Request Idempotency Patterns

1. The Real Problem This Exists to Solve

When clients retry failed requests (due to network errors, timeouts, or uncertainty), the same operation may execute multiple times on the server, causing duplicate charges, double inventory deduction, or inconsistent state. Idempotency ensures that repeating the same request produces the same result without side effects.

Real production scenario:

- Payment API: User clicks "Pay $100" button
- Request sent to server
- Server processes payment successfully
- Response lost due to network timeout
- Client doesn't receive confirmation
- User clicks "Pay" again (thinks first attempt failed)
- **Without idempotency:**
  - Second request processed as new payment
  - User charged $200 (double charge)
  - Customer dispute, refund required
  - Trust damaged, regulatory issues
- **With idempotency:**
  - Second request includes idempotency key from first request
  - Server detects duplicate, returns cached result
  - No duplicate charge
  - User correctly charged $100 once

The fundamental problem: Network is unreliable. Clients can't distinguish between "request failed" and "request succeeded but response lost". Without idempotency, retries cause duplicate operations.

Without idempotency:

- Retries create duplicates
- No way to safely retry
- Users experience double charges, duplicate orders
- Complex cleanup/reconciliation needed
- Lost revenue from scared users

With idempotency:

- Safe to retry any request
- Same result regardless of retry count
- No duplicate side effects
- Better user experience
- Simpler error handling

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: No Idempotency (Hope for the Best)

```
// Incorrect: Process every request as new operation
app.post('/api/payment', async (req, res) => {
```

```
  const { userId, amount, cardToken } = req.body;

  // Charge card every time this endpoint is called
  const charge = await stripe.charges.create({
    amount: amount * 100,
    currency: 'usd',
    source: cardToken,
    description: `Payment from user ${userId}`,
  });

  // Deduct from user balance
  await db.query(
    'UPDATE accounts SET balance = balance - $1 WHERE user_id = $2',
    [amount, userId]
  );

  res.json({ chargeId: charge.id, success: true });
});
```

Why it seems reasonable:

- Simple implementation
- No extra complexity
- Works fine if network perfect
- Standard CRUD pattern

How it breaks:

- Client timeout → retry → double charge
- Load balancer retry → duplicate processing
- User double-click → two charges
- No deduplication mechanism
- Can't distinguish retry from new request

Production symptoms:

- Customer reports: "I was charged twice!"
- Database shows duplicate transactions 1 second apart
- Customer support spends hours investigating
- Refunds required (processing fees lost)
- Stripe dispute fees ($15 per dispute)
- Trust issues, negative reviews

❌ Incorrect Approach #2: Client-Side Deduplication Only

```
// Incorrect: Client tracks requests, server doesn't
// Client code
let inFlight = new Set<string>();

async function makePayment(amount: number) {
  const requestId = `payment-${Date.now()}`;

  if (inFlight.has(requestId)) {
```

```
    console.log('Request already in flight, skipping');
    return;
  }

  inFlight.add(requestId);

  try {
    const result = await fetch('/api/payment', {
      method: 'POST',
      body: JSON.stringify({ amount }),
    });
    return result.json();
  } finally {
    inFlight.delete(requestId);
  }
}

// Server code (no idempotency)
app.post('/api/payment', async (req, res) => {
  await processPayment(req.body);
  res.json({ success: true });
});
```

Why it seems reasonable:

- Prevents duplicate requests from client
- Simple client-side tracking
- No server changes needed

How it breaks:

- Doesn't work across page refreshes (client state lost)
- Doesn't work if user opens multiple tabs
- Doesn't work with mobile app (multiple devices)
- Doesn't prevent load balancer retries
- Doesn't prevent server-side retries
- Fails if client crashes and restarts

Production symptoms:

- Works in testing (single session)
- Fails in production (multiple tabs, devices)
- User opens 2 browser tabs → 2 charges
- Mobile app in background → retry → duplicate
- Client-side deduplication bypassed easily

