draupnir — Manual

version 0.1.0

generated 2026-07-11

---

rendered by **nornir** · typst engine

Contents

1 Design 3
2 draupnir — design 4
2.1 Charter 4
2.2 Ownership — one home per capability (the no-duplication rule) 4
2.3 Types & traits (the real core) 5
2.4 Dependencies (reuse, don't reinvent — Brokkr's minimal-dependency law) 5
2.5 Extraction plan — moving instance-firing out of Skidbladnir (staged) 5
2.6 Why a separate repo 6
2.7 Docs 6
3 draupnir — guide 7
3.1 Add it 7
3.2 Describe an instance 7
3.3 Fire it up 7
3.4 Boot a fleet from one ISO (the ring) 8
3.5 Where it sits 8

Design

draupnir — design

Draupnir is the low-level boot / provisioning library of the nordisk constellation. It fires up a runtime instance from one [`BootSpec`] across three backends and drives its power lifecycle. Named for **Draupnir**, Odin's gold ring that drips eight identical copies of itself every ninth night — here a *light nod*: booting a **fleet of identical machines from one ISO** is the natural extension, not the core. The core is **one library, three boot backends**.

Charter

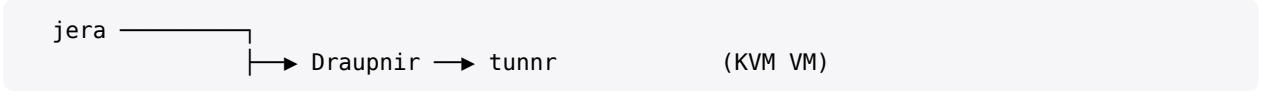
- **One `Boot` trait, three backends** — Draupnir's own code is the thin unifying seam; each backend *delegates* to an existing engine (Brokkr's minimal-dependency law: reuse, don't reinvent).
  - **KVM** → drives tunnr's `tunnr_vm::boot_test(BootSpec) -> BootHandle` primitive. Draupnir does **not** reimplement VM boot.
  - **Container** → an OCI runtime crate (podman/Docker REST).
  - **Redfish** → a BMC's out-of-band Redfish REST API to boot **bare metal**: insert a virtual-media ISO, set the one-time boot override, power on. This is the capability Draupnir uniquely owns.
- **Honest stubs.** The trait wiring + the pure bookkeeping (spec validation, the fleet fan-out) are real and unit-tested; every unimplemented backend internal is an honest `todo!("draupnir: …")` — never a fake-green stub or a sentinel string.
- **Lean by default.** A plain `cargo build` is **pure std, zero dependencies**. Each live backend sits behind a feature; a consumer pulls only what it drives.

Ownership — one home per capability (the no-duplication rule)

Draupnir is the **shared low-level boot lib**. Two high-level consumers depend on it **directly**; neither re-implements boot code. Draupnir knows nothing about jobs or services — it only fires up and controls instances.

Component	Layer	Owns	Depends on
Draupnir	low-level boot lib	the `Boot` / `Lifecycle` / `VirtualMedia` traits, `BootSpec` / `Machine` types, spec validation + fleet fan-out, and the **three backend adapters** (kvm/container/redfish)	tunnr (KVM); an OCI runtime crate; a Redfish client crate
tunnr	KVM boot backend	the QEMU + OVMF (UEFI) + KVM boot primitive (`BootSpec` / `boot_test` / `BootHandle`)	— (Draupnir's KVM backend drives it)
jera (edda job handler)	thin job policy	*job* semantics only — a job is `process | VM | container` ; the durable job ledger	**Draupnir** (directly, for VM/container job instances)
Skidbladnir	high-level service/airgap layer	service/systemd lifecycle, airgap pack/move/unfold, orchestration	**Draupnir** (directly, for service instances)

Call direction — two independent consumers, one shared engine:



```
Skidbladnir ──┘              ├─▶ OCI runtime      (container / redis service)
                             └─▶ Redfish BMC      (bare-metal virtual-media)
```

No-duplication rule. Instance *booting* lives in exactly one place: Draupnir. jera does not shell out to QEMU; Skidbladnir does not carry its own KVM/container drivers; neither reimplements the other's boot path. tunnr is the sole KVM boot primitive, reached only *through* Draupnir's KVM backend.

