location="localhost:5000"
insecure=true
```

For more, see containers-registries.conf.

Private registries

It's common to use a private repository when deploying a fleet of bootc instances.

In addition to registry configuration, private registries require authentication. This is configured by placing an `auth.json` file at `/etc/ostree/auth.json`.

For more, see auth.json

Disconnected and offline updates

It is common (a best practice even) to maintain systems which default to being disconnected from the public Internet.

Pulling updates from a local mirror

Everything in the section remapping and mirroring images applies to bootc as well.

Performing offline updates via USB

In a usage scenario where the operating system update is in a fully disconnected environment and you want to perform updates via e.g. inserting a USB drive, one can do this by copying the desired OS container image to e.g. an `oci` directory:

```
skopeo copy docker://quay.io/exampleos/myos:latest oci:/path/to/
filesystem/myos.oci
```

Then once the USB device containing the `myos.oci` OCI directory is mounted on the target, use

```
bootc switch --transport oci /var/mnt/usb/myos.oci
```

The above command is only necessary once, and thereafter will be idempotent. Then, use `bootc upgrade --apply` to fetch and apply the update from the USB device.

This process can all be automated by creating systemd units that look for a USB device with a specific label, mount (optionally with LUKS for example), and then trigger the bootc upgrade.

Logically Bound Images

About logically bound images

This feature enables an association of container "app" images to a base bootc system image. Use cases for this include:

- Logging (e.g. journald->remote log forwarder container)
- Monitoring (e.g. Prometheus node_exporter)
- Configuration management agents
- Security agents

These types of things are commonly not updated outside of the host, and there's a secondary important property: We *always* want them present and available on the host, possibly from very early on in the boot. In contrast with default usage of tools like `podman` or `docker`, images may be pulled dynamically *after* the boot starts; requiring functioning networking, etc. For example if the remote registry is unavailable temporarily, the host system may run for a longer period of time without log forwarding or monitoring, which can be very undesirable.

Another simple way to say this is that logically bound images allow you to reference container images with the same confidence you can with `ExecStart=` in a systemd unit.

The term "logically bound" was created to contrast with physically bound images. There are some trade-offs between the two approaches. Some benefits of logically bound images are:

- The bootc system image can be updated without re-downloading the app image bits.
- The app images can be updated without modifying the bootc system image, this would be especially useful for development work

Using logically bound images

Each image is defined in a Podman Quadlet `.image` or `.container` file. An image is selected to be bound by creating a symlink in the `/usr/lib/bootc/bound-images.d` directory pointing to a `.image` or `.container` file.

With these defined, during a `bootc upgrade` or `bootc switch` the bound images defined in the new bootc image will be automatically pulled into the bootc image storage, and are available to container runtimes such as podman by explicitly configuring them to point to the bootc storage as an "additional image store", via e.g.:

```
podman --storage-opt=additionalimagestore=/usr/lib/bootc/storage run
<image> ...
```

An example Containerfile

```
FROM quay.io/myorg/myimage:latest

COPY ./my-app.image /usr/share/containers/systemd/my-app.image
COPY ./another-app.container /usr/share/containers/systemd/another-
app.container

RUN ln -s /usr/share/containers/systemd/my-app.image /usr/lib/bootc/
bound-images.d/my-app.image && \
    ln -s /usr/share/containers/systemd/another-app.container /usr/lib/
bootc/bound-images.d/another-app.container
```

In the `.container` definition, you should use:

```
GlobalArgs=--storage-opt=additionalimagestore=/usr/lib/bootc/storage
```

NOTE: Do *not* attempt to globally enable `/usr/lib/bootc/storage` in `/etc/containers/storage.conf`; only use the bootc storage for logically bound images, not also floating images. For more, see below.

Pull secret

Images are fetched using the global bootc pull secret by default (`/etc/ostree/auth.json`). It is not yet supported to configure `PullSecret` in these image definitions.

Garbage collection

The bootc image store is owned by bootc; images will be garbage collected when they are no longer referenced by a file in `/usr/lib/bootc/bound-images.d`.

Installation

Logically bound images must be present in the default container store (`/var/lib/containers`) when invoking bootc install; the images will be copied into the target system and present directly at boot, alongside the bootc base image.

Limitations

The *only* field parsed and honored by bootc currently is the `Image` field of a `.image` or `.container` file.

Other pull-relevant flags such as `PullSecret=` for example are not supported (see above). Another example unsupported flag is `Arch` (the default host architecture is always used).

There is no mechanism to inject arbitrary arguments to the `podman pull` (or equivalent) invocation used by bootc. However, many properties used for container registry interaction can be configured via containers-registries.conf and apply to all commands operating on that image.

It is not currently supported in general to launch "rootless" containers from system-owned image stores in general, whether from `/var/lib/containers` or the `/usr/lib/bootc/storage` . There is no integration between bootc and "rootless" storage today, and none is planned. Instead, it's recommended to ensure that your "system" or "rootful" containers drop privileges. More in e.g. https://github.com/containers/podman/discussions/13728.

Distro/OS installer support

At the current time, logically bound images are not supported by Anaconda.

Comparison with default podman systemd units

In the comparison below, the term "floating" will be used for non-logically bound images. These images are often fetched by e.g. podman-systemd and may be upgraded, added or

removed independently of the host upgrade lifecycle.

Lifecycle

- **Floating image:** The images are downloaded by the machine the first time it starts (requiring networking typically). Tools such as `podman auto-update` can be used to upgrade them independently of the host.
- **Logically bound image:** The images are referenced by the bootable container and are ensured to be available when the (bootc based) server starts. The image is always upgraded via `bootc upgrade` and appears read-only to other processes (e.g. `podman`).

Upgrades, rollbacks and garbage collection

- **Floating image:** Managed by the user (`podman auto-update`, `podman image prune`). This can be triggered at anytime independent of the host upgrades or rollbacks, and host upgrades/rollbacks do not affect the set of images.
- **Logically bound image:** Managed exclusively by `bootc` during upgrades. The logically bound images corresponding to rollback deployments will also be retained. `bootc` performs garbage collection of unused images.

"rootless" container image

- **Floating image:** Supported.
- **Logically bound image:** Not supported (`bootc` cannot be invoked as non-root). Instead, it's recommended to just drop most privileges for launched logically bound containers.

Avoid using /usr/lib/bootc/storage for floating images

Because images and in particular *layers* of images can be removed over time as the OS upgrades, if you attempt to globally enable `/usr/lib/bootc/storage` in the global `/etc/containers/storage.conf` that would also apply to "floating" container images (i.e. the default `podman run` and other runtimes), it can cause a bug where floating images

can later fail if layers that were reused in the LBI storage are removed. In the future, this restriction may be lifted, but at the current time you can only configure this additional storage for logically bound images.

Booting local builds

In some scenarios, you may want to boot a *locally* built container image, in order to apply a persistent hotfix to a specific server, or as part of a development/testing scenario.

Building a new local image

At the current time, the bootc host container storage is distinct from that of the `podman` container runtime storage (default configuration in `/var/lib/containers`).

It not currently streamlined to export the booted host container storage into the podman storage.

Hence today, to replicate the exact container image the host has booted, take the container image referenced in `bootc status` and turn it into a `podman pull` invocation.

Next, craft a container build file with your desired changes:

```
FROM <image>
RUN apt|dnf upgrade https://example.com/systemd-hotfix.package
```

Copying an updated image into the bootc storage

This command is straightforward; we just need to tell bootc to fetch updates from `containers-storage`, which is the local "application" container runtime (podman) storage:

```
$ bootc switch --transport containers-storage quay.io/fedora/fedora-
bootc:40
```

From there, the new image will be queued for the next boot and a `reboot` will apply it.

For more on valid transports, see containers-transports.

NAME

bootc - Deploy and transactionally in-place with bootable container images

SYNOPSIS

bootc [*OPTIONS...*] <*SUBCOMMAND*>

DESCRIPTION

Deploy and transactionally in-place with bootable container images.

The `bootc` project currently uses ostree-containers as a backend to support a model of bootable container images. Once installed, whether directly via `bootc install` (executed as part of a container) or via another mechanism such as an OS installer tool, further updates can be pulled and `bootc upgrade`.

SUBCOMMANDS

Command	Description
bootc upgrade	Download and queue an updated container image to apply
bootc switch	Target a new container image reference to boot
bootc rollback	Change the bootloader entry ordering; the deployment under `rollback` will be queued for the next boot, and the current will become rollback. If there is a `staged` entry (an unapplied, queued upgrade) then it will be discarded
bootc edit	Apply full changes to the host specification
bootc status	Display status
bootc usr-overlay	Add a transient writable overlayfs on `/usr`
bootc install	Install the running container to a target
bootc container	Operations which can be executed as part of a container build
bootc composefs-finalize-staged	

VERSION

NAME

bootc-status - Display status

SYNOPSIS

bootc status [*OPTIONS...*]

DESCRIPTION

Display status.

If standard output is a terminal, this will output a description of the bootc system state. If standard output is not a terminal, output a YAML-formatted object using a schema intended to match a Kubernetes resource that describes the state of the booted system.

Parsing output via programs

Either the default YAML format or `--format=json` can be used. Do not attempt to explicitly parse the output of `--format=humanreadable` as it will very likely change over time.

Programmatically detecting whether the system is deployed via bootc

Invoke e.g. `bootc status --json`, and check if `status.booted` is not `null`.

Detecting rpm-ostree vs bootc

There is no "bootc runtime". When used with the default ostree backend, bootc and tools like rpm-ostree end up sharing the same code and doing effectively the same thing. Hence, there isn't a mechanism to detect if a system "is bootc" or "is rpm-ostree".

However, if the `incompatible` flag is set on a deployment, then there are layered packages and `rpm-ostree` must be used for mutation.

OPTIONS

--format=*FORMAT*

```
The output format

Possible values:
- humanreadable
- yaml
- json
```

--format-version=*FORMAT_VERSION*

```
The desired format version. There is currently one supported version,
which is exposed as both `0` and `1`. Pass this option to explicitly
request it; it is possible that another future version 2 or newer will
be supported in the future
```

--booted

```
Only display status for the booted deployment
```

-v, **--verbose**

```
Include additional fields in human readable format
```

EXAMPLES

Show current system status:

```
bootc status
```

Show status in JSON format:

```
bootc status --format=json
```

Show detailed status with verbose output:

```
bootc status --verbose
```

Show only booted deployment status:

```
bootc status --booted
```

SEE ALSO

bootc(8), **bootc-upgrade**(8), **bootc-switch**(8), **bootc-rollback**(8)

VERSION

NAME

bootc-upgrade - Download and queue an updated container image to apply

SYNOPSIS

bootc upgrade [*OPTIONS...*]

DESCRIPTION

Download and queue an updated container image to apply.

This does not affect the running system; updates operate in an "A/B" style by default.

A queued update is visible as `staged` in `bootc status`.

Checking for Updates

The `--check` option allows you to verify if updates are available without downloading the full image layers. This only downloads the updated manifest and image configuration (typically kilobyte-sized metadata), making it much faster than a full upgrade.

Applying Updates

Currently by default, the update will be applied at shutdown time via `ostree-finalize-staged.service`. There is also an explicit `bootc upgrade --apply` verb which will automatically take action (rebooting) if the system has changed.

The `--apply` option currently always reboots the system. In the future, this command may detect cases where no kernel changes are queued and perform a userspace-only restart instead.

However, in the future this is likely to change such that reboots outside of a `bootc upgrade --apply` do *not* automatically apply the update in addition.

Soft Reboot

The `--soft-reboot` option configures soft reboot behavior when used with `--apply`:

- `required`: The operation will fail if soft reboot is not available on the target system
- `auto`: Uses soft reboot if available on the target system, otherwise falls back to a regular reboot

Soft reboot allows faster system restart by avoiding full hardware reboot when possible.

OPTIONS

--quiet

```
Don't display progress
```

--check

```
Check if an update is available without applying it
```

--apply

```
Restart or reboot into the new target image
```

--soft-reboot=*SOFT_REBOOT*

```
Configure soft reboot behavior

Possible values:
- required
- auto
```

--download-only

```
Download and stage the update without applying it
```

--from-downloaded

```
Apply a staged deployment that was previously downloaded with --
download-only
```

EXAMPLES

Check for available updates:

```
bootc upgrade --check
```

Upgrade and immediately apply the changes:

```
bootc upgrade --apply
```

Upgrade with soft reboot if possible:

```
bootc upgrade --apply --soft-reboot=auto
```

SEE ALSO

bootc(8), **bootc-switch**(8), **bootc-status**(8), **bootc-rollback**(8)

VERSION

NAME

bootc-switch - Target a new container image reference to boot

SYNOPSIS

bootc switch [*OPTIONS...*] <*TARGET*>

DESCRIPTION

Target a new container image reference to boot.

This is almost exactly the same operation as `upgrade`, but additionally changes the container image reference instead.

Usage

A common pattern is to have a management agent control operating system updates via container image tags; for example, `quay.io/exampleos/someuser:v1.0` and `quay.io/exampleos/someuser:v1.1` where some machines are tracking `:v1.0`, and as a rollout progresses, machines can be switched to `v:1.1`.

It is also supported to provide explicit digests, via e.g. `bootc switch quay.io/exampleos/someuser@sha256:9cca0703342e24806a9f64e08c053dca7f2cd90f10529af8ea872afb0a0c77d4`. When you do this, `bootc upgrade` will always be a no-op. In this model, upgrades are then always triggered by further `switch` operations.

Applying Changes

The `--apply` option will automatically take action (rebooting) if the system has changed after switching to the new image. Currently, this option always reboots the system. In the future, this command may detect cases where no kernel changes are queued and perform a userspace-only restart instead.

Soft Reboot

The `--soft-reboot` option configures soft reboot behavior when used with `--apply`:

- `required`: The operation will fail if soft reboot is not available on the target system
- `auto`: Uses soft reboot if available on the target system, otherwise falls back to a regular reboot

Soft reboot allows faster system restart by avoiding full hardware reboot when possible.

OPTIONS

TARGET

```
Target image to use for the next boot

This argument is required.
```

--quiet

```
Don't display progress
```

--apply

```
Restart or reboot into the new target image
```

--soft-reboot=*SOFT_REBOOT*

```
Configure soft reboot behavior

Possible values:
- required
- auto
```

--transport=*TRANSPORT*

```
The transport; e.g. registry, oci, oci-archive, docker-daemon,
containers-storage.  Defaults to `registry`

Default: registry
```

--enforce-container-sigpolicy

```
This is the inverse of the previous `--target-no-signature-verification`
(which is now a no-op)
```

--retain

```
Retain reference to currently booted image
```

EXAMPLES

Switch to a different image version:

```
bootc switch quay.io/exampleos/myapp:v1.1
```

Switch and immediately apply the changes:

```
bootc switch --apply quay.io/exampleos/myapp:v1.1
```

Switch with soft reboot if possible:

```
bootc switch --apply --soft-reboot=auto quay.io/exampleos/myapp:v1.1
```

SEE ALSO

bootc(8), **bootc-upgrade**(8), **bootc-status**(8), **bootc-rollback**(8)

VERSION

NAME

bootc-rollback - Change the bootloader entry ordering

SYNOPSIS

bootc rollback [*OPTIONS...*]

DESCRIPTION

Change the bootloader entry ordering; the deployment under `rollback` will be queued for the next boot, and the current will become rollback. If there is a `staged` entry (an unapplied, queued upgrade) then it will be discarded.

Note that absent any additional control logic, if there is an active agent doing automated upgrades (such as the default `bootc-fetch-apply-updates.timer` and associated `.service`) the change here may be reverted. It's recommended to only use this in concert with an agent that is in active control.

A systemd journal message will be logged with `MESSAGE_ID=26f3b1eb24464d12aa5e7b544a6b5468` in order to detect a rollback invocation.

Note on Rollbacks and the `/etc` Directory

When you perform a rollback (e.g., with `bootc rollback`), any changes made to files in the `/etc` directory won't carry over to the rolled-back deployment. The `/etc` files will revert to their state from that previous deployment instead.

This is because `bootc rollback` just reorders the existing deployments. It doesn't create new deployments. The `/etc` merges happen when new deployments are created.

OPTIONS

--apply

```
Restart or reboot into the rollback image
```

--soft-reboot=*SOFT_REBOOT*

```
Configure soft reboot behavior

Possible values:
- required
- auto
```

EXAMPLES

Rollback to the previous deployment:

```
bootc rollback
```

Rollback and immediately apply the changes:

```
bootc rollback --apply
```

Rollback with soft reboot if possible:

```
bootc rollback --apply --soft-reboot=auto
```

SEE ALSO

bootc(8), **bootc-upgrade**(8), **bootc-switch**(8), **bootc-status**(8)

VERSION

NAME

bootc-usr-overlay - Adds a transient writable overlayfs on `/usr` that will be discarded on reboot

SYNOPSIS

bootc usr-overlay [*OPTIONS...*]

DESCRIPTION

Adds a transient writable overlayfs on `/usr` that will be discarded on reboot.

USE CASES

A common pattern is wanting to use tracing/debugging tools, such as `strace` that may not be in the base image. A system package manager such as `apt` or `dnf` can apply changes into this transient overlay that will be discarded on reboot.

/ETC AND /VAR

However, this command has no effect on `/etc` and `/var` - changes written there will persist. It is common for package installations to modify these directories.

UNMOUNTING

Almost always, a system process will hold a reference to the open mount point. You can however invoke `umount -l /usr` to perform a "lazy unmount".

VERSION

NAME

bootc-fetch-apply-updates.service

DESCRIPTION

This service causes `bootc` to perform the following steps:

- Check the source registry for an updated container image
- If one is found, download it
- Reboot

This service also comes with a companion `bootc-fetch-apply-updates.timer` systemd unit. The current default systemd timer shipped in the upstream project is enabled for daily updates.

However, it is fully expected that different operating systems and distributions choose different defaults.

CUSTOMIZING UPDATES

Note that all three of these steps can be decoupled; they are:

- `bootc upgrade --check`
- `bootc upgrade`
- `bootc upgrade --apply`

SEE ALSO

bootc(1)

VERSION

NAME

bootc-status-updated.path

DESCRIPTION

This unit watches the `bootc` root directory (/ostree/bootc) for modification, and triggers the companion `bootc-status-updated.target` systemd unit.

The `bootc` program updates the mtime on its root directory when the contents of `bootc status` changes as a result of an update/upgrade/edit/switch/rollback operation.

SEE ALSO

bootc(1), **bootc-status-updated.target**(5)

VERSION

NAME

bootc-status-updated.target

DESCRIPTION

This unit is triggered by the companion `bootc-status-updated.path` systemd unit. This target is intended to enable users to add custom services to trigger as a result of `bootc status` changing.

Add the following to your unit configuration to active it when `bootc status` changes:

```
[Install]
WantedBy=bootc-status-updated.target
```

SEE ALSO

bootc(1), **bootc-status-updated.path**(5)

VERSION

Using bootc via API

At the current time, bootc is primarily intended to be driven via a fork/exec model. The core CLI verbs are stable and will not change.

Using `bootc edit` and `bootc status --json`

While bootc does not depend on Kubernetes, it does currently also offer a Kubernetes *style* API, especially oriented towards the spec and status and other conventions.

In general, most use cases of driving bootc via API are probably most easily done by forking off `bootc upgrade` when desired, and viewing `bootc status --json --format-version=1`.

JSON Schema

The current API `org.containers.bootc/v1` is stable. In order to support the future introduction of a v2 or newer format, please change your code now to explicitly request `--format-version=1` as referenced above. (Available since bootc 0.1.15, `--format-version=0` in bootc 0.1.14).

There is a JSON schema generated from the Rust source code available here: host-v1.schema.json.

A common way to use this is to run a code generator such as go-jsonschema on the input schema.

Installing "bootc compatible" images

A key goal of the bootc project is to think of bootable operating systems as container images. Docker/OCI container images are just tarballs wrapped with some JSON. But in order to boot a system (whether on bare metal or virtualized), one needs a few key components:

- bootloader
- kernel (and optionally initramfs)
- root filesystem (xfs/ext4/btrfs etc.)

The bootloader state is managed by the external bootupd project which abstracts over bootloader installs and upgrades. The invocation of `bootc install` will always run `bootupd` to handle bootloader installation to the target disk. The default expectation is that bootloader contents and install logic come from the container image in a `bootc` based system.

The Linux kernel (and optionally initramfs) is embedded in the container image; the canonical location is `/usr/lib/modules/$kver/vmlinuz`, and the initramfs should be in `initramfs.img` in that directory.

The `bootc install` command bridges the two worlds of a standard, runnable OCI image and a bootable system by running tooling logic embedded in the container image to create the filesystem and bootloader setup dynamically. This requires running the container via `--privileged`; it uses the running Linux kernel on the host to write the file content from the running container image; not the kernel inside the container.

There are two sub-commands: `bootc install to-disk` and `boot install to-filesystem`.

However, nothing *else* (external) is required to perform a basic installation to disk - the container image itself comes with a baseline self-sufficient installer that sets things up ready to boot.

Internal vs external installers

The `bootc install to-disk` process only sets up a very simple filesystem layout, using the default filesystem type defined in the container image, plus hardcoded requisite

platform-specific partitions such as the ESP.

In general, the `to-disk` flow should be considered mainly a "demo" for the `bootc install to-filesystem` flow, which can be used by "external" installers today. For example, in the Fedora/CentOS bootc project project, there are two "external" installers in Anaconda and `bootc-image-builder`.

More on this below.

Executing `bootc install`

The two installation commands allow you to install the container image either directly to a block device (`bootc install to-disk`) or to an existing filesystem (`bootc install to-filesystem`).

The installation commands **MUST** be run **from** the container image that will be installed, using `--privileged` and a few other options. This means you are (currently) not able to install `bootc` to an existing system and install your container image. Failure to run `bootc` from a container image will result in an error.

Here's an example of using `bootc install` (root/elevated permission required):

```
podman run --rm --privileged --pid=host -v /var/lib/containers:/var/lib/
containers -v /dev:/dev --security-opt label=type:unconfined_t <image>
bootc install to-disk /path/to/disk
```

Note that while `--privileged` is used, this command will not perform any destructive action on the host system. Among other things, `--privileged` makes sure that all host devices are mounted into container. `/path/to/disk` is the host's block device where `<image>` will be installed on.

The `--pid=host --security-opt label=type:unconfined_t` today make it more convenient for bootc to perform some privileged operations; in the future these requirements may be dropped.

The `-v /var/lib/containers:/var/lib/containers` option is required in order for the container to access its own underlying image, which is used by the installation process.

Jump to the section for `install to-filesystem` later in this document for additional

information about that method.

"day 2" updates, security and fetch configuration

By default the `bootc install` path will find the pull specification used for the `podman run` invocation and use it to set up "day 2" OS updates that `bootc update` will use.

For example, if you invoke `podman run --privileged ... quay.io/examplecorp/exampleos:latest bootc install ...` then the installed operating system will fetch updates from `quay.io/examplecorp/exampleos:latest`. This can be overridden via `--target_imgref`; this is handy in cases like performing installation in a manufacturing environment from a mirrored registry.

By default, the installation process will verify that the container (representing the target OS) can fetch its own updates.

Additionally note that to perform an upgrade with a target image reference set to an authenticated registry, you must provide a pull secret. One path is to embed the pull secret into the image in `/etc/ostree/auth.json`.

Configuring the default root filesystem type

To use the `to-disk` installation flow, the container should include a root filesystem type. If it does not, then each user will need to specify `install to-disk --filesystem`.

To set a default filesystem type for `bootc install to-disk` as part of your OS/distribution base image, create a file named `/usr/lib/bootc/install/00-<osname>.toml` with the contents of the form:

```
[install.filesystem.root]
type = "xfs"
```

Configuration files found in this directory will be merged, with higher alphanumeric values taking precedence. If for example you are building a derived container image from the above OS, you could create a `50-myos.toml` that sets `type = "btrfs"` which will override the prior setting.

For other available options, see bootc-install-config.

Installing an "unconfigured" image

The bootc project aims to support generic/general-purpose operating systems and distributions that will ship unconfigured images. An unconfigured image does not have a default password or SSH key, etc.

For more information, see Image building and configuration guidance.

More advanced installation with `to-filesystem`

The basic `bootc install to-disk` logic is really a pretty small (but opinionated) wrapper for a set of lower level tools that can also be invoked independently.

The `bootc install to-disk` command is effectively:

- `mkfs.$fs /dev/disk`
- `mount /dev/disk /mnt`
- `bootc install to-filesystem --karg=root=UUID=<uuid of /mnt> --imgref $self /mnt`

There may be a bit more involved here; for example configuring `--block-setup tpm2-luks` will configure the root filesystem with LUKS bound to the TPM2 chip, currently via systemd-cryptenroll.

Some OS/distributions may not want to enable it at all; it can be configured off at build time via Cargo features.

Using `bootc install to-filesystem`

The usual expected way for an external storage system to work is to provide `root=<UUID>` and `rootflags` kernel arguments to describe to the initial RAM disk how to find and mount the root partition. For more on this, see the below section discussing mounting the root filesystem.

Note that if a separate `/boot` is needed (e.g. for LUKS) you will also need to provide `--boot-mount-spec UUID=...` .

The `bootc install to-filesystem` command allows an operating system or distribution to ship a separate installer that creates more complex block storage or filesystem setups, but reuses the "top half" of the logic. For example, a goal is to change Anaconda to use this.

Postprocessing after to-filesystem

Some installation tools may want to inject additional data, such as adding an `/etc/hostname` into the target root. At the current time, bootc does not offer a direct API to do this. However, the backend for bootc is ostree, and it is possible to enumerate the deployments via ostree APIs.

You can use `ostree admin --sysroot=/path/to/target --print-current-dir` to find the newly created deployment directory. For detailed examples and usage, see the Injecting configuration before first boot section under `to-existing-root` documentation below.

We hope to provide a bootc-supported method to find the deployment in the future.

However, for tools that do perform any changes, there is a new `bootc install finalize` command which is optional, but recommended to run as the penultimate step before unmounting the target filesystem.

This command will perform some basic sanity checks and may also perform fixups on the target root. For example, a direction currently for bootc is to stop using `/etc/fstab`. While `install finalize` does not do this today, in the future it may automatically migrate `etc/fstab` to `rootflags` kernel arguments.

Using `bootc install to-disk --via-loopback`

Because every `bootc` system comes with an opinionated default installation process, you can create a raw disk image that you can boot via virtualization. Run these commands as root:

```
truncate -s 10G myimage.raw
podman run --rm --privileged --pid=host --security-opt
label=type:unconfined_t -v /dev:/dev -v /var/lib/containers:/var/lib/
containers -v .:/output <yourimage> bootc install to-disk --generic-
image --via-loopback /output/myimage.raw
```

Notice that we use `--generic-image` for this use case.

Set the environment variable `BOOTC_DIRECT_IO=on` to create the loopback device with direct-io enabled.

Using `bootc install to-existing-root`

This is a variant of `install to-filesystem`, which maximizes convenience for using an existing Linux system, converting it into the target container image. Note that the `/boot` (and `/boot/efi`) partitions *will be reinitialized* - so this is a somewhat destructive operation for the existing Linux installation.

Also, because the filesystem is reused, it's required that the target system kernel support the root storage setup already initialized.

The core command should look like this (root/elevated permission required):

```
podman run --rm --privileged -v /dev:/dev -v /var/lib/containers:/var/
lib/containers -v /:/target \
             --pid=host --security-opt label=type:unconfined_t \
             <image> \
             bootc install to-existing-root
```

It is assumed in this command that the target rootfs is pased via `-v /:/target` at this time.

As noted above, the data in `/boot` will be wiped, but everything else in the existing operating `/` is **NOT** automatically cleaned up. This can be useful, because it allows the new image to access data from the previous host system. For example, container images, database, user home directory data, and config files in `/etc` are all available after the subsequent reboot in `/sysroot` (which is the "physical root").

However, previous mount points or subvolumes will not be automatically mounted in the new system, e.g. a btrfs subvolume for /home will not be automatically mounted to /sysroot/home. These filesystems will persist and can be handled any way you want like manually mounting them or defining the mount points as part of the bootc image.

Managing configuration: before and after reboot

There are two distinct scenarios for managing configuration with `to-existing-root`:

Before rebooting (injecting new configuration): You can inject new configuration files into the newly installed deployment before the first boot. This is useful for adding custom `/etc/fstab` entries, systemd mount units, or other fresh configuration that the new system should have from the start.

After rebooting (migrating old configuration): You can copy or migrate configuration and data from the old system (now accessible at `/sysroot`) to the new system. This is useful for preserving network settings, user accounts, or application data from the previous installation.

Before reboot: Injecting new configuration

After running `bootc install to-existing-root`, you may want to inject configuration files (such as `/etc/fstab`, systemd units, or other configuration) into the newly installed system before rebooting. The new deployment is located in the ostree repository structure at:

```
/target/ostree/deploy/<stateroot>/deploy/<checksum>.<serial>/
```

Where `<stateroot>` defaults to `default` unless specified via `--stateroot`.

To find and modify the newly installed deployment:

```
# Get the deployment path
DEPLOY_PATH=$(ostree admin --sysroot=/target --print-current-dir)