❌ Incorrect Approach #3: Database Unique Constraint Only

```
// Incorrect: Rely only on DB unique constraint
// Database schema
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT,
```

```
  amount DECIMAL,
  created_at TIMESTAMP,
  UNIQUE(user_id, amount, created_at)  -- Try to prevent duplicates
);

// Server code
app.post('/api/order', async (req, res) => {
  try {
    await db.query(
      'INSERT INTO orders (user_id, amount, created_at) VALUES ($1, $2, NOW())',
      [req.body.userId, req.body.amount]
    );
    res.json({ success: true });
  } catch (err) {
    if (err.code === '23505') {  // Unique constraint violation
      res.json({ success: true, message: 'Already processed' });
    } else {
      throw err;
    }
  }
});
```

Why it seems reasonable:

- Database enforces uniqueness
- Simple to implement
- Automatic deduplication
- Relies on database guarantees

How it breaks:

- User might legitimately order same amount twice
- Constraint can't distinguish retry from new request
- If order amount is $50, user can never order $50 again
- Time-based uniqueness breaks with clock skew
- Doesn't work for updates/deletes (only inserts)
- Can't handle partial failures (payment succeeded, inventory failed)

Production symptoms:

- User orders $50 item
- Retry blocked by unique constraint
- Later tries to order different $50 item → rejected
- User confused: "Why can't I order this?"
- Legitimate requests blocked

❌ Incorrect Approach #4: Request Deduplication Without State Storage

```
// Incorrect: Check for duplicates but don't store result
const recentRequests = new Set<string>();

app.post('/api/payment', async (req, res) => {
  const idempotencyKey = req.headers['idempotency-key'];
```

```
  if (recentRequests.has(idempotencyKey)) {
    // Duplicate detected, but don't know what result was!
    return res.json({ error: 'Duplicate request' });
  }

  recentRequests.add(idempotencyKey);

  const result = await processPayment(req.body);

  // Not storing result for future retries
  res.json(result);
});
```

Why it seems reasonable:

- Detects duplicates
- Prevents duplicate processing
- Simple in-memory tracking

How it breaks:

- First request succeeds → returns result
- First request response lost in network
- Second request arrives → detected as duplicate
- But server doesn't have original result to return!
- Returns error instead of success
- Client thinks payment failed (actually succeeded)
- User tries different payment method → double charged

Production symptoms:

- First attempt: Payment succeeds
- Retry: Server returns "Duplicate request" error
- User sees error, tries again with different card
- Both payments went through
- Customer charged twice on different cards

3. Correct Mental Model (How It Actually Works)

Idempotency means executing an operation multiple times has the same effect as executing it once. For HTTP APIs, this requires:

The Idempotency Key Model

```
Client generates unique key per logical operation:
  idempotency-key: "payment-abc123-2024-01-15-uuid"

First request:
  1. Server receives request with key "payment-abc123"
  2. Checks if key exists in storage → No
  3. Stores key with status "PROCESSING"
  4. Processes payment
```

```
  5. Stores key with status "COMPLETED" + result
  6. Returns result to client

Retry (response lost):
  1. Server receives same request, same key
  2. Checks if key exists → Yes, status "COMPLETED"
  3. Returns stored result (no reprocessing)
  4. Client gets result from first attempt

Result: Same outcome whether request sent once or 100 times
```

State Machine for Idempotency

```
States:
  1. NOT_STARTED (key doesn't exist)
  2. PROCESSING (request in progress)
  3. COMPLETED (success, result stored)
  4. FAILED (failure, error stored)

Transitions:
  NOT_STARTED → PROCESSING (first request)
  PROCESSING → COMPLETED (success)
  PROCESSING → FAILED (error)
  PROCESSING → PROCESSING (concurrent duplicate, wait)
  COMPLETED → COMPLETED (retry, return stored result)
  FAILED → FAILED (retry, return stored error)
```

