Request Coalescing (Singleflight)

1. The Real Problem This Exists to Solve

When multiple clients simultaneously request the same resource, naive implementations make duplicate backend calls for identical data. This creates unnecessary load amplification that can overload databases, caches, or APIs.

Real production scenario:

- A popular post goes viral on social media
- 10,000 users click the link within 1 second
- Your application server receives 10,000 identical requests for `/post/12345`
- Each request triggers a database query: `SELECT * FROM posts WHERE id = 12345`
- Database receives 10,000 identical queries for the same row
- Database CPU spikes to 100%, query queue grows, latency increases to 5s
- More users retry, sending 20,000 more requests
- Database crashes or becomes unresponsive

The fundamental problem: **concurrent identical requests are not deduplicated**. Each request independently fetches the same data, creating a thundering herd at the backend.

Without request coalescing:

- 1000 concurrent requests = 1000 backend calls
- Backend load scales linearly with concurrent requests, not unique requests
- Cache misses cause N×amplification (where N = concurrent requests)
- System cannot handle traffic spikes even when data is cacheable

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: No Deduplication

```
// Incorrect: Every request independently fetches data
async function getPost(postId: string): Promise<Post> {
  return await database.query('SELECT * FROM posts WHERE id = ?', [postId]);
}

app.get('/post/:id', async (req, res) => {
  const post = await getPost(req.params.id);
  res.json(post);
});
```

Why it seems reasonable:

- Simple, straightforward code
- Each request gets fresh data
- No coordination between requests needed

How it breaks:

- 1000 concurrent requests for the same post = 1000 database queries
- During cache miss or cache expiration, all requests hit the database simultaneously

- Database connections exhausted
- Query queue builds up, latency increases from 10ms to 10s
- Timeouts trigger, clients retry, amplifying the problem

Production symptoms:

- Database CPU spikes correlate with popular content views
- Connection pool exhaustion errors: "too many connections"
- Query latency p99 is 100x higher than p50
- Slow query log shows the same query repeated thousands of times
- System cannot handle traffic spikes despite ample server capacity

❌ Incorrect Approach #2: Process-Level Caching Without Coalescing

```
// Incorrect: Cache results but don't coalesce in-flight requests
const cache = new Map<string, { data: Post; expiry: number }>();

async function getPost(postId: string): Promise<Post> {
  const cached = cache.get(postId);
  if (cached && cached.expiry > Date.now()) {
    return cached.data;
  }

  // Every request during cache miss hits DB
  const post = await database.query('SELECT * FROM posts WHERE id = ?', [postId]);

  cache.set(postId, {
    data: post,
    expiry: Date.now() + 60000, // 60 second TTL
  });

  return post;
}
```

Why it seems reasonable:

- Caching reduces database load for repeated requests
- Simple to implement with Map or LRU cache
- Works well for steady-state traffic

How it breaks:

- During cache miss window, all concurrent requests see "not in cache"
- All N concurrent requests proceed to query the database
- Database receives burst of N identical queries
- After first response, subsequent responses arrive but are redundant
- The cache helps steady-state but not the critical cache-miss storm

Production symptoms:

- Cache hit rate is 99%+ but database still has load spikes
- Load spikes happen exactly at cache expiry boundaries (every 60s)
- Database slow query log shows bursts of identical queries at regular intervals
- p99 latency shows periodic spikes (cache expiry stampede)

- Adding more cache servers doesn't solve the problem

❌ Incorrect Approach #3: Locking Without Per-Key Granularity

```typescript
// Incorrect: Global lock prevents all concurrent cache fills
const cache = new Map<string, Post>();
const cacheLock = new Mutex();

async function getPost(postId: string): Promise<Post> {
  if (cache.has(postId)) {
    return cache.get(postId)!;
  }

  // Global lock blocks ALL requests, even for different posts
  await cacheLock.lock();
  try {
    // Double-check after acquiring lock
    if (cache.has(postId)) {
      return cache.get(postId)!;
    }

    const post = await database.query('SELECT * FROM posts WHERE id = ?', [postId]);
    cache.set(postId, post);
    return post;
  } finally {
    cacheLock.unlock();
  }
}
```

