Contents

RADIANT v5.0.2 - Source Export Part 2: Lambda APIs

---

4. Grimoire API Handler

File: `packages/infrastructure/lambda/grimoire-api/index.ts`

Purpose: REST API handler for The Grimoire. Provides endpoints for CRUD operations on heuristics and statistics retrieval.

Endpoints: - `GET /heuristics` - List heuristics with optional domain/search filters - `POST /heuristics` - Add a new heuristic - `DELETE /heuristics/:id` - Remove a heuristic - `POST /heuristics/:id/reinforce` - Adjust confidence (positive/negative) - `GET /stats` - Get Grimoire statistics

```
/**
 * Grimoire API Lambda Handler
 * RADIANT v5.0.2 - System Evolution
 */

import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { success, handleError, noContent } from '../shared/response';
import {
  withSecureDBContext,
  extractAuthFromEvent,
  isTenantAdmin
} from '../shared/services/db-context.service';
import {
  ValidationError,
  UnauthorizedError,
  NotFoundError
} from '../shared/errors';

export async function handler(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  const method = event.httpMethod;
  const path = event.path;
  const pathParts = path.split('/').filter(Boolean);

  try {
    const authContext = extractAuthFromEvent(event);

    if (!authContext.tenantId) {
      return handleError(new UnauthorizedError('Tenant ID required'));
    }

    // Route: GET /heuristics
    if (method === 'GET' && pathParts[pathParts.length - 1] === 'heuristics') {
      return await listHeuristics(event, authContext);
    }

    // Route: POST /heuristics
    if (method === 'POST' && pathParts[pathParts.length - 1] === 'heuristics') {
      return await addHeuristic(event, authContext);
    }

    // Route: DELETE /heuristics/:id
    if (method === 'DELETE' && pathParts.includes('heuristics')) {
      const id = pathParts[pathParts.length - 1];
      return await deleteHeuristic(id, authContext);
    }

    // Route: POST /heuristics/:id/reinforce
    if (method === 'POST' && pathParts.includes('reinforce')) {
      const id = pathParts[pathParts.length - 2];
      return await reinforceHeuristic(id, event, authContext);
    }

    // Route: GET /stats
    if (method === 'GET' && pathParts[pathParts.length - 1] === 'stats') {
      return await getStats(authContext);
    }

    return handleError(new NotFoundError('Route not found'));
  } catch (error) {
    return handleError(error);
  }
}

async function listHeuristics(event: APIGatewayProxyEvent, authContext: any) {
  const domain = event.queryStringParameters?.domain;
  const search = event.queryStringParameters?.search;
  const limit = parseInt(event.queryStringParameters?.limit || '100', 10);

  return withSecureDBContext(authContext, async (client) => {
    let query = `
      SELECT id, domain, heuristic_text, confidence_score,
             source_execution_id, created_at, updated_at, expires_at
      FROM knowledge_heuristics
      WHERE expires_at > NOW()
    `;
    const params: any[] = [];
    let paramIndex = 1;

    if (domain) {
      query += ` AND domain = $${paramIndex++}`;
      params.push(domain);
    }

    if (search) {
      query += ` AND heuristic_text ILIKE $${paramIndex++}`;
      params.push(`%${search}%`);
    }

    query += ` ORDER BY confidence_score DESC, created_at DESC LIMIT $${paramIndex}`;
    params.push(limit);

    const result = await client.query(query, params);

    return success({ heuristics: result.rows });
  });
}

async function addHeuristic(event: APIGatewayProxyEvent, authContext: any) {
  const body = JSON.parse(event.body || '{}');
  const { domain, heuristic_text } = body;

  if (!heuristic_text?.trim()) {
    return handleError(new ValidationError('heuristic_text is required'));
  }

  const validDomains = ['general', 'medical', 'financial', 'legal', 'technical', 'creative'];
  const domainValue = validDomains.includes(domain) ? domain : 'general';

  return withSecureDBContext(authContext, async (client) => {
    const result = await client.query(`
      INSERT INTO knowledge_heuristics (tenant_id, domain, heuristic_text, confidence_score)
      VALUES ($1, $2, $3, 0.5)
      RETURNING id, domain, heuristic_text, confidence_score, created_at
    `, [authContext.tenantId, domainValue, heuristic_text.trim()]);

    return success({ heuristic: result.rows[0] }, 201);
  });
}

async function deleteHeuristic(id: string, authContext: any) {
  if (!isTenantAdmin(authContext)) {
    return handleError(new UnauthorizedError('Admin access required'));
  }
  return withSecureDBContext(authContext, async (client) => {
    const result = await client.query(
      'DELETE FROM knowledge_heuristics WHERE id = $1 RETURNING id',
      [id]
    );

    if (result.rowCount === 0) {
      return handleError(new NotFoundError('Heuristic not found'));
    }

    return noContent();
  });
}

async function reinforceHeuristic(id: string, event: APIGatewayProxyEvent, authContext: any) {
  const body = JSON.parse(event.body || '{}');
  const { positive } = body;
  const adjustment = positive ? 0.05 : -0.05;

  return withSecureDBContext(authContext, async (client) => {
    const result = await client.query(`
      UPDATE knowledge_heuristics
      SET confidence_score = GREATEST(0.1, LEAST(1.0, confidence_score + $1)),
          expires_at = CASE WHEN $2 THEN expires_at + INTERVAL '30 days' ELSE expires_at END
      WHERE id = $3
      RETURNING id, confidence_score
    `, [adjustment, positive, id]);

    if (result.rowCount === 0) {
      return handleError(new NotFoundError('Heuristic not found'));
    }

    return success({ heuristic: result.rows[0] });
  });
}

async function getStats(authContext: any) {
  return withSecureDBContext(authContext, async (client) => {
    const statsResult = await client.query(`
      SELECT
        COUNT(*) as total_heuristics,
        COUNT(*) FILTER (WHERE confidence_score >= 0.8) as total_high_confidence,
        COUNT(*) FILTER (WHERE expires_at < NOW() + INTERVAL '7 days') as total_expiring_soon
      FROM knowledge_heuristics
      WHERE expires_at > NOW()
    `);
    const byDomainResult = await client.query(`
      SELECT domain, COUNT(*) as total, AVG(confidence_score) as avg_confidence,
             COUNT(*) FILTER (WHERE confidence_score >= 0.8) as high_confidence,
             COUNT(*) FILTER (WHERE expires_at < NOW() + INTERVAL '7 days') as expiring_soon,
             MAX(created_at) as last_added
      FROM knowledge_heuristics
      WHERE expires_at > NOW()
      GROUP BY domain
    `);

    const stats = statsResult.rows[0];
    const byDomain: Record<string, any> = {};
    for (const row of byDomainResult.rows) {
      byDomain[row.domain] = {
        total: parseInt(row.total, 10),
        avg_confidence: parseFloat(row.avg_confidence),
        high_confidence: parseInt(row.high_confidence, 10),
        expiring_soon: parseInt(row.expiring_soon, 10),
        last_added: row.last_added
      };
    }

    return success({
      total_heuristics: parseInt(stats.total_heuristics, 10),
      total_high_confidence: parseInt(stats.total_high_confidence, 10),
      total_expiring_soon: parseInt(stats.total_expiring_soon, 10),
      by_domain: byDomain,
      domain_count: Object.keys(byDomain).length
    });
  });
}
```