The Storage Requirements

```
Idempotency store must contain:
  - Key: Unique identifier from client
  - Status: PROCESSING | COMPLETED | FAILED
  - Result: Full response body (for COMPLETED)
  - Error: Error details (for FAILED)
  - Timestamp: For cleanup/expiry
  - Lock: For handling concurrent requests

TTL: Store for 24 hours (or longer for critical operations)
```

Key Properties

1. Safe to retry: Client can retry any request without fear **2. Exactly-once semantics:** Operation executes once regardless of retries **3. Response replay:** Retry gets same response as original **4. Time-bound:** Keys expire after reasonable period

4. Correct Design & Algorithm

Strategy 1: Database-Backed Idempotency Store

```
Table: idempotency_keys
  key VARCHAR PRIMARY KEY
  status ENUM('processing', 'completed', 'failed')
  result JSONB
  created_at TIMESTAMP
  expires_at TIMESTAMP

Algorithm:
  1. BEGIN TRANSACTION
  2. INSERT idempotency_key with status='processing'
     (or UPDATE if exists and check status)
  3. If key exists and status='completed': return stored result
  4. If key exists and status='processing': wait/retry
  5. Process request
  6. UPDATE status='completed', result=<result>
  7. COMMIT
  8. Return result
```

Strategy 2: Redis-Backed Idempotency

```
Redis keys:
  idem:{key}:status = "processing" | "completed" | "failed"
  idem:{key}:result = <json>
  TTL: 24 hours

Algorithm with Redis:
  1. SET idem:{key}:status "processing" NX EX 86400
  2. If SET failed (key exists):
     - GET idem:{key}:status
     - If "completed": return idem:{key}:result
     - If "processing": wait and retry
  3. Process request
  4. SET idem:{key}:result <result>
  5. SET idem:{key}:status "completed"
  6. Return result
```

Strategy 3: Event Sourcing Idempotency

```
Event store ensures idempotency naturally:
  - Event ID = idempotency key
  - Duplicate events rejected at write
  - Read from event store to get result
```

