# Format Police

Image format conversion service

## What is Format Police?

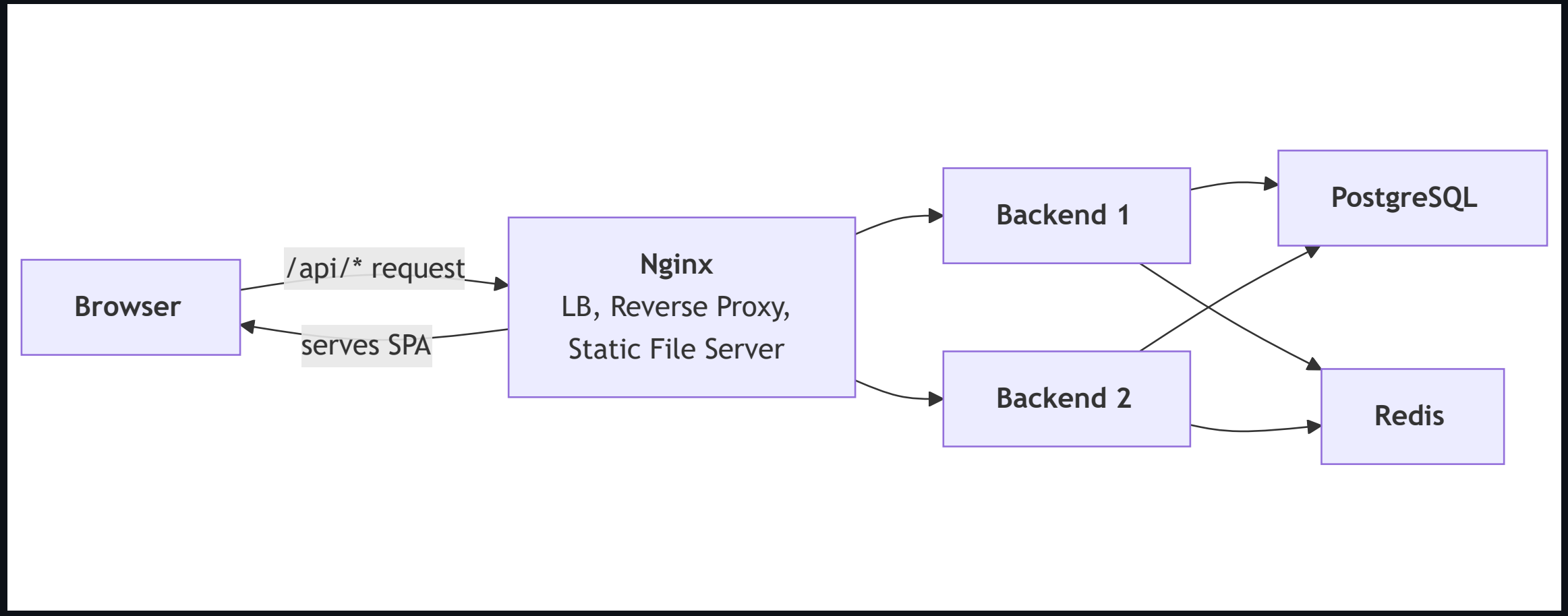
A web service where authenticated users upload an image and receive it converted to a different format (JPEG / PNG / GIF).

- **Auth:** JWT authentication with refresh token rotation
- **Rate limiting:** Redis, 10 req / 60 s per user
- **Scaling:** horizontal scaling + transparent failover
- **Deployment:** fully containerised (Docker Compose)

PART 1

# Architecture & Design

# System Architecture




Rendering model: CSR — Nginx serves static files once; all rendering happens in the browser.

## Request Flow

```
Request
  → RequireAuth    (JWT signature + expiry check)
  → RateLimit      (Redis INCR, reject if > 10)
  → ConvertHandler (decode image → re-encode)
```

### Short-circuit paths:

- **401:** missing / invalid / expired JWT → frontend auto-refreshes and retries once
- **429:** rate limit exceeded
- **400:** unsupported format or file too large

## Design Decision: JWT + Refresh Token Rotation

- **Access token:** JWT HS256, 15-min TTL, kept in JS memory only, sent as `Authorization: Bearer`
- **Refresh token:** 32 random bytes, 7-day TTL, HttpOnly cookie
  - Only the SHA-256 hash is stored in the DB
- **Client-side:** proactive refresh fires 60 s before expiry; on 401 retries once after refresh

## Design Decision: Distributed Rate Limiting

**Problem** — two stateless backends with local counters would allow 20 req/min instead of 10.

**Solution** — shared Redis counter:

```
key  = "rate:limit:user:<id>:<epoch / 60>"
INCR key
if result == 1: EXPIRE key 60
if result > 10: return 429
```

## Design Decision: Horizontal Scaling & Security

**Load balancing** — round-robin (Nginx default)

**Failover** — passive health checking via `max_fails=1 fail_timeout=10s`
After 1 failed request within 10 s the backend is excluded for 10 s, then retried automatically.

| Concern | Choice |
| --- | --- |
| Passwords | bcrypt |
| Refresh token storage | SHA-256 hash in DB only |
| Cookies | HttpOnly, SameSite=Strict |
| Upload size | 10 MB limit |

PART 2

# Load Test Results & Demo

## Load Test

### Scenario 1 — single-user burst (25 req, 5 concurrent)

- Rate limit enforced **exactly**: 10 × `200 OK`, 15 × `429`
- Successful requests: **3.8 ms** average
- 429 short-circuits: **~1.6–3 ms** (no image work done)

### Scenario 2 — 9 users × 9 requests (8×8 px test image)

| Metric | Value |
|---|---|
| Combined throughput | **465 req/s** * |
| P50 latency | 49.6 ms |
| Dominant cost | `resp wait` 35 ms (~97% of total) |

* *WSL2, single machine — all components co-located*

## Bottleneck Analysis

| Component | Latency share | Scales with |
|---|---|---|
| JWT validation | < 1 ms | Constant (in-memory) |
| Redis rate limit | 1–4 ms | Concurrent connections |
| Image decode | ~0.5 ms | Image pixel count (CPU) |
| Image encode | ~0.5 ms | Pixel count × quality |
| Nginx overhead | < 0.5 ms | Connection count |
| PostgreSQL | 0 ms | Not in `/api/convert` path |

- **Small images:** Redis I/O dominates; every request pays a round-trip for `INCR + EXPIRE`
- **Large images:** CPU dominates; decode/encode scale O(width × height)

## Demo

```
docker compose up --build
# open http://localhost
```

1. **Register** a new user — show bcrypt hash in the DB
2. **Login** → inspect JWT in DevTools; confirm HttpOnly cookie
3. **Convert** PNG → JPEG — check logs for which backend responded
4. **Hit rate limit** — 11 rapid requests; 11th returns `429`
5. **Kill a backend** — `docker compose stop backend1`; conversions still succeed
6. **Revive** — `docker compose start backend1`; both backends reappear in logs

# Questions?