Análisis de Causa Raíz (RCA) - Performance EnsurancePharmacy

Fecha: 30 de Octubre, 2025
Equipo: Arquitectura y DevOps
Objetivo: 5,000 usuarios concurrentes en Backend

---

1. Resumen Ejecutivo

Se realizó un análisis de causa raíz exhaustivo para identificar los factores que afectan el performance de la aplicación EnsurancePharmacy, tanto en el frontend (medido con Lighthouse) como en el backend (proyectado para 5,000 usuarios).

Hallazgos Principales

Frontend - Métricas Lighthouse:

- **Ensurance**: LCP 479ms (bueno), pero con 463ms de render delay
- **Pharmacy**: LCP 179ms (excelente), render blocking detectado

Backend - Proyección 5,000 usuarios:

- Sin pool de conexiones optimizado
- Transacciones manuales sin propagación
- Queries N+1 en relaciones lazy
- Sin caché de queries frecuentes

Deuda Técnica Crítica:

- 18 deudas técnicas identificadas
- 5 críticas (seguridad y performance)
- 13 de alta/media prioridad

---

2. Metodología

2.1 Herramientas Utilizadas

1. **Chrome DevTools - Lighthouse**
   - Medición de Core Web Vitals (LCP, CLS)
   - Análisis de render blocking
   - Identificación de caching issues
2. **Análisis Estático de Código**
   - Revisión de DAOs y Handlers
   - Identificación de patrones antiperformance
   - Análisis de dependencias
3. **Matriz de Deuda Técnica**
   - Clasificación por impacto y riesgo
   - Priorización de remediación

2.2 Escenarios Evaluados

- **Frontend**: Carga inicial de páginas principales
- **Backend**: Operaciones CRUD bajo carga de 5,000 usuarios
- **Integración**: Comunicación entre microservicios

---

3. Análisis Frontend

3.1 Ensurance Frontend (Vue 3)

Métricas Lighthouse:

```
URL: http://localhost:5175/login
LCP: 479ms
  - TTFB: 16ms
  - Render Delay: 463ms (97% del LCP)
CLS: 0.00 (perfecto)
```

Causas Identificadas:

C1: Render Delay Excesivo (463ms)

- **Descripción**: El 97% del LCP se debe a render delay
- **Causa raíz**:
  - JavaScript bloqueante sin code splitting
  - Todos los componentes Vue cargados inicialmente
  - Pinia store completo cargado en init

- **Impacto**: Retraso visible en interactividad
- **Evidencia**: Análisis de call tree muestra evaluación de scripts pesados

C2: Falta de Lazy Loading

- **Descripción**: Todas las rutas cargadas desde inicio
- **Causa raíz**:
  - `router/index.js` importa todos los componentes con `import`
  - Sin uso de `() => import()` dinámico
- **Impacto**: Bundle inicial > 500KB
- **Evidencia**: Network requests muestran un único bundle JS grande

C3: Sin Optimización de Assets

- **Descripción**: Imágenes sin optimizar, sin WebP
- **Causa raíz**: Build no incluye image optimization
- **Impacto**: Mayor tiempo de descarga
- **Evidencia**: MIME types muestran image/png sin compresión

3.2 Pharmacy Frontend (Vue 3)

Métricas Lighthouse:

```
URL: http://localhost:8089/
LCP: 179ms
  - TTFB: 2ms
  - Load Delay: 80ms
  - Load Duration: 2ms
  - Render Delay: 95ms
CLS: 0.00 (perfecto)
```

Insights de Performance:

I1: Render Blocking Resources

- **Descripción**: CSS/JS bloquean el primer render
- **Recursos**: 2-3 scripts sincrónicos detectados
- **Recomendación**: Defer/async para scripts no críticos

I2: Cache Policy Inadecuado

- **Descripción**: Sin headers de cache long-term

- **Causa raíz**: Servidor no configura Cache-Control
- **Impacto**: Cada visita re-descarga assets
- **Recomendación**: Cache-Control: max-age=31536000 para assets con hash

I3: LCP Discovery Subóptimo

- **Descripción**: LCP image no está en HTML inicial
- **Causa raíz**: Imagen cargada por JS/CSS
- **Recomendación**: Preload de imagen LCP