5. Full Production-Grade Implementation

```typescript
interface IdempotencyRecord {
  key: string;
  status: 'processing' | 'completed' | 'failed';
  result?: any;
```

```
  error?: string;
  createdAt: Date;
  expiresAt: Date;
}

class IdempotencyManager {
  constructor(
    private db: Database,
    private ttlSeconds: number = 86400  // 24 hours
  ) {}

  /**
   * Execute operation with idempotency
   */
  async execute<T>(
    idempotencyKey: string,
    operation: () => Promise<T>
  ): Promise<IdempotencyResult<T>> {
    // Check if key already exists
    const existing = await this.getRecord(idempotencyKey);

    if (existing) {
      switch (existing.status) {
        case 'completed':
          return {
            result: existing.result,
            isRetry: true,
            originalTimestamp: existing.createdAt,
          };

        case 'failed':
          throw new IdempotentOperationFailedError(
            existing.error || 'Operation failed previously',
            existing.createdAt
          );

        case 'processing':
          // Another request is processing, wait and retry
          return await this.waitAndRetry(idempotencyKey);
      }
    }

    // New request, create processing record
    try {
      await this.createProcessingRecord(idempotencyKey);
    } catch (err) {
      // Race condition: another request created record
      return this.execute(idempotencyKey, operation);
    }

    // Execute operation
    try {
```

```
    const result = await operation();

    // Store result
    await this.markCompleted(idempotencyKey, result);

    return {
      result,
      isRetry: false,
      originalTimestamp: new Date(),
    };
  } catch (error) {
    // Store error
    await this.markFailed(idempotencyKey, error.message);
    throw error;
  }
}

/**
 * Get existing idempotency record
 */
private async getRecord(key: string): Promise<IdempotencyRecord | null> {
  const record = await this.db.query(
    `SELECT * FROM idempotency_keys
     WHERE key = $1 AND expires_at > NOW()`,
    [key]
  );

  return record.rows[0] || null;
}

/**
 * Create processing record
 */
private async createProcessingRecord(key: string): Promise<void> {
  const expiresAt = new Date(Date.now() + this.ttlSeconds * 1000);

  try {
    await this.db.query(
      `INSERT INTO idempotency_keys (key, status, created_at, expires_at)
       VALUES ($1, 'processing', NOW(), $2)`,
      [key, expiresAt]
    );
  } catch (err) {
    if (err.code === '23505') {  // Unique constraint violation
      throw new DuplicateKeyError();
    }
    throw err;
  }
}

/**
 * Mark operation as completed
```

```
 */
private async markCompleted(key: string, result: any): Promise<void> {
  await this.db.query(
    `UPDATE idempotency_keys
     SET status = 'completed', result = $2
     WHERE key = $1`,
    [key, JSON.stringify(result)]
  );
}

/**
 * Mark operation as failed
 */
private async markFailed(key: string, error: string): Promise<void> {
  await this.db.query(
    `UPDATE idempotency_keys
     SET status = 'failed', error = $2
     WHERE key = $1`,
    [key, error]
  );
}

/**
 * Wait for concurrent request to complete
 */
private async waitAndRetry<T>(key: string): Promise<IdempotencyResult<T>> {
  const maxAttempts = 10;
  const delayMs = 100;

  for (let i = 0; i < maxAttempts; i++) {
    await this.sleep(delayMs);

    const record = await this.getRecord(key);

    if (!record) {
      throw new Error('Idempotency record disappeared');
    }

    if (record.status === 'completed') {
      return {
        result: record.result,
        isRetry: true,
        originalTimestamp: record.createdAt,
      };
    }

    if (record.status === 'failed') {
      throw new IdempotentOperationFailedError(
        record.error || 'Operation failed',
        record.createdAt
      );
    }
```

```
    }

    throw new Error('Timeout waiting for concurrent operation');
  }

  private sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
  }

  /**
   * Cleanup expired keys
   */
  async cleanup(): Promise<number> {
    const result = await this.db.query(
      `DELETE FROM idempotency_keys WHERE expires_at < NOW()`
    );
    return result.rowCount;
  }
}

interface IdempotencyResult<T> {
  result: T;
  isRetry: boolean;
  originalTimestamp: Date;
}

class IdempotentOperationFailedError extends Error {
  constructor(message: string, public originalTimestamp: Date) {
    super(message);
    this.name = 'IdempotentOperationFailedError';
  }
}

class DuplicateKeyError extends Error {
  constructor() {
    super('Duplicate idempotency key');
    this.name = 'DuplicateKeyError';
  }
}

// Express middleware
const idempotencyManager = new IdempotencyManager(db);

function requireIdempotency(req: Request, res: Response, next: NextFunction) {
