Technical Programming Guide - Spotify Follow-Swarm

Tech Stack Recommendations

Backend

- **Language:** Node.js (Express.js) or Python (FastAPI)
- **Database:** PostgreSQL for relational data, Redis for caching
- **Queue:** Bull (Node.js) or Celery (Python)
- **Authentication:** Passport.js or AuthLib

Frontend

- **Framework:** React or Next.js
- **Styling:** Tailwind CSS
- **State Management:** Redux or Zustand
- **Charts:** Recharts or Chart.js

Infrastructure

- **Hosting:** AWS EC2/ECS or Google Cloud Run
- **Database:** AWS RDS or Google Cloud SQL
- **Queue:** AWS SQS or Google Pub/Sub
- **Storage:** S3 for static assets

Database Schema

```sql
```

```sql
-- Users table
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    spotify_id VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE,
    display_name VARCHAR(255),
    profile_image_url TEXT,
    subscription_tier VARCHAR(50) DEFAULT 'free',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- OAuth tokens table
CREATE TABLE oauth_tokens (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    access_token TEXT NOT NULL,
    refresh_token TEXT NOT NULL,
    expires_at TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Follow relationships table
CREATE TABLE follows (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    follower_id UUID REFERENCES users(id) ON DELETE CASCADE,
    followed_id UUID REFERENCES users(id) ON DELETE CASCADE,
    status VARCHAR(50) DEFAULT 'pending', -- pending, completed, failed
    attempted_at TIMESTAMP,
    completed_at TIMESTAMP,
    error_message TEXT,
    UNIQUE(follower_id, followed_id)
);

-- Queue jobs table
CREATE TABLE queue_jobs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    job_type VARCHAR(50), -- follow_batch, refresh_token, etc.
    status VARCHAR(50) DEFAULT 'pending',
```

```
    payload JSONB,
    attempts INT DEFAULT 0,
    max_attempts INT DEFAULT 3,
    scheduled_for TIMESTAMP,
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
    error_message TEXT
);

-- Analytics table
CREATE TABLE analytics (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    event_type VARCHAR(100),
    event_data JSONB,
    created_at TIMESTAMP DEFAULT NOW()
);
```

Core API Endpoints

Authentication Endpoints

```javascript
// Spotify OAuth initiation
GET /auth/spotify
// Redirects to Spotify OAuth page

// OAuth callback
GET /auth/callback
// Handles Spotify callback, creates/updates user, stores tokens

// Logout
POST /auth/logout
// Clears session, optionally revokes tokens
```

User Management Endpoints

```javascript
```

```
// Get current user profile
GET /api/user/profile
Response: { user: {...}, stats: {...} }

// Update subscription
POST /api/user/subscription
Body: { tier: 'pro' }

// Get follow progress
GET /api/user/follow-progress
Response: { total: 1000, completed: 750, pending: 250 }
```

Follow Management Endpoints

```
javascript

// Trigger follow sync
POST /api/follows/sync
// Queues follow jobs for current user

// Get follow status
GET /api/follows/status
Response: { following: [], followers: [], pending: [] }

// Pause/resume follows
POST /api/follows/pause
POST /api/follows/resume
```

Spotify API Integration

OAuth Flow Implementation

```
javascript
```

```
// spotify-auth.js
const SpotifyWebApi = require('spotify-web-api-node');

const spotifyApi = new SpotifyWebApi({
  clientId: process.env.SPOTIFY_CLIENT_ID,
  clientSecret: process.env.SPOTIFY_CLIENT_SECRET,
  redirectUri: process.env.SPOTIFY_REDIRECT_URI
});

// Generate auth URL
function getAuthUrl() {
  const scopes = [
    'user-follow-modify',
    'user-follow-read',
    'user-read-private',
    'user-read-email'
  ];

  return spotifyApi.createAuthorizeURL(scopes, 'state-key');
}

// Exchange code for tokens
async function handleCallback(code) {
  const data = await spotifyApi.authorizationCodeGrant(code);

  return {
    accessToken: data.body['access_token'],
    refreshToken: data.body['refresh_token'],
    expiresIn: data.body['expires_in']
  };
}

// Refresh access token
async function refreshAccessToken(refreshToken) {
  spotifyApi.setRefreshToken(refreshToken);
  const data = await spotifyApi.refreshAccessToken();

  return {
    accessToken: data.body['access_token'],
    expiresIn: data.body['expires_in']
```

```
  };
}
```