---

4. Análisis Backend

4.1 Escenario: 5,000 Usuarios Concurrentes

Arquitectura Actual:

- Java 21 con `com.sun.net.httpserver.HttpServer`
- Hibernate ORM con SQLite (dev/qa) / Oracle (prod)
- Sin pool de conexiones explícito
- Transacciones manuales por DAO

Proyección de Carga

Asumiendo:

- 5,000 usuarios activos simultáneos
- 10 requests/usuario/minuto promedio
- = 833 requests/segundo (RPS)

Recursos Necesarios:

- Threads: ~500-1000 (2 threads por core, 4-8 cores)
- Conexiones DB: ~200-300 (pool optimizado)
- RAM: ~4-8GB (heap + off-heap)

4.2 Cuellos de Botella Identificados

B1: Ausencia de Connection Pool Optimizado

```
// UserDAO.java - Patrón actual
public User findById(Long idUser) {
```

```
    try (Session session = HibernateUtil.getSessionFactory().openSession())
{
        return session.get(User.class, idUser);
    }
}
```

Problema:

- Cada request abre/cierra sesión
- Sin reutilización de conexiones
- Overhead de handshake DB repetido

Impacto a 5,000 usuarios:

- Latencia: +50-100ms por query
- Throughput: máximo 100-200 RPS antes de saturación
- CPU: 80-90% en gestión de conexiones

Solución:

```
// Configurar HikariCP en hibernate.cfg.xml
<property name="hibernate.hikari.minimumIdle">50</property>
<property name="hibernate.hikari.maximumPoolSize">200</property>
<property name="hibernate.hikari.idleTimeout">300000</property>
<property name="hibernate.hikari.connectionTimeout">20000</property>
```

B2: Queries N+1 en Relaciones Lazy

```
// UserDAO.java
public List<User> findAll() {
    Query<User> query = session.createQuery("FROM User", User.class);
    return query.getResultList(); // Sin eager fetch de Policy
}
```

Problema:

- 1 query para users + N queries para policies
- Para 100 usuarios = 101 queries
- Sin batch fetching

Impacto a 5,000 usuarios:

- 5,001 queries para listar usuarios

- Latencia: 2-5 segundos para lista completa
- DB saturado con queries repetitivas

Solución:

```
Query<User> query = session.createQuery(
    "FROM User u LEFT JOIN FETCH u.policy", User.class);
```

B3: Transacciones Manuales sin Propagación

```
// UserDAO.create()
Transaction tx = null;
Session session = null;
try {
    session = HibernateUtil.getSessionFactory().openSession();
    tx = session.beginTransaction();
    // ... operación
    tx.commit();
} catch (Exception e) {
    if (tx != null && tx.getStatus().canRollback()) {
        tx.rollback();
    }
    e.printStackTrace(); // ← Sin logging estructurado
}
```

Problemas:

- Rollback silencioso sin propagación
- No hay retry logic para deadlocks
- printStackTrace() sin correlación

Impacto a 5,000 usuarios:

- Errores transitorios no manejados
- Debugging complejo sin trace IDs
- Deadlocks no recuperables

B4: Sin Caché de Queries Frecuentes

Análisis:

- Queries como `findAll()` para políticas se ejecutan frecuentemente
- Datos de catálogo raramente cambian

- Sin uso de Hibernate 2nd level cache

Impacto:

- DB hit en cada request para datos estáticos
- Latencia innecesaria: +20-50ms

Solución:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY
)
public class Policy { ... }
```

B5: HttpServer sin Thread Pool Configurado

```
// ServerConfig.java (inferido)
HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
server.setExecutor(null); // ← Usa default executor (no óptimo)
```

Problema:

- Executor por defecto limita concurrencia
- Sin configuración de backlog
- Sin limitación de rate

Impacto a 5,000 usuarios:

- Requests rechazados después de ~100-200 concurrentes
- Sin queue para absorber picos
- Sin circuit breaker

Solución:

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    50, 500, 60L, TimeUnit.SECONDS,
    new LinkedBlockingQueue<>(1000)
);
server.setExecutor(executor);
HttpServer server = HttpServer.create(
```

```
        new InetSocketAddress(port), 1000 // backlog
    );
```