  const idempotencyKey = req.headers['idempotency-key'] as string;

  if (!idempotencyKey) {
    return res.status(400).json({
      error: 'Missing Idempotency-Key header',
      message: 'POST requests must include Idempotency-Key header',
    });
  }
```

```
  if (idempotencyKey.length < 16 || idempotencyKey.length > 255) {
    return res.status(400).json({
      error: 'Invalid Idempotency-Key',
      message: 'Key must be 16-255 characters',
    });
  }

  req.idempotencyKey = idempotencyKey;
  next();
}

// Payment endpoint with idempotency
app.post('/api/payment', requireIdempotency, async (req, res) => {
  try {
    const result = await idempotencyManager.execute(
      req.idempotencyKey,
      async () => {
        // Actual payment processing
        const charge = await stripe.charges.create({
          amount: req.body.amount * 100,
          currency: 'usd',
          source: req.body.cardToken,
          idempotency_key: req.idempotencyKey,  // Stripe also supports idempotency!
        });

        // Deduct from user balance
        await db.query(
          'UPDATE accounts SET balance = balance - $1 WHERE user_id = $2',
          [req.body.amount, req.body.userId]
        );

        // Create order record
        const order = await db.query(
          'INSERT INTO orders (user_id, amount, stripe_charge_id) VALUES ($1, $2,
$3) RETURNING id',
          [req.body.userId, req.body.amount, charge.id]
        );

        return {
          orderId: order.rows[0].id,
          chargeId: charge.id,
          amount: req.body.amount,
        };
      }
    );

    // Add header indicating if this was a retry
    if (result.isRetry) {
      res.setHeader('Idempotent-Replayed', 'true');
      res.setHeader('Original-Request-Time',
result.originalTimestamp.toISOString());
```

```
    }

    res.json(result.result);
  } catch (error) {
    if (error instanceof IdempotentOperationFailedError) {
      res.status(409).json({
        error: 'Operation failed previously',
        message: error.message,
        originalTimestamp: error.originalTimestamp,
      });
    } else {
      res.status(500).json({ error: 'Payment failed' });
    }
  }
});

// Client implementation
class IdempotentClient {
  async makePayment(amount: number, userId: string, cardToken: string): Promise<any>
{
    const idempotencyKey = this.generateIdempotencyKey('payment', userId);

    const maxRetries = 3;
    let lastError: Error | null = null;

    for (let attempt = 0; attempt < maxRetries; attempt++) {
      try {
        const response = await fetch('/api/payment', {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
            'Idempotency-Key': idempotencyKey,
          },
          body: JSON.stringify({ amount, userId, cardToken }),
        });

        if (response.ok) {
          const wasRetry = response.headers.get('Idempotent-Replayed') === 'true';
          const result = await response.json();

          if (wasRetry) {
            console.log('Payment was already processed, retrieved cached result');
          }

          return result;
        }

        if (response.status === 409) {
          // Operation failed previously, don't retry
          throw new Error('Payment failed on previous attempt');
        }
```

```
        throw new Error(`HTTP ${response.status}`);
      } catch (error) {
        lastError = error;

        if (attempt < maxRetries - 1) {
          const backoff = Math.min(1000 * Math.pow(2, attempt), 5000);
          await this.sleep(backoff);
        }
      }
    }

    throw lastError;
  }

  private generateIdempotencyKey(operation: string, userId: string): string {
    // Use combination of operation, user, and timestamp
    // Key should be stable across retries within same logical operation
    const timestamp = Date.now();
    const randomness = Math.random().toString(36).substring(7);
    return `${operation}-${userId}-${timestamp}-${randomness}`;
  }

  private sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
  }
}

// Database schema
const schema = `
CREATE TABLE idempotency_keys (
  key VARCHAR(255) PRIMARY KEY,
  status VARCHAR(20) NOT NULL CHECK (status IN ('processing', 'completed',
'failed')),
  result JSONB,
  error TEXT,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  expires_at TIMESTAMP NOT NULL,
  INDEX idx_expires_at (expires_at)
);

-- Cleanup job (run periodically)
-- DELETE FROM idempotency_keys WHERE expires_at < NOW();
`;

// Periodic cleanup job
setInterval(async () => {
  const deleted = await idempotencyManager.cleanup();
  console.log(`Cleaned up ${deleted} expired idempotency keys`);
}, 3600000);  // Every hour
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Payment Processing

