Operational Efficiency & Caching

Overview

This document describes the caching strategy, data storage, and API efficiency optimisations in Spotify Genre Sorter.

Caching Strategy

Genre Cache

Attribute	Value
Key Pattern	`genre_cache_{spotifyUserId}`
TTL	1 hour (3600 seconds)
Storage	Cloudflare KV
Invalidation	On playlist creation

The genre cache stores:

- Sorted genre list with counts
- Track IDs per genre
- Total tracks/artists/genres
- Cache timestamp

Cache Flow:

```
GET /api/genres
  └─> Check genre_cache_{userId}
      ├─> HIT: Return cached data + fromCache: true
      └─> MISS: Fetch from Spotify → Cache → Return
```

Chunk Cache (Progressive Loading)

Attribute	Value
Key Pattern	`genre_chunk_{userId}_{offset}_{limit}`
TTL	1 hour (3600 seconds)
Purpose	Cache partial results for large libraries

For libraries >1000 tracks, progressive loading fetches in chunks of 500 tracks. Each chunk is cached separately to avoid re-fetching.

Session Cache

Attribute	Value
Key Pattern	`session:{uuid}`
TTL	7 days

Contents	OAuth tokens, user info

OAuth State

Attribute	Value
Key Pattern	`state:{uuid}`
TTL	10 minutes
Purpose	CSRF protection during OAuth flow
Behaviour	Single-use, deleted after verification

API Call Efficiency

Batch Operations

Operation	Batch Size	Spotify Limit
Get liked tracks	50 per request	50 max
Get artists	50 per request	50 max
Add tracks to playlist	100 per request	100 max
Get playlists	50 per request	50 max

Subrequest Budget

Cloudflare Workers free tier: 50 subrequests per invocation.

Per-chunk budget (500 tracks):

```
Track requests:  10 (500 ÷ 50 = 10)
Artist requests: ~10 (assuming ~500 unique artists)
Overhead:         1 (user info)
─────────────────────
Total:          ~21 subrequests (under 50 limit)
```

Retry Logic

All Spotify API calls use exponential backoff:

```
Attempt 1: Immediate
Attempt 2: 1000ms delay
Attempt 3: 2000ms delay
```

Respects Spotify's `Retry-After` header for 429 responses.

Data Storage Analysis

What We Store

Data	Location	Necessity	GDPR
Session ID	Cookie	Required	Functional
Spotify tokens	KV	Required	Consent given
Spotify user ID	KV	Required	Consent given
GitHub username	KV	Optional	Consent given
Genre cache	KV	Performance	Temporary (1hr)
Hall of Fame	KV	Feature	Display name only

What We DON'T Store

- Music library data (fetched on demand)
- Listening history
- Playlist contents
- Personal preferences (stored in localStorage)

Data Retention

Data Type	Retention
Sessions	7 days (auto-expire)
Genre cache	1 hour (auto-expire)
OAuth state	10 minutes (auto-expire)
User registry	Indefinite (minimal data)

Performance Optimisations

1. Parallel Artist Fetching

Artists are fetched in parallel batches rather than sequentially:

```
// Fetch all artist batches concurrently
const artistBatches = chunks.map(chunk =>
  spotifyFetch(`/artists?ids=${chunk.join(',')}`, token)
);
const results = await Promise.all(artistBatches);
```

2. Early Cache Check

Cache is checked before any Spotify API calls:

```
// Check cache FIRST
const cached = await kv.get(cacheKey);
if (cached) return cached;
```

```
// Only then fetch from Spotify
```

3. Incremental UI Updates

Frontend merges genre data incrementally rather than replacing:

```
// Merge new genres with existing
for (const genre of newGenres) {
  const existing = accumulatedGenres.get(genre.name);
  if (existing) {
    existing.count += genre.count;
    existing.trackIds.push(...genre.trackIds);
  } else {
    accumulatedGenres.set(genre.name, genre);
  }
}
```

4. Rate Limiting Cleanup

Rate limit map is cleaned probabilistically (1% of requests) to avoid memory growth:

```
if (Math.random() < 0.01) {
  // Clean up expired entries
}
```

Bundle Size

Component	Size
Hono framework	~14 KB
Application code	~45 KB
Embedded frontend	~35 KB
Total	~94 KB

Cloudflare Workers limit: 1 MB (compressed), 10 MB (uncompressed)

Cold Start Performance

Cloudflare Workers have near-zero cold start times due to:

- V8 isolate architecture
- Global edge deployment
- No container/VM overhead

Typical time-to-first-byte: <50ms

Memory Usage

Per-Request Memory

Operation	Peak Memory
Small library (<500 tracks)	~2 MB
Medium library (500-1000 tracks)	~5 MB
Large library (progressive)	~3 MB per chunk

Rate Limiter Memory

- Map entry size: ~50 bytes per IP
- Cleanup threshold: entries older than 2 minutes
- Maximum expected size: <1 MB

Cache Hit Rate Estimation

Based on typical usage patterns:

Cache	Expected Hit Rate
Genre cache	60-70% (1hr TTL, repeat views)
Chunk cache	40-50% (progressive loads)
Session	95%+ (7 day TTL)

Recommendations

Implemented

- [x] Genre caching with 1-hour TTL
- [x] Progressive loading for large libraries
- [x] Batch API calls
- [x] Exponential backoff retry
- [x] In-memory rate limiting

Future Considerations

1. **Cache warming**: Pre-fetch popular genres on login
2. **Compression**: Compress cached genre data (currently JSON)
3. **Cache metrics endpoint**: Add cache hit/miss counters
4. **User data export**: GDPR compliance self-service

Last updated: December 2025