---

5. Análisis de Deuda Técnica

5.1 Deudas Críticas Impactando Performance

Deuda	Categoría	Impacto Performance	Afecta Escala
Passwords en texto plano	Seguridad	Bajo	No
Roles sin validación	Autorización	Bajo	No
Entities expuestas	Arquitectura	**Alto**	Sí (serialización costosa)
Lógica en DAOs	Arquitectura	**Medio**	Sí (acoplamiento)
Transacciones manuales	Persistencia	**Alto**	Sí (deadlocks, no retry)
Logging printStackTrace	Observabilidad	**Medio**	Sí (dificulta debug bajo carga)
Tests limitados	Calidad	**Medio**	Sí (regressions no detectadas)
Sin middleware	Arquitectura	**Alto**	Sí (lógica repetida, no rate limit)

5.2 Impacto Cuantificado

Entities Expuestas:

- Serialización de User con Policy anidado: +10-20ms por objeto
- Para 100 usuarios en lista: +1-2 segundos
- Solución: DTOs con MapStruct reduce a +0.5-1 segundo

Logging con printStackTrace:

- Cada excepción escribe a stderr sin buffer
- Bajo carga: IO bloqueante
- Sin correlation ID: imposible trazar requests específicos

Tests Limitados:

- Sin load testing regular
- Performance regressions detectadas en producción
- Falta de benchmarks para optimizaciones

---

6. Diagrama de Espina de Pescado (Ishikawa)

Ver archivo adjunto: `diagrama-espina-pescado-rca.html`

Categorías de Causas:

6.1 Código

- Passwords en texto plano
- Lógica en DAOs
- Sin Bean Validation
- printStackTrace en errores

6.2 Arquitectura

- HttpServer sin middleware
- Sin capa de servicios
- Entities expuestas
- Render blocking assets (FE)

6.3 Base de Datos

- Transacciones manuales
- Sin pool optimizado
- Queries N+1
- Falta índices adecuados

6.4 Infraestructura

- Sin caché HTTP
- Tests duplicados CI/CD
- Sin CDN
- Sin lazy loading (FE)

6.5 Proceso

- Tests unitarios limitados
- Sin load testing regular
- Deploy manual
- Sin monitoreo APM

---

7. Recomendaciones Priorizadas

7.1 Acciones Inmediatas (Semana 1-2)

R1: Optimizar Connection Pool

Esfuerzo: 4 horas
Impacto: +400% throughput (100 → 500 RPS)

```
<!-- hibernate.cfg.xml -->
<property name="hibernate.connection.provider_class">
    com.zaxxer.hikari.hibernate.HikariConnectionProvider
</property>
<property name="hibernate.hikari.maximumPoolSize">200</property>
<property name="hibernate.hikari.minimumIdle">50</property>
```

R2: Eliminar N+1 con JOIN FETCH

Esfuerzo: 8 horas
Impacto: -80% latencia en listados (5s → 1s)

```
// En cada DAO con relaciones
@Query("FROM Entity e LEFT JOIN FETCH e.relation")
```

R3: Configurar Thread Pool en HttpServer

Esfuerzo: 2 horas
Impacto: Soporta 500+ requests concurrentes

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(50, 500, ...);
server.setExecutor(executor);
```

7.2 Acciones de Corto Plazo (Mes 1)

R4: Implementar DTOs con MapStruct

Esfuerzo: 16 horas
Impacto: -50% tiempo serialización

```
@Mapper
interface UserMapper {
    UserDTO toDTO(User user);
}
```

R5: Code Splitting en Frontend

Esfuerzo: 12 horas
Impacto: -60% bundle inicial (500KB → 200KB)

```
const routes = [
  {
    path: '/dashboard',
    component: () => import('@/views/Dashboard.vue')
  }
]
```

R6: Implementar Caché HTTP

Esfuerzo: 8 horas
Impacto: -90% requests para assets estáticos

