SiteLog Team

# SiteLog

### Architektur, Auth & Refresh Token

#### Tagesrapport-Tool für Baustellen

Verteiltes System mit Load Balancing & Failover

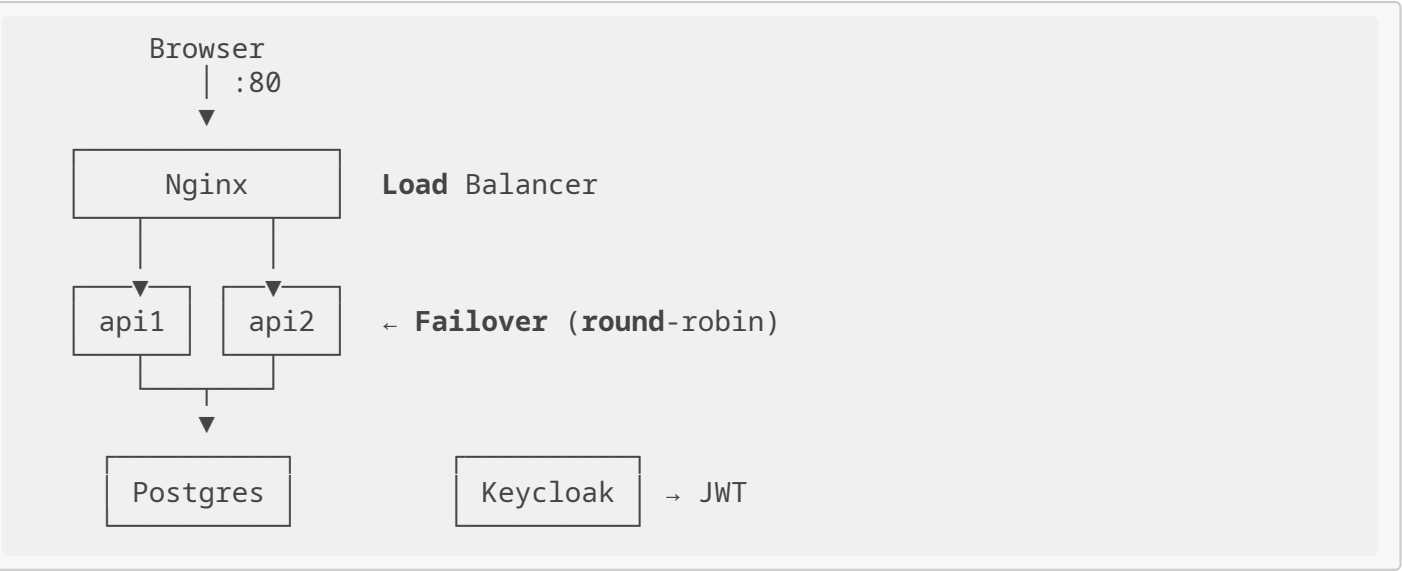
---

# Agenda

1. **Architektur-Überblick** — Komponenten & Datenfluss
2. **Tech Stack** — Was steckt drin
3. **Auth-Flow** — Login mit Keycloak
4. **Sicherheit** — OWASP JWT-Best-Practices
5. **Load Test & Failover** — Zahlen aus dem echten Setup

---

# Architektur — Big Picture




**Load Balancing:** Nginx round-robin → `api1` / `api2` **Failover:** `proxy_next_upstream`, `max_fails=3`, `fail_timeout=10s` **Datenbank:** geteilte PostgreSQL für beide API-Instanzen

---

# Tech Stack

| Komponente | Technologie |
| --- | --- |

| | |
|---|---|
| Frontend | Next.js 16 (App Router, TS) |
| Auth | Keycloak 26 + NextAuth.js v5 |
| Backend | ASP.NET Core Web API (.NET 10) |
| Datenbank | PostgreSQL 17 |
| Load Balancer | Nginx 1.27 |
| Orchestrierung | Docker Compose |