# Add a systemd mount unit
cat > ${DEPLOY_PATH}/etc/systemd/system/data.mount <<EOF
[Unit]
Description=Data partition

[Mount]
What=UUID= ...
Where=/data
Type=xfs

[Install]
WantedBy=local-fs.target
EOF
```

Injecting kernel arguments for local state

An alternative approach is to key machine-local configuration from kernel arguments via

the `--karg` option to `bootc install to-existing-root`.

For example with filesystem mounts, systemd offers a `systemd.mount-extra` that can be used instead of `/etc/fstab`:

```
bootc install to-existing-root \
  --karg="systemd.mount-extra=UUID=<uuid>:/data:xfs:defaults"
```

The `systemd.mount-extra` syntax is: `source:path:type:options`

After reboot: Migrating data from the old system

After rebooting into the new bootc system, the previous root filesystem is mounted at `/sysroot`. You can then migrate configuration and data from the old system to the new one.

Important: Any data from `/etc` that you want to use in the new system must be manually copied from `/sysroot/etc` to `/etc` after rebooting into the new system. There is currently no automated mechanism for migrating this configuration data. This applies to network configurations, user accounts, application settings, and other system configuration stored in `/etc`.

For example, after rebooting:

```
# Copy network configuration from the old system
cp /sysroot/etc/sysconfig/network-scripts/ifcfg-eth0 /etc/sysconfig/
network-scripts/

# Copy application configuration
cp -r /sysroot/etc/myapp /etc/
```

A special case is using the `--root-ssh-authorized-keys` flag during installation to automatically inherit root's SSH keys (which may have been injected from e.g. cloud instance userdata via a tool like `cloud-init`). To do this, add `--root-ssh-authorized-keys /target/root/.ssh/authorized_keys` to the install command.

See also the bootc-install-to-existing-root(8) man page for more details.

Using `system-reinstall-bootc`

This is a separate binary included with bootc. It is an opinionated, interactive CLI that wraps `bootc install to-existing-root`. See bootc install to-existing-root for details on the installation operation.

`system-reinstall-bootc` can be run from an existing Linux system. It will pull the supplied image, prompt to setup SSH keys for accessing the system, and run `bootc install to-existing-root` with all the bind mounts and SSH keys configured.

It will also add the `bootc-destructive-cleanup.service` systemd unit that will run on first boot to cleanup parts of the previous system. The cleanup actions can be configured per distribution by creating a script and packaging it similar to this one for Fedora.

Using `bootc install to-filesystem --source-imgref <imgref>`

By default, `bootc install` has to be run inside a podman container. With this assumption, it can escape the container, find the source container image (including its layers) in the podman's container storage and use it to create the image.

When `--source-imgref <imgref>` is given, `bootc` no longer assumes that it runs inside podman. Instead, the given container image reference (see containers-transports(5) for accepted formats) is used to fetch the image. Note that `bootc install` still has to be run inside a chroot created from the container image. However, this allows users to use a different sandboxing tool (e.g. bubblewrap).

This argument is mainly useful for 3rd-party tooling for building disk images from bootable containers (e.g. based on osbuild).

Finding and configuring the physical root filesystem

On a bootc system, the "physical root" is different from the "logical root" of the booted container. For more on that, see filesystem. This section is about how the physical root filesystem is discovered.

Systems using systemd will often default to using systemd-fstab-generator and/or systemd-gpt-auto-generator. Support for the latter though for the root filesystem is conditional on EFI and a bootloader implementing the bootloader interface.

Outside of the discoverable partition model, a common baseline default for installers is to set `root=UUID=` (and optionally `rootflags=`) kernel arguments as machine specific state. When using `install to-filesystem`, you should provide these as explicit kernel arguments.

Some installation tools may want to generate an `/etc/fstab`. An important consideration is that when composefs is on by default (as it is expected to be) it will no longer work to have an entry for `/` in `/etc/fstab` (or a systemd `.mount` unit) that handles remounting the rootfs with updated options after exiting the initrd.

In general, prefer using the `rootflags` kernel argument for that use case; it ensures that the filesystem is mounted with the correct options to start, and avoid having an entry for `/` in `/etc/fstab`.

The physical root is mounted at `/sysroot`. It is an option for legacy `/etc/fstab` references for `/` to use `/sysroot` by default, but `rootflags` is preferred.

Configuring machine-local state

Per the filesystem section, `/etc` and `/var` are machine-local state by default. If you want to inject additional content after the installation process, at the current time this can be done by manually finding the target "deployment root" which will be underneath `/ostree/deploy/<stateroot>/deploy/`.

You can use `ostree admin --sysroot=/path/to/target --print-current-dir` to find the deployment directory. For detailed examples, see Injecting configuration before first boot.

Installation software such as Anaconda do this today to implement generic `%post` scripts and the like.

However, it is very likely that a generic bootc API to do this will be added.

NAME

bootc-install - Install the running container to a target

SYNOPSIS

bootc install [*OPTIONS...*] <*SUBCOMMAND*>

DESCRIPTION

Install the running container to a target.

Understanding installations

OCI containers are effectively layers of tarballs with JSON for metadata; they cannot be booted directly. The `bootc install` flow is a highly opinionated method to take the contents of the container image and install it to a target block device (or an existing filesystem) in such a way that it can be booted.

For example, a Linux partition table and filesystem is used, and the bootloader and kernel embedded in the container image are also prepared.

A bootc installed container currently uses OSTree as a backend, and this sets it up such that a subsequent `bootc upgrade` can perform in-place updates.

An installation is not simply a copy of the container filesystem, but includes other setup and metadata.

Secure Boot Keys

When installing with `systemd-boot`, bootc can let `systemd-boot` can handle enrollment of Secure Boot keys by putting signed EFI signature lists in `/usr/lib/bootc/install/secureboot-keys` which will copy over into `ESP/loader/keys` after bootloader installation. The keys will be copied to `loader/keys` subdirectory of the ESP.

after installing `systemd-boot` to the system. More information on how key enrollment works with `systemd-boot` is available in the systemd-boot man page.

SUBCOMMANDS

Command	Description
bootc install to-disk	Install to the target block device
bootc install to-filesystem	Install to an externally created filesystem structure
bootc install to-existing-root	Install to the host root filesystem
bootc install finalize	Execute this as the penultimate step of an installation using `install to-filesystem`
bootc install ensure-completion	Intended for use in environments that are performing an ostree-based installation, not bootc
bootc install print-configuration	Output JSON to stdout that contains the merged installation configuration as it may be relevant to calling processes using `install to-filesystem` that in particular want to discover the desired root filesystem type from the container image

VERSION

NAME

bootc-install-config.toml

DESCRIPTION

The `bootc install` process supports some basic customization. This configuration file is in TOML format, and will be discovered by the installation process in via "drop-in" files in `/usr/lib/bootc/install` that are processed in alphanumerical order.

The individual files are merged into a single final installation config, so it is supported for e.g. a container base image to provide a default root filesystem type, that can be overridden in a derived container image.

install

This is the only defined toplevel table.

The `install` section supports two subfields:

- `block`: An array of supported `to-disk` backends enabled by this base container image; if not specified, this will just be `direct`. The only other supported value is `tpm2-luks`. The first value specified will be the default. To enable both, use `block = ["direct", "tpm2-luks"]`.
- `filesystem`: See below.
- `kargs`: An array of strings; this will be appended to the set of kernel arguments.
- `match_architectures`: An array of strings; this filters the install config.

filesystem

There is one valid field:

- `root`: An instance of "filesystem-root"; see below

filesystem-root

There is one valid field:

`type` : This can be any basic Linux filesystem with a `mkfs.$fstype` . For example, `ext4` , `xfs` , etc.

Examples

```
[install.filesystem.root]
type = "xfs"
[install]
kargs = ["nosmt", "console=tty0"]
```

SEE ALSO

bootc(1)

VERSION

NAME

bootc-install-to-disk - Install to the target block device

SYNOPSIS

bootc install to-disk [*OPTIONS...*] <*DEVICE*>

DESCRIPTION

Install to the target block device.

This command must be invoked inside of the container, which will be installed. The container must be run in `--privileged` mode, and hence will be able to see all block devices on the system.

The default storage layout uses the root filesystem type configured in the container image, alongside any required system partitions such as the EFI system partition. Use `install to-filesystem` for anything more complex such as RAID, LVM, LUKS etc.

Partitioning details

The default as of bootc 1.11 uses the Discoverable Partitions Specification for the generated root filesystem, as well as any required system partitions such as the EFI system partition.

Note that by default when used with "type 1" bootloader setups (i.e. non-UKI) a kernel argument `root=UUID=<uuid of filesystem>` is injected by default.

When used with the composefs backend and UKIs, it's recommended that a bootloader implementing the DPS specification is used and that the root partition is auto-discovered.

OPTIONS

DEVICE

Target block device for installation. The entire device will be wiped

This argument is required.

--wipe

Automatically wipe all existing data on device

--block-setup=*BLOCK_SETUP*

Target root block device setup

Possible values:
- direct
- tpm2-luks

--filesystem=*FILESYSTEM*

Target root filesystem type

Possible values:
- xfs
- ext4
- btrfs

--root-size=*ROOT_SIZE*

Size of the root partition (default specifier: M). Allowed specifiers: M (mebibytes), G (gibibytes), T (tebibytes)

--source-imgref=*SOURCE_IMGREF*

Install the system from an explicitly given source

--target-transport=*TARGET_TRANSPORT*

The transport; e.g. oci, oci-archive, containers-storage. Defaults to `registry`

Default: registry

--target-imgref=*TARGET_IMGREF*

Specify the image to fetch for subsequent updates

--enforce-container-sigpolicy

This is the inverse of the previous `--target-no-signature-verification` (which is now a no-op). Enabling this option enforces that `/etc/containers/policy.json` includes a default policy which requires signatures

--run-fetch-check

Verify the image can be fetched from the bootc image. Updates may fail when the installation host is authenticated with the registry but the pull secret is not in the bootc image

--skip-fetch-check

Verify the image can be fetched from the bootc image. Updates may fail when the installation host is authenticated with the registry but the pull secret is not in the bootc image

--disable-selinux

Disable SELinux in the target (installed) system

--karg=*KARG*

Add a kernel argument. This option can be provided multiple times

--root-ssh-authorized-keys=*ROOT_SSH_AUTHORIZED_KEYS*

The path to an `authorized_keys` that will be injected into the `root` account

--generic-image

Perform configuration changes suitable for a "generic" disk image. At the moment:

--bound-images=*BOUND_IMAGES*

```
How should logically bound images be retrieved

Possible values:
- stored
- skip
- pull

Default: stored
```

--stateroot=*STATEROOT*

```
The stateroot name to use. Defaults to `default`
```

--via-loopback

```
Instead of targeting a block device, write to a file via loopback
```

--composefs-backend

```
If true, composefs backend is used, else ostree backend is used

Default: false
```

--insecure

```
Make fs-verity validation optional in case the filesystem doesn't
support it

Default: false
```

--bootloader=*BOOTLOADER*

```
The bootloader to use

Possible values:
- grub
- systemd
```

--uki-addon=*UKI_ADDON*

```
Name of the UKI addons to install without the ".efi.addon" suffix. This
option can be provided multiple times if multiple addons are to be
installed
```

EXAMPLES

Install to a disk, wiping all existing data:

```
bootc install to-disk --wipe /dev/sda
```

Install with a specific root filesystem type:

```
bootc install to-disk --filesystem xfs /dev/nvme0n1
```

Install with TPM2 LUKS encryption:

```
bootc install to-disk --block-setup tpm2-luks /dev/sda
```

Install with custom kernel arguments:

```
bootc install to-disk --karg=nosmt --karg=console=ttyS0 /dev/sda
```

SEE ALSO

bootc(8), **bootc-install**(8), **bootc-install-to-filesystem**(8)

VERSION

NAME

bootc-install-to-filesystem - Install to an externally created filesystem structure

SYNOPSIS

bootc install to-filesystem [*OPTIONS...*] <*ROOT_PATH*>

DESCRIPTION

Install to an externally created filesystem structure.

In this variant of installation, the root filesystem alongside any necessary platform partitions (such as the EFI system partition) are prepared and mounted by an external tool or script. The root filesystem is currently expected to be empty by default.

OPTIONS

ROOT_PATH

```
Path to the mounted root filesystem

This argument is required.
```

--root-mount-spec=*ROOT_MOUNT_SPEC*

```
Source device specification for the root filesystem.  For example,
`UUID=2e9f4241-229b-4202-8429-62d2302382e1`. If not provided, the UUID
of the target filesystem will be used. This option is provided as some
use cases might prefer to mount by a label instead via e.g.
`LABEL=rootfs`
```

--boot-mount-spec=*BOOT_MOUNT_SPEC*

```
Mount specification for the /boot filesystem
```

--replace=*REPLACE*

Initialize the system in-place; at the moment, only one mode for this is implemented. In the future, it may also be supported to set up an explicit "dual boot" system

Possible values:
- wipe
- alongside

--acknowledge-destructive

If the target is the running system's root filesystem, this will skip any warnings

--skip-finalize

The default mode is to "finalize" the target filesystem by invoking `fstrim` and similar operations, and finally mounting it readonly. This option skips those operations. It is then the responsibility of the invoking code to perform those operations

--source-imgref=*SOURCE_IMGREF*

Install the system from an explicitly given source

--target-transport=*TARGET_TRANSPORT*

The transport; e.g. oci, oci-archive, containers-storage. Defaults to `registry`

Default: registry

--target-imgref=*TARGET_IMGREF*

Specify the image to fetch for subsequent updates

--enforce-container-sigpolicy

This is the inverse of the previous `--target-no-signature-verification` (which is now a no-op). Enabling this option enforces that `/etc/containers/policy.json` includes a default policy which requires signatures

--run-fetch-check

Verify the image can be fetched from the bootc image. Updates may fail when the installation host is authenticated with the registry but the pull secret is not in the bootc image

--skip-fetch-check

Verify the image can be fetched from the bootc image. Updates may fail when the installation host is authenticated with the registry but the pull secret is not in the bootc image

--disable-selinux

Disable SELinux in the target (installed) system

--karg=*KARG*

Add a kernel argument. This option can be provided multiple times

--root-ssh-authorized-keys=*ROOT_SSH_AUTHORIZED_KEYS*

The path to an `authorized_keys` that will be injected into the `root` account

--generic-image

Perform configuration changes suitable for a "generic" disk image. At the moment:

--bound-images=*BOUND_IMAGES*

How should logically bound images be retrieved

Possible values:
- stored
- skip
- pull

Default: stored

--stateroot=*STATEROOT*