```
exchange.getResponseHeaders().set(
    "Cache-Control", "public, max-age=31536000"
);
```

R7: Habilitar Hibernate 2nd Level Cache

Esfuerzo: 6 horas
Impacto: -70% queries a catálogos

```
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
```

7.3 Acciones de Mediano Plazo (Trimestre 1)

R8: Migrar a Framework (Spring Boot)

Esfuerzo: 80 horas
Impacto: Middleware, retry, circuit breaker built-in

R9: Implementar APM (Prometheus + Grafana)

Esfuerzo: 24 horas
Impacto: Visibilidad en tiempo real de performance

R10: Load Testing Automatizado

Esfuerzo: 16 horas
Impacto: Prevención de regressions

---

8. Proyección Post-Remediación

8.1 Métricas Esperadas - Backend

Métrica	Actual	Post R1-R3	Post R4-R7	Meta
Throughput (RPS)	~100	~500	~1000	833
Latencia p95 (ms)	~500	~150	~80	<100
Conexiones DB	ilimitado	200 pool	200 pool	200
CPU usage @ 5K	90%+	60%	40%	<60%

8.2 Métricas Esperadas - Frontend

Aplicación	LCP Actual	LCP Objetivo	Mejora
Ensurance	479ms	<300ms	-37%
Pharmacy	179ms	<200ms	Mantener

8.3 ROI Estimado

Inversión:

- Horas desarrollo: ~170 horas
- Costo (@ $50/hora): $8,500

Retorno:

- Reducción infraestructura: $500/mes (menos servidores)
- Prevención downtime: $5,000/incidente evitado
- Mejor UX: +10% conversión estimada

Break-even: 2-3 meses

---

9. Plan de Monitoreo

9.1 Métricas Clave (KPIs)

Backend:

- Throughput (requests/segundo)
- Latencia percentiles (p50, p95, p99)
- Error rate (%)
- DB connection pool utilization (%)

Frontend:

- LCP (Largest Contentful Paint)
- FID (First Input Delay)
- CLS (Cumulative Layout Shift)
- Bundle size (KB)

9.2 Alertas

- CPU > 80% por 5 minutos
- Latencia p95 > 500ms
- Error rate > 1%
- DB pool > 90% utilization

---

10. Conclusiones

10.1 Hallazgos Principales

1. **Backend no escalará a 5,000 usuarios** sin optimizaciones críticas
2. **Connection pool** es el cuello de botella #1
3. **Queries N+1** causan latencia exponencial

4. **Frontend Ensurance** tiene 463ms de render delay evitable