Follow Engine Implementation

javascript

```
// follow-engine.js
class FollowEngine {
  constructor(spotifyApi, db) {
    this.spotifyApi = spotifyApi;
    this.db = db;
    this.maxFollowsPerHour = 30;
    this.batchSize = 50;
  }

  async processFollowBatch(userId) {
    // Get pending follows
    const pendingFollows = await this.db.query(
      `SELECT * FROM follows
       WHERE follower_id = $1 AND status = 'pending'
       LIMIT $2`,
      [userId, this.batchSize]
    );

    // Get user's token
    const token = await this.getUserToken(userId);
    this.spotifyApi.setAccessToken(token);

    // Process follows with throttling
    for (const follow of pendingFollows.rows) {
      try {
        await this.executeFollow(follow);
        await this.delay(this.calculateDelay());
      } catch (error) {
        await this.handleFollowError(follow, error);
      }
    }
  }

  async executeFollow(follow) {
    // Follow the artist
    await this.spotifyApi.followArtists([follow.spotify_id]);

    // Update database
    await this.db.query(
      `UPDATE follows
       SET status = 'completed', completed_at = NOW()
```

```
      WHERE id = $1`,
      [follow.id]
    );
  }

  calculateDelay() {
    // Random delay between 2-4 minutes
    const minDelay = 120000; // 2 minutes
    const maxDelay = 240000; // 4 minutes
    return Math.random() * (maxDelay - minDelay) + minDelay;
  }

  delay(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
  }
}
```

Queue System Architecture

Job Queue Implementation

javascript

```
// queue-manager.js
const Bull = require('bull');
const followQueue = new Bull('follow-queue', {
  redis: {
    host: process.env.REDIS_HOST,
    port: process.env.REDIS_PORT
  }
});

// Add follow job
async function addFollowJob(userId, priority = 0) {
  return await followQueue.add(
    'process-follows',
    { userId },
    {
      priority,
      attempts: 3,
      backoff: {
        type: 'exponential',
        delay: 60000 // Start with 1 minute
      }
    }
  );
}

// Process follow jobs
followQueue.process('process-follows', async (job) => {
  const { userId } = job.data;
  const engine = new FollowEngine(spotifyApi, db);

  await engine.processFollowBatch(userId);

  // Check if more follows pending
  const remaining = await checkRemainingFollows(userId);
  if (remaining > 0) {
    // Re-queue with delay
    await addFollowJob(userId, -1);
  }
});
```

Rate Limiting & Throttling

Implementation Strategy

javascript

```
// rate-limiter.js
class RateLimiter {
  constructor(redis) {
    this.redis = redis;
  }

  async checkLimit(userId, action) {
    const key = `rate:${userId}:${action}`;
    const hour = Math.floor(Date.now() / 3600000);
    const dayKey = `${key}:day:${Math.floor(Date.now() / 86400000)}`;

    // Check hourly limit
    const hourCount = await this.redis.incr(`${key}:${hour}`);
    await this.redis.expire(`${key}:${hour}`, 3600);

    if (hourCount > 30) {
      throw new Error('Hourly rate limit exceeded');
    }

    // Check daily limit
    const dayCount = await this.redis.incr(dayKey);
    await this.redis.expire(dayKey, 86400);

    if (dayCount > 500) {
      throw new Error('Daily rate limit exceeded');
    }

    return true;
  }

  async getWaitTime(userId, action) {
    // Calculate how long until rate limit resets
    const key = `rate:${userId}:${action}`;
    const hour = Math.floor(Date.now() / 3600000);
    const ttl = await this.redis.ttl(`${key}:${hour}`);

    return ttl > 0 ? ttl * 1000 : 0;
  }
}
```

