Proxy Integration - Proxy-Cheap Implementation Guide

Proxy-Cheap Service Overview

Service Configuration

- **Provider**: Proxy-Cheap Static Residential Proxies
- **Cost**: $1.27/month per dedicated IP
- **Target Configuration**: 5 static IPs for 5 pen name personas
- **Total Monthly Cost**: $6.35 + platform fees
- **Geographic Coverage**: Philippines (Manila, Cebu)

Proxy Types Available

- **Static Residential IPs**: Dedicated IP per persona (recommended)
- **Rotating Residential**: Shared IP pool (not suitable for persona consistency)
- **Mobile Proxies**: 4G/5G connections (premium option for advanced detection avoidance)

API Integration Architecture

Proxy-Cheap API Endpoints

```
interface ProxyCheapAPI {
  baseUrl: 'https://api.proxy-cheap.com/v1';
  authentication: {
    method: 'API_KEY';
    header: 'X-API-Key';
  };
}

// Core API endpoints for automation
interface ProxyCheapEndpoints {
  // Proxy management
  listProxies: 'GET /proxies';
  getProxy: 'GET /proxies/{proxyId}';
  createProxy: 'POST /proxies';
  deleteProxy: 'DELETE /proxies/{proxyId}';

  // Health monitoring
  testConnection: 'POST /proxies/{proxyId}/test';
```

```
  getStats: 'GET /proxies/{proxyId}/stats';

  // Geographic targeting
  listLocations: 'GET /locations';
  getLocationDetails: 'GET /locations/{countryCode}/
{city}';
}
```

Proxy Configuration Schema

```
interface ProxyConfiguration {
  // Proxy-Cheap identifiers
  proxyId: string; // Provider's internal ID
  proxyType: 'static_residential';

  // Connection details
  endpoint: {
    host: string; // proxy.proxy-cheap.com
    port: number; // 8080, 8081, etc.
    protocol: 'HTTP' | 'SOCKS5';
  };

  // Authentication
  credentials: {
    username: string;
    password: string; // Encrypted in database
  };

  // Geographic assignment
  location: {
    country: 'PH';
    city: 'Manila' | 'Cebu';
    timezone: 'Asia/Manila';
  };

  // Persona assignment
  assignedPersona?: {
    personaId: string;
    assignedAt: Date;
    lastUsed: Date;
  };
```

```
  // Health metrics
  health: {
    status: 'healthy' | 'degraded' | 'failed';
    lastCheck: Date;
    responseTimeMs: number;
    uptimePercentage: number;
  };
}
```

Proxy Management Service

Core Proxy Manager Class

```
class ProxyManager {
  private apiClient: ProxyCheapClient;
  private database: DatabaseConnection;
  private cache: Redis;

  constructor(config: ProxyConfig) {
    this.apiClient = new ProxyCheapClient(config.apiKey);
    this.database = config.database;
    this.cache = config.redis;
  }

  // Assign dedicated IP to persona
  async assignProxyToPersona(personaId: string, location:
'Manila' | 'Cebu'): Promise<ProxyAssignment> {
    // 1. Check available proxies in target location
    const availableProxies = await
this.getAvailableProxies(location);

    if (availableProxies.length === 0) {
      // 2. Create new proxy if none available
      const newProxy = await this.createProxy(location);
      availableProxies.push(newProxy);
    }

    // 3. Assign proxy to persona
    const proxy = availableProxies[0];
    await this.database.updateProxyAssignment(proxy.id, {
```

```
      assignedPersonaId: personaId,
      assignedAt: new Date()
    });

    // 4. Cache assignment for quick lookup
    await this.cache.setex(`persona:${personaId}:proxy`,
3600, JSON.stringify(proxy));

    return {
      personaId,
      proxyId: proxy.id,
      connectionDetails:
this.formatConnectionDetails(proxy)
    };
  }

  // Get proxy connection details for persona
  async getPersonaProxy(personaId: string):
Promise<ProxyConnection | null> {
    // Check cache first
    const cached = await this.cache.get(`persona:$
{personaId}:proxy`);
    if (cached) {
      return JSON.parse(cached);
    }

    // Fallback to database
    const assignment = await
this.database.getProxyByPersona(personaId);
    if (!assignment) return null;

    // Verify proxy is still healthy
    const healthCheck = await
this.testProxyConnection(assignment.proxyId);
    if (!healthCheck.healthy) {
      await this.handleUnhealthyProxy(assignment.proxyId,
personaId);
      return null;
    }
```

```
    return this.formatConnectionDetails(assignment);
  }

  // Health monitoring
  async monitorProxyHealth(): Promise<void> {
    const allProxies = await
this.database.getAllProxyAssignments();

    for (const proxy of allProxies) {
      try {
        const health = await
this.testProxyConnection(proxy.proxyId);

        await this.database.updateProxyHealth(proxy.id, {
          status: health.healthy ? 'healthy' : 'degraded',
          lastCheck: new Date(),
          responseTimeMs: health.responseTime,
          uptimePercentage: this.calculateUptime(proxy.id)
        });

        // Alert if proxy is failing
        if (!health.healthy) {
          await this.handleUnhealthyProxy(proxy.proxyId,
proxy.assignedPersonaId);
        }

      } catch (error) {
        console.error(`Health check failed for proxy $
{proxy.proxyId}:`, error);
        await this.markProxyAsFailed(proxy.id);
      }
    }
  }
}
```