Why it seems reasonable:

- Prevents duplicate database queries during cache miss
- Double-checked locking pattern is common
- Ensures only one request fills the cache

How it breaks:

- Global lock serializes ALL cache misses, not just identical ones
- Request for post/1 blocks request for post/2, even though they're independent
- Throughput collapses to 1 request at a time during cache misses
- 100 concurrent requests for 100 different posts = 100× slower than parallel
- Creates artificial bottleneck worse than the original problem

Production symptoms:

- Low database load (good) but terrible latency (bad)
- Requests queue up behind the lock, latency = N × query_time
- Single slow query blocks all other cache fills
- Throughput drops to single-digit requests/second during cache misses
- System cannot use parallelism even for different resources

3. Correct Mental Model (How It Actually Works)

Request coalescing (also called singleflight) deduplicates **in-flight** requests for the same resource. When multiple concurrent requests arrive for the same key:

1. First request starts the work (database query, API call)
2. Subsequent requests **wait for the first request to complete**
3. All requests receive the same result
4. Only one backend call is made, regardless of concurrent request count

The key data structure is an **in-flight map**:

```
Key (post/123) → Promise<Result> + Subscriber List

Request A arrives → creates Promise, starts work → in-flight["post/123"] = Promise
Request B arrives → sees in-flight["post/123"] → waits on existing Promise
Request C arrives → sees in-flight["post/123"] → waits on existing Promise
...
Work completes → resolve Promise → notify A, B, C with same result
Remove from in-flight map
```

State Machine

```
Idle State (key not in-flight)
     │
     ├─ Request A arrives
     ├─ Add to in-flight map: key → Promise
     ├─ Start backend work
     ▼
In-Flight State
     │
     ├─ Requests B, C, D arrive
     ├─ Return existing Promise (no new work)
     ▼
Work Completes
     │
     ├─ Resolve Promise with result
     ├─ All waiting requests receive result
     ├─ Remove from in-flight map
     ▼
Idle State
```

Critical Properties

Atomicity: Checking "is key in-flight?" and "add key to in-flight" must be atomic. Otherwise race condition:

```
Thread A: check in-flight["post/123"] → not found
Thread B: check in-flight["post/123"] → not found
Thread A: add in-flight["post/123"], start work
Thread B: add in-flight["post/123"], start work (DUPLICATE)
```

Cleanup: Must remove key from in-flight map after completion (success or failure). Otherwise memory leak and future requests wait forever on stale promise.

Per-Key Locking: Only requests for the same key wait on each other. Requests for different keys proceed in parallel.

4. Correct Design & Algorithm

Core Algorithm

```
function coalesce(key, workFn):
    lock(key)
    try:
        if in_flight[key] exists:
            return in_flight[key]  // Reuse existing promise

        promise = new Promise()
        in_flight[key] = promise

        unlock(key)  // Release lock before doing work

        try:
            result = await workFn()
            promise.resolve(result)
            return result
        catch error:
            promise.reject(error)
            throw error
        finally:
            delete in_flight[key]  // Always cleanup
    finally:
        if lock still held:
            unlock(key)
```

Key Design Decisions

1. Per-key locking vs global lock:

- Use per-key lock (or lock-free CAS) to avoid blocking unrelated keys
- Global lock creates unnecessary serialization

2. When to release lock:

- Release lock BEFORE starting work
- Hold lock only during in-flight map check/update
- This allows concurrent requests for the same key to quickly find existing promise

3. Error handling:

- If work fails, all waiting requests receive the same error
- Remove from in-flight map so next request can retry
- Avoid caching errors (unless intentional negative caching)