---

5. Governor API Handler

File: `packages/infrastructure/lambda/governor-api/index.ts`

Purpose: REST API handler for the Economic Governor. Manages configuration and provides statistics.

Endpoints: - `GET /config` - Get domain configurations - `PUT /config` - Update domain mode - `GET /statistics` - Get savings statistics - `GET /recent` - Get recent routing decisions - `POST /analyze` - Analyze a prompt's complexity

```
/**
 * Governor API Lambda Handler
 * RADIANT v5.0.2 - System Evolution
 */
```

```
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { success, handleError } from '../shared/response';
import {
  withSecureDBContext,
  extractAuthFromEvent,
  isTenantAdmin
} from '../shared/services/db-context.service';
import {
  ValidationError,
  UnauthorizedError,
  NotFoundError
} from '../shared/errors';
import { EconomicGovernor, GovernorMode } from '../shared/services/governor';

const VALID_MODES: GovernorMode[] = ['performance', 'balanced', 'cost_saver', 'off'];

export async function handler(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  const method = event.httpMethod;
  const path = event.path;
  const pathParts = path.split('/').filter(Boolean);

  try {
    const authContext = extractAuthFromEvent(event);

    if (!authContext.tenantId) {
      return handleError(new UnauthorizedError('Tenant ID required'));
    }

    // Route: GET /config
    if (method === 'GET' && pathParts[pathParts.length - 1] === 'config') {
      return await getConfig(authContext);
    }

    // Route: PUT /config
    if (method === 'PUT' && pathParts[pathParts.length - 1] === 'config') {
      return await updateConfig(event, authContext);
    }

    // Route: GET /statistics
    if (method === 'GET' && pathParts[pathParts.length - 1] === 'statistics') {
      return await getStatistics(event, authContext);
    }

    // Route: GET /recent
    if (method === 'GET' && pathParts[pathParts.length - 1] === 'recent') {
      return await getRecentDecisions(event, authContext);
    }
    // Route: POST /analyze
    if (method === 'POST' && pathParts[pathParts.length - 1] === 'analyze') {
      return await analyzePrompt(event, authContext);
    }

    return handleError(new NotFoundError('Route not found'));
  } catch (error) {
    return handleError(error);
  }
}

async function getConfig(authContext: any) {
  return withSecureDBContext(authContext, async (client) => {
    const result = await client.query(`
      SELECT domain, governor_mode as mode, updated_at
      FROM decision_domain_config
      ORDER BY domain
    `);

    return success({ domains: result.rows });
  });
}

async function updateConfig(event: APIGatewayProxyEvent, authContext: any) {
  if (!isTenantAdmin(authContext)) {
    return handleError(new UnauthorizedError('Admin access required'));
  }

  const body = JSON.parse(event.body || '{}');
  const { domain, mode } = body;

  if (!domain) {
    return handleError(new ValidationError('domain is required'));
  }

  if (!VALID_MODES.includes(mode)) {
    return handleError(new ValidationError(`mode must be one of: ${VALID_MODES.join(', ')}`));
  }

  return withSecureDBContext(authContext, async (client) => {
    await client.query(`
      INSERT INTO decision_domain_config (tenant_id, domain, governor_mode, updated_at)
      VALUES ($1, $2, $3, NOW())
      ON CONFLICT (tenant_id, domain)
      DO UPDATE SET governor_mode = $3, updated_at = NOW()
    `, [authContext.tenantId, domain, mode]);

    return success({ domain, mode, updated: true });
  });
}

async function getStatistics(event: APIGatewayProxyEvent, authContext: any) {
  const days = parseInt(event.queryStringParameters?.days || '30', 10);

