DNS Blocklist Manager

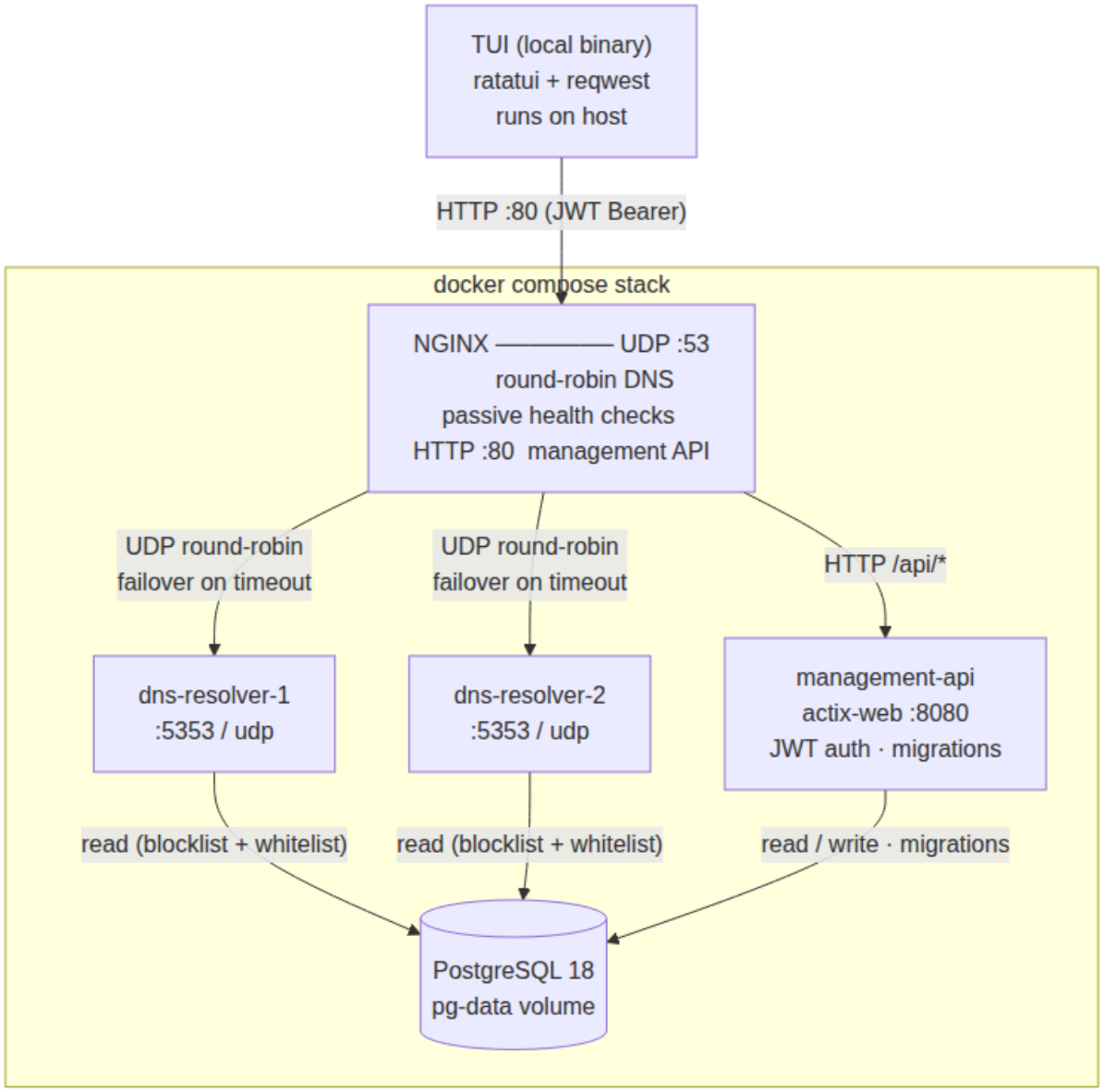
1 DNS Blocklist Manager

A distributed DNS filtering system built in Rust. DNS queries are load-balanced across multiple resolver instances. A REST management API handles configuration and statistics. A Ratatui TUI is the user-facing control plane.