> **This revises the earlier "Skidbladnir is the KVM/container mechanism master" decision.** Skidbladnir now **delegates instance-firing to Draupnir** and becomes a *consumer* of it (service/systemd, airgap, orchestration stay in Skidbladnir). jera likewise depends on Draupnir directly rather than re-rolling VM/container boot.

Types & traits (the real core)

- `enum Backend { Kvm, Container, Redfish }` — selects the driver.
- `enum ImageSource { KernelRootfs{kernel,rootfs}, Disk, OciImage, Iso }` — the bootable payload; `suits(Backend)` gates which backend can boot which source.
- `struct BootSpec { name, backend, image, mem_mb, cores, cmdline, env, bmc }` — one self-contained request; `validate()` rejects mismatches (ISO to KVM; a Redfish spec with no BMC; a non-Redfish spec carrying a BMC) **before** any backend is touched.
- `struct BmcEndpoint { host, username, system_id }` — the out-of-band Redfish endpoint. **No credential field** — the secret is supplied at drive time, so a spec never carries one.
- `trait Boot { fn boot(&self, &BootSpec) -> Result<Machine>; }`
- `trait Lifecycle { power_on / power_off / status }`
- `trait VirtualMedia { insert_media / eject_media / set_boot_override }` — Redfish only.
- `fn plan_fleet(&BootSpec, n) -> Vec<BootSpec>` — the ring drips N identically- payloaded, distinctly-named specs (fleet-from-one-ISO).

Dependencies (reuse, don't reinvent — Brokkr's minimal-dependency law)

Draupnir's own code is the thin `Boot` glue; each backend delegates to an existing crate. Chosen (and why):

- **`tunnr-vm`** (path/optional, feature `backend-tunnr`) — the KVM boot backend *is* tunnr's primitive; wiring it as an optional path dep (interim until tunnr publishes) is exactly how `jera` does it, so there is one KVM boot path in the constellation, not two.
- **OCI runtime** — **`bollard`** (planned, feature `backend-oci`) — the async podman/Docker REST client `jera` already uses for its zero-shell container path; reusing it keeps a single container engine rather than a second bespoke one.
- **Redfish** — **`redfish-codegen`** (a generated, typed Rust Redfish client), or a thin **`reqwest`** wrapper over the handful of endpoints we need (`Managers/{id}/VirtualMedia` insert/eject, `Systems/{id}` `Boot` override, `ComputerSystem.Reset`) if the generated client's surface is heavier than the three actions warrant (planned, feature `backend-redfish`). Either way Draupnir does not hand-roll the Redfish protocol.
- **znippy** (nordisk, if/when an image/ISO content-cache is needed) — reuse Skidbladnir's content-addressed archive rather than a new one.

The network-touching backends (OCI, Redfish) sit behind OFF-by-default features so the default build stays pure-std and offline-green; only their justified crates are pulled when a consumer enables them.

Extraction plan — moving instance-firing out of Skidbladnir (staged)

Instance-firing code currently lives in **`Skidbladnir/src/machine.rs`** (behind its `machine` / `kvm` / `container` features): the `MachineController` trait with `kvm::KvmController` (qemu/KVM + cloud-init NoCloud seed) and `container::ContainerController` (podman/docker), plus `MachineSpec` / `Handle` / `ImageCache`. That is exactly Draupnir's remit. **Do not rip it now** — the scaffold is created first; the move is staged:

1. **Land Draupnir** (this repo) as the `Boot` / `Lifecycle` seam + backend adapters. *(done — this scaffold.)*
2. **Wire the KVM backend** to `tunnr_vm::boot_test` under `backend-tunnr` (signature already mapped in `src/kvm.rs`); port Skidbladnir's cloud-init NoCloud seed authoring into the KVM adapter.
3. **Wire the container backend** to `bollard` under `backend-oci`; fold in `ContainerController`'s start/stop/exec/wait_ready readiness model.
4. **Add the Redfish backend** (`backend-redfish`) — net-new capability, no Skidbladnir source to move.
5. **Repoint Skidbladnir** (and jera) at Draupnir: delete `machine.rs`'s duplicate drivers, keep only the high-level service/airgap logic that *calls* Draupnir. Skidbladnir's `ImageCache` either moves to Draupnir or is replaced by the znippy-backed image store.
6. **Repoint jera**'s `vm` / `container` seams at Draupnir instead of its own `tunnr-vm` path dep, so jera keeps only job policy.