Security Considerations

Token Security

javascript

```javascript
// Encrypt tokens at rest
const crypto = require('crypto');

function encryptToken(token) {
  const cipher = crypto.createCipher('aes-256-cbc', process.env.ENCRYPTION_KEY);
  let encrypted = cipher.update(token, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  return encrypted;
}

function decryptToken(encryptedToken) {
  const decipher = crypto.createDecipher('aes-256-cbc', process.env.ENCRYPTION_KEY);
  let decrypted = decipher.update(encryptedToken, 'hex', 'utf8');
  decrypted += decipher.final('utf8');
  return decrypted;
}
```

Input Validation

javascript

```
// validation.js
const Joi = require('joi');

const schemas = {
  subscription: Joi.object({
    tier: Joi.string().valid('free', 'pro', 'premium').required()
  }),

  followSync: Joi.object({
    immediate: Joi.boolean().default(false),
    priority: Joi.number().min(0).max(10).default(5)
  })
};

function validate(schema, data) {
  const result = schema.validate(data);
  if (result.error) {
    throw new ValidationError(result.error.details[0].message);
  }
  return result.value;
}
```

Monitoring & Logging

Metrics to Track

```
javascript
```

```javascript
// metrics.js
const prometheus = require('prom-client');

// Define metrics
const followsProcessed = new prometheus.Counter({
  name: 'follows_processed_total',
  help: 'Total number of follows processed',
  labelNames: ['status']
});

const apiLatency = new prometheus.Histogram({
  name: 'spotify_api_latency_seconds',
  help: 'Spotify API response time',
  labelNames: ['endpoint']
});

const queueSize = new prometheus.Gauge({
  name: 'queue_size',
  help: 'Current size of follow queue',
  labelNames: ['status']
});

// Track metrics
function trackFollow(status) {
  followsProcessed.inc({ status });
}

function trackApiCall(endpoint, duration) {
  apiLatency.observe({ endpoint }, duration);
}
```

Error Handling

Comprehensive Error Strategy

```javascript
```

```
// error-handler.js
class ErrorHandler {
  async handleError(error, context) {
    // Log error
    logger.error({
      message: error.message,
      stack: error.stack,
      context
    });

    // Categorize error
    if (error.statusCode === 429) {
      // Rate limit - back off
      await this.handleRateLimit(context);
    } else if (error.statusCode === 401) {
      // Token expired - refresh
      await this.refreshUserToken(context.userId);
    } else if (error.statusCode >= 500) {
      // Server error - retry
      await this.scheduleRetry(context);
    } else {
      // Client error - mark as failed
      await this.markAsFailed(context, error.message);
    }
  }

  async handleRateLimit(context) {
    // Exponential backoff
    const delay = Math.pow(2, context.attempts) * 60000;
    await this.scheduleRetry(context, delay);
  }
}
```

Testing Strategy

Unit Tests

javascript

```
// follow-engine.test.js
describe('FollowEngine', () => {
  it('should respect rate limits', async () => {
    const engine = new FollowEngine(mockApi, mockDb);
    const start = Date.now();

    await engine.processFollowBatch('user-123');

    const duration = Date.now() - start;
    expect(duration).toBeGreaterThan(120000); // Should take >2 min
  });

  it('should handle API errors gracefully', async () => {
    mockApi.followArtists.mockRejectedValue(new Error('API Error'));

    const engine = new FollowEngine(mockApi, mockDb);
    await engine.processFollowBatch('user-123');

    // Should mark as failed but not throw
    expect(mockDb.query).toHaveBeenCalledWith(
      expect.stringContaining('status = \'failed\''),
      expect.any(Array)
    );
  });
});
```