Two deployment modes are supported: **Docker Compose** (local dev / production) and **Kubernetes** (production, Cilium L2 load balancing). Both use PostgreSQL.

1.1 Architecture

1.1.1 Docker Compose



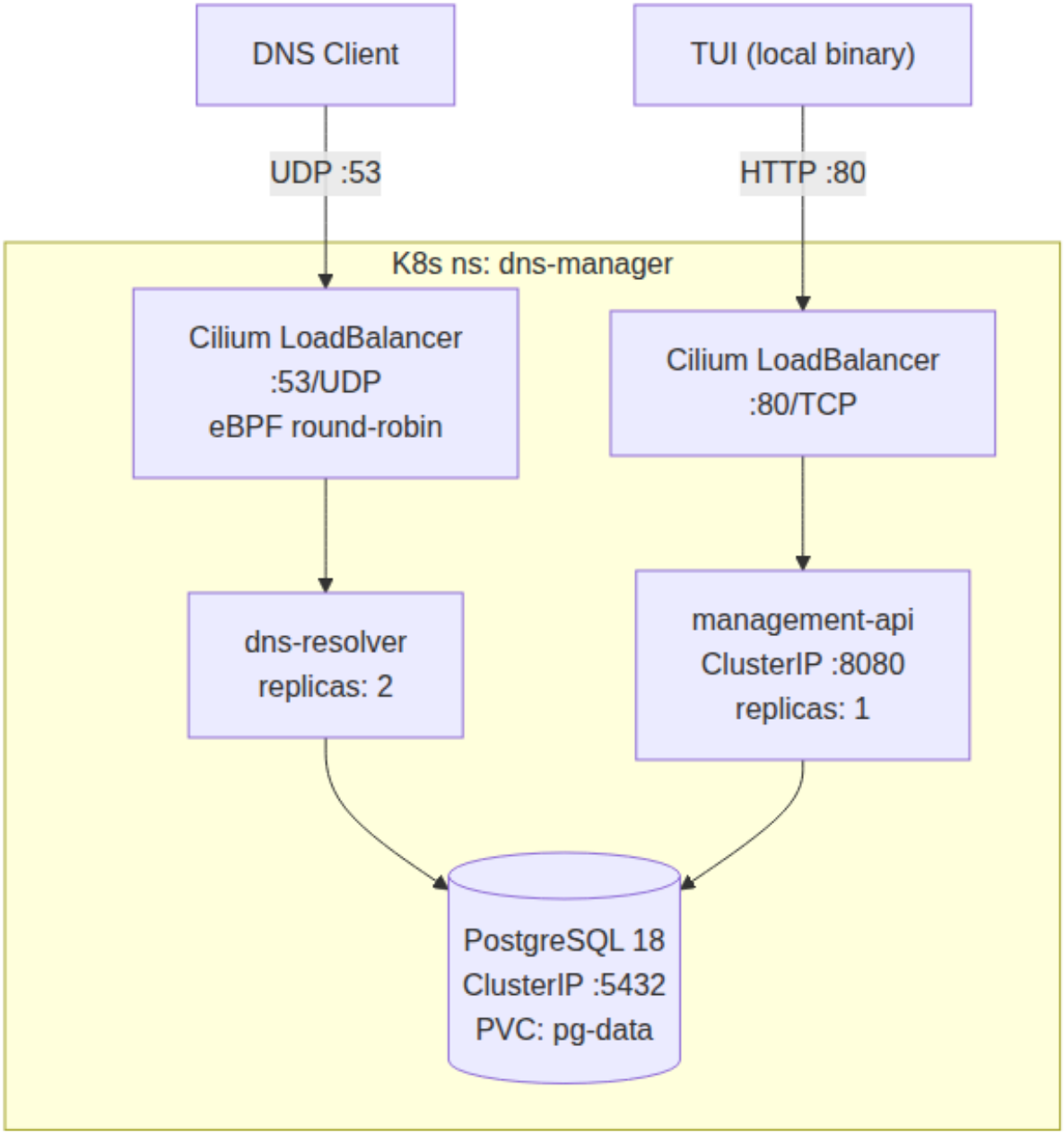
The Compose stack ships two environment profiles:

File	DNS port mapping	Use case
`compose.override.yml`	host `:5354` → NGINX `:53/udp`	**Dev** — avoids conflict with `systemd-resolved` on port 53
`compose.prod.yml`	host `:53` → NGINX `:53/udp`	**Production** — standard DNS port

Dev mode is the default (`docker compose up` auto-merges `compose.override.yml`). This seperation needs to be done, because on many systems you are not allowed to map to 53, since the system own DNS service is running on it and we don't want to breake that. For production, use:

```
docker compose -f compose.yaml -f compose.prod.yml up -d
```

1.1.2 Kubernetes



To deploy on Kubernetes, after changing the secrets:

```
kubectl apply -k ./k8s/
```

1.2 Quick Start

1.2.1 Docker

1.2.1.1 Docker Compose (development)

```
# 1. Configure
cp .env.example .env
# Edit .env: set JWT_SECRET (min 32 chars)

# 2. Start all services (dev mode — DNS on port 5354)
docker compose up -d

# 3. Verify DNS works
dig @localhost -p 5354 google.com
dig @localhost -p 5354 doubleclick.net   # → 0.0.0.0 if a blocklist is active

# 4. Start the TUI
API_URL=http://localhost:80 cargo run --manifest-path tui/Cargo.toml

# 5. In the TUI: press F5 to register, then L → select a blocklist → Enter
```

1.2.1.2 Docker Compose (production)

```
# DNS binds to port 53 — requires no other process on that port
docker compose -f compose.yaml -f compose.prod.yml up -d

dig @localhost google.com
```

1.2.2 Kubernetes

```
# 1. Edit secrets and Cilium IP range
vim k8s/secret.yaml       # set JWT_SECRET, POSTGRES_PASSWORD, DATABASE_URL
vim k8s/cilium-lb.yaml    # set CIDR to an available IP range on your network

# 2. Deploy
kubectl apply -k ./k8s/

# 3. Watch pods come up
kubectl get pods -n dns-manager -w

# 4. Find the external IPs assigned by Cilium
kubectl get svc -n dns-manager

# 5. Test DNS (replace <EXTERNAL-IP> with the dns-resolver LoadBalancer IP)
dig @<EXTERNAL-IP> google.com

# 6. Point the TUI at the management API LoadBalancer IP
API_URL=http://<EXTERNAL-IP> cargo run --manifest-path tui/Cargo.toml
```

1.3 Services

1.3.1 Docker Compose Stack

Service	Image	Port
`postgres`	`postgres:18-alpine`	internal `:5432`
`nginx`	`nginx:alpine` (custom build)	`:5354/udp` DNS (dev), `:53/udp` DNS (prod), `:80` HTTP
`management-api`	`$DOCKERHUB_USERNAME/management-api:latest`	internal `:8080`
`dns-resolver-1`	`$DOCKERHUB_USERNAME/dns-resolver:latest`	internal `:5353/udp`
`dns-resolver-2`	`$DOCKERHUB_USERNAME/dns-resolver:latest`	internal `:5353/udp`

Network isolation: `backend-net` is marked `internal: true` — PostgreSQL and the data plane have no direct internet access. Only `frontend-net` (NGINX, API, resolvers) can reach the outside.