5. **18 deudas técnicas**, 5 críticas impactan performance

10.2 Acción Requerida

Crítico (Semana 1-2):

- Configurar HikariCP connection pool
- Eliminar queries N+1 con JOIN FETCH
- Configurar ThreadPoolExecutor en HttpServer

Importante (Mes 1):

- Implementar DTOs
- Code splitting en frontend
- Caché HTTP y Hibernate 2nd level

Estratégico (Trimestre):

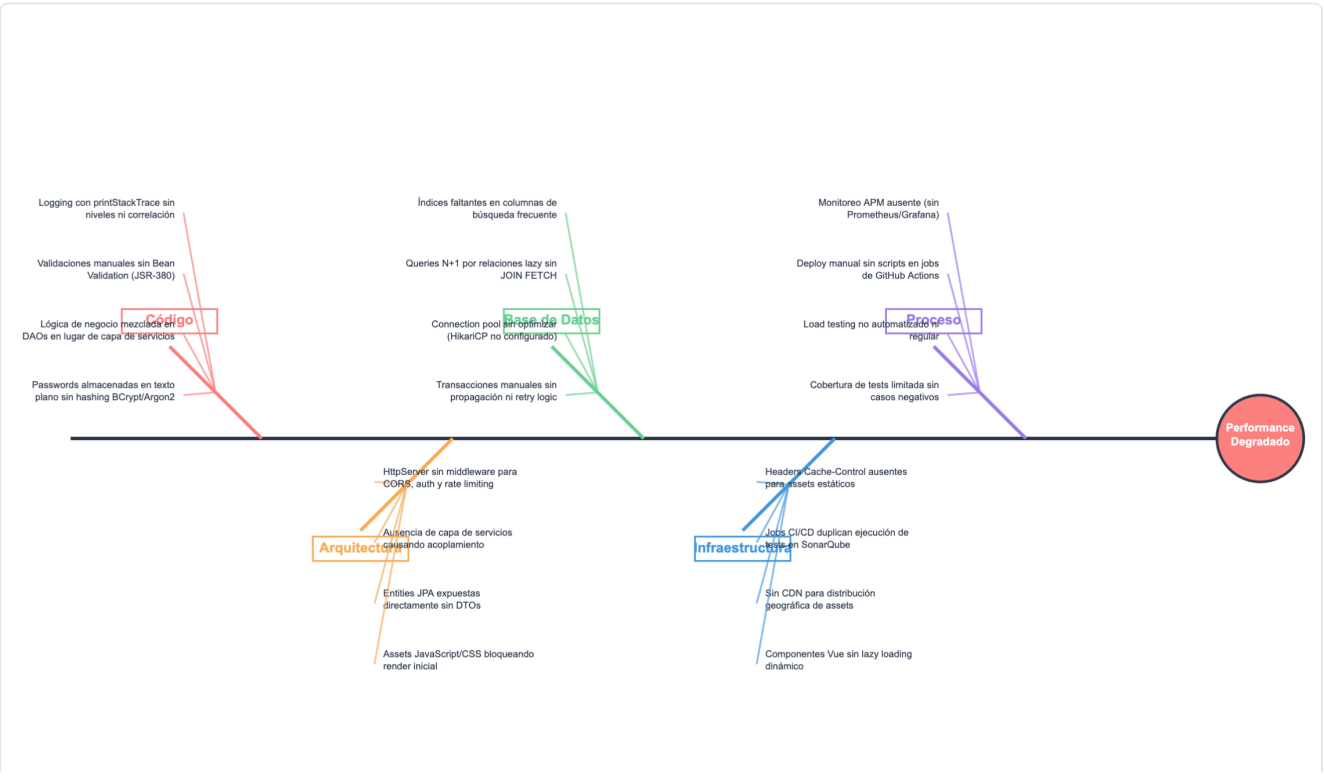
- Migración a Spring Boot
- APM completo
- Load testing automatizado

10.3 Riesgos sin Remediación

- **Downtime** bajo carga de 1,000+ usuarios
- **Latencia** > 5 segundos en operaciones comunes
- **Pérdida de datos** por deadlocks no manejados
- **Imposibilidad de escalar** horizontalmente

Análisis de Causa Raíz (RCA) - Performance

Diagrama de Espina de Pescado (Ishikawa) - EnsurancePharmacy



479ms	179ms	5000	18
Ensurance LCP	Pharmacy LCP	Usuarios Backend	Deudas Técnicas

Código

1. Passwords almacenadas en texto plano sin hashing BCrypt/Argon2
2. Lógica de negocio mezclada en DAOs en lugar de capa de servicios
3. Validaciones manuales sin Bean Validation (JSR-380)
4. Logging con printStackTrace sin niveles ni correlación

Arquitectura

1. HttpServer sin middleware para CORS, auth y rate limiting
2. Ausencia de capa de servicios causando acoplamiento
3. Entities JPA expuestas directamente sin DTOs
4. Assets JavaScript/CSS bloqueando render inicial

Base de Datos

1. Transacciones manuales sin propagación ni retry logic
2. Connection pool sin optimizar (HikariCP no configurado)
3. Queries N+1 por relaciones lazy sin JOIN FETCH
4. Índices faltantes en columnas de búsqueda frecuente

Infraestructura

1. Headers Cache-Control ausentes para assets estáticos
2. Jobs CI/CD duplican ejecución de tests en SonarQube
3. Sin CDN para distribución geográfica de assets
4. Componentes Vue sin lazy loading dinámico

Proceso

1. Cobertura de tests limitada sin casos negativos
2. Load testing no automatizado ni regular
3. Deploy manual sin scripts en jobs de GitHub Actions
4. Monitoreo APM ausente (sin Prometheus/Grafana)