Distributed Habit Tracker

A fault-tolerant web service with JWT auth and load balancing

Cedric Cathomas

DSy Challenge Task FS 2026 — OST

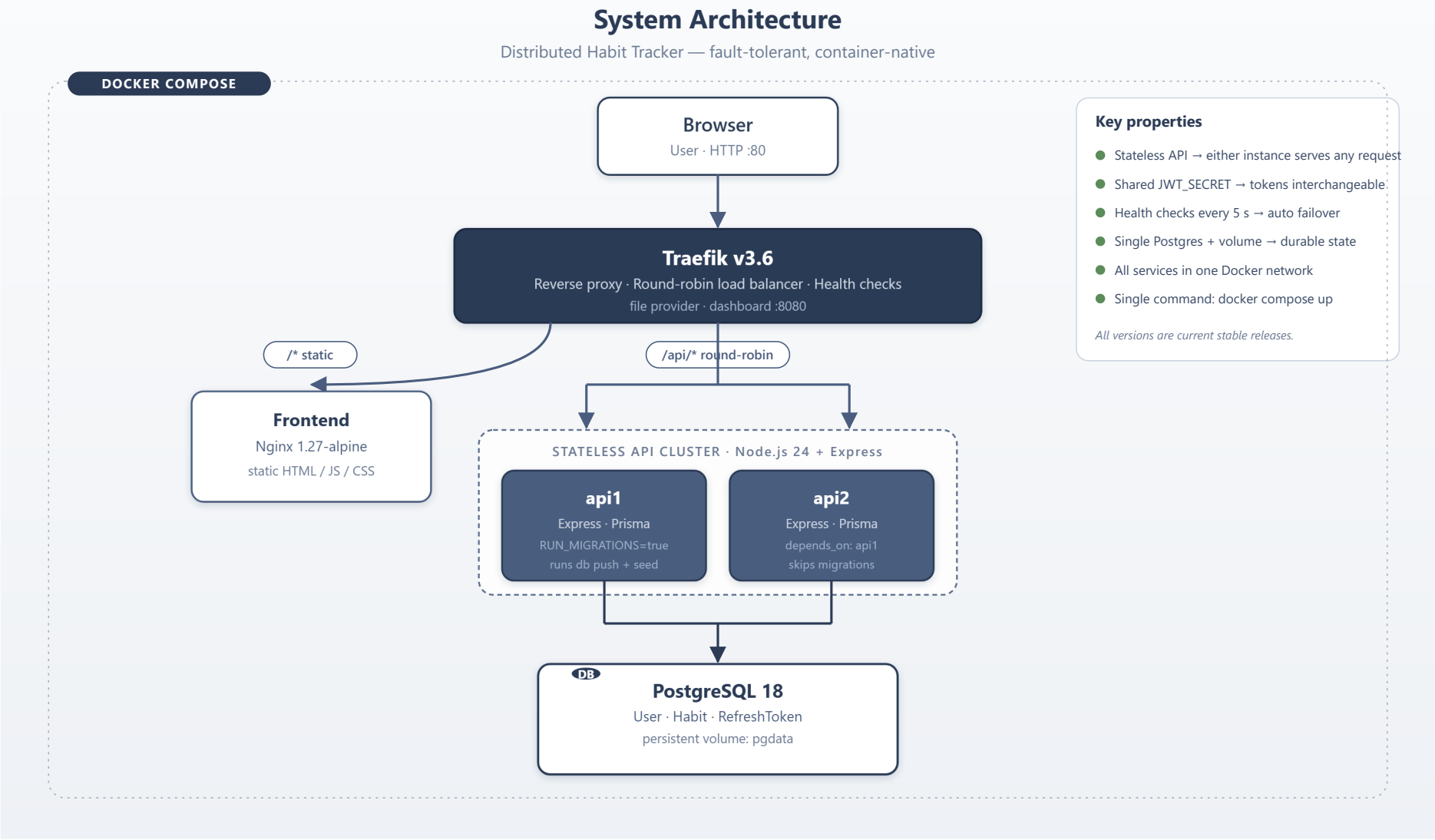
The Problem

Design and implement a **distributed system** where a service instance can fail without taking the system down.