Steps 5–6 are the actual code removal and happen only after Draupnir's backends are real (green under their features) — never a half-landed swap.

Why a separate repo

Booting/provisioning is a distinct, versionable capability with two independent consumers (jera, Skidbladnir). Its own leaf workspace lets both depend on one lean crate without dragging each other's dep trees, and lets the boot engine evolve on its own cadence — every backend a unit-testable Rust seam, not an opaque script.

Docs

This `.nornir/` is the **source** of the repo's docs (crisp holger-pattern): `design.md` (this file) · `guide.md` (usage) · `for-idiots.md` (the beginner walk-through).

draupnir — guide

Draupnir fires up a runtime instance from one `BootSpec` across three backends — KVM (via tunnr), OCI container, and Redfish bare-metal — and drives its power lifecycle. This is the usage walk-through; the architecture is in `design.md`.

Add it

```
[dependencies]
draupnir = { version = "0.1" }                # pure-std default: types + traits + validation
# draupnir = { version = "0.1", features = ["backend-tunnr"] }   # live KVM boot via tunnr
```

The default build has **no dependencies**. Each live backend is a feature:

feature	backend	delegates to
(default)	—	the trait surface + `BootSpec` / `Machine` + validation + `plan_fleet`
`backend-tunnr`	KVM VM	`tunnr_vm::boot_test`
`backend-oci` *(planned)*	OCI container	`bollard`
`backend-redfish` *(planned)*	bare metal	Redfish REST client

Describe an instance

```
use draupnir::{BootSpec, BmcEndpoint};

// A KVM appliance (kernel + rootfs) — booted through tunnr.
let vm = BootSpec::kvm_kernel_rootfs("wg-appliance", "/boot/bzImage", "/img/rootfs.cpio.gz");

// A redis service instance — an OCI container.
let redis = BootSpec::container("cache", "docker.io/library/redis:7")
    .with_env("REDIS_ARGS", "--save 60 1000");

// A bare-metal node — a Redfish virtual-media ISO boot.
let node = BootSpec::redfish_iso(
    "node-42",
    "/images/rhel-9.iso",
    BmcEndpoint {
        host: "https://bmc-42.dc.example".into(),
        username: "admin".into(),
        system_id: "System.Embedded.1".into(),
    },
);

// Reject an inconsistent spec before touching a backend.
vm.validate().unwrap();
node.validate().unwrap();
```

Fire it up

```
use draupnir::{Boot, kvm::KvmBoot, container::ContainerBoot, redfish::RedfishBoot};
```

```
let machine = KvmBoot::new().boot(&vm)?;          // feature backend-tunnr
let cache   = ContainerBoot::new().boot(&redis)?; // feature backend-oci
let bare    = RedfishBoot::new().boot(&node)?;    // feature backend-redfish
```

Each `boot()` returns a `Machine { id, spec_name, backend, power }`. Drive it with the `Lifecycle` trait (`power_on` / `power_off` / `status`). For Redfish, the `VirtualMedia` trait exposes the raw steps (`insert_media` / `set_boot_override` / `eject_media`) that `boot()` sequences for you.

Without its feature, a backend's `boot()` returns `Err(Error::Unsupported(..))` telling you which feature to enable — it never pretends to boot.

Boot a fleet from one ISO (the ring)

```
use draupnir::plan_fleet;

// One ISO -> eight identically-payloaded, distinctly-named specs (node-1 … node-8).
for member in plan_fleet(&node, 8) {
    RedfishBoot::new().boot(&member)?;
}
```

Where it sits

jera (job handler) and Skidbladnir (service/airgap) both depend on Draupnir directly and call down into it — neither reimplements boot. Draupnir drives tunnr for the KVM backend. See `design.md` for the ownership table.