1.3.2 Kubernetes Stack

Service	Image	Port
`postgres`	`postgres:18-alpine`	internal `:5432`
`management-api`	`ghcr.io/linxli/management-api:latest`	internal `:8080`, external `:80`
`dns-resolver` (x2)	`ghcr.io/linxli/dns-resolver:latest`	internal `:5353/udp`, external `:53/udp`

All Rust service containers use a two-stage build: a `rustlang/rust:nightly-slim` builder and a `gcr.io/distroless/cc-debian12` runtime (20 MB, no shell, no package manager). Both `management-api` and `dns-resolver` images include a tiny native HTTP healthcheck binary compiled in the builder — no `curl` needed at runtime.

1.4 Kubernetes Deployment

1.4.1 Prerequisites

- Kubernetes cluster with Cilium (for L2 `LoadBalancer` services)
- `kubectl` with Kustomize support (v1.14+)

1.4.2 Apply

```
kubectl apply -k ./k8s/
```

1.4.3 Configuration

Edit k8s/secret.yaml before applying:

```
stringData:
  JWT_SECRET: "change-me-to-at-least-32-random-chars!!"
  POSTGRES_PASSWORD: "change-me-generate-with-openssl-rand"
  DATABASE_URL: "postgres://postgres:change-me-generate-with-openssl-rand@
postgres.dns-manager.svc.cluster.local:5432/dns_manager"
```

Edit k8s/cilium-lb.yaml to match an available IP range on your local network:

```
blocks:
  - cidr: "192.168.1.240/28"  # replace with your range
```

1.4.4 K8s Resources

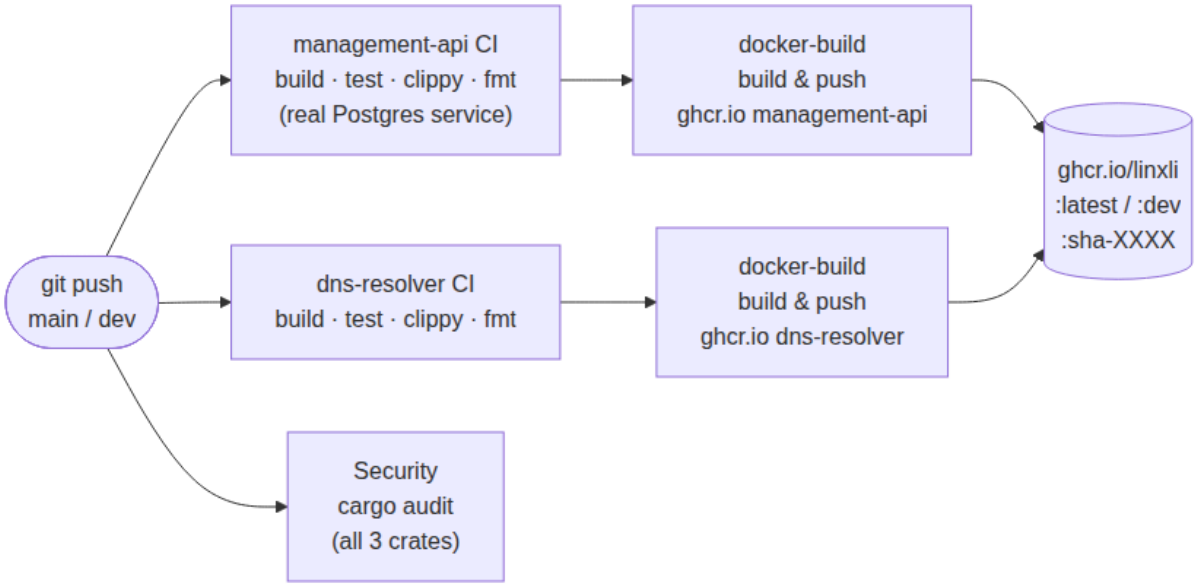
Resource	Kind	Details
`dns-manager`	Namespace	Isolates all workloads
`dns-manager-secrets`	Secret	`JWT_SECRET`, `POSTGRES_PASSWORD`, `DATABASE_URL`
`dns-manager-config`	ConfigMap	All non-secret env vars
`pg-data`	PersistentVolumeClaim	Postgres data directory
`postgres`	Deployment	`postgres:18-alpine`, replicas: 1
`management-api`	Deployment	`ghcr.io/linxli/management-api`, replicas: 1
`dns-resolver`	Deployment	`ghcr.io/linxli/dns-resolver`, replicas: 2
`dns-resolver`	Service (LoadBalancer)	UDP :53 → pod :5353, Cilium eBPF round-robin
`dns-manager-http`	Service (LoadBalancer)	TCP :80 → pod :8080
`management-api`	Service (ClusterIP)	Internal :8080
`postgres`	Service (ClusterIP)	Internal :5432
`dns-manager-pool`	CiliumLoadBalancerIPPool	IP range for LB services
`dns-manager-l2-policy`	CiliumL2AnnouncementPolicy	L2 ARP announcement