Hard requirements:

- Load balancing across multiple instances + failover
- Containerized
- Simple frontend
- JWT authentication following OWASP guidelines
- Persistent storage
- `docker compose up` brings up the entire system
- Load test against a JWT-protected endpoint

Architecture



Tech Stack

Layer	Choice	Why
Frontend	HTML / vanilla JS, Nginx 1.27	Simple, no build step
Load Balancer	Traefik v3.6	Container-native, file provider
API (×2)	Node.js 24 + Express 4.19	Fast, mainstream, easy to explain
ORM	Prisma 5.13	Type-safe queries, auto migrations
Database	PostgreSQL 18	Robust, persistent volume
Auth	jsonwebtoken 9, bcryptjs	Standard JWT + OWASP password hash
Load Test	wrk (alpine)	Lightweight, direct HTTP benchmark
Orchestration	Docker Compose	Single-command spin-up

All versions are current stable releases.

Design Decisions (1/2)

Two stateless API instances

- Identical Docker image, only `INSTANCE_ID` and `RUN_MIGRATIONS` differ
- All state lives in Postgres → either instance can serve any request
- This is what makes failover *work*

Migration race avoidance

- `prisma db push` running on both instances simultaneously = race condition
- Solution: `RUN_MIGRATIONS=true` on `api1` only; `api2` `depends_on: api1`

Shared `JWT_SECRET` across instances

- Tokens issued by `api1` verify on `api2` — required for seamless failover

Design Decisions (2/2)

Separate Nginx for the frontend

- Static files don't burden the API containers
- Mirrors real-world separation (CDN / static host vs. application server)

Prisma over raw SQL

- Schema-as-code in `schema.prisma`
- `prisma db push` creates tables on first startup automatically
- No manual migration step required

File-based Traefik configuration

- Explicit, reviewable in git
- Two routers: `/api/*` (priority 10) and `/*` (priority 1)
- Health checks every 5 s detect failed instances

Authentication Design

Access + Refresh token flow (OWASP-aligned):

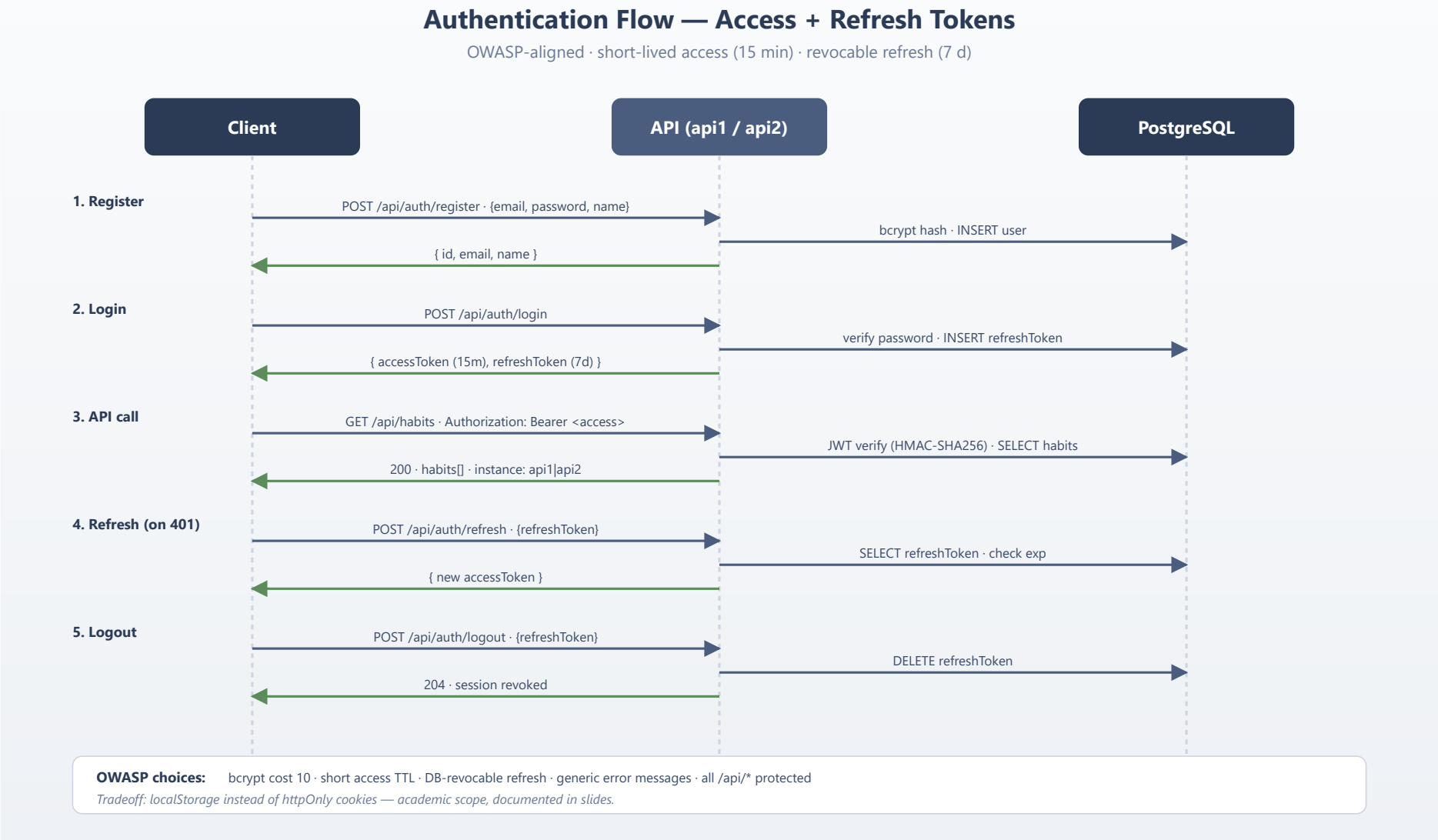
Token	Lifetime	Storage	Revocable?
Access	15 min	Stateless (JWT)	No (short-lived)
Refresh	7 days	Database row	Yes (delete row)