4. Timeout handling:

- If work times out, reject all waiting requests
- Cleanup in-flight entry

- Allow next request to retry

5. Full Production-Grade Implementation

```
interface CoalescedCall<T> {
  promise: Promise<T>;
  subscribers: number;
  startTime: number;
}

class Singleflight {
  private inFlight: Map<string, CoalescedCall<any>> = new Map();
  private readonly maxSubscribers: number;
  private readonly maxDuration: number;

  constructor(options?: {
    maxSubscribers?: number;
    maxDuration?: number;
  }) {
    this.maxSubscribers = options?.maxSubscribers ?? Infinity;
    this.maxDuration = options?.maxDuration ?? 30000; // 30 seconds
  }

  /**
   * Execute work function with request coalescing.
   * Multiple concurrent calls with the same key will share a single execution.
   *
   * @param key Unique identifier for this work (e.g., "post:123")
   * @param work Function to execute (e.g., database query)
   * @returns Result of work function
   */
  async execute<T>(key: string, work: () => Promise<T>): Promise<T> {
    // Fast path: check if already in-flight
    const existing = this.inFlight.get(key);
    if (existing) {
      existing.subscribers++;

      // Safety check: prevent unbounded subscriber growth
      if (existing.subscribers > this.maxSubscribers) {
        throw new Error(
          `Too many subscribers for key ${key}: ${existing.subscribers}`
        );
      }

      // Safety check: prevent stuck in-flight entries
      const age = Date.now() - existing.startTime;
      if (age > this.maxDuration) {
        this.inFlight.delete(key);
        throw new Error(
          `In-flight entry for ${key} exceeded max duration: ${age}ms`
        );
      }

      return existing.promise;
    }

    // Slow path: create new in-flight entry
    let resolve: (value: T) => void;
    let reject: (error: Error) => void;

    const promise = new Promise<T>((res, rej) => {
      resolve = res;
      reject = rej;
    });

    const call: CoalescedCall<T> = {
      promise,
      subscribers: 1,
      startTime: Date.now(),
    };

    this.inFlight.set(key, call);

    try {
      const result = await work();
      resolve!(result);
      return result;
    } catch (error) {
      reject!(error as Error);
      throw error;
    } finally {
      // Always cleanup, even on error
      this.inFlight.delete(key);
    }
  }

  /**
   * Get current in-flight status (for monitoring)
   */
  getStats(): {
    inFlightCount: number;
    totalSubscribers: number;
    oldestCallAge: number | null;
  } {
    let totalSubscribers = 0;
    let oldestCallAge: number | null = null;
    const now = Date.now();

    this.inFlight.forEach((call) => {
      totalSubscribers += call.subscribers;
      const age = now - call.startTime;
      if (oldestCallAge === null || age > oldestCallAge) {
        oldestCallAge = age;
      }
    });

    return {
      inFlightCount: this.inFlight.size,
      totalSubscribers,
      oldestCallAge,
    };
  }

  /**
   * Clear all in-flight entries (use cautiously, may break pending promises)
   */
  clear(): void {
    this.inFlight.clear();
  }
}

// Example usage with database
class PostService {
  private singleflight = new Singleflight({ maxSubscribers: 10000 });

  async getPost(postId: string): Promise<Post> {
    return this.singleflight.execute(`post:${postId}`, async () => {
      console.log(`[Singleflight] Fetching post ${postId} from database`);
      const result = await database.query(
        'SELECT * FROM posts WHERE id = ?',
        [postId]
      );
      return result.rows[0];
    });
  }
}

// Example usage with external API
class WeatherService {
  private singleflight = new Singleflight({ maxDuration: 5000 });

  async getWeather(city: string): Promise<Weather> {
    return this.singleflight.execute(`weather:${city}`, async () => {
      console.log(`[Singleflight] Calling weather API for ${city}`);
      const response = await fetch(`https://api.weather.com/v1/${city}`);
      return response.json();
    });
  }
}

// Example with caching layer
class CachedPostService {
  private cache = new Map<string, { data: Post; expiry: number }>();
  private singleflight = new Singleflight();
  async getPost(postId: string): Promise<Post> {
    // Check cache first
    const cached = this.cache.get(postId);
    if (cached && cached.expiry > Date.now()) {
      return cached.data;
    }

    // Cache miss: use singleflight to coalesce DB queries
    return this.singleflight.execute(`post:${postId}`, async () => {
      const post = await database.query(
        'SELECT * FROM posts WHERE id = ?',
        [postId]
      );

      // Update cache
      this.cache.set(postId, {
        data: post,
        expiry: Date.now() + 60000, // 60s TTL
      });

      return post;
    });
  }
}

// Express middleware for monitoring
app.get('/metrics/singleflight', (req, res) => {
  const stats = singleflight.getStats();
  res.json({
    in_flight_count: stats.inFlightCount,
    total_subscribers: stats.totalSubscribers,
    oldest_call_age_ms: stats.oldestCallAge,
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Database Query Deduplication

When multiple users request the same resource simultaneously:

```
class UserService {
  private singleflight = new Singleflight();

  async getUserProfile(userId: string): Promise<User> {
    return this.singleflight.execute(`user:${userId}`, async () => {
      return await db.query('SELECT * FROM users WHERE id = ?', [userId]);
    });
  }
}
```

```
// 1000 concurrent requests for user/123
// Without singleflight: 1000 DB queries
// With singleflight: 1 DB query, 999 requests wait and share result
```

Why this works:

- User profiles change infrequently (minutes to hours)
- Sharing result among concurrent requests is safe
- Dramatically reduces DB load during traffic spikes
- Latency is the same (all requests wait for same query)

Pattern 2: Cache Miss Storm Prevention

When cache expires, prevent thundering herd to database:

```
class ProductService {
  private cache = new LRU<string, Product>({ max: 10000 });
  private singleflight = new Singleflight();

  async getProduct(productId: string): Promise<Product> {
    const cached = this.cache.get(productId);
    if (cached) return cached;

    // Cache miss: coalesce DB queries
    return this.singleflight.execute(`product:${productId}`, async () => {
      const product = await db.query(
        'SELECT * FROM products WHERE id = ?',
        [productId]
      );
      this.cache.set(productId, product);
      return product;
    });
  }
}
```

Why this works:

- Cache expiry is a known stampede trigger
- Singleflight ensures only one request refills cache
- All concurrent requests during miss share the fill operation
- Works with any cache (Redis, memcached, in-memory)

Pattern 3: External API Rate Limit Protection

When calling third-party APIs with rate limits:

```
class GitHubService {
  private singleflight = new Singleflight({ maxDuration: 10000 });

  async getRepoInfo(owner: string, repo: string): Promise<RepoInfo> {
    const key = `github:${owner}/${repo}`;
    return this.singleflight.execute(key, async () => {
      const response = await fetch(
        `https://api.github.com/repos/${owner}/${repo}`,
        { headers: { Authorization: `token ${GITHUB_TOKEN}` } }
      );
      return response.json();
    });
  }
}
```

Why this works:

- External APIs have rate limits (e.g., 5000 requests/hour)
- Multiple users viewing same repo shouldn't count against limit multiple times
- Coalescing reduces API calls dramatically
- Helps stay within rate limits even during traffic spikes

Pattern 4: Expensive Computation Deduplication

When performing CPU-intensive operations:

```
class ReportService {
  private singleflight = new Singleflight({ maxDuration: 120000 });

  async generateMonthlyReport(userId: string, month: string): Promise<Report> {
    const key = `report:${userId}:${month}`;
    return this.singleflight.execute(key, async () => {
      // Expensive: aggregate millions of records, compute statistics
      const records = await db.query(
        'SELECT * FROM transactions WHERE user_id = ? AND month = ?',
        [userId, month]
      );
      return computeStatistics(records); // CPU-intensive
    });
  }
}
```

Why this works:

- Report generation takes 10-30 seconds
- User might click "generate" button multiple times impatiently
- Without coalescing: multiple 30s computations run in parallel
- With coalescing: subsequent clicks wait for in-progress computation

7. Failure Modes & Edge Cases

Race Condition: In-Flight Map Check and Insert

Problem: Two requests check in-flight map simultaneously, both see "not found", both start work.

Mitigation:

- Use atomic operations (CAS, Mutex)
- JavaScript single-threaded event loop makes this naturally atomic
- In multi-threaded languages, use locks or atomic operations

```
// JavaScript: naturally atomic due to event loop
const existing = this.inFlight.get(key);  // atomic
if (!existing) {
  this.inFlight.set(key, promise);  // atomic
}
```

Memory Leak: In-Flight Entry Not Cleaned Up

Problem: Work function throws exception, in-flight entry never removed, future requests hang.

Symptoms:

- Requests timeout waiting for in-flight promise
- Memory grows unbounded (in-flight map never shrinks)
- Debugging shows stale entries from hours ago

Mitigation:

- Always cleanup in finally block
- Set max duration, reject requests waiting on old entries
- Monitor in-flight map size