All pods run with a hardened `securityContext`: `runAsNonRoot`, `allowPrivilegeEscalation: false`, `capabilities.drop: ALL`, `seccompProfile: RuntimeDefault`.

`dns-resolver` pods use an init container (`wait-for-api`) that polls the management API health endpoint before the resolver starts, guaranteeing migrations have run before any resolver connects to PostgreSQL.

1.5 CI / CD

Each crate has its own GitHub Actions workflow. Docker images are only pushed if tests and linting pass.



Workflow	File	What it does
management-api CI	`.github/workflows/management-api.yml`	Test (with Postgres service) → clippy/fmt → build & push to ghcr.io
dns-resolver CI	`.github/workflows/dns-resolver.yml`	Test → clippy/fmt → build & push to ghcr.io
Security	`.github/workflows/security.yml`	`cargo audit` for each crate (dns-resolver, management-api, tui)

Images are published to `ghcr.io/linxli/dns-resolver` and `ghcr.io/linxli/management-api`.

1.6 Crate Overview

1.6.1 `dns-resolver/`

Pure DNS server. No HTTP server (except a health endpoint on port 9090).

On startup, the resolver reads its configuration from environment variables, opens a PostgreSQL connection pool, loads the initial blocklist, and begins listening for UDP DNS packets. Incoming queries are checked against an in-memory blocklist (backed by two thread-safe `HashSet` globals for the effective set and the whitelist), which is automatically reloaded every 30 seconds. Allowed domains are resolved asynchronously via `tokio::net::lookup_host`. Query statistics are tracked with atomic counters and flushed to the database every 5 seconds.

Effective block set (recomputed on every reload):

```
effective = (blocklist_entries WHERE is_active=1  ∪  blacklist) \ whitelist
```

Subdomain matching: a query for `ads.example.com` also checks `example.com`.

Dependencies: `tokio`, `hickory-proto`, `hickory-resolver`, `sqlx`

1.6.2 `management-api/`

JWT-authenticated REST API. Owns all database writes and runs migrations.

The API reads its configuration (JWT secret, DB URL, listen address) from environment variables and manages a PostgreSQL connection pool with automatic migrations. Authentication uses HS256 JWTs via the `jsonwebtoken` crate, with an Actix-web extractor that validates `Authorization: Bearer` headers. User registration and login endpoints hash passwords with argon2. The core functionality exposes paginated CRUD endpoints for whitelist and blacklist management, blocklist source activation (which downloads and bulk-inserts entries), and a filtered view of the current effective block set. A stats endpoint aggregates recent query data and top blocked domains, and a health endpoint serves as a liveness probe. Integration tests cover JWT handling, auth flows, all CRUD operations, and stats.

