Contents

SECTION 14: CENTRALIZED ERROR LOGGING (v3.1.0)

14.1 Error Logging Database Schema

```sql
-- migrations/024_centralized_error_logging.sql

CREATE TABLE error_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES tenants(id),
    user_id UUID REFERENCES users(id),
    error_code VARCHAR(50) NOT NULL,
    error_type VARCHAR(50) NOT NULL,
    error_message TEXT NOT NULL,
    stack_trace TEXT,
    request_id VARCHAR(100),
    source_service VARCHAR(100),
    source_function VARCHAR(200),
    context JSONB DEFAULT '{}',
    severity VARCHAR(20) NOT NULL DEFAULT 'error',
    resolved BOOLEAN DEFAULT false,
    resolved_at TIMESTAMPTZ,
    resolved_by UUID REFERENCES users(id),
    resolution_notes TEXT,
    created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE error_patterns (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    pattern_name VARCHAR(100) NOT NULL,
    error_code_pattern VARCHAR(100),
    message_pattern TEXT,
    occurrence_count INTEGER DEFAULT 1,
    first_seen TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP,
    last_seen TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP,
    auto_resolve_enabled BOOLEAN DEFAULT false,
    auto_resolve_action JSONB
```

```
);

CREATE INDEX idx_error_logs_tenant ON error_logs(tenant_id, created_at DESC);
CREATE INDEX idx_error_logs_code ON error_logs(error_code);
CREATE INDEX idx_error_logs_unresolved ON error_logs(resolved) WHERE resolved = false;

ALTER TABLE error_logs ENABLE ROW LEVEL SECURITY;

-- Allow admins to see all errors, users only their own
CREATE POLICY error_logs_policy ON error_logs USING (
    tenant_id = current_setting('app.current_tenant_id')::UUID OR
    current_setting('app.user_role') = 'admin'
);
```

14.2 Error Logger Service

```
// packages/core/src/services/error-logger.ts

import { Pool } from 'pg';
import { SNSClient, PublishCommand } from '@aws-sdk/client-sns';

interface ErrorContext {
    tenantId?: string;
    userId?: string;
    requestId?: string;
    sourceService: string;
    sourceFunction: string;
    additionalContext?: Record<string, any>;
}

type ErrorSeverity = 'debug' | 'info' | 'warning' | 'error' | 'critical';

export class ErrorLogger {
    private pool: Pool;
    private sns: SNSClient;
    private alertTopicArn: string;

    constructor(pool: Pool, alertTopicArn: string) {
        this.pool = pool;
        this.sns = new SNSClient({});
        this.alertTopicArn = alertTopicArn;
    }

    async log(
        error: Error,
        context: ErrorContext,
        severity: ErrorSeverity = 'error'
    ): Promise<string> {
```

```
        const errorCode = this.extractErrorCode(error);
        const errorType = error.constructor.name;

        const result = await this.pool.query(`
            INSERT INTO error_logs (
                tenant_id, user_id, error_code, error_type, error_message,
                stack_trace, request_id, source_service, source_function,
                context, severity
            ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11)
            RETURNING id
        `, [
            context.tenantId,
            context.userId,
            errorCode,
            errorType,
            error.message,
            error.stack,
            context.requestId,
            context.sourceService,
            context.sourceFunction,
            JSON.stringify(context.additionalContext || {}),
            severity
        ]);

        const errorId = result.rows[0].id;

        // Check for patterns and auto-resolve
        await this.checkPatterns(errorCode, error.message);

        // Send alerts for critical errors
        if (severity === 'critical') {
            await this.sendAlert(errorId, error, context);
        }

        return errorId;
    }

    async getUnresolvedErrors(
        tenantId?: string,
        limit: number = 50
    ): Promise<any[]> {
        const whereClause = tenantId ? 'AND tenant_id = $2' : '';
        const params = tenantId ? [limit, tenantId] : [limit];

        const result = await this.pool.query(`
            SELECT * FROM error_logs
            WHERE resolved = false ${whereClause}
            ORDER BY
```

```
            CASE severity
                WHEN 'critical' THEN 1
                WHEN 'error' THEN 2
                WHEN 'warning' THEN 3
                ELSE 4
            END,
            created_at DESC
        LIMIT $1
    `, params);

    return result.rows;
}

async resolve(
    errorId: string,
    resolvedBy: string,
    notes: string
): Promise<void> {
    await this.pool.query(`
        UPDATE error_logs
        SET resolved = true, resolved_at = NOW(), resolved_by = $2, resolution_notes = $3
        WHERE id = $1
    `, [errorId, resolvedBy, notes]);
}

private extractErrorCode(error: Error): string {
    if ('code' in error) return String((error as any).code);
    if (error.message.includes('ECONNREFUSED')) return 'CONN_REFUSED';
    if (error.message.includes('timeout')) return 'TIMEOUT';
    if (error.message.includes('rate limit')) return 'RATE_LIMIT';
    return 'UNKNOWN';
}

private async checkPatterns(errorCode: string, message: string): Promise<void> {
    await this.pool.query(`
        INSERT INTO error_patterns (pattern_name, error_code_pattern, message_pattern)
        VALUES ($1, $2, $3)
        ON CONFLICT DO NOTHING
    `, [`pattern_${errorCode}`, errorCode, message.substring(0, 100)]);

    await this.pool.query(`
        UPDATE error_patterns
        SET occurrence_count = occurrence_count + 1, last_seen = NOW()
        WHERE error_code_pattern = $1
    `, [errorCode]);
}

private async sendAlert(errorId: string, error: Error, context: ErrorContext): Promise<voi
```

```
        await this.sns.send(new PublishCommand({
            TopicArn: this.alertTopicArn,
            Subject: `[RADIANT CRITICAL] ${error.message.substring(0, 50)}`,
            Message: JSON.stringify({
                errorId,
                message: error.message,
                service: context.sourceService,
                function: context.sourceFunction,
                tenantId: context.tenantId,
                timestamp: new Date().toISOString()
            })
        }));
    }
}
```