```
try {
  const result = await work();
  return result;
} finally {
  this.inFlight.delete(key);  // ALWAYS cleanup
}
```

Error Amplification: One Failure Fails All

Problem: Work function fails, all waiting requests receive same error.

Symptoms:

- 1000 concurrent requests, database query fails once, 1000 users see error
- Single network timeout causes mass request failure

Mitigation:

- This is often desired behavior (fail fast)
- For transient errors, next request will retry
- Consider caching errors with short TTL (negative caching)
- Implement circuit breaker to stop coalescing during persistent failures

```
catch (error) {
  // Remove from in-flight so next request can retry
  this.inFlight.delete(key);
  throw error;
}
```

Slow Request Blocks Fast Retries

Problem: First request is slow (10s), subsequent requests wait, even though retry might be faster.

Symptoms:

- All requests wait for slow database replica
- Retry with different replica could succeed faster

Mitigation:

- Set max duration for in-flight entries
- After timeout, allow new requests to retry
- Consider adaptive timeout based on p99 latency

```
const age = Date.now() - existing.startTime;
if (age > this.maxDuration) {
  this.inFlight.delete(key);
  // Fall through to start new request
}
```

Unbounded Subscriber Growth

Problem: 1 million concurrent requests wait on same key, causes memory pressure.

Symptoms:

- Promise subscriber list grows to millions of entries
- Memory pressure triggers GC pauses
- OOM errors

Mitigation:

- Set max subscribers per key
- Reject requests beyond limit (backpressure)
- Monitor subscriber count

```
if (existing.subscribers > this.maxSubscribers) {
  throw new Error('Too many subscribers, request rejected');
}
```

8. Performance Characteristics & Tradeoffs

Load Reduction

Without singleflight:

- 1000 concurrent requests = 1000 backend calls
- Backend load: O(N) where N = concurrent requests

With singleflight:

- 1000 concurrent requests = 1 backend call
- Backend load: O(1) regardless of concurrent requests

Benefit: 100-1000x reduction in backend load during traffic spikes

Latency Impact

All requests wait for the slowest request (first one):

- Without: each request independent, avg latency = p50 backend latency
- With: all requests wait for same call, latency = that specific call's latency

If first request is slow (p99), all subsequent requests pay that latency cost.

Tradeoff: Reduces load but couples request latencies

Memory Overhead

In-flight map:

- One entry per unique in-flight key
- Each entry: ~100 bytes (key + promise + metadata)
- 1000 concurrent unique keys = 100KB
- Negligible

Promise subscribers:

- Each waiting request adds minimal overhead
- JavaScript promises don't store subscriber list (internal)
- Negligible memory cost

CPU Overhead

Map operations:

- get(): O(1)
- set(): O(1)
- delete(): O(1)

Per-request overhead: <1μs

Negligible compared to backend work (1ms-1s).

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Using Singleflight for Mutations

Why engineers do it: "It deduplicates requests, let's use it everywhere!"

What breaks: Multiple users submit same mutation (e.g., "like post"), only first one succeeds.

```
// WRONG: Don't coalesce writes
async function likePost(userId: string, postId: string): Promise<void> {
  await singleflight.execute(`like:${postId}`, async () => {
    await db.query('INSERT INTO likes VALUES (?, ?)', [userId, postId]);
  });
}

// User A and User B both like post 123
// Only User A's like is saved, User B's is deduplicated
```

Detection: User reports submitting action multiple times but it only happens once.

Fix: Only use singleflight for idempotent reads.

```
// Correct: Don't coalesce mutations
async function likePost(userId: string, postId: string): Promise<void> {
  // Include userId in key to prevent deduplication across users
  await db.query('INSERT INTO likes VALUES (?, ?)', [userId, postId]);
}
```

Mistake 2: Not Including All Parameters in Key

Why engineers do it: Simplify key generation.

What breaks: Different queries share the same result.