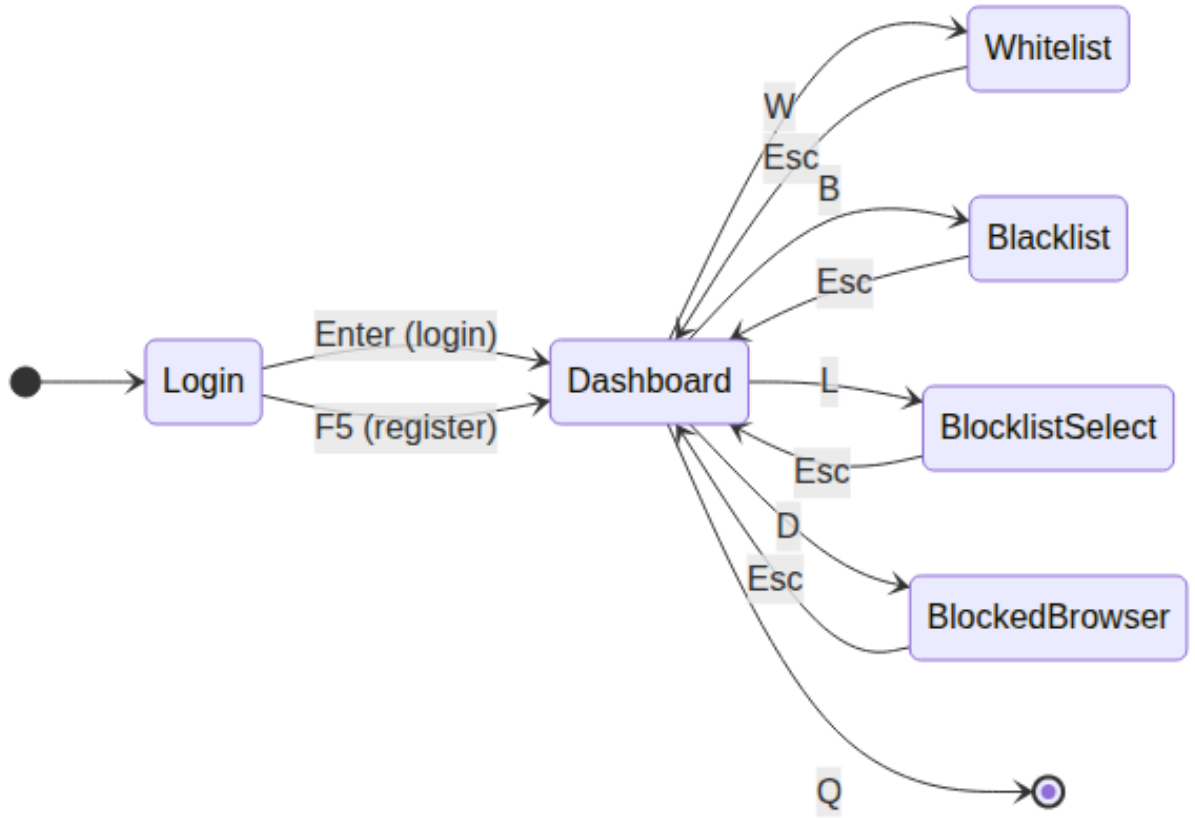
Dependencies: `actix-web`, `sqlx`, `jsonwebtoken`, `argon2`, `reqwest`, `uuid`, `serde`

1.6.3 `tui/`

Local Ratatui TUI. Connects to the management API over HTTP. Has its own Dockerfile but is not part of `docker compose up` — it requires a real TTY and is typically run locally with `cargo run`.

The TUI initializes the terminal in raw mode with an alternate screen and runs a 250 ms poll loop for keyboard input and auto-refresh ticks. All API communication goes through `reqwest`, with response types mirroring the management API. The `App` struct holds the full UI state across all screens (login, dashboard, whitelist, blacklist, blocklist selector, blocked-domain browser), with enums controlling which screen is active and whether the user is filtering or adding entries. Each screen has a dedicated renderer: a centered login dialog with masked password input, a dashboard with stats bar and top-blocked-domains table, a shared list view for both whitelist and blacklist (supporting filter, add, and delete), a blocklist selector where Enter downloads and activates, and a blocked-domain browser where W whitelists the selected domain.