  return withSecureDBContext(authContext, async (client) => {
    const summaryResult = await client.query(`
      SELECT
        COUNT(*) as total_decisions,
        AVG(complexity_score) as avg_complexity,
        SUM(savings_amount) as total_savings,
        COUNT(*) FILTER (WHERE original_model != selected_model) as model_swaps,
        COUNT(*) FILTER (WHERE complexity_score <= 4) as simple_tasks,
        COUNT(*) FILTER (WHERE complexity_score BETWEEN 5 AND 8) as medium_tasks,
        COUNT(*) FILTER (WHERE complexity_score >= 9) as complex_tasks
      FROM governor_savings_log
      WHERE created_at > NOW() - INTERVAL '1 day' * $1
    `, [days]);

    const dailyResult = await client.query(`
      SELECT
        DATE(created_at) as day,
        COUNT(*) as decisions,
        SUM(savings_amount) as savings,
        AVG(complexity_score) as avg_complexity
      FROM governor_savings_log
      WHERE created_at > NOW() - INTERVAL '1 day' * $1
      GROUP BY DATE(created_at)
      ORDER BY day DESC
    `, [days]);

    const byModeResult = await client.query(`
      SELECT
        governor_mode as mode,
        COUNT(*) as count,
        SUM(savings_amount) as savings
      FROM governor_savings_log
      WHERE created_at > NOW() - INTERVAL '1 day' * $1
      GROUP BY governor_mode
    `, [days]);

    const summary = summaryResult.rows[0];

    return success({
      period: { days },
      summary: {
        totalDecisions: parseInt(summary.total_decisions, 10),
        avgComplexity: parseFloat(summary.avg_complexity) || 0,
        totalSavings: parseFloat(summary.total_savings) || 0,
        modelSwaps: parseInt(summary.model_swaps, 10),
        taskDistribution: {
          simple: parseInt(summary.simple_tasks, 10),
          medium: parseInt(summary.medium_tasks, 10),
          complex: parseInt(summary.complex_tasks, 10)
        }
      },
      daily: dailyResult.rows,
      byMode: byModeResult.rows
    });
  });
}

async function getRecentDecisions(event: APIGatewayProxyEvent, authContext: any) {
  const limit = parseInt(event.queryStringParameters?.limit || '20', 10);

  return withSecureDBContext(authContext, async (client) => {
    const result = await client.query(`
      SELECT
        id, execution_id, original_model, selected_model,
        complexity_score, savings_amount, governor_mode as mode,
        reason, created_at
      FROM governor_savings_log
      ORDER BY created_at DESC
      LIMIT $1
    `, [limit]);

    return success({
      decisions: result.rows.map(row => ({
        id: row.id,
        executionId: row.execution_id,
        originalModel: row.original_model,
        selectedModel: row.selected_model,
        complexityScore: row.complexity_score,
        savingsAmount: parseFloat(row.savings_amount),
        mode: row.mode,
        reason: row.reason,
        createdAt: row.created_at
      }))
    });
  });
}

async function analyzePrompt(event: APIGatewayProxyEvent, authContext: any) {
  const body = JSON.parse(event.body || '{}');
  const { prompt, model, domain } = body;
  if (!prompt) {
    return handleError(new ValidationError('prompt is required'));
  }

  const governor = new EconomicGovernor();
  const decision = await governor.evaluateTask(
    { type: 'analyze', prompt, context: {} },
    { agent_id: 'api', role: 'analyzer', model: model || 'gpt-4o' },
    domain || 'general'
  );

  return success({
    complexityScore: decision.complexityScore,
    recommendedModel: decision.selectedModel,
    originalModel: decision.originalModel,
    estimatedSavings: decision.estimatedSavings,
    reason: decision.reason
  });
}
```

Continued in GRIMOIRE-GOVERNOR-SOURCE-PART3.md