**Zwei separate Postgres-Instanzen:** - App-DB (sitelog) - Keycloak-DB (isoliert)

---

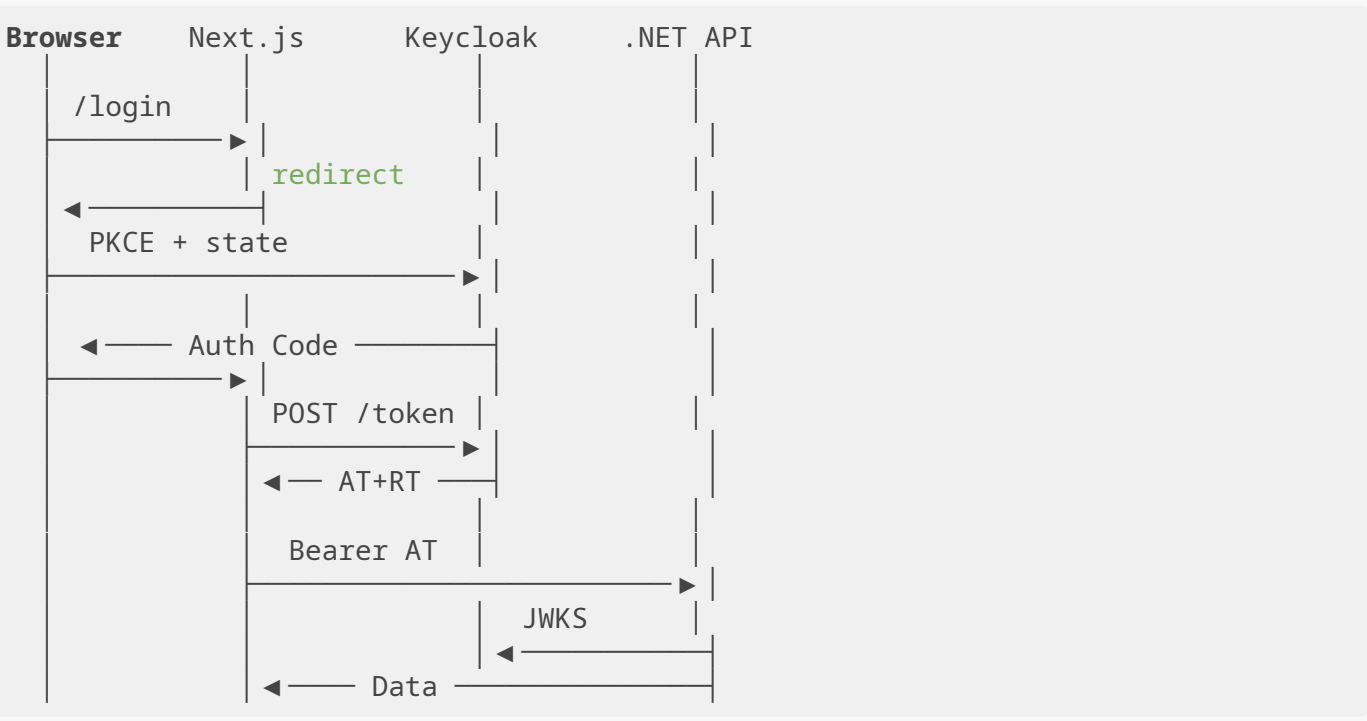
# Auth-Flow — Übersicht

```
1. User klickt "Anmelden"
2. Redirect → Keycloak (PKCE-Flow)
3. User gibt Credentials ein
4. Keycloak → Authorization Code zurück
5. NextAuth tauscht Code → Access + Refresh Token
6. Tokens server-side in Session-Cookie (httpOnly)
7. Server Components rufen .NET API mit Bearer
8. .NET API validiert via Keycloak JWKS
```

**Wichtig:** - PKCE schützt vor Code-Interception - State-Check verhindert CSRF - `iss`-Claim-Validierung (RFC 9207, Mix-Up Attack)

---

# Auth-Flow — Sequenz




---

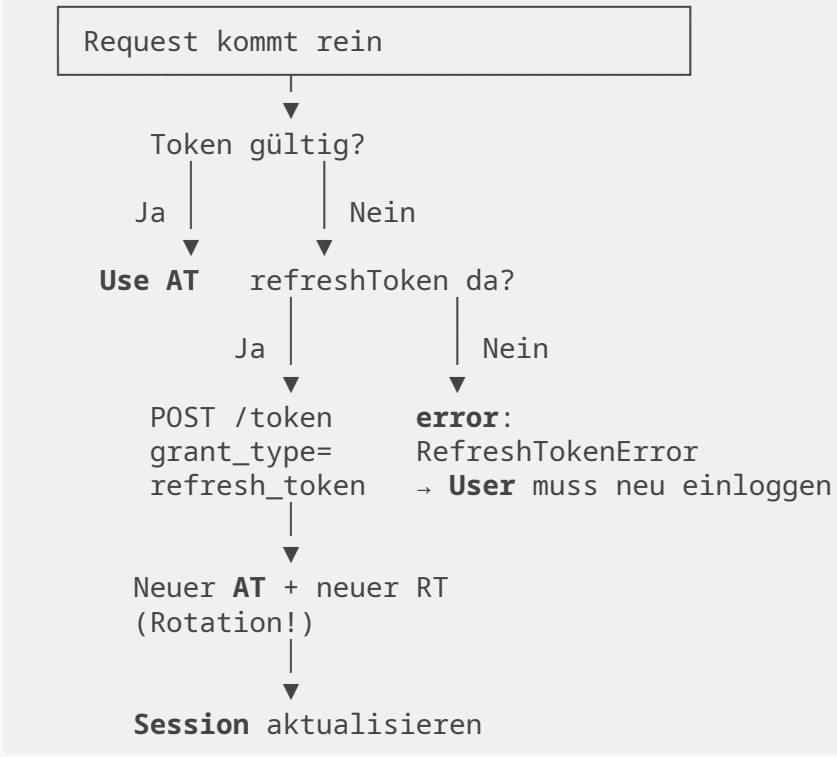
# Refresh Token — Code

```
// frontend/src/lib/auth.ts (gekürzt)
async jwt({ token, account }) {
  if (account) {
    return { ...token,
      accessToken: account.access_token,
      refreshToken: account.refresh_token,
      expiresAt: account.expires_at };
  }

  // Token noch gültig?
  if (Date.now() < token.expiresAt * 1000) return token;

  // → Refresh
  const res = await fetch(`${KC}/protocol/openid-connect/token`, {
    method: "POST",
    body: new URLSearchParams({
      grant_type: "refresh_token",
      client_id, client_secret,
      refresh_token: token.refreshToken,
    }),
  });
  const refreshed = await res.json();
  return { ...token,
    accessToken: refreshed.access_token,
    refreshToken: refreshed.refresh_token ?? token.refreshToken,
    expiresAt: Math.floor(Date.now()/1000) + refreshed.expires_in };
}
```