Screens and navigation:



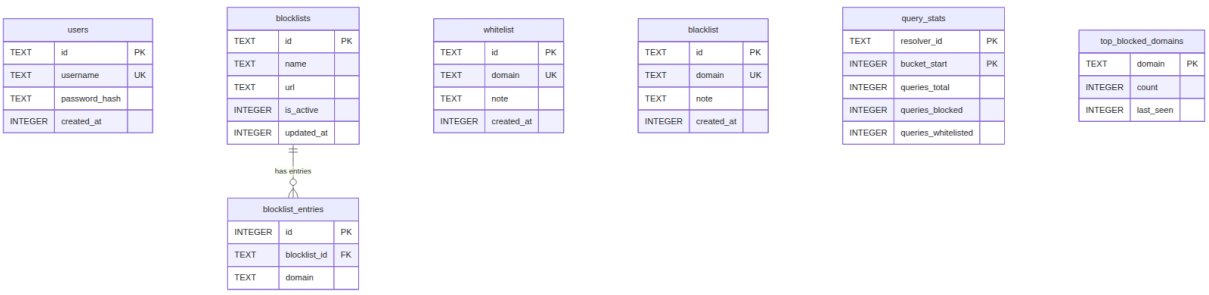
Dashboard auto-refreshes stats every 10 s on poll timeout.

Key bindings (full help is shown in the TUI):

Key	Action
`F5`	Register new account (login screen)
`A`	Add domain
`D` / `Delete`	Delete selected entry
`W`	Whitelist domain (blocked-domain browser)
`/`	Live filter
`Esc`	Back / clear

Dependencies: `ratatui`, `crossterm`, `reqwest`, `tokio`, `serde`, `serde_json`, `dotenvy`

1.7 Database Schema



Migrations live in `management-api/migrations/` and are embedded at compile time via `sqlx::migrate!`, running automatically at `management-api` startup.

Both Docker Compose and Kubernetes use PostgreSQL 18 (Alpine).

1.8 REST API Reference

All endpoints except `/api/auth/*` and `/api/health` require:

```
Authorization: Bearer <token>
```

Method	Path	Description
`POST`	`/api/auth/register`	Register (username >= 3 chars, password >= 8 chars); returns JWT
`POST`	`/api/auth/login`	Login; returns JWT
`GET`	`/api/health`	Liveness probe
`GET`	`/api/stats`	5-min aggregate stats + resolver breakdown
`GET`	`/api/stats/top-domains`	Top blocked domains by count
`GET/POST/DELETE`	`/api/whitelist[/:id]`	Paginated CRUD
`GET/POST/DELETE`	`/api/blacklist[/:id]`	Paginated CRUD
`GET`	`/api/blocked`	Browse current effective block set
`GET/POST`	`/api/blocklists`	List sources / add custom source
`PUT`	`/api/blocklists/:id/activate`	Download + activate blocklist

1.9 Environment Variables

1.9.1 Docker Compose Variables

Configured via `.env` (copy from `.env.example`):

```
# Required
JWT_SECRET=<min 32 chars>          # openssl rand -base64 48
DOCKERHUB_USERNAME=<your-dockerhub-username>
IMAGE_TAG=latest                   # optional, defaults to 'latest'
ALLOWED_ORIGIN=http://localhost    # CORS origin for the management API
```

Set in `compose.yaml` (generally no need to change):

```
# management-api
DATABASE_URL=postgres://postgres:password@postgres:5432/dns_manager
LISTEN_ADDR=0.0.0.0:8080
JWT_EXPIRY_HOURS=1

# dns-resolver (per-instance)
DATABASE_URL=postgres://postgres:password@postgres:5432/dns_manager
```

```
RESOLVER_ID=resolver-1            # resolver-2 for second instance
LISTEN_ADDR=0.0.0.0:5353
UPSTREAM_DNS=1.1.1.1:53
BLOCKLIST_RELOAD_INTERVAL_SECS=30
STATS_FLUSH_INTERVAL_SECS=5

# tui (run on host)
API_URL=http://localhost:80
```