Browser Integration with SOCKS5

```
interface BrowserProxyConfig {
  personaId: string;
  proxyConnection: ProxyConnection;
  browserOptions: PuppeteerLaunchOptions;
```

```
}

class PersonaBrowserManager {
  private activeInstances = new Map<string, Browser>();
  private proxyManager: ProxyManager;

  async launchPersonaBrowser(config: BrowserProxyConfig):
Promise<Browser> {
    const proxyConfig = config.proxyConnection;

    // Configure Puppeteer with SOCKS5 proxy
    const browserOptions: PuppeteerLaunchOptions = {
      headless: 'new',
      args: [
        `--proxy-server=socks5://${proxyConfig.host}:$
{proxyConfig.port}`,
        '--disable-web-security',
        '--disable-features=VizDisplayCompositor',
        '--no-sandbox',
        '--disable-setuid-sandbox'
      ],
      ignoreDefaultArgs: ['--disable-extensions'],
      ...config.browserOptions
    };

    const browser = await puppeteer.launch(browserOptions);

    // Authenticate with proxy
    const page = await browser.newPage();
    await page.authenticate({
      username: proxyConfig.credentials.username,
      password: proxyConfig.credentials.password
    });

    // Verify IP matches expected location
    await this.verifyProxyLocation(page,
proxyConfig.expectedLocation);

    // Cache browser instance
    this.activeInstances.set(config.personaId, browser);
```

```
    return browser;
  }

  private async verifyProxyLocation(page: Page,
expectedLocation: string): Promise<void> {
    try {
      // Check current IP and location
      await page.goto('https://ipapi.co/json/');
      const response = await page.evaluate(() =>
document.body.innerText);
      const ipData = JSON.parse(response);

      if (ipData.city !== expectedLocation) {
        throw new Error(`Proxy location mismatch. Expected:
${expectedLocation}, Got: ${ipData.city}`);
      }

      console.log(`✅ Proxy verified: ${ipData.ip} in $
{ipData.city}, ${ipData.country_name}`);

    } catch (error) {
      console.error('Proxy location verification failed:',
error);
      throw error;
    }
  }
}
```