```
// WRONG: Key doesn't include all parameters
async function getComments(postId: string, limit: number): Promise<Comment[]> {
  return singleflight.execute(`comments:${postId}`, async () => {
    return db.query('SELECT * FROM comments WHERE post_id = ? LIMIT ?',
      [postId, limit]
    );
  });
}

// Request 1: getComments('123', 10)
// Request 2: getComments('123', 100)  ← Gets 10 comments, not 100!
```

Detection: Users report seeing wrong number of results inconsistently.

Fix: Include all query parameters in key.

```
// Correct: Include all parameters in key
async function getComments(postId: string, limit: number): Promise<Comment[]> {
  return singleflight.execute(`comments:${postId}:${limit}`, async () => {
    return db.query('SELECT * FROM comments WHERE post_id = ? LIMIT ?',
      [postId, limit]
    );
  });
}
```

Mistake 3: Coalescing User-Specific Data

Why engineers do it: Looks like duplicate requests.

What breaks: User A sees User B's private data.

```
// WRONG: Key doesn't include user context
async function getPrivateProfile(profileId: string): Promise<Profile> {
  return singleflight.execute(`profile:${profileId}`, async () => {
    // Query checks current user's permissions
    return db.query('SELECT * FROM profiles WHERE id = ? AND ...');
  });
}

// User A requests profile 123 (has permission)
// User B requests profile 123 (no permission) ← Gets User A's result!
```

Detection: Privacy violation, users see data they shouldn't.

Fix: Include viewer context in key, or don't coalesce user-specific queries.

```
// Correct: Include viewer in key
async function getPrivateProfile(
  viewerId: string,
  profileId: string
): Promise<Profile> {
  return singleflight.execute(
    `profile:${profileId}:viewer:${viewerId}`,
    async () => {
      return db.query(
        'SELECT * FROM profiles WHERE id = ? AND visible_to = ?',
        [profileId, viewerId]
      );
    }
  );
}
```

Mistake 4: Not Handling Errors

Why engineers do it: Assume work always succeeds.

What breaks: One error causes all subsequent requests to fail permanently.

```
// WRONG: Error leaves stale entry in map
async execute(key: string, work: () => Promise<T>): Promise<T> {
  if (this.inFlight.has(key)) {
    return this.inFlight.get(key)!;
  }

  const promise = work();
  this.inFlight.set(key, promise);
  return promise;
  // Never deletes on error!
}
```

Detection: Request fails once, all future requests fail or hang forever.

Fix: Always cleanup in finally.

```
// Correct: Cleanup on both success and error
try {
  const result = await work();
  return result;
} catch (error) {
  throw error;
} finally {
  this.inFlight.delete(key);
}
```

Mistake 5: Coalescing Across Incompatible Cache Layers

Why engineers do it: "Reduce database load."

What breaks: Cached data is stale, but singleflight serves it to all requests.

```
// WRONG: Cache and singleflight don't coordinate
const cache = new Map();
const singleflight = new Singleflight();

async function getData(key: string): Promise<Data> {
  // Check cache
  if (cache.has(key)) return cache.get(key);

  // Coalesce database queries
  return singleflight.execute(key, async () => {
    const data = await db.query('SELECT * FROM data WHERE id = ?', [key]);
    cache.set(key, data);
    return data;
  });
}

// Cache eviction happens, but singleflight key still in-flight
// Subsequent requests get stale data from in-flight promise
```

Detection: Data is stale even though cache was invalidated.

Fix: Don't cache the promise itself, only cache the result after completion.

10. When NOT to Use This (Anti-Patterns)

Anti-Pattern 1: Write Operations

Never coalesce writes, inserts, updates, deletes:

```
// NEVER do this
async function createOrder(order: Order): Promise<string> {
  return singleflight.execute('create-order', async () => {
    return db.insert('orders', order);
  });
}
// Multiple users creating orders will only create ONE order
```

Anti-Pattern 2: User-Specific Results

Don't coalesce when results depend on caller identity:

```
// NEVER do this
async function getMyNotifications(userId: string): Promise<Notification[]> {
  return singleflight.execute('notifications', async () => {
    return db.query('SELECT * FROM notifications WHERE user_id = ?', [userId]);
  });
}
// All users share same userId's notifications
```

Anti-Pattern 3: Real-Time Data

Don't coalesce when data must be fresh (stock prices, live scores):

```
// Don't do this for real-time data
async function getStockPrice(symbol: string): Promise<number> {
  return singleflight.execute(`stock:${symbol}`, async () => {
    return exchange.getCurrentPrice(symbol);
  });
}
// Users get stale price from milliseconds ago
```

Use singleflight for reference data, not real-time data.

Anti-Pattern 4: Low Traffic Endpoints

Don't add complexity when traffic doesn't justify it:

```
// Unnecessary: this endpoint gets 1 req/minute
async function getAdminConfig(): Promise<Config> {
  return singleflight.execute('admin-config', async () => {
    return db.query('SELECT * FROM config');
  });
}
```

Singleflight adds complexity. Only use when concurrent requests are common.

11. Related Concepts (With Contrast)

Caching

Difference: Caching stores results across time. Singleflight deduplicates concurrent in-flight requests.

When to combine: Use both. Singleflight prevents cache stampede at expiry. Cache provides long-term storage.

```
// Combined: Cache + Singleflight
async function getData(key: string): Promise<Data> {
  const cached = cache.get(key);
  if (cached) return cached;

  return singleflight.execute(key, async () => {
    const data = await db.query('...');
    cache.set(key, data);
    return data;
  });
}
```

Memoization

Difference: Memoization caches function results based on arguments. Singleflight coalesces concurrent calls.

Memoization stores result permanently. Singleflight stores only during execution.

Request Batching

Difference: Batching combines multiple different requests into one backend call. Singleflight combines duplicate requests into one call.

Example:

- Batching: get user/1, user/2, user/3 → SELECT * WHERE id IN (1,2,3)
- Singleflight: get user/1, user/1, user/1 → SELECT * WHERE id = 1

When to combine: Use both. Batch different requests; coalesce identical requests.

Circuit Breaker

Difference: Circuit breaker prevents calls to failing services. Singleflight deduplicates calls to working services.

When to combine: Use both. Circuit breaker for failure protection; singleflight for load protection.

12. Production Readiness Checklist

Before deploying singleflight to production:

Metrics to Monitor

- [ ] In-flight map size (number of unique keys)
- [ ] Total subscribers across all in-flight calls
- [ ] Oldest in-flight call age (detect stuck entries)
- [ ] Singleflight hit rate (% of requests that coalesced)
- [ ] Backend call reduction (before vs after)

Logging Requirements

- [ ] Log when new in-flight call starts (first request for key)
- [ ] Log subscriber count when threshold exceeded (e.g., >1000)
- [ ] Log when in-flight entry exceeds max duration
- [ ] Log when singleflight entry completes (success/failure, latency)

Limits and Safeguards

- [ ] Set max subscribers per key (prevent memory pressure)

- [ ] Set max duration per in-flight entry (prevent stuck entries)
- [ ] Implement cleanup of stale entries (cron job)
- [ ] Set max in-flight map size (prevent unbounded growth)

Load Testing Considerations

- [ ] Test with 1000 concurrent identical requests (verify coalescing)
- [ ] Test with 1000 concurrent different requests (verify no blocking)
- [ ] Test error handling (verify cleanup on failure)
- [ ] Test timeout handling (verify cleanup on timeout)
- [ ] Measure backend call reduction (before/after)
- [ ] Verify latency doesn't increase (should be same or better)

Rollout Strategy

- [ ] Deploy to 1% of traffic, verify metrics improve
- [ ] Monitor backend load reduction
- [ ] Monitor for errors or stuck requests
- [ ] Gradually increase to 10%, 50%, 100%
- [ ] Have feature flag to disable instantly if issues arise

Alerting

- [ ] Alert if in-flight map size > 10,000 (memory leak)
- [ ] Alert if oldest call age > 60s (stuck request)
- [ ] Alert if subscriber count > 10,000 (potential DoS)
- [ ] Alert if singleflight hit rate drops suddenly (indicates key mismatch bug)