```
The stateroot name to use. Defaults to `default`
```

--composefs-backend

```
If true, composefs backend is used, else ostree backend is used

Default: false
```

--insecure

```
Make fs-verity validation optional in case the filesystem doesn't
support it

Default: false
```

--bootloader=*BOOTLOADER*

```
The bootloader to use

Possible values:
- grub
- systemd
```

--uki-addon=*UKI_ADDON*

```
Name of the UKI addons to install without the ".efi.addon" suffix. This
option can be provided multiple times if multiple addons are to be
installed
```

VERSION

NAME

bootc-install-to-existing-root - Install to the host root filesystem

SYNOPSIS

bootc install to-existing-root [*OPTIONS...*] [*ROOT_PATH*]

DESCRIPTION

Install to the host root filesystem.

This is a variant of `install to-filesystem` that is designed to install "alongside" the running host root filesystem. Currently, the host root filesystem's `/boot` partition will be wiped, but the content of the existing root will otherwise be retained, and will need to be cleaned up if desired when rebooted into the new root.

Managing configuration: before and after reboot

When using `to-existing-root`, there are two distinct scenarios for managing configuration files:

1. **Before rebooting**: Injecting new configuration into the newly installed system
2. **After rebooting**: Migrating configuration from the old system to the new system

Before reboot: Injecting new configuration

If you need to inject new configuration files (such as custom `/etc/fstab` entries, systemd mount units, or other configuration) into the newly installed system before rebooting, you can find the deployment directory in the ostree repository structure. The new deployment is located at:

```
/ostree/deploy/<stateroot>/deploy/<checksum>.<serial>/
```

Where `<stateroot>` defaults to `default` unless you specified a different value with `--`

`stateroot`.

To find the path to the newly installed deployment:

```
# Get the full deployment path directly
DEPLOY_PATH=$(ostree admin --sysroot=/target --print-current-dir)
```

This will return the full path, for example: `/target/ostree/deploy/default/deploy/807f233831a03d315289a4ba29c1670d8bd326d4569eabee7a84f25327997307.0`

You can then modify files in that deployment. For example, to add systemd mount units:

```
# Get deployment path
DEPLOY_PATH=$(ostree admin --sysroot=/target --print-current-dir)
# Add a systemd mount unit
vi ${DEPLOY_PATH}/etc/systemd/system/data.mount
```

Injecting kernel arguments for local state

A better approach for machine-local configuration like filesystem mounts is to inject kernel arguments during installation. Kernel arguments are ideal for local/machine-specific state in a bootc system.

For filesystem mounts, use `systemd.mount-extra` instead of `/etc/fstab`:

```
# Add a mount via kernel argument (preferred over /etc/fstab)
bootc install to-existing-root \
  --karg="systemd.mount-extra=UUID=<uuid>:/data:xfs:defaults"
```

The `systemd.mount-extra` syntax is: `source:path:type:options`

You can also inject other local kernel arguments for machine-specific configuration:

```
# Add console settings for serial access
bootc install to-existing-root --karg="console=ttyS0,115200"

# Add storage-specific options
bootc install to-existing-root --karg="rootflags=subvol=root"
```

This approach is cleaner than editing configuration files because kernel arguments are explicitly designed for local/machine-specific state in a bootc system.

Note: In the future, this functionality will be provided via a dedicated bootc API to make finding and modifying the deployment more straightforward.

After reboot: Migrating data from the old system

After rebooting into the new bootc system, the previous root filesystem data is accessible at `/sysroot` (the "physical root"). This allows you to migrate data from the old system to the new one.

Important: Any configuration data from `/etc` that you want to use in the new system must be **manually copied** from `/sysroot/etc` to `/etc` after rebooting. There is currently no automated mechanism for migrating this data.

For example, to migrate configuration after rebooting:

```
# After rebooting into the new system
# Copy network configuration from the old system
cp /sysroot/etc/sysconfig/network-scripts/ifcfg-eth0 /etc/sysconfig/
network-scripts/

# Copy application configuration
cp -r /sysroot/etc/myapp /etc/

# Selectively merge configuration files
vi /etc/resolv.conf  # Add nameservers from /sysroot/etc/resolv.conf

# For user accounts, use proper tools
vipw  # Carefully review and merge users from /sysroot/etc/passwd
```

This applies to network configurations, user accounts, application settings, and other system configuration stored in `/etc`. Review files in `/sysroot/etc` and manually copy or merge what you need into `/etc`.

Note: For filesystem mounts from `/etc/fstab` in the old system, consider using kernel arguments (via `systemd.mount-extra`) injected before reboot instead of migrating the fstab entries. See the "Injecting kernel arguments" section above.

OPTIONS

ROOT_PATH

Path to the mounted root; this is now not necessary to provide. Historically it was necessary to ensure the host rootfs was mounted at here via e.g. `-v /:/target`

--replace=*REPLACE*

Configure how existing data is treated

Possible values:
- wipe
- alongside

Default: alongside

--source-imgref=*SOURCE_IMGREF*

Install the system from an explicitly given source

--target-transport=*TARGET_TRANSPORT*

The transport; e.g. oci, oci-archive, containers-storage. Defaults to `registry`

Default: registry

--target-imgref=*TARGET_IMGREF*

Specify the image to fetch for subsequent updates

--enforce-container-sigpolicy

This is the inverse of the previous `--target-no-signature-verification` (which is now a no-op). Enabling this option enforces that `/etc/containers/policy.json` includes a default policy which requires signatures

--run-fetch-check

Verify the image can be fetched from the bootc image. Updates may fail when the installation host is authenticated with the registry but the pull secret is not in the bootc image

--skip-fetch-check

Verify the image can be fetched from the bootc image. Updates may fail when the installation host is authenticated with the registry but the pull secret is not in the bootc image

--disable-selinux

Disable SELinux in the target (installed) system

--karg=*KARG*

Add a kernel argument. This option can be provided multiple times

--root-ssh-authorized-keys=*ROOT_SSH_AUTHORIZED_KEYS*

The path to an `authorized_keys` that will be injected into the `root` account

--generic-image

Perform configuration changes suitable for a "generic" disk image. At the moment:

--bound-images=*BOUND_IMAGES*

How should logically bound images be retrieved

Possible values:
- stored
- skip
- pull

Default: stored

--stateroot=*STATEROOT*

The stateroot name to use. Defaults to `default`

--acknowledge-destructive

```
Accept that this is a destructive action and skip a warning timer
```

--cleanup

```
Add the bootc-destructive-cleanup systemd service to delete files from
the previous install on first boot
```

--composefs-backend

```
If true, composefs backend is used, else ostree backend is used

Default: false
```

--insecure

```
Make fs-verity validation optional in case the filesystem doesn't
support it

Default: false
```

--bootloader=*BOOTLOADER*

```
The bootloader to use

Possible values:
- grub
- systemd
```

--uki-addon=*UKI_ADDON*

```
Name of the UKI addons to install without the ".efi.addon" suffix. This
option can be provided multiple times if multiple addons are to be
installed
```

VERSION

bootc is read-only when run in a default container

Currently, running e.g. `podman run <someimage> bootc upgrade` will not work. There are a variety of reasons for this, such as the basic fact that by default a `docker|podman run <image>` doesn't know where to update itself; the image reference is not exposed into the target image (for security/operational reasons).

Supported operations

There are only two supported operations in a container environment today:

- `bootc status` : This can reliably be used to detect whether the system is actually booted via bootc or not.
- `bootc container lint` : See man/bootc-container-lint.8.md.

Testing bootc in a container

Eventually we would like to support having bootc run inside a container environment primarily for testing purposes. For this, please see the tracking issue.

NAME

bootc-container-lint - Perform relatively inexpensive static analysis checks as part of a container build

SYNOPSIS

bootc container lint [*OPTIONS*...]

DESCRIPTION

Perform relatively inexpensive static analysis checks as part of a container build.

This is intended to be invoked via e.g. `RUN bootc container lint` as part of a build process; it will error if any problems are detected.

OPTIONS

--rootfs=*ROOTFS*

```
Operate on the provided rootfs

Default: /
```

--fatal-warnings

```
Make warnings fatal
```

--list

```
Instead of executing the lints, just print all available lints. At the
current time, this will output in YAML format because it's reasonably
human friendly. However, there is no commitment to maintaining this
exact format; do not parse it via code or scripts
```

--skip=*SKIP*

```
Skip checking the targeted lints, by name. Use `--list` to discover the
set of available lints
```

--no-truncate

```
Don't truncate the output. By default, only a limited number of entries
are shown for each lint, followed by a count of remaining entries
```

VERSION

"bootc compatible" images

It is a toplevel goal of this project to tightly integrate with the OCI ecosystem and make booting containers a normal activity.

However, there are a number of basic requirements and integration points, some of which have distribution-specific variants.

Further at the current time, the bootc project makes a lot of use of ostree, and this can appear in the base image requirements.

ostree-in-container

With bootc 1.1.3 or later, it is no longer required to have a `/ostree` directory present in the base image.

To generate container images which do include `/ostree` from scratch, the underlying `ostree container` tooling is designed to operate on an existing ostree commit, and the `ostree container encapsulate` command can turn the commit into an OCI image. If you already have a pipeline which prdouces ostree commits as an output (e.g. using osbuild to produce `ostree` commit artifacts), then this allows a seamless transition to a bootc/OCI compatible ecosystem.

Higher level base image build tooling

A well tested tool to produce compatible base images is `rpm-ostree compose image`, which is used by the Fedora base image.

Standard image content

The bootc project provides a baseimage reference set of configuration files for base images. In particular at the current time the content defined by `base` must be used (or recreated). There is also suggested integration there with e.g. `dracut` to ensure the

initramfs is set up, etc.

Standard metadata for bootc compatible images

It is strongly recommended to do:

```
LABEL containers.bootc 1
```

This will signal that this image is intended to be usable with `bootc`.

Deriving from existing base images

It's important to emphasize that from one of these specially-formatted base images, every tool and technique for container building applies! In other words it will Just Work to do

```
FROM <bootc base image>
RUN dnf -y install foo && dnf clean all
```

You can then use `podman build`, `buildah`, `docker build`, or any other container build tool to produce your customized image. The only requirement is that the container build tool supports producing OCI container images.

Kernel

The Linux kernel (and optionally initramfs) is embedded in the container image; the canonical location is `/usr/lib/modules/$kver/vmlinuz`, and the initramfs should be in `initramfs.img` in that directory. You should *not* include any content in `/boot` in your container image. Bootc will take care of copying the kernel/initramfs as needed from the container image to `/boot`.

Future work for supporting UKIs will follow the recommendations of the uapi-group in Locations for Distribution-built UKIs Installed by Package Managers.

The `bootc container lint` command will check this.

The `ostree container commit` command

You may find some references to this; it is no longer very useful and is not recommended.

The bootloader setup

At the current time bootc relies on the bootupd project which handles bootloader installs and upgrades. The invocation of `bootc install` will always run `bootupd` to perform installations. Additionally, `bootc upgrade` will currently not upgrade the bootloader; you must invoke `bootupctl update`.

SELinux

Container runtimes such as `podman` and `docker` commonly apply a "coarse" SELinux policy to running containers. See container-selinux. It is very important to understand that non-bootc base images do not (usually) have any embedded `security.selinux` metadata at all; all labels on the toplevel container image are *dynamically* generated per container invocation, and there are no individually distinct e.g. `etc_t` and `usr_t` types.

In contrast, with the current OSTree backend for bootc, it is possible to include label metadata (and precomputed ostree checksums) in special metadata files in `/sysroot/ostree` that correspond to components of the base image. This is optional as of bootc v1.1.3.

File content in derived layers will be labeled using the default file contexts (from `/etc/selinux`). For example, you can do this (as of bootc 1.1.0):

```
RUN semanage fcontext -a -t httpd_sys_content_t "/web(/.*)?"
```

(This command will write to `/etc/selinux/$policy/policy/`.)

It will currently not work to do e.g.:

```
RUN chcon -t foo_t /usr/bin/foo
```

Because the container runtime state will deny the attempt to "physically" set the `security.selinux` extended attribute.

In the future, it is likely however that we add support for handling the `security.selinux` extended attribute in tar streams; but this can only currently be done with a custom build process.

Toplevel directories

In particular, a common problem is that inside a container image, it's easy to create arbitrary toplevel directories such as e.g. `/app` or `/aimodel` etc. But in some SELinux policies such as Fedora derivatives, these will be labeled as `default_t` which few domains can access.

References:

- https://github.com/ostreedev/ostree-rs-ext/issues/510

composefs

It is strongly recommended to enable the ostree composefs backend (but not strictly required) for bootc.

A reference enablement file to do so is in the base image content referenced above.

More in ostree-prepare-root.

Filesystem

As noted in other chapters, the bootc project currently depends on the ostree project for storing the base container image. Additionally there is a containers/storage instance for logically bound images.

However, bootc is intending to be a "fresh, new container-native interface", and ostree is an implementation detail.

First, it is strongly recommended that bootc consumers use the ostree composefs backend; to do this, ensure that you have a `/usr/lib/ostree/prepare-root.conf` that contains at least