Proxy Setup Automation

Initial Proxy Provisioning

```
class ProxyProvisioner {
  private proxyCheapClient: ProxyCheapClient;

  async setupPersonaProxies(personas: PersonaConfig[]):
Promise<ProxySetupResult[]> {
    const results: ProxySetupResult[] = [];
```

```
    for (const persona of personas) {
      try {
        // 1. Create static residential proxy in target
location
        const proxyRequest = {
          type: 'static_residential',
          location: {
            country: 'PH',
            city: persona.preferredLocation // Manila or
Cebu
          },
          duration: '30_days', // Monthly billing
          sticky_session: true // Maintain same IP for
entire month
        };

        const proxy = await
this.proxyCheapClient.createProxy(proxyRequest);

        // 2. Store proxy configuration in database
        await this.database.createProxyAssignment({
          proxyId: proxy.id,
          proxyIp: proxy.ip,
          proxyPort: proxy.port,
          proxyUsername: proxy.username,
          proxyPasswordEncrypted: await
this.encrypt(proxy.password),
          countryCode: 'PH',
          city: persona.preferredLocation,
          assignmentStatus: 'assigned',
          assignedPersonaId: persona.id,
          assignedAt: new Date(),
          monthlyCostUsd: 1.27
        });

        // 3. Test proxy connection
        const testResult = await
this.testProxyConnection(proxy);

        results.push({
```

```
          personaId: persona.id,
          proxyId: proxy.id,
          success: testResult.success,
          connectionDetails: proxy,
          error: testResult.error
        });

      } catch (error) {
        results.push({
          personaId: persona.id,
          success: false,
          error: error.message
        });
      }
    }

    return results;
  }

  // Cost monitoring and management
  async calculateMonthlyCosts(): Promise<ProxyCostSummary>
{
    const assignments = await
this.database.getActiveProxyAssignments();

    return {
      totalProxies: assignments.length,
      costPerProxy: 1.27,
      totalMonthlyCost: assignments.length * 1.27,
      breakdown: assignments.map(proxy => ({
        personaName: proxy.persona.penName,
        location: proxy.city,
        cost: 1.27,
        usage: proxy.monthlyRequests
      }))
    };
  }
}
```

Proxy Health Monitoring System

```
class ProxyHealthMonitor {
  private checkInterval = 15 * 60 * 1000; // 15 minutes
  private monitoringActive = false;

  startMonitoring(): void {
    if (this.monitoringActive) return;

    this.monitoringActive = true;

    setInterval(async () => {
      await this.runHealthChecks();
    }, this.checkInterval);

    console.log('✅ Proxy health monitoring started');
  }

  private async runHealthChecks(): Promise<void> {
    const activeProxies = await
this.database.getActiveProxyAssignments();

    const healthChecks = activeProxies.map(proxy =>
      this.checkProxyHealth(proxy)
    );

    const results = await Promise.allSettled(healthChecks);

    // Process results and handle failures
    results.forEach((result, index) => {
      const proxy = activeProxies[index];

      if (result.status === 'fulfilled' &&
result.value.healthy) {
        this.updateProxyStatus(proxy.id, 'healthy',
result.value.responseTime);
      } else {
        this.handleUnhealthyProxy(proxy.id,
proxy.assignedPersonaId);
      }
    });
  }
```

```
  private async checkProxyHealth(proxy: ProxyAssignment):
Promise<HealthResult> {
    const startTime = Date.now();

    try {
      // Test proxy connection with simple HTTP request
      const response = await this.makeProxyRequest(proxy,
'https://httpbin.org/ip');
      const responseTime = Date.now() - startTime;

      return {
        healthy: response.status === 200,
        responseTime,
        ip: response.data?.origin
      };

    } catch (error) {
      return {
        healthy: false,
        responseTime: Date.now() - startTime,
        error: error.message
      };
    }
  }

  private async handleUnhealthyProxy(proxyId: string,
personaId: string): Promise<void> {
    console.warn(`🚨 Unhealthy proxy detected: ${proxyId}
for persona: ${personaId}`);

    // 1. Mark proxy as degraded
    await this.database.updateProxyStatus(proxyId,
'degraded');

    // 2. Attempt to get replacement proxy
    const persona = await
this.database.getPersona(personaId);
    const replacementProxy = await
this.proxyManager.assignProxyToPersona(
```

```
      personaId,
      persona.fictionalLocation
    );

    // 3. Notify if replacement successful
    if (replacementProxy) {
      console.log(`✅ Replacement proxy assigned: ${
{replacementProxy.proxyId}`);
    } else {
      console.error(`❌ Failed to assign replacement proxy
for persona: ${personaId}`);
      // Could implement email/SMS alerts here
    }
  }
}
```