# Refresh Token — Ablauf



## Sicherheit — OWASP JWT

- **JWT nur von Keycloak** — keine Eigenimplementierung
- **RS256** (Public-Key-Signatur) — kein HS256-Mix-Up möglich
- **Kurze AT-Lebensdauer:** 15 Min
- **Refresh Token Rotation** — gestohlenes RT wird beim nächsten Refresh ungültig
- **Kein Token in** `localStorage` → NextAuth Session-Cookie, `httpOnly`, `SameSite`
- **PKCE + State** im Authorization Code Flow
- **Backend-Validierung** via Keycloak JWKS-Endpoint → Signatur, `iss`, `aud`, `exp`

## Load Test — Normalbetrieb

**Setup:** `hey -n 1000 -c 20` gegen `/api/daily-reports` Beide API-Instanzen aktiv, JWT auf jedem Request

| Metrik | Wert |
|---|---|
| Total Requests | 1000 |
| **Fehler** | **0 / 1000** |
| Requests/sec | **3'542** |
| Latenz p50 | 4.4 ms |
| Latenz p95 | 12.6 ms |
| Latenz p99 | 23.9 ms |

**Verbesserung Apr 14 → Apr 28:** +3.2× Throughput, p99 –95 %

## Load Test — Sättigung (Ramp-Up)

**Setup:** `max-load-test.sh`, c=50/100/200/500/1000, 20 s pro Stufe

| Concurrency | Req/s | p95 ms | p99 ms | Fehler |
|---|---|---|---|---|
| **50** | 478 | 18 | 4666 | **0** |
| 100 | 403 | 280 | 4725 | 246 |
| 200 | 271 | 4584 | 5371 | 238 |
| 500 | 521 | 5195 | 9572 | 292 |
| | | | | |

| | | | | |
|---|---|---|---|---|
| 1000 | 2543 | 5114 | 9364 | 464 |

**Saubere Zone: c ≤ 50** (~480 req/s, p95 < 20 ms, 0 Fehler) **Sättigung ab c=100** — Bottleneck vermutlich DB-Connection-Pool

---

# Failover-Demo — Ablauf

```
Phase 1              Phase 2              Phase 3
─────────            ─────────            ─────────
api1 + api2          api1 only            api1 + api2
beide healthy        api2 stopped         api2 restored
     │                    │                    │
     ▼                    ▼                    ▼
 3'280 r/s            65 r/s               485 r/s
 p95 = 8 ms           p95 = 3'002 ms       p95 = 298 ms
 0 Fehler             0 Fehler             0 Fehler
                      (Detection-Penalty)  (Cold-Start)
```

**Nginx:** `max_fails=3` , `fail_timeout=10s` , `proxy_next_upstream` → Ausfall wird erkannt, Requests transparent umgeleitet

---

# Failover — Was passiert wirklich?

**Phase 2 (api2 down):** - 190 / 200 Requests → direkt an api1, p50 = 2.1 ms (normal) - 10 / 200 Requests → trafen api2 → ~3 s Timeout → Retry auf api1 - **Bimodale Verteilung** in den Metriken, aber: - **0 / 200 Fehler beim Client**

**Phase 3 (api2 zurück):** - p95 = 298 ms statt 8 ms → **Cold-Start-Penalty** - JIT-Compilation der .NET-Runtime - Leerer DB-Connection-Pool - Nach Aufwärmphase wieder normal

**Take-away:** Failover funktioniert ohne Datenverlust, mit erwarteter Latenz-Spitze während der Detection-Phase.

---

# Zusammenfassung

| Frage | Antwort |
|---|---|
| Wer stellt Tokens aus? | Keycloak (zentral) |
| Wer validiert? | .NET API via JWKS |
| Wo läuft Refresh? | Next.js (NextAuth `jwt` -Callback) |
| Lebensdauer AT? | 15 Min |
| Lebensdauer RT? | 30 Min idle / 10h max |
| Rotation? | Ja, bei jedem Refresh |
| | |

| | |
|---|---|
| Storage? | httpOnly Session-Cookie |

---

# Demo / Fragen?

**Live-Demo** - `docker compose up` - Login-Flow zeigen - `docker compose stop api2` → Failover demonstrieren - Token-Refresh in DevTools sichtbar machen

**Code-Referenzen:** - `frontend/src/lib/auth.ts` — NextAuth Config - `backend/SiteLog.Api/Program.cs` — JWT Bearer Setup - `docs/architecture.md` — Vollständige Doku

**Danke!**