

ForumZ

Ein verteiltes Forum

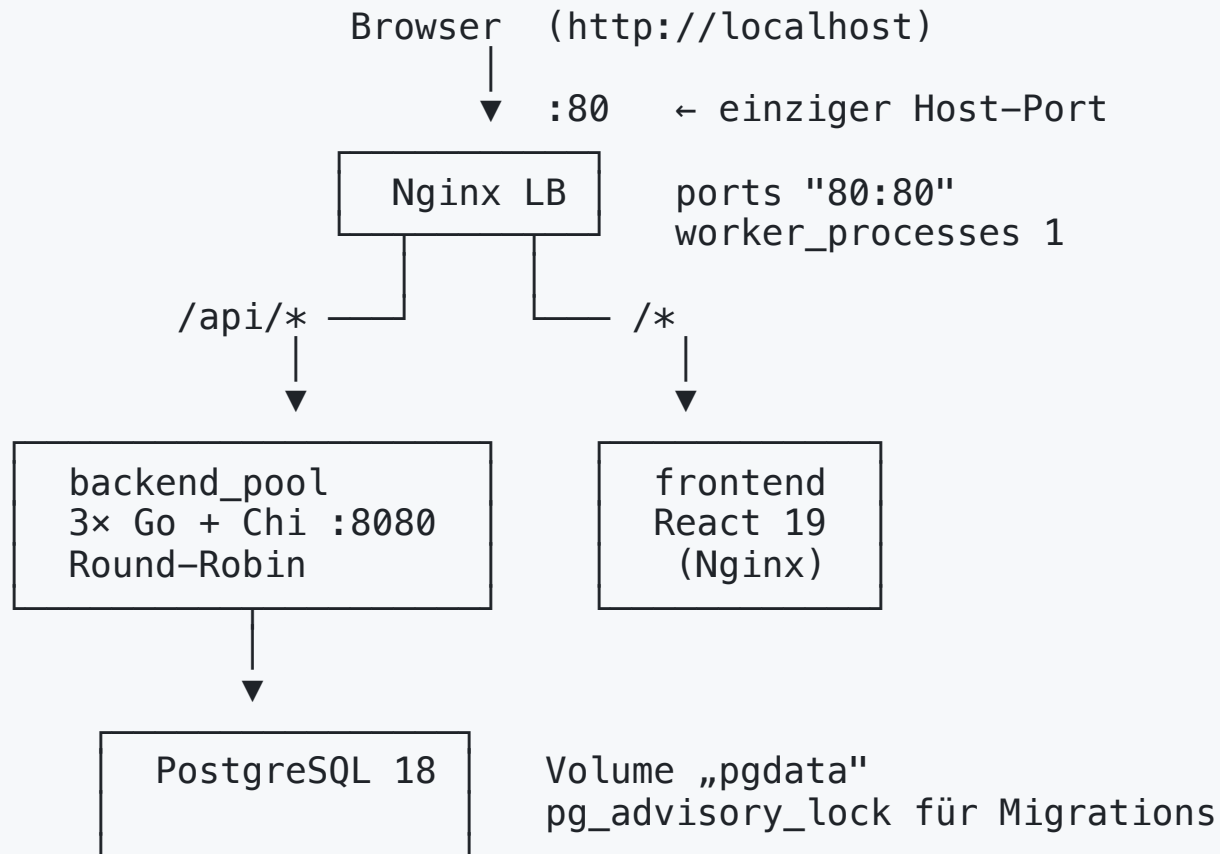
Distributed Systems · Jahr 3 / Semester 2

Lenny Merlo · 17.05.2026

Anforderungen

- Load Balancing über mehrere Instanzen **mit Failover**
- Containerisiert (Docker)
- Einfaches Frontend
- JWT-Authentifizierung **nach OWASP-Guidelines**
- Persistenter Storage
- Aktuelle stabile Library-Versionen
- Fresh clone → **ein Befehl** bringt alles hoch
- Lasttest gegen JWT-Endpoint + dokumentierte Auswertung

Architektur



Stateless Backends + statische React-SPA. Kein Container außer dem LB hat einen Host-Port. Zwei Docker-Bridges (`backend_net`, `frontend_net`) — der LB sitzt auf beiden, Frontend kommt nicht direkt an die DB.

Tech-Stack

Schicht	Technologie
Backend	Go 1.26 · Chi v5.2 · pgx v5.9 · golang-jwt/v5 · Argon2id
Datenbank	PostgreSQL 18
Load Balancer	Nginx 1.30 — Round-Robin + passiver Failover
Frontend	React 19 · TypeScript 6 · Vite 8 · Zustand · React Router 7
Container	Docker Compose v2
Lasttest	k6

Alle Versionen aktuell-stable und in `go.mod` / `package.json` / `docker-compose.yml` gepinned.

Design-Entscheidungen

Stateless Backends — keine Session-Daten im Speicher, jede Instanz kann jeden Request bedienen.
Voraussetzung für echtes Load Balancing.

Migrations beim Start mit Advisory Lock — Alle 3 Replicas starten simultan, `pg_advisory_lock` serialisiert die Migration. Keine separate Migrations-Pipeline → erfüllt „*single command bootstrap*“.

Passiver Failover statt Health Probes — `proxy_connect_timeout 1s` + `proxy_next_upstream` lassen Nginx transparent auf den nächsten Backend springen; nach 1 Fehler wird der defekte Backend für 10 s aus dem Pool genommen. Einfach, kein Service-Discovery-Stack.

Seed via Migration — `005_seed_demo_data.sql` legt Demo-User + Threads an → System ist sofort benutzbar.