Error Handling & Failover

Proxy Connection Failures

```
interface ProxyFailoverStrategy {
  maxRetries: number;
  retryDelayMs: number;
  fallbackBehavior: 'pause_persona' | 'use_backup_proxy' |
'direct_connection';
}

class ProxyFailoverManager {
  private failoverConfig: ProxyFailoverStrategy = {
    maxRetries: 3,
    retryDelayMs: 5000,
    fallbackBehavior: 'use_backup_proxy'
  };

  async handleProxyFailure(
    personaId: string,
    failedProxyId: string,
    context: FailureContext
  ): Promise<FailoverResult> {
```

```
    console.log(`🔄 Handling proxy failure for persona ${personaId}`);

    // 1. Log failure for analysis
    await this.logProxyFailure(personaId, failedProxyId, context);

    // 2. Attempt failover based on strategy
    switch (this.failoverConfig.fallbackBehavior) {
      case 'use_backup_proxy':
        return await this.assignBackupProxy(personaId);

      case 'pause_persona':
        return await this.pausePersonaActivities(personaId);

      case 'direct_connection':
        console.warn('⚠️ Using direct connection - persona may be detectable');
        return { success: true, usingDirectConnection: true };

      default:
        throw new Error('Invalid failover strategy');
    }
  }

  private async assignBackupProxy(personaId: string): Promise<FailoverResult> {
    try {
      const persona = await this.database.getPersona(personaId);
      const backupProxy = await this.proxyManager.assignProxyToPersona(
        personaId,
        persona.fictionalLocation
      );

      return {
```

```
        success: true,
        newProxyId: backupProxy.proxyId,
        message: 'Backup proxy assigned successfully'
      };

    } catch (error) {
      return {
        success: false,
        error: `Backup proxy assignment failed: $
{error.message}`
      };
    }
  }
}
```

Security & Authentication

Proxy Credential Management

```
class ProxyCredentialManager {
  private encryptionKey: string;

  constructor(encryptionKey: string) {
    this.encryptionKey = encryptionKey;
  }

  async encryptProxyCredentials(credentials:
ProxyCredentials): Promise<string> {
    const cipher = crypto.createCipher('aes-256-gcm',
this.encryptionKey);
    const encrypted =
cipher.update(JSON.stringify(credentials), 'utf8', 'hex');
    const final = cipher.final('hex');
    const tag = cipher.getAuthTag();

    return JSON.stringify({
      encrypted: encrypted + final,
      tag: tag.toString('hex'),
      algorithm: 'aes-256-gcm'
    });
  }
```

```
  async decryptProxyCredentials(encryptedData: string):
Promise<ProxyCredentials> {
    const data = JSON.parse(encryptedData);
    const decipher = crypto.createDecipher('aes-256-gcm',
this.encryptionKey);

    decipher.setAuthTag(Buffer.from(data.tag, 'hex'));

    const decrypted = decipher.update(data.encrypted,
'hex', 'utf8');
    const final = decipher.final('utf8');

    return JSON.parse(decrypted + final);
  }
}
```

Performance Optimization

Proxy Connection Pooling

```
class ProxyConnectionPool {
  private pools = new Map<string, ProxyPool>();
  private maxConnectionsPerProxy = 5;

  async getConnection(personaId: string):
Promise<ProxyConnection> {
    const proxyId = await
this.getProxyIdForPersona(personaId);

    if (!this.pools.has(proxyId)) {
      this.pools.set(proxyId, new ProxyPool(proxyId,
this.maxConnectionsPerProxy));
    }

    const pool = this.pools.get(proxyId)!;
    return await pool.acquire();
  }

  async releaseConnection(connection: ProxyConnection):
Promise<void> {
```

```
    const pool = this.pools.get(connection.proxyId);
    if (pool) {
      await pool.release(connection);
    }
  }
}
```

Monitoring & Analytics

Proxy Performance Metrics

```
interface ProxyMetrics {
  proxyId: string;
  personaId: string;
  responseTimeMs: number;
  requestsPerHour: number;
  successRate: number;
  bandwidthUsageGB: number;
  costPerRequest: number;
  locationAccuracy: number; // % of requests from correct
location
}

class ProxyAnalytics {
  async generateDailyReport():
Promise<ProxyPerformanceReport> {
    const metrics = await this.collectDailyMetrics();

    return {
      date: new Date().toISOString().split('T')[0],
      totalRequests: metrics.reduce((sum, m) => sum +
m.requestsPerHour * 24, 0),
      averageResponseTime:
this.calculateAverage(metrics.map(m => m.responseTimeMs)),
      overallSuccessRate:
this.calculateAverage(metrics.map(m => m.successRate)),
      totalCost: metrics.reduce((sum, m) => sum +
(m.costPerRequest * m.requestsPerHour * 24), 0),
      performanceByPersona: metrics.map(m => ({
        personaName: m.personaId,
        performance: this.calculatePerformanceScore(m)
```

```
        }))
      };
    }
}
```