```
[composefs]
enabled = true
```

This will ensure that the entire `/` is a read-only filesystem which is very important for achieving correct semantics.

Understanding container build/runtime vs deployment

When run *as a container* (e.g. as part of a container build), the filesystem is fully mutable in order to allow derivation to work. For more on container builds, see build guidance.

The rest of this document describes the state of the system when "deployed" to a physical or virtual machine, and managed by `bootc`.

Timestamps

bootc uses ostree, which currently squashes all timestamps to zero. This is now viewed as an implementation bug and will be changed in the future. For more information, see this tracker issue.

Understanding physical vs logical root with `/sysroot`

When the system is fully booted, it is into the equivalent of a `chroot`. The "physical" host root filesystem will be mounted at `/sysroot`. For more on this, see filesystem: sysroot.

This `chroot` filesystem is called a "deployment root". All the remaining filesystem paths below are part of a deployment root which is used as a final target for the system boot. The target deployment is determined via the `ostree=` kernel commandline argument.

/usr

The overall recommendation is to keep all operating system content in `/usr`, with directories such as `/bin` being symbolic links to `/usr/bin`, etc. See UsrMove for example.

However, with composefs enabled `/usr` is not different from `/`; they are part of the same immutable image. So there is not a fundamental need to do a full "UsrMove" with a bootc system.

/usr/local

The OSTree upstream recommendation suggests making `/usr/local` a symbolic link to `/var/usrlocal`. But because the emphasis of a bootc-oriented system is on users deriving custom container images as the default entrypoint, it is recommended here that base images configure `/usr/local` be a regular directory (i.e. the default).

Projects that want to produce "final" images that are themselves not intended to be derived from in general can enable that symbolic link in derived builds.

/etc

The `/etc` directory contains mutable persistent state by default; however, it is supported (and encouraged) to enable the `etc.transient` config option, see below as well.

When in persistent mode, it inherits the OSTree semantics of performing a 3-way merge across upgrades. In a nutshell:

- The *new default* `/etc` is used as a base
- The diff between current and previous `/etc` is applied to the new `/etc`
- Locally modified files in `/etc` different from the default `/usr/etc` (of the same deployment) will be retained

You can view the state via `ostree admin config-diff`. Note that the "diff" here includes metadata (uid, gid, extended attributes), so changing any of those will also mean that updated files from the image are not applied.

The implementation of this defaults to being executed by `ostree-finalize-staged.service` at shutdown time, before the new bootloader entry is created.

The rationale for this design is that in practice today, many components of a Linux system end up shipping default configuration files in `/etc`. And even if the default package doesn't, often the software only looks for config files there by default.

Some other image-based update systems do not have distinct "versions" of `/etc` and it may be populated only set up at install time, and untouched thereafter. But that creates "hysteresis" where the state of the system's `/etc` is strongly influenced by the initial image version. This can lead to problems where e.g. a change to `/etc/sudoers` (to give one simple example) would require external intervention to apply.

For more on configuration file best practices, see Building.

To emphasize again, it's recommended to enable `etc.transient` if possible, though when using that you may need to store some machine-specific state in e.g. the kernel commandline if applicable.

/usr/etc

The `/usr/etc` tree is generated client side and contains the default container image's view of `/etc`. This should generally be considered an internal implementation detail of bootc/ostree. Do *not* explicitly put files into this location, it can create undefined behavior. There is a check for this in `bootc container lint`.

/var

Content in `/var` persists by default; it is however supported to make it or subdirectories mount points (whether network or `tmpfs`). There is exactly one `/var`. If it is not a distinct partition, then it is automatically made a bind from `/ostree/deploy/$stateroot/var` and shared across "deployments" (bootloader entries).

You may include content in `/var` in your image - and reference base images may have a few basic directories such as `/var/tmp` (in order to ease use in container builds).

However, it is very important to understand that content included in `/var` in the container image acts like a Docker `VOLUME /var`. This means its contents are unpacked *only from the initial image* - subsequent changes to `/var` in a container image are not automatically applied.

A common case is for applications to want some directory structure (e.g. `/var/lib/postgresql`) to be pre-created. It's recommended to use systemd tmpfiles.d for this. An even better approach where applicable is StateDirectory= in units.

As of bootc 1.1.6, the `bootc container lint` command will check for missing `tmpfiles.d` entries and warn.

Note this is very different from the handling of `/etc`. The rationale for this is that `/etc` is relatively small configuration files, and the expected configuration files are often bound to the operating system binaries in `/usr`.

But `/var` has arbitrarily large data (system logs, databases, etc.). It would also not be expected to be rolled back if the operating system state is rolled back. A simple example is that an `apt|dnf downgrade postgresql` should not affect the physical database in general in `/var/lib/postgres`. Similarly, a bootc update or rollback should not affect this application data.

Having `/var` separate also makes it work cleanly to "stage" new operating system updates before applying them (they're downloaded and ready, but only take effect on reboot).

In general, this is the same rationale for Docker `VOLUME`: decouple the application code from its data.

Other directories

It is not supported to ship content in `/run` or `/proc` or other API Filesystems in container images.

Besides those, for other toplevel directories such as `/usr` `/opt`, they will be lifecycled with the container image.

`/opt`

In the default suggested model of using composefs (per above) the `/opt` directory will be read-only, alongside other toplevels such as `/usr`.

Some software (especially "3rd party" deb/rpm packages) expect to be able to write to a subdirectory of `/opt` such as `/opt/examplepkg`.

See building images for recommendations on how to build container images and adjust the filesystem for cases like this.

However, for some use cases, it may be easier to allow some level of mutability. There are two options for this, each with separate trade-offs: transient roots and state overlays.

Other toplevel directories

Creating other toplevel directories and content (e.g. `/afs`, `/arbitrarymountpoint`) or in general further nested data is supported - just create the directory as part of your container image build process (e.g. `RUN mkdir /arbitrarymountpoint`). These directories will be lifecycled with the container image state, and appear immutable by default, the same as all other directories such as `/usr` and `/opt`.

Mounting separate filesystems there can be done by the usual mechanisms of `/etc/fstab`, systemd `.mount` units, etc.

SELinux for arbitrary toplevels

Note that operating systems using SELinux may use a label such as `default_t` for unknown toplevel directories, which may not be accessible by some processes. In this situation you currently may need to also ensure a label is defined for them in the file contexts.

Enabling transient root

This feature enables a fully transient writable rootfs by default. To do this, set the

```
[root]
transient = true
```

option in `/usr/lib/ostree/prepare-root.conf`. In particular this will allow software to write (transiently, i.e. until the next reboot) to all top-level directories, including `/usr` and `/opt`, with symlinks to `/var` for content that should persist.

This can be combined with `etc.transient` as well (below).

More on prepare-root: https://ostreedev.github.io/ostree/man/ostree-prepare-root.html

Note that regenerating the initramfs is required when changing this file.

Dynamic mountpoints with transient-ro

The `transient-ro` option allows privileged users to create dynamic toplevel mountpoints at runtime while keeping the filesystem read-only by default. This is particularly useful for applications that need to bind mount host paths that may be platform-specific or dynamic.

Use cases

This feature addresses scenarios where:

- Applications need to bind mount host directories that match the host's absolute paths
- Platform-specific mountpoints are required (e.g., `/Users` on macOS)
- Dynamic mountpoints need to be created after deployment but before application startup
- The filesystem should remain read-only for regular processes

Configuration

To enable this feature, add the following to `/usr/lib/ostree/prepare-root.conf`:

```
[root]
transient-ro = true
```

How it works

When `transient-ro=true` is set:

1. The overlayfs upper directory is mounted read-only by default
2. Privileged processes can remount it as writable only in a new mount namespace, and perform arbitrary changes there, such as creating new toplevel mountpoints
3. These mountpoints persist for the current boot but do not survive reboots or upgrades
4. Regular processes continue to see a read-only filesystem

A privileged process can achieve this using standard Linux commands. For example:

```
# unshare -m -- /bin/sh -c 'mount -o remount,rw / && mkdir /new-
mountpoint'
```

Example: Podman machine integration

A common use case is with `podman machine` on macOS, where the VM needs to bind mount host paths like `/Users/username` into the VM. With `transient-ro`, the system can:

1. Create the `/Users` directory dynamically at runtime
2. Bind mount the host's `/Users` directory to the VM's `/Users`
3. Keep the rest of the filesystem read-only for security

Enabling transient etc

The default (per above) is to have `/etc` persist. If however you do not need to use it for any per-machine state, then enabling a transient `/etc` is a great way to reduce the amount of possible state drift. Set the

```
[etc]
transient = true
```

option in `/usr/lib/ostree/prepare-root.conf`.

This can be combined with `root.transient` as well (above).

More on prepare-root: https://ostreedev.github.io/ostree/man/ostree-prepare-root.html

Note that regenerating the initramfs is required when changing this file.

Enabling state overlays

This feature enables a writable overlay on top of `/opt` (or really, any toplevel or subdirectory baked into the image that is normally read-only).

The semantics here are somewhat nuanced:

- Changes persist across reboots by default
- During updates, new files from the container image override any locally modified version

The advantages are:

- It makes it very easy to make compatible applications that install into `/opt`.
- In contrast to transient root (above), a smaller surface of the filesystem is mutable.

The disadvantages are:

- There is no equivalent to this feature in the Docker/Podman ecosystem.
- It allows for some temporary state drift until the next update.

To enable this feature, instantiate the `ostree-state-overlay@.service` unit template on the target path. For example, for `/opt`:

```
RUN systemctl enable ostree-state-overlay@opt.service
```

More generally dealing with `/opt`

Both transient root and state overlays above provide ways for packages that install in `/opt` to operate. However, for maximum immutability the best approach is simply to symlink just the parts of the `/opt` needed into `/var`. See the section on `/opt` in Image building and configuration guidance for a more concrete example.

Increased filesystem integrity with fsverity

The bootc project uses composefs by default for the root filesystem (using ostree's support for composefs). However, the default configuration as recommended for base images uses composefs in a mode that does not require signatures or fsverity.

bootc supports with ostree's model of hard requiring fsverity for underlying objects. Enabling this also causes bootc to error out at install time if the target filesystem does not enable fsverity.

To enable this, inside your container build update `/usr/lib/ostree/prepare-root.conf` with:

```
[composefs]
enabled = verity
```

At the current time, there is no default recommended mechanism to check the integrity of the upper composefs. For more information about this, see this tracking issue.

Note that the default `/etc` and `/var` mounts are unaffected by this configuration. Because `/etc` in particular can easily contain arbitrary executable code (`/etc/systemd/system` unit files), many deployment scenarios that want to hard require fsverity will also want a "transient etc" model.

Caveats

Does not apply to logically bound images

The logically bound images store is currently implemented using a separate mechanism

and configuring fsverity for the bootc storage has no effect on it.

Enabling fsverity across upgrades

At the current time the integration is only for installation; there is not yet support for automatically ensuring that fsverity is enabled when upgrading from a state with `composefs.enabled = yes` to `composefs.enabled = verity`. Because older objects may not have fsverity enabled, the new system will likely fail at runtime to access these older files across the upgrade.

Filesystem: Physical /sysroot

The bootc project uses ostree as a backend, and maps fetched container images to a deployment.

stateroot

The underlying `ostree` CLI and API tooling expose a concept of `stateroot`, which is not yet exposed via `bootc`. The `stateroot` used by `bootc install` is just named `default`.

The stateroot concept allows having fully separate parallel operating system installations with fully separate `/etc` and `/var`, while still sharing an underlying root filesystem.

In the future, this functionality will be exposed and used by `bootc`.

/sysroot mount

When booted, the physical root will be available at `/sysroot` as a read-only mount point and the logical root `/` will be a bind mount pointing to a deployment directory under `/sysroot/ostree`. This is a key aspect of how `bootc upgrade` operates: it fetches the updated container image and writes the base image files (using OSTree storage to `/sysroot/ostree/repo`).

Beyond that and debugging/introspection, there are few use cases for tooling to operate on the physical root.

bootc-owned container storage

For logically bound images, bootc maintains a dedicated containers/storage instance using the `overlay` backend (the same type of thing that backs `/var/lib/containers`).

This storage is accessible via a `/usr/lib/bootc/storage` symbolic link which points into `/sysroot`. (Avoid directly referencing the `/sysroot` target)

At the current time, this storage is *not* used for the base bootable image. This unified storage issue tracks unification.

Expanding the root filesystem

One notable use case that *does* need to operate on `/sysroot` is expanding the root filesystem.

Some higher level tools such as e.g. `cloud-init` may (reasonably) expect the `/` mount point to be the physical root. Tools like this will need to be adjusted to instead detect this and operate on `/sysroot`.

Growing the block device

Fundamentally bootc is agnostic to the underlying block device setup. How to grow the root block device depends on the underlying storage stack, from basic partitions to LVM. However, a common tool is the growpart utility from `cloud-init`.

Growing the filesystem

The systemd project ships a systemd-growfs tool and corresponding `systemd-growfs@` services. This is a relatively thin abstraction over detecting the target root filesystem type and running the underlying tool such as `xfs_growfs`.

At the current time, most Linux filesystems require the target to be mounted writable in order to grow. Hence, an invocation of `system-growfs /sysroot` or `xfs_growfs /sysroot` will need to be further wrapped in a temporary mount namespace.

Using a `MountFlags=slave` drop-in stanza for `systemd-growfs@sysroot.service` is recommended, along with an `ExecStartPre=mount -o remount,rw /sysroot`.

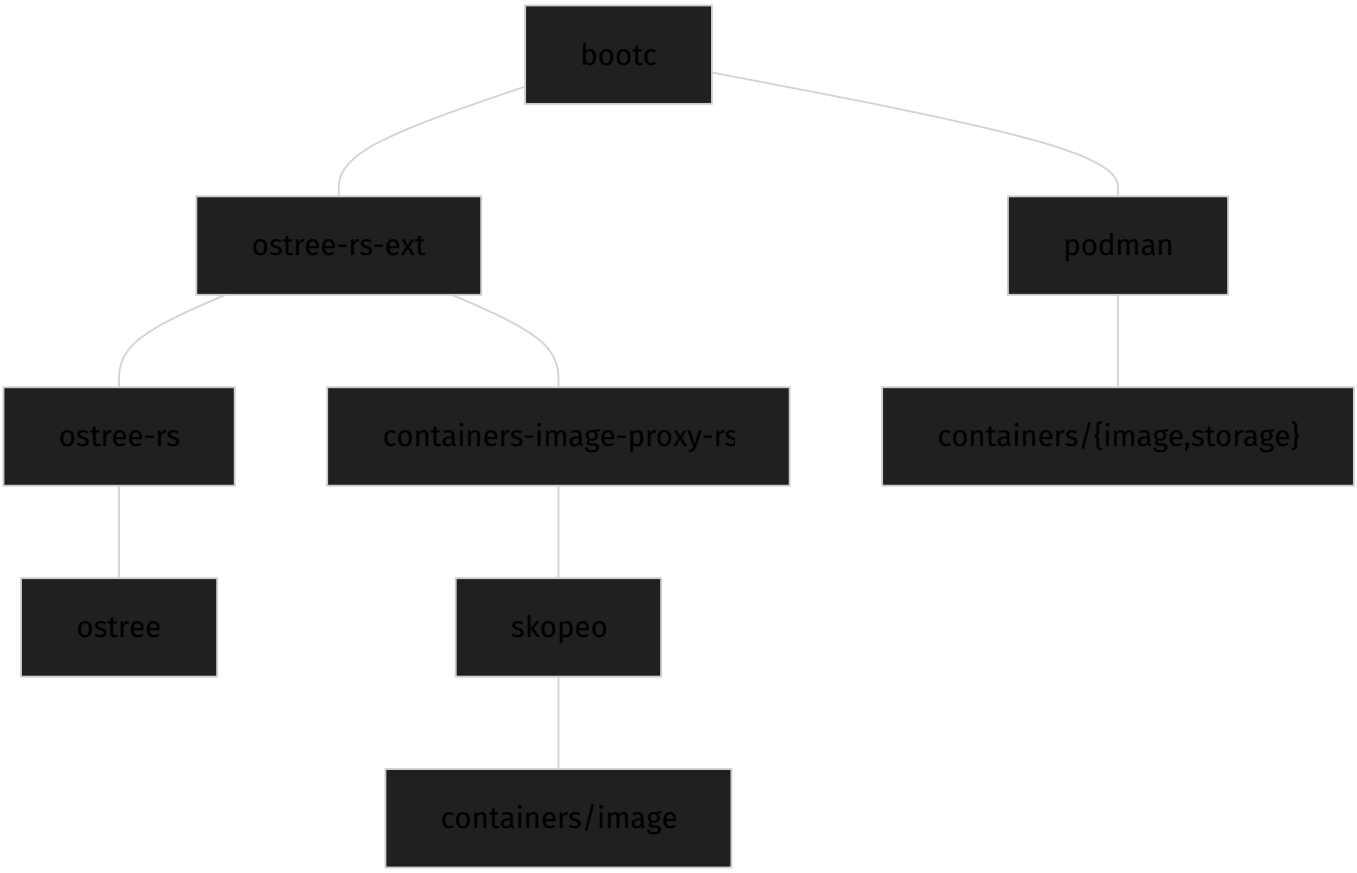
Detecting bootc/ostree systems

See the package managers section on "Detecting image based systems".

Container storage

The bootc project uses ostree and specifically the ostree-rs-ext Rust library which handles storage of container images on top of an ostree-based system for the booted host, and additionally there is a containers/storage instance for logically bound images.

Architecture



There were two high level goals that drove the design of the current system architecture:

- Support seamless in-place migrations from existing ostree systems
- Avoid requiring deep changes to the podman stack

A simple way to explain the current architecture is that podman uses two Go libraries:

- https://github.com/containers/image
- https://github.com/containers/storage

Whereas ostree uses a custom container storage, not `containers/storage`.

Mapping container images to ostree

OCI images are effectively just a standardized format of tarballs wrapped with JSON - specifically "layers" of tarballs.

The ostree-rs-ext project maps layers to OSTree commits. Each layer is stored separately, under an ostree "ref" (like a git branch) under the `ostree/container/` namespace:

```
$ ostree refs ostree/container
```

Layers

The `ostree/container/blob` namespace tracks storage of a container layer identified by its blob ID (sha256 digest).

Images

At the current time, ostree always boots into a "flattened" filesystem tree. This is generated as both a hardlinked checkout as well as a composefs image.

The flattened tree is constructed and committed into the `ostree/container/image` namespace. The commit metadata also includes the OCI manifest and config objects.

This is implemented in the ostree-rs-ext/container module.

SELinux labeling

See the SELinux section of Image layout.

Origin files

ostree has the concept of an `origin` file which defines the source of truth for upgrades.

The container image reference for each deployment is included in its origin.

Booting

A core aspect of this entire design is that once a container image is fetched into the ostree storage, from there on it just appears as an "ostree commit", and so all code built on top can work with it.

For example, the `ostree-prepare-root.service` which runs in the initramfs is currently agnostic to whether the filesystem tree originated from an OCI image or some other mechanism; it just targets a prepared flattened filesystem tree.

This is what is referenced by the `ostree=` kernel commandline.

Logically bound images

In addition to the base image, bootc supports logically bound images.

Bootloaders in `bootc`

`bootc` supports two ways to manage bootloaders.

bootupd

bootupd is a project explicitly designed to abstract over and manage bootloader installation and configuration. Today it primarily supports GRUB+shim. There are pending patches for it to support systemd-boot as well.

When you run `bootc install`, it invokes `bootupctl backend install` to install the bootloader to the target disk or filesystem. The specific bootloader configuration is determined by the container image and the target system's hardware.

Currently, `bootc` only runs `bootupd` during the installation process. It does **not** automatically run `bootupctl update` to update the bootloader after installation. This means that bootloader updates must be handled separately, typically by the user or an automated system update process.

systemd-boot

If bootupd is not present in the input container image, then systemd-boot will be used by default (except on s390x).

s390x

bootc uses `zipl`.

---

bootc image

Experimental features are subject to change or removal. Please do provide feedback on them.

Tracking issue: https://github.com/bootc-dev/bootc/issues/690

Using `bootc image copy-to-storage`

This experimental command is intended to aid in booting local builds.

Invoking this command will default to copying the booted container image into the `containers-storage:` area as used by e.g. `podman`, under the image tag `localhost/bootc` by default. It can then be managed independently; used as a base image, pushed to a registry, etc.

Run `bootc image copy-to-storage --help` for more options.

Example workflow:

```
$ bootc image copy-to-storage
$ cat Containerfile
FROM localhost/bootc
...
$ podman build -t localhost/bootc-custom .
$ bootc switch --transport containers-storage localhost/bootc-custom
```

---

composefs backend

Experimental features are subject to change or removal. Please do provide feedback on them.

Tracking issue: https://github.com/bootc-dev/bootc/issues/1190

Overview

The composefs backend is an experimental alternative storage backend that uses composefs-rs instead of ostree for storing and managing bootc system deployments.

Status: Experimental. The composefs backend is under active development and not yet suitable for production use. The feature is always compiled in as of bootc v1.10.1.

Key Benefits

- **Native container integration**: Direct use of container image formats without the ostree layer
- **UKI support**: First-class support for Unified Kernel Images (UKIs) and systemd-boot
- **Sealed images**: Enables building cryptographically sealed, securely-bootable images
- **Simpler architecture**: Reduces dependency on ostree as an implementation detail

Building Sealed Images

Using `just build-sealed`

This is an entrypoint focused on *bootc development* itself - it builds bootc from source.

```
just build-sealed
```

We are working on documenting individual steps to build a sealed image outside of this

tooling.

How Sealed Images Work

A sealed image includes:

- A Unified Kernel Image (UKI) that combines kernel, initramfs, and boot parameters
- The composefs fsverity digest embedded in the kernel command line
- Secure Boot signatures on both the UKI and systemd-boot loader

The UKI is placed in `/boot/EFI/Linux/` and includes the composefs digest in its command line:

```
composefs=${COMPOSEFS_FSVERITY} root=UUID= ...
```

This enables the boot chain to verify the integrity of the root filesystem.

Installation

When installing a composefs-backend system, use:

```
bootc install to-disk /dev/sdX
```

Note: Sealed images will require fsverity support on the target filesystem by default.

Testing Composefs

To run the composefs integration tests:

```
just test-composefs
```

This builds a sealed image and runs the composefs test suite using `bcvk` (bootc VM tooling).

Current Limitations

- **Experimental**: In particular, the on-disk formats are subject to change
- **UX refinement**: The user experience for building and managing sealed images is still being improved

Related Issues

- #1190 - composefs-native backend (main tracker)
- #1498 - Sealed image build UX + implementation
- #1703 - OCI config mismatch issues
- #20 - Unified storage (long-term goal)
- #806 - UKI/systemd-boot tracker

Additional Resources

- See filesystem.md for information about composefs in the standard ostree backend
- See bootloaders.md for bootloader configuration details

NAME

bootc-root-setup.service

DESCRIPTION

This service runs in the initramfs to set up the root filesystem when composefs is enabled. It is only activated when the `composefs` kernel command line parameter is present.

The service performs the following operations:

- Mounts the composefs image specified in the kernel command line
- Sets up `/etc` and `/var` directories from the deployment state
- Optionally configures transient overlays based on the configuration file
- Prepares the root filesystem for switch-root

This service runs after `sysroot.mount` and `ostree-prepare-root.service`, and before `initrd-root-fs.target`.

CONFIGURATION FILE

The service reads an optional configuration file at `/usr/lib/composefs/setup-root-conf.toml`. If this file does not exist, default settings are used.

WARNING: The configuration file format and composefs integration are experimental and subject to change.

Configuration Options

The configuration file uses TOML format with the following sections:

`[root]`

- `transient` (boolean): If true, mounts the root filesystem as a transient overlay. This

makes all changes to `/` ephemeral and lost on reboot. Default: false.

`[etc]`

- `mount` (string): Mount type for `/etc`. Options: "none", "bind", "overlay", "transient". Default: "bind".
- `transient` (boolean): Shorthand for `mount = "transient"`. Default: false.

`[var]`

- `mount` (string): Mount type for `/var`. Options: "none", "bind", "overlay", "transient". Default: "bind".
- `transient` (boolean): Shorthand for `mount = "transient"`. Default: false.

Example Configuration

```
[root]
transient = false

[etc]
mount = "bind"

[var]
mount = "overlay"
```

EXPERIMENTAL STATUS

The composefs integration, including this service and its configuration file format, is experimental and subject to change.

SEE ALSO

bootc(8)

VERSION

bootc internals fsck

Experimental features are subject to change or removal. Please do provide feedback on them.

Using `bootc internals fsck`

This command expects a booted system, and performs consistency checks in a read-only fashion.

Factory reset with `bootc install reset`

This is an experimental feature; use `--experimental` flag to acknowledge.

Overview

The `bootc install reset` command allows you to perform a non-destructive factory reset of an existing bootc system. This creates a fresh installation state in a new stateroot while preserving the existing system's files on disk. After rebooting into the new deployment, you can still access the old system's data by examining files in `/sysroot/ostree/deploy/<old-stateroot>/`.

How it works

When you run `bootc install reset`:

1. A new stateroot is created with an automatically generated name (format: `state-<year>-<serial>`, e.g., `s2025-0`)
2. A fresh deployment is created in the new stateroot using the currently booted image (or optionally a different image via `--target-imgref`)
3. Kernel arguments related to root filesystem configuration are automatically inherited from the current deployment
4. The `/boot` fstab entry is preserved from the current system if it exists
5. The new deployment becomes the default boot target

After rebooting, you'll be running in a completely fresh system state:

- `/etc` contains only the configuration from the container image
- `/var` is empty (no user data or state from the previous system)
- The old stateroot's files remain on disk at `/sysroot/ostree/deploy/<old-stateroot>/` and can be accessed for data recovery or inspection

Usage

Basic usage (reset to the same image currently running):

```
bootc install reset --experimental
```

Reset and switch to a different image:

```
bootc install reset --experimental --target-imgref quay.io/example/
myimage:latest
```

Reset with custom stateroot name:

```
bootc install reset --experimental --stateroot production-2025
```

Reset and immediately reboot:

```
bootc install reset --experimental --apply
```

Add custom kernel arguments:

```
bootc install reset --experimental --karg=console=ttyS0,115200n8
```

Skip inheriting root filesystem kernel arguments:

```
bootc install reset --experimental --no-root-kargs
```

Kernel arguments

By default, `bootc install reset` automatically inherits kernel arguments from the currently booted deployment that are related to root filesystem configuration. This includes:

- `root=` - Root device specification
- `rootflags=` - Root filesystem mount options
- `rd.*` arguments - Initramfs arguments (e.g., for LVM, LUKS, network root)
- Kernel arguments defined in `/usr/lib/bootc/kargs.d/` and `/etc/bootc/kargs.d/`

You can:

- Add additional kernel arguments with `--karg` (can be specified multiple times)
- Skip automatic root filesystem argument inheritance with `--no-root-kargs`

Use cases

- **Development/testing**: Quickly return to a clean state while preserving the ability to boot back to your development environment
- **Troubleshooting**: Reset to a known-good state without losing access to the problematic deployment for debugging
- **System refresh**: Start fresh after accumulating configuration changes, while keeping the old state accessible
- **Image testing**: Test a new image version in a separate stateroot before committing to it

Cleaning up the old stateroot

After performing a factory reset and rebooting into the new stateroot, the old stateroot remains on disk at `/sysroot/ostree/deploy/<old-stateroot>/`. This allows you to access files from the previous system if needed.

Once you no longer need the old stateroot, you can remove it to free up disk space:

1. First, remove any remaining deployments from the old stateroot:

```
# List all deployments to find the old stateroot's deployment index
ostree admin status

# Remove the old deployment(s) by index
# The index is shown in the output (e.g., "1" for the second deployment)
ostree admin undeploy <index>
```

2. After all deployments from the old stateroot are removed, you can delete the stateroot directory:

```
# Replace "default" with your old stateroot name if different
mount -o remount,rw /sysroot
rm -rf /sysroot/ostree/deploy/default
```

Note: You cannot remove the stateroot directory while deployments still exist in it. OSTree protects deployment directories with filesystem-level mechanisms, so you must undeploy them first using `ostree admin undeploy`.

Limitations

- This command requires `--experimental` flag as the feature is still under development
- Only works on systems already running bootc (not for initial installations)
- The old stateroot is not automatically removed and will consume disk space until manually deleted (see "Cleaning up the old stateroot" section above)

See also

- `bootc switch` - Switch to a different container image
- `bootc status` - View current deployment status

Interactive progress with `--progress-fd`

This is an experimental feature; tracking issue: https://github.com/bootc-dev/bootc/issues/1016

While the `bootc status` tooling allows a client to discover the state of the system, during interactive changes such as `bootc upgrade` or `bootc switch` it is possible to monitor the status of downloads or other operations at a fine-grained level with `--progress-fd`.

The format of data output over `--progress-fd` is JSON Lines which is a series of JSON objects separated by newlines (the intermediate JSON content is guaranteed not to contain a literal newline).

You can find the JSON schema describing this version here: progress-v0.schema.json.

Deploying a new image with either switch or upgrade consists of three stages: `pulling`, `importing`, and `staging`. The `pulling` step downloads the image from the registry, offering per-layer and progress in each message. The `importing` step imports the image into storage and consists of a single step. Finally, `staging` runs a variety of staging tasks. Currently, they are staging the image to disk, pulling bound images, and removing old images.

Note that new stages or fields may be added at any time.

Importing and staging are affected by disk speed and the total image size. Pulling is affected by network speed and how many layers invalidate between pulls. Therefore, a large image with a good caching strategy will have longer importing and staging times, and a small bespoke container image will have negligible importing and staging times.

Packaging and Integration

This document describes how to build and package bootc for distribution in operating systems.

Build Requirements

- Rust toolchain (see `rust-toolchain.toml` for the version)
- `coreutils` and `make`

Basic Build Commands

The primary build targets are:

```
make all
```

This builds:

- Binary artifacts (`cargo build --release`)
- Man pages (via `cargo xtask manpages`)

The built binaries are placed in `target/release/` :

- `bootc` - The main bootc CLI
- `system-reinstall-bootc` - System reinstallation tool
- `bootc-initramfs-setup` - Initramfs setup utility

Installation

The `install` target supports the standard `DESTDIR` variable for staged installations, which is essential for packaging:

```
make install DESTDIR=/path/to/staging/root
```

The install target handles:

- Binary installation to `$(prefix)/bin`
- Man pages to `$(prefix)/share/man/man{5,8}`
- systemd units to `$(prefix)/lib/systemd/system`
- Documentation and examples to `$(prefix)/share/doc/bootc`
- Dracut module to `/usr/lib/dracut/modules.d/51bootc`
- Base image configuration files

Optional Installation Targets

install-ostree-hooks

For distributions that need bootc to provide compatibility with `ostree container` commands:

```
make install-ostree-hooks DESTDIR=/tmp/stage
```

This creates symbolic links in `$(prefix)/libexec/libostree/ext/` for:

- `ostree-container`
- `ostree-ima-sign`
- `ostree-provisional-repair`

Source Packaging

Vendored Dependencies

bootc is written in Rust and has numerous dependencies. For distribution packaging, we recommend using a vendored tarball of Rust crates to ensure reproducible builds and avoid network access during the build process.

Generating the Vendor Tarball

Use the `cargo xtask package` command to generate both source and vendor tarballs:

```
cargo xtask package
```

This creates two files in the `target/` directory:

- `bootc-<version>.tar.zstd` - Source tarball with git archive contents
- `bootc-<version>-vendor.tar.zstd` - Vendored Rust dependencies

The source tarball includes a `.cargo/vendor-config.toml` file that configures cargo to use the vendored dependencies.

Using Vendored Dependencies in Builds

When building with vendored dependencies:

1. Extract both tarballs into your build directory
2. Extract the vendor tarball to create a `vendor/` directory
3. Ensure `.cargo/vendor-config.toml` is in place (included in source tarball)
4. Build normally with `make all`

The cargo build will automatically use the vendored crates instead of fetching from crates.io.

Version Management

The version is derived from git tags. The `cargo xtask package` command automatically determines the version:

- If the current commit has a tag: uses the tag (e.g., `v1.0.0` becomes `1.0.0`)
- Otherwise: generates a timestamp-based version with commit hash (e.g., `202501181430.g1234567890`)

This ensures that development snapshots have monotonically increasing version numbers.

Cargo Features

The build respects the `CARGO_FEATURES` environment variable. By default, the Makefile

auto-detects whether to enable the `rhsm` (Red Hat Subscription Manager) feature based on the build environment's `/usr/lib/os-release`.

To explicitly control features:

```
make all CARGO_FEATURES="rhsm"
```

Integration Testing

For distributions that want to include integration tests, use:

```
make install-all DESTDIR=/tmp/stage
```

This installs:

- Everything from `make install`
- Everything from `make install-ostree-hooks`
- The integration test binary as `bootc-integration-tests`

Base image content

Alongside building the binary here, you may also want to prepare a base image. For that, see bootc-images.

Additional Resources

- See `Makefile` for all available targets and variables
- See `crates/xtask/src/xtask.rs` for cargo xtask implementation details
- See `contrib/packaging/bootc.spec` for an example RPM spec file that uses all of the above.

Package manager integration

A toplevel goal of bootc is to encourage a default model where Linux systems are built and delivered as (container) images. In this model, the default usage of package managers such as `apt` and `dnf` will be at container build time.

However, one may end up shipping the package manager tooling onto the end system. In some cases this may be desirable even, to allow workflows with transient overlays using e.g. `bootc usroverlay`.

Detecting image-based systems

bootc is not the only image based system; there are many. A common emphasis is on having the operating system content in `/usr`, and for that filesystem to be mounted read-only at runtime.

A first recommendation here is that package managers should detect if `/usr` is read-only, and provide a useful error message referring users to documentation guidance.

An example of a non-bootc case is "Live CD" environments, where the *physical media* is readonly. Some Live operating system environments end up mounting a transient writable overlay (whether via e.g. devicemapper or overlayfs) that make the system appear writable, but it's arguably clearer not to do so by default. Detecting `/usr` as read-only here and providing the same information would make sense.

To specifically detect if bootc is in use, you can parse its JSON status (if the binary is present) to tell if a system is tracking an image. The following command succeeds if an image is *not* being tracked: `test $(bootc status --format=json | jq .spec.image) = null`.

The `/run/ostree-booted` file

This is created by ostree, and hence created by bootc (with the ostree) backend. You can use it to detect ostree. However, *most* cases should instead detect via one of the recommendations above.

Running a read-only system via podman/docker

The historical default for docker (inherited into podman) is that the `/` is a writable (but transient) overlayfs. However, e.g. `podman` supports a `--read-only` flag, and Kubernetes pods offer a `securityContext.readOnlyRootFilesystem` flag.

Running containers in production in this way is a good idea, for exactly the same reasons that bootc defaults to mounting the system read-only.

Ensure that your package manager offers a useful error message in this mode. Today for example:

```
$ podman run --read-only --rm -ti debian apt update
Reading package lists ... Done
E: List directory /var/lib/apt/lists/partial is missing. - Acquire (30:
Read-only file system)
$ podman run --read-only --rm -ti quay.io/fedora/fedora:40 dnf -y
install strace
Config error: [Errno 30] Read-only file system: '/var/log/dnf.log': '/
var/log/dnf.log'
```

However note that both of these fail on `/var` being read-only; in a default bootc model, it won't be. A more accurate check is thus closer to:

```
$ podman run --read-only --rm -ti --tmpfs /var quay.io/fedora/fedora:40
dnf -y install strace
...
Error: Transaction test error:
  installing package strace-6.9-1.fc40.x86_64 needs 2MB more space on
the / filesystem


$ podman run --read-only --rm --tmpfs /var -ti debian /bin/sh -c 'apt
update && apt -y install strace'
...
dpkg: error processing archive /var/cache/apt/archives/
libunwind8_1.6.2-3_amd64.deb (--unpack):
 unable to clean up mess surrounding './usr/lib/x86_64-linux-gnu/
libunwind-coredump.so.0.0.0' before installing another version: Read-
only file system
```

These errors message are misleading and confusing for the user. A more useful error may look like e.g.:

```
$ podman run --read-only --rm --tmpfs /var -ti debian /bin/sh -c 'apt
update && apt -y install strace'
error: read-only /usr detected, refusing to operate. See `man apt-image-
based` for more information.
```

Transient overlays

Today there is a simple `bootc usroverlay` command that adds a transient writable overlayfs for `/usr`. This makes many package manager operations work; conceptually it is similar to the writable overlay that many "Live CDs" use. However, one cannot change the kernel this way for example.

An optional integration that package managers can do is to detect this transient overlay situation and inform the user that the changes will be ephemeral.

Persistent changes

A bootc system by default *does* have a writable, persistent data store that holds multiple container image versions (more in filesystem).

Systems such as rpm-ostree implement a "hybrid" mechanism where packages can be persistently layered and re-applied; the system effectively does a "local build", unioning the intermediate filesystems.

One aspect of how rpm-ostree implements this is by caching individual unpacked RPMs as ostree commits in the ostree repo.

This section will be expanded later; you may also be able to find more information in booting local builds.

Relationship with other projects

bootc is the key component in a broader mission of bootable containers. Here's its relationship to other moving parts.

Relationship with podman

It gets a bit confusing to talk about shipping bootable operating systems in container images. Again, to be clear: we are reusing container images as:

- A build mechanism (including running *as* a standard OCI container image)
- A transport mechanism

But, actually when a bootc container is booted, podman (or docker, etc.) is not involved. The storage used for the operating system content is distinct from `/var/lib/containers`. `podman image prune --all` will not delete your operating system.

That said, a toplevel goal of bootc is alignment with the https://github.com/containers ecosystem, which includes podman. But more specifically at a technical level, today bootc uses skopeo and hence indirectly containers/image as a way to fetch container images.

This means that bootc automatically also honors many of the knobs available in `/etc/containers` - specifically things like containers-registries.conf.

In other words, if you configure `podman` to pull images from your local mirror registry, then `bootc` will automatically honor that as well.

The simple way to say it is: A goal of `bootc` is to be the bootable-container analogue for `podman`, which runs application containers. Everywhere one might run `podman`, one could also consider using `bootc`.

Relationship with Image Builder (osbuild)

There is a new bootc-image-builder project that is dedicated to the intersection of these two!

Relationship with Kubernetes

Just as `podman` does not depend on a Kubernetes API server, `bootc` will also not depend on one.

However, there are also plans for `bootc` to also understand Kubernetes API types. See configmap/secret support for example.

Perhaps in the future we may actually support some kind of `Pod` analogue for representing the host state. Or we may define a CRD which can be used inside and outside of Kubernetes.

Relationship with ostree

OSTree provides many things:

1. a git-like repo for OS data from which you can check out an entire rootfs
2. a bootloader integration layer
3. a transport layer for pulling content over HTTP

With bootc, the OSTree transport layer is not used. Instead, content is pulled as OCI containers using `skopeo` as mentioned above. However, this content is then imported into the local OSTree repo to perform a deployment checkout. The role of OSTree may further shrink in the future, especially as tighter integration with podman and composefs occurs, but it will remain an important part of the bootc stack (in particular the bootloader integration layer and management of deployment roots).

Relationship with rpm-ostree

As mentioned above, bootc uses OSTree as a backing model, and so does rpm-ostree. Hence, when using a container source, `rpm-ostree upgrade` and `bootc upgrade` are effectively equivalent; you can use either command.

Differences from rpm-ostree

- The ostree project never tried to have an opinionated "install" mechanism, but bootc does with `bootc install to-filesystem`
- Bootc has additional features such as `/usr/lib/bootc/kargs.d` and logically bound images.

Client side changes

Currently all functionality for client-side changes such as `rpm-ostree install` or `rpm-ostree initramfs --enable` continue to work, because of the shared base.

However, as soon as you mutate the system in this way, `bootc upgrade` will error out as it will not understand how to upgrade the system. The bootc project currently takes a relatively hard stance that system state should come from a container image.

The way kernel argument work also uses ostree on the backend in both cases, so using e.g. `rpm-ostree kargs` will also work on a system updating via bootc.

Overall, rpm-ostree is used in several important projects and will continue to be maintained for many years to come.

However, for use cases which want a "pure" image based model, using `bootc` will be more appealing. bootc also does not e.g. drag in dependencies on `libdnf` and the RPM stack.

bootc also has the benefit of starting as a pure Rust project; and while it doesn't have an IPC mechanism today, the surface of such an API will be significantly smaller.

Further, bootc does aim to include some of the functionality of zincati.

But all this said: *It will be supported to use both bootc and rpm-ostree together*; they are not exclusive. For example, `bootc status` at least will still function even if packages are layered.

Future bootc <-> podman binding

All the above said, it is likely that at some point bootc will switch to hard binding with podman. This will reduce the role of ostree, and hence break compatibility with rpm-ostree. When such work lands, we will still support at least a "one way" transition from an ostree backend. But once this happens there are no plans to teach rpm-ostree to use

podman too.

Relationship with Fedora CoreOS (and Silverblue, etc.)

Per above, it is a toplevel goal to support a seamless, transactional update from existing OSTree based systems, which includes these Fedora derivatives.

For Fedora CoreOS specifically, see this tracker issue.

See also OstreeNativeContainerStable.

How does the use of OCI artifacts intersect with this effort?

The "bootc compatible" images are OCI container images; they do not rely on the OCI artifact specification or OCI referrers API.

It is foreseeable that users will need to produce "traditional" disk images (i.e. raw disk images, qcow2 disk images, Amazon AMIs, etc.) from the "bootc compatible" container images using additional tools. Therefore, it is reasonable that some users may want to encapsulate those disk images as an OCI artifact for storage and distribution. However, it is not a goal to use `bootc` to produce these "traditional" disk images nor to facilitate the encapsulation of those disk images as OCI artifacts.

Relationship with systemd "particles"

There is an excellent vision blog entry that puts together a coherent picture for how a systemd (and uapi-group.org) oriented Linux based operating system can be put together, and the rationale for doing so.

The "bootc vision" aligns with parts of this, but differs in emphasis and also some important technical details - and some of the emphasis and details have high level ramifications. Simply stated: related but different.

System emphasis

The "particle" proposal mentions that the desktop case is most interesting; the bootc belief is that servers are equally important and interesting. In practice, this is not a real point of differentiation, because the systemd project has done an excellent job in catering to all use cases (desktop, embedded, server) etc.

An important aspect related to this is that the bootc project exists and must interact with many ecosystems, from "systemd-oriented Linux" to Android and Kubernetes. Hence, we would not explicitly compare with just ChromeOS, but also with e.g. Kairos and many others.

Design goals

Many of the toplevel design goals do overall align. It is clear that e.g. Discoverable Disk Images and OCI images align on managing systems in an image-oriented fashion.

A difference on goal 11

Goal 11 states:

> Things should not require explicit installation. i.e. every image should be a live image. For installation it should be sufficient to dd an OS image onto disk.

The `bootc install` approach is explicitly intending to support things such as e.g. static IP addresses provisioned via kernel arguments at install time; it is not a goal for installations to be equivalent to `dd`. The bootc creator has experience with systems that install this way, and it creates practical problems in nontrivial scenarios such as "Advanced Format" disk drives, etc.

New Goal: An explicit alignment with cloud-native

The bootc project has an explicit goal to to take formats, cues and inspiration from the container and cloud-native ecosystem. More on this in several sections below.

New Goal: Continued explicit support for "unlocked" systems

A strong emphasis of the particle approach is "sealed" systems that chain from Secure Boot. bootc aims to support the same. And in practice, nothing in "particles" strictly requires Secure Boot etc.

However, bootc has a stronger emphasis on continuing to support "unlocked" systems into the foreseeable future in which key (even root level) operating system changes can be that are outside of an explicit signed state and feel *equally* first class, not just "developer system extensions".

Or stated more simply, it will be explicitly supported to create bootc-based operating systems that boot as e.g. a cloud instance or as desktop machine that defaults to an unlocked state and provides good ergonomics in this scenario for managing user owned state across operating system upgrades too.

Hermetic `/usr`

One of the biggest differences starts with this. The idea of having the entire operating system self-contained in `/usr` is a good one. However, there is an immense amount of prior history and details that make this hard to support in many generalized cases.

This tracking issue is a good starting point - it's mostly about `/etc` (see below).

bootc design: Carve out sub mounts

Instead, the bootc model allows arbitrary directory roots starting from `/` to be included in the base operating system image.

This first notable difference is rooted in bootc taking a stronger cue from the opencontainers ecosystem (including docker/podman/Kubernetes). There are no restrictions on application container filesystem layout (everything is ephemeral by default, and persistence must be explicit); bootc aims to be closer to this.

There is still alignment: bootc design does *strongly encourage* operating system state to live underneath `/usr` - it should be the default place for all operating system executable binaries and default configuration. It should be read-only by default.

`/etc`

Today, the bootc project uses ostree as a backend, and a key semantic ostree provides for `/etc` is a "3 way merge".

This has several important differences. First, it means that `/etc` does get updated by default for unchanged configuration files.

The default proposal for "particle" OSes to deal with "legacy" config files in `/etc` is to copy them on first OS install (e.g. `/usr/share/factory`).

This creates serious problems for all the software (for example, OpenSSH) that put config files there; - having the default configuration updated (e.g. for a security issue) for a package manager but not an image based update is not viable.

However a key point of alignment between the two is that we still aim to have `/etc` exist and be useful! Writing files there, whether from `vi` or config management tooling must continue to work. Both bootc and systemd "particle" systems should still Feel Like Unix - in contrast to e.g. Android.

At the current time, this is implemented in ostree; as bootc moves towards stronger integration with podman, it is likely that this logic will simply be moved into bootc instead on top of podman. Alternatively perhaps, podman itself may grow some support for specifying this merge semantic for containers.

Other persistent state: `/var`

Supporting arbitrary toplevel files in `/` on operating system updates conflicts with a desire to have e.g. `/home` be persistent by default.

Hence, bootc emphasizes having e.g. `/home` → `/var/home` as a default symlink in base images.

Aside from `/home` and `/etc`, it is common on most Linux systems to have most persistent state under `/var`, so this is not a major point of difference otherwise.

Other toplevel files/directories

Even the operating systems have completed "UsrMerge" still have legacy compatibility symlinks required in `/`, e.g. `/bin` → `/usr/bin`. We still need to support shipping these for many cases, and they are an important part of operating system state. Having them not be explicitly managed by OS updates is hence suboptimal.

Related to this, bootc will continue to support operating systems that have not completed UsrMerge.

Discoverable Disk images and booting

The bootc project will not use Discoverable Disk Images. Instead, we orient as strongly around opencontainers/image-spec i.e. OCI/Docker images.

This is the biggest technical difference that strongly influences many other aspects of operating system design and experience.

It is an explicit goal of the bootc project that it should feel as natural as possible for someone familiar with "application containers" from podman/Docker/Kubernetes to take their tools and knowledge and apply that to the base operating system too.

Technical heart: composefs

There is a very strong security rationale behind much of the design proposal of "particles" and DDIs. It is absolutely true today, quoting the blog:

> That said, I think [OCI has] relatively weak properties, in particular when it comes to security, since immutability/measurements and similar are not provided. This means, unlike for system extensions and portable services a complete trust chain with attestation and per-app cryptographically protected data is much harder to implement sanely.

The composefs project aims to close this gap, and the bootc project will use it, and has an explicit goal to align with e.g. podman in using it too.

Effectively, everywhere one might use a DDI, bootc will usually support a container image. (However for some things like system configuration files, bootc may aim to instead support e.g. plain ConfigMap files which are signed for example).

System booting

The bootloader

The strong emphasis of the UAPI-group is on UEFI. However, the world is a bit broader than that; the bootc project also will explicitly continue to support:

- GNU Grub for multiple reasons; among them that unfortunately x86 BIOS systems will not disappear entirely in the next 10 years even.
- Android Boot - because some hardware manufacturers ship it, and we want to support operating systems that must work on this hardware.
- zipl because it's how things work on s390x, and there is significant alignment in terms of emphasizing a "unified kernel" style flow.

Boot loader configs

bootc aims to align with the idea of generic bootloader-independent config files where possible; today it uses ostree. For more on this, see ostree and bootloaders.

The kernel and initramfs

There is agreement that in order to achieve integrity, there must be a strong link between the kernel and the first userspace code that executes in the initial RAM disk.

Building on the bootloader statement above: bootc will support UKI, but not require it.

The root filesystem

In the bootc model, the root filesystem defaults to a single physical Linux filesystem (e.g. `xfs`, `ext4`, `btrfs` etc.). It is of course supported to mount other partitions and filesystems; doing so is encouraged even for `/var`., where one ends up with some space constraints around the OS `/usr` partition due to dm-verity.

This is a rather large difference already from particles; the root filesystem contains the operating system too; it is not a separate partition. One thing this helps significantly with is dealing with the "space management" problems that dm-verity introduces (need for a partition to have unused empty space to grow, and also a fixed-size ultimate capacity limit).

Locating the root

bootc does not mandate or emphasize any particular way to locate the root filesystem; parts of the discoverable partitions specification specifically the "root partition" may be used. Or, the root filesystem can be found the traditional way, via a local `root=` kernel argument.

Another point of contrast from the particle emphasis is that while we encourage encrypting the root filesystem, it is not required. Particularly some use cases in cloud environments perform encryption at the hypervisor level and do not want additional overhead of doing so per virtual machine.

Locating the base container image

Until this point, we have been operating under external constraints; no one is creating a bootloader that directly understands how to start a container image, for example. We've gotten as far as running a Linux userspace in the initial RAM disk, and the physical root filesystem is mounted.

Here, we circle back to composefs. One can think of composefs as effectively a way to

manage something like dm-verity, but using files.

What bootc builds on top of that is to target a specific container image rootfs that is part of the "physical" root. Today, this is implemented again using ostree, via the `ostree=` kernel commandline argument. In the future, it is likely to be a `bootc.image`. However, integration with other bootloaders (such as Android Boot) require us to interact with externally-specified fixed kernel arguments.

Ultimately, the initramfs will contain logic to find the desired root container, which again is just a set of files stored in the "physical" root filesystem.

Chaining integrity from the initramfs

One can think of composefs as effectively a way to manage something like dm-verity, but supporting multiple ones stored inside a standard Linux filesystem.

For "sealed" systems, the bootc project suggests a default model where there is an "ephemeral key" that binds the UKI (or equivalent) and the real root. For a bit more on this, see ostree and composefs. Effectively, at image build time an "ephemeral" key is generated which signs the composefs digest of the container image. The public half of this key is injected into the UKI, which is itself signed e.g. for Secure Boot.

At boot time, the initramfs will use its embedded public key to verify the composefs digest of the target root - and from there, overlayfs in the Linux kernel combined with fs-verity will continually verify the integrity of all operating system root files we use.

At the current time, there is not one single standardized approach for signing composefs images. Ultimately, a composefs image has a digest, and signing and verification of that digest can be done via any signing tool. For more on this, see this issue.

bootc itself will not mandate one mechanism currently. However, it is very likely that we will ship an optionally-enabled opinionated mechanism that uses basic ed25519 signatures for example.

This is effectively equivalent to the particle approach of embedding a verity root hash into the kernel commandline - it means that the booted Linux kernel will *only* be capable of mounting that one specific root filesystem. Note that this model is effectively the same as e.g. Fedora uses to sign kernel modules.

However, an "ephemeral key" is not the only valid way to do things; for some operating system creators it may be very desirable to continue to be able to make root OS image

changes without changing the UKI (and hence re-signing it). Instead, another valid approach is to simply maintain a persistent public/private keypair. This allows disconnecting the build of userspace and kernel, but also means that there is less strict verification between kernel and userspace (e.g. downgrade attacks become possible).

Chaining integrity to configuration and application containers

composefs is explicitly designed to be useful as a backend for "application" containers (e.g. podman). There is again not one single mechanism for signing and verification; in some use cases, it may be enough to boot the operating system enough to implement "network as source of truth" - for example, the public keys for verification of application containers might be fetched from a remote server. Then before any application containers are run, we dynamically fetch the relevant keys from a server which was trusted.

The bootc project will align with podman in general, and make it easy to implement a mechanism that chains keys stored alongside the operating system into composefs-signed application containers.

Configuration (effectively starting from `/etc` and the kernel commandline) in a "sealed" system is a complex topic. Many operating system builds will want to disable the default "etc merge" and make `/etc` always lifecycle bound with the OS: commonly writable but ephemeral.

This topic is covered more in the next section.

Modularity

A goal of "particles" is to add integrity into "general purpose" Linux OSes and distributions - supporting a world where there are a lot of users that simply directly install an OS from an upstream OS such as Debian or Fedora. This has a lot of implications; among them that e.g. the Secure Boot signatures etc. are made by the OS creator, not the user.

A big emphasis for the bootc project in contrast a design where it is normal and expected for many users to *derive* (via standard container build technology) from the base image produced by the OS upstream.

This is just a difference in emphasis: "particles" can clearly be built fully customized by

the end customer, and bootc fully supports booting "stock" images.

But still: the bootc project will again much more strongly push any scenario that desires truly strong integrity towards making and managing custom derived builds.

Extensions and security

In "unlocked" scenarios, the bootc project will continue to support a "traditional Unix" feeling where persistent changes to `/etc` can be written and maintained. Similarly, it will continue to be supported to have machine-local kernel arguments. There is significant value in migrating "package based" systems to "image based" systems, even if they are still "unsigned" or "unlocked".

The particle model calls for tools like confext that use DDIs. The "backend" of this (managing merged dynamic filesystem trees with overlayfs) and its relationship with systemd units is still relevant, but the bootc approach will again not expose DDIs to the user. Instead, our approach will take cues from the cloud-native world and use e.g. Kubernetes ConfigMap and support signatures on these.

More Modularity: Secondary OS installs

This uses OCI containers, which will work the same as the host.

Developer Mode

This topic heavily diverges between the "unlocked" and "sealed" cases. In the unlocked case, the bootc project aims to still continue to make it feel very "first class" to perform arbitrary machine-local mutations. Instead of managing overlay DDIs, `bootc` will make it trivial and obvious to use local container builds using any standard container build tooling.

Package managers

In order to ease the transition for users coming from package systems, the bootc project

suggests that package managers like `apt` and `dnf` etc. learn how to become a frontend for "local" container builds too. In other words, `apt|dnf install foo` would become shorthand for a container build like:

```
FROM <localhost>
RUN apt|dnf install foo
```

Transitioning from unlocked, mutable local state to server-built images

Building on the above, a key point of `bootc` is to make it easy and obvious how to go from an "unlocked" system with potential unmanaged state towards a system built and managed using standard OCI container image build systems and tooling. For example, there should be a command like `apt|dnf print-containerfile`. (The problem is more complex than this of course, as we would likely want to capture some changes from `/etc` - but also some of those changes may include secrets, which are their own sub-topic)

Democratizing Code Signing

Strong alignment here.

Running the OS itself in a container

This is equally obvious to do when the host and the linked container runtime (e.g. podman) again use the same tools.

Parameterizing Kernels

In "unlocked" scenarios (per above) we will continue to use bootloader configuration that is unsigned.

We will not (in contrast to particles) try to strongly support a "partially sealed, general purpose" model. More on this below.

Most cases for "sealed" systems will want to entirely lock the kernel commandline, not even using a bootloader at all and hence there is no mechanism to configure it locally at all. However, as discussed in various venues around UKI, "sealed" systems can become complex to deploy where there is a need for machine (or machine-type) specific kernel arguments:

- Deploying the RT kernel often wants to use isolcpus=.
- Setting static IP addresses on the kernel commandline to enable network bound disk encryption for the rootfs

The bootc project default approach for this is to lean into the container-native world, using derivation to create a machine-independent "base image", then create derived, machine (or machine-class) specific images that are in turn signed.

Updating Images

A big differentiation here is that bootc will reuse container technology for fetching updates. The operating system and application containers will be signed with e.g. sigstore or similar for network fetching. The signature will cover the composefs digest, which enables continuous verification.

Managing storage of container images using composefs is more complex than `systemd-sysupdate` writing to a partition, but significantly more flexible. For more on this, see upstream composefs.

Kernel in images

The bootc and particle approaches are aligned on storing the kernel binary in `/usr/lib/modules/$kver`. On the bootc side, a key bit here is that bootc will extract the kernel and initramfs (or just UKI) and put it in the appropriate place - this is implemented as a transactional operation. There are significant details that can vary for how this works (because unlike particles, bootc aims to support non-EFI setups as well), but the high level idea is similar.

Boot Counting + Assessment

This topic relates to the previous one; because of multiple bootloaders, there is not one single approach. The systemd automatic boot assessment is good where it can be used, but we also will support e.g. Android bootloaders.

Picking the Newest Version

Because the storage of images is not just files or partitions, bootc will not expose to the user/administrator a semantic of `strvercmp` or package-manager oriented versioning semantics. Instead, the implementation of "latest" will be implemented in a more Kubernetes-oriented fashion of having "local" API objects with spec and status. This makes it easy and obvious for higher level management (e.g. cluster) tooling to orchestrate updates in a Kubernetes-style fashion.

Home Directory Management

The bootc project will not do anything with this. We will support systemd-homed where users want it, but in many dedicated servers and managed devices the idea of persistent user "home directories" are more of an anti-pattern.

Partition Setup

The biggest difference again here is that bootc is oriented closer to a single root partition by default that includes the OS, system/app containers and persistent local state all as one unit.

Trust chain

In contrast to particles, the bootc project does not aim to by default emphasize a model of using sysexts from the initramfs because its primary use case occurs when using a "partially sealed" system. And per above (re kernels) it is insufficient for other cases.

Without this in the mix then, the trust chain is simple to describe: the kernel+initramfs

are verified by the bootloader, the initramfs contains the key and logic necessary to verify the composefs digest of the root, and the root starts to verify everything else.

File System Choice

As mentioned above, any Linux filesystem is valid for the root. For "sealed" systems using composefs will cover integrity and there is not a distinct need for dm-integrity.

OS Installation vs. OS Instantiation

The bootc project is just less partition-oriented and more towards multiple-composefs-in-root oriented. However the high level goal is shared of making it easy to "re-provision" and keeping the install-time flow as close as possible.

Building Images According to this Model

This is a key point of bootc: we aim for operating systems and distributions to ship their own bootc-compatible base images that can be used as a default derivation source. These images are just OCI images that will follow simple rules (as mentioned above, the kernel is found in `/usr/lib/modules/$kver/vmlinuz`) for example for the extra state to boot.

However in order to enable "sealed" systems (using signed composefs digests), the container build system will need support for this. But, it is a goal to standardize the composefs metadata needed alongside the OCI, and to support this in the broader container ecosystem of tools (e.g. docker, podman) as well as bootc.

Final words

This document is obviously very heavily inspired by the original blog.

A point of divergence is that a goal of the bootc project *is* to strongly influence the

existing operating systems and distributions and help them migrate their customers into an image-based world - and to make practical compromises in order to aid that goal.

But, the bootc project strongly agrees with the idea of finding common ground (the "50% shared" case). At a practical level, this project will take a hard dependency on systemd *and* on the container ecosystem, extending bridges where they exist, working on shared standards and approaches between the two.

Internals (rustdoc)

This section provides rustdoc API documentation for bootc's internal crates. These are intended for developers working on bootc itself, not for external consumption.

Core crates

- bootc-lib - Core bootc implementation
- bootc - CLI frontend

Supporting crates

- ostree-ext - Extension APIs for OSTree
- bootc-mount - Internal mount utilities
- bootc-kernel-cmdline - Kernel command line parsing
- bootc-initramfs-setup - Initramfs setup code
- etc-merge - /etc merge handling

Utility crates

- bootc-internal-utils - Internal utilities
- bootc-internal-blockdev - Block device handling
- bootc-sysusers - systemd-sysusers implementation
- bootc-tmpfiles - systemd-tmpfiles implementation

External git crates

These crates are pulled from git and are not published to crates.io (so not on docs.rs).

- composefs - Core composefs library
- composefs-boot - Boot support for composefs

- composefs-oci - OCI integration for composefs