JWT-Auth nach OWASP (1/2)

OWASP JWT Cheat Sheet, serverseitig:

Risiko	Umsetzung
Algorithm-Confusion	HS256-Whitelist via <code>jwt.WithValidMethods</code> blockt <code>alg:none</code>
Lange Token-Lebenszeit	15 min , <code>exp</code> + <code>nbf</code> + <code>iss</code> erzwungen, 30 s Clock-Skew
Schwaches Secret	HS256-Key $\geq$ 64 Zeichen , Hard-Fail beim Start
Schwache Passwörter	Argon2id, min. 12 Zeichen
Logout	Server löscht Fingerprint-Cookie → JWT sofort unbrauchbar

JWT-Auth nach OWASP (2/2) — Token Sidejacking

Problem: Ein gestohlenes JWT (XSS, Logging, Proxy) reicht normalerweise zum Impersonieren.

Lösung aus dem Cheat Sheet:

1. Beim Login generieren wir ein 50-Byte-Random in einem `__Secure-Fgp`-Cookie
→ `HttpOnly` · `Secure` · `SameSite=Strict` · `Path=/api`
 2. SHA-256 davon wird als `userFingerprint`-Claim **ins JWT** geschrieben
 3. `RequireAuth`-Middleware vergleicht Cookie-Hash **constant-time** gegen Claim
- Stolen JWT ohne Cookie = 401. Stolen Cookie ohne JWT = 401.
→ Logout muss nur den Cookie löschen, **keine DB-Denylist** nötig.

Demo — was ich gleich zeige

1. **System läuft schon** (`docker compose up --build` ist durch)
2. **Frontend** im Browser → Login als `demo` / `demo1234`
3. **Load Balancing + Failover** — Loop läuft kontinuierlich, mitten drin `backend2` killen

```
# Terminal 1 — Endlos-Loop (Ctrl+C zum Stoppen)
while true; do curl -s http://localhost/api/health | jq -r .served_by; sleep 0.3; done

# Terminal 2 — Backend killen, dann live zuschauen wie 2 aus dem Stream verschwindet
docker compose stop backend2
docker compose start backend2    # später wieder hochfahren
```

→ vor `stop`: zyklert 1 / 2 / 3, danach: nur noch 1 / 3, kein Client-Fehler

4. **OWASP-Härtung** — Request mit JWT ohne `__Secure-Fgp`-Cookie → 401
5. **Lasttest** — k6 (Ergebnisse auf nächstem Slide)

Lasttest — Ergebnis

Setup: 50 VUs · 2 min sustained · MacBook + Docker Desktop · 3 Replicas

Endpoint: GET /api/users/me — JWT + Fingerprint-Cookie

```
checks_total .....: 126 624    (1 055/s)
checks_succeeded .....: 100.00 %
http_req_duration ....: avg=6.3 ms · med=5.5 ms · p95=13.3 ms · max=62.5 ms
http_req_failed .....: 0.00 %
http_reqs .....: 42 208      (352/s)
vus .....: max=50
```

→ ~**352 req/s** über 3 Replicas (≈ 117 req/s pro Instanz)

→ Jeder Request läuft den vollen OWASP-Pfad (JWT-Decode + SHA-256 + ct-cmp)

→ Null Fehler, p95 unter 15 ms

Bottleneck-Analyse

Primärer Bottleneck: DB-Connection-Pool

`MaxConns = 10` pro Backend → **30 Connections** insgesamt (Postgres-Default: 100).

Bei > 200 VUs queuen Requests auf eine freie Connection.

Skalierungs-Optionen:

- `MaxConns` hochsetzen — bounded durch Postgres `max_connections`
- **PgBouncer** zwischen Backends und DB (server-seitiges Pooling)
- Read-Replicas für lese-lastige Endpoints
- Mehr Backend-Replicas hinter dem LB

OWASP-Hardening ist **nicht** der Flaschenhals — $1 \times \text{SHA-256} + 1 \times \text{ct-compare} \approx \text{Mikrosekunden}$.

Repo-Wegweiser

backend-go/internal/auth/auth.go	JWT + Fingerprint-Cookie + RequireAuth
backend-go/internal/api/users.go	register · login · logout · me
backend-go/internal/db/db.go	pgxpool + Migrations + Advisory Lock
backend-go/internal/config/config.go	64-char-Secret-Check
docker/nginx-lb/nginx.conf	LB + Failover-Regeln
docker-compose.yml	db · 3× backend · LB · frontend
migrations/	Schema + Seed (005)
load-test/k6-load-test.js	k6 gegen /api/users/me

Alle Responses enthalten `X-Served-By: N` Header und `"served_by": "N"` im JSON — LB-Wirkung von außen sichtbar.

Bewusst nicht gemacht (Q&A)

- **Refresh-Tokens** — 15-min Access-Token + Logout-Cookie-Clear deckt OWASP ab.
- **Distributed Tracing** — Out-of-Scope, strukturierte Logs via Go `slog`.
- **Service Discovery** — statische Compose-Hostnames, kein Consul/etcd bei 3 Backends.
- **Circuit Breaker** — Backends crashen bei DB-Ausfall hart; in Produktion Retry + Backoff.
- **Sticky Sessions** — Backends sind stateless, daher unnötig.
- **TLS** — gehört in der echten Welt vor den LB (ALB/CloudFront/Let's Encrypt).

Fragen?

Repo: <https://gitlab.ost.ch/lenny.merlo/forum2000>

Hochfahren: `docker compose up --build` → <http://localhost>

Lasttest: `k6 run load-test/k6-load-test.js`

Danke!