```
app.post('/api/subscribe', requireIdempotency, async (req, res) => {
  const result = await idempotencyManager.execute(req.idempotencyKey, async () => {
    const subscription = await stripe.subscriptions.create({
      customer: req.body.customerId,
      items: [{ price: req.body.priceId }],
      idempotency_key: req.idempotencyKey,
    });

    await db.query(
      'UPDATE users SET subscription_id = $1, subscribed_at = NOW() WHERE id = $2',
      [subscription.id, req.user.id]
    );

    return { subscriptionId: subscription.id };
  });

  res.json(result.result);
});
```

Pattern 2: Inventory Reservation

```
app.post('/api/cart/reserve', requireIdempotency, async (req, res) => {
  const result = await idempotencyManager.execute(req.idempotencyKey, async () => {
    const reserved = await db.query(
      `UPDATE inventory
       SET reserved = reserved + $1, available = available - $1
       WHERE product_id = $2 AND available >= $1
       RETURNING reserved, available`,
      [req.body.quantity, req.body.productId]
    );

    if (reserved.rowCount === 0) {
      throw new Error('Insufficient inventory');
    }

    return { reserved: true, remaining: reserved.rows[0].available };
  });

  res.json(result.result);
});
```

Pattern 3: Webhook Processing

```
app.post('/webhooks/stripe', async (req, res) => {
  const event = req.body;

  // Use webhook ID as idempotency key
  const idempotencyKey = `webhook-stripe-${event.id}`;

  try {
    await idempotencyManager.execute(idempotencyKey, async () => {
      await processStripeWebhook(event);
      return { processed: true };
    });

    res.json({ received: true });
  } catch (error) {
    // Even if processing fails, acknowledge receipt
    res.json({ received: true, error: error.message });
  }
});
```

7. Failure Modes & Edge Cases

Concurrent Requests

Problem: Two requests with same key arrive simultaneously.

Mitigation: Database unique constraint + retry logic handles this.

Key Expiry During Processing

Problem: Long-running operation, key expires mid-processing.

Mitigation: Set TTL longer than max operation time.

Partial Failures

Problem: Payment succeeds but database update fails.

Mitigation: Use transactions, or store partial state in idempotency record.

Client Key Generation

Problem: Client generates different keys for same logical operation.

Mitigation: Client must store key, reuse on retry.

8. Performance Characteristics & Tradeoffs

Storage Cost

- Store result for each idempotency key (24 hours)
- With 1M requests/day: 1M records in database
- Cleanup required

Latency Impact

- First request: +5ms (INSERT idempotency record)
- Retry: +2ms (SELECT + return cached result)
- Minimal overhead

Scalability

- Database bottleneck at high scale
- Use Redis for better performance
- Partition by key prefix

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

Mistake 1: Key Not Unique Enough

Fix: Include user ID, operation type, and timestamp in key.

Mistake 2: TTL Too Short

Fix: TTL must be longer than max processing time + retry window.

Mistake 3: Not Storing Failures

Fix: Store failed operations, return same error on retry.

Mistake 4: Client Generates New Key on Retry

Fix: Client must persist and reuse key across retries.

Mistake 5: No Cleanup

Fix: Periodic job to delete expired keys.

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

GET Requests

GET should be idempotent by nature (HTTP spec), no special handling needed.

Idempotent by Design

If operation is naturally idempotent (UPDATE SET status=X), don't add complexity.

High Cardinality

Don't use for operations with billions of unique keys (storage explosion).

11. Related Concepts (With Contrast)

Transactions

Difference: Transactions ensure atomicity. Idempotency prevents duplicates across retries.

Exactly-Once Semantics

Related: Idempotency enables exactly-once processing despite retries.

Deduplication

Difference: Deduplication detects duplicates. Idempotency returns same result.

12. Production Readiness Checklist

Implementation

- [ ] Idempotency table/store created
- [ ] Unique constraint on key
- [ ] TTL/expiry logic implemented
- [ ] Cleanup job scheduled

Client

- [ ] Client generates stable idempotency keys
- [ ] Client reuses keys on retry
- [ ] Client handles 409 (operation failed previously)

Monitoring

- [ ] Track idempotency hit rate (retries)
- [ ] Alert on high retry rate
- [ ] Monitor storage growth

Testing

- [ ] Test retry returns same result
- [ ] Test concurrent requests
- [ ] Test expired keys cleaned up
- [ ] Load test with retries