Security choices:

- Passwords hashed with **bcrypt cost factor 10**
- Generic `"Invalid credentials"` error → no user enumeration
- Refresh token = 320 bits of crypto-random entropy
- All `/api/*` endpoints protected by JWT middleware
- Cascading delete on user → orphan refresh tokens cleaned up

Documented tradeoffs: localStorage instead of httpOnly cookies, no rate limiting, no HTTPS in dev. (conscioous choice, because its just for academic and learning purposes)

Authentication — End-to-End Flow



Failover Strategy

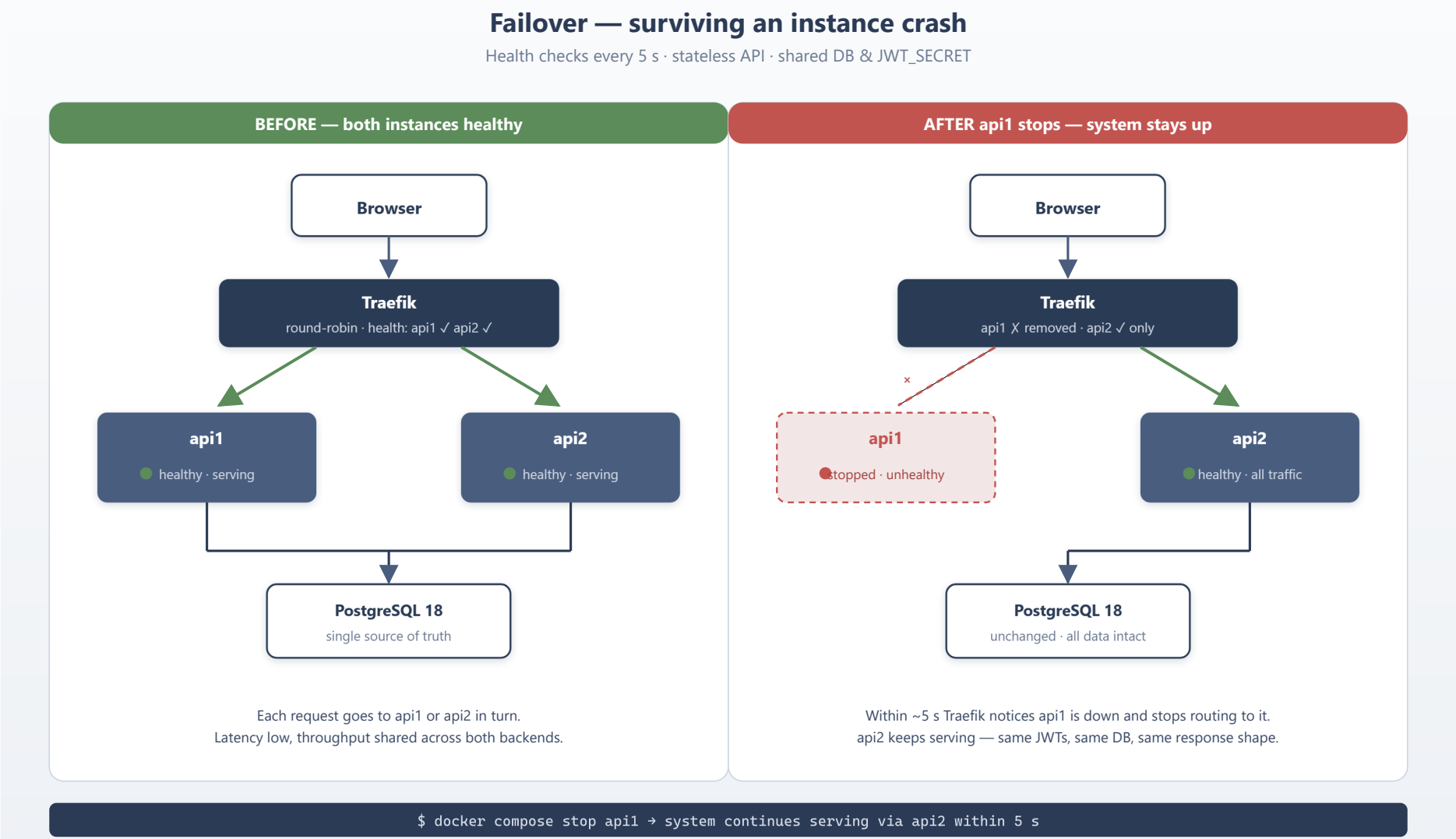
Why it works:

1. **Stateless API** — no in-memory sessions, no local cache
2. **Shared database** — single source of truth
3. **Shared** `JWT_SECRET` — tokens are interchangeable across instances
4. **Traefik active health checks** — `GET /health` every 5 s, removes failed backend from pool
5. **Round-robin** — even distribution across healthy backends

Live demo:

```
docker compose stop api1
# system continues serving via api2 within 5s
```

Failover — Before / After



Single-Command Startup

```
docker compose up --build
```

What happens automatically:

1. Postgres starts
2. `api1` runs `waitForDb.js` (retry-until-ready), then `prisma db push` + `seed.js`
3. `api2` waits for `api1`, skips migrations, starts Express
4. Traefik + Nginx come up
5. App live at `http://localhost`, dashboard at `http://localhost:8080`

No manual setup steps required.

Demo (5 min)

Demo Walkthrough

1. Bring up the system

```
docker compose up --build
```

2. Application flow — `http://localhost`

- Login as `demo@example.com` / `demo1234` -> Create / delete habits -> Show the `Served by: api1 / api2`

3. Failover

```
docker compose stop api1
```

- App still works
- Traefik dashboard (`:8080`) shows `api1` unhealthy

4. Load test

```
docker compose run --rm --build loadtest -c50 -d5s -H "Authorization: Bearer <token>" "http://api1:3000/api/me"
```

Load Test & Bottlenecks

Load Test Results

Setup: wrk, 50 connections, 2 threads, 5 seconds, target `GET /api/me` (JWT-protected)

Metric	Value
Total requests	10,576
Throughput	**~2,439 req/s**
Avg latency	27.68 ms
Stdev latency	32.00 ms
Max latency	448 ms
Socket timeouts	72

System sustains ~2,439 req/s under 50 concurrent connections with avg latency under 28 ms.

Where Are the Bottlenecks?

1. Single Postgres instance, no connection pooling

- Each request opens a Prisma connection, runs a SELECT
- First component to saturate under heavier load

2. JWT verification is CPU-bound

- HMAC-SHA256 signature check on every request
- Negligible at 100 req/s, matters at 10 000+

3. No caching layer

- Every `/api/me` hits the DB even though profile data is static

Scaling path: add `api3`, introduce **PgBouncer** for DB pooling, add **Redis** for hot reads (cache)

> Note: 100 timeouts at 50 connections point to DB connection exhaustion — the first bottleneck to address.

Summary

- All 7 task requirements are fulfilled
- Architecture is **simple, stateless, and survives instance failure**
- ~107 req/s sustained with 0 % errors and p95 = 12 ms
- Tradeoffs are **conscious and documented**
- Single command (`docker compose up`) brings up the whole stack

Repository: ct-fs2026

Cedric Cathomas — DSy FS 2026