1.9.2 Kubernetes Variables

Configured via k8s/secret.yaml and k8s/configmap.yaml:

```
# Secret (edit before applying)
JWT_SECRET=<min 32 chars>
POSTGRES_PASSWORD=<generate with: openssl rand -base64 24>
DATABASE_URL=postgres://postgres:<password>@postgres.dns-
manager.svc.cluster.local:5432/dns_manager

# ConfigMap
RUST_LOG=info
LISTEN_ADDR=0.0.0.0:8080              # management-api
DNS_LISTEN_ADDR=0.0.0.0:5353          # dns-resolver
HEALTH_ADDR=0.0.0.0:9090             # dns-resolver health endpoint
UPSTREAM_DNS=1.1.1.1:53
BLOCKLIST_RELOAD_INTERVAL_SECS=30
STATS_FLUSH_INTERVAL_SECS=5
JWT_EXPIRY_HOURS=24
```

1.10 Failover

1.10.1 Docker Compose — NGINX passive health checks

NGINX's `stream` module load-balances UDP traffic across both resolvers. If a resolver stops responding, NGINX detects the failure via `proxy_timeout` and automatically routes all queries to the healthy replica (1 s failover). After `fail_timeout` (10 s), NGINX retries the failed backend.

```
docker compose stop dns-resolver-1
dig @localhost -p 5354 google.com      # resolver-2 handles all queries
docker compose start dns-resolver-1    # NGINX re-adds after 10s
```

1.10.2 Kubernetes Failover

```
# Scale down to 1 replica
kubectl scale deployment dns-resolver -n dns-manager --replicas=1
dig @<EXTERNAL-IP> google.com         # still works

# Scale back up
kubectl scale deployment dns-resolver -n dns-manager --replicas=2
```

1.11 Load Testing

Three scripts live in `loadtest/` :

Script	What it tests	Tool needed
`dns-stress.sh`	DNS correctness (blocked → `0.0.0.0`, allowed → real IP) + UDP throughput	`dig` (`apt install dnsutils`)
`script.js`	HTTP management API (`/api/stats`, `/api/whitelist`)	k6
`run-all.sh`	Runs both of the above in sequence	both

```
# Run everything
./loadtest/run-all.sh

# Tune with env vars
DNS_HOST=127.0.0.1  DNS_DURATION=60  DNS_CONCURRENCY=50 \
API_URL=http://localhost:80  USERNAME=loadtest  PASSWORD=loadtest123 \
./loadtest/run-all.sh
```

k6 stages: ramp to 10 VUs (30 s) → sustain 50 VUs (60 s) → ramp down (10 s). Thresholds: `p(95) < 500 ms`, `error rate < 1 %`.

1.12 Tech Stack

Component	Crates / Tools
DNS resolver	`hickory-proto 0.25`, `hickory-resolver 0.25`
Async runtime	`tokio 1`
HTTP framework	`actix-web 4`, `actix-cors 0.7`
Database	`sqlx 0.8` + PostgreSQL 16
Auth	`jsonwebtoken 9` (HS256), `argon2 0.5`
HTTP client	`reqwest 0.12` (management-api), `reqwest 0.11` (TUI)
TUI	`ratatui 0.29`, `crossterm 0.28`
Container runtime	`gcr.io/distroless/cc-debian12` (~20 MB, no shell)
Load balancer (compose)	NGINX (stream + http)
Load balancer (k8s)	Cilium eBPF L2
K8s config	Kustomize
Load testing	k6, `dig` (dnsutils)
CI/CD	GitHub Actions
Image registry	`ghcr.io/linxli` (GitHub Container Registry)
Rust edition	2024 (nightly) — `dns-resolver`, `management-api`; 2021 — `tui`