# Guía Paso a Paso: Optimización de APIs

## 🎯 Objetivo

Optimizar las APIs más pesadas de INMOVA para alcanzar tiempos de respuesta < 500ms usando:
- Redis cache para datos frecuentes
- Queries Prisma optimizadas
- Monitoring de performance

---

## 📊 Prioridades de Optimización

### APIs a Optimizar Primero (Alto Impacto)

1. ⭐⭐⭐ **Críticas (Optimizar YA)**
   - `/api/dashboard` - Se accede en cada visita
   - `/api/buildings` - Lista principal, muy frecuente
   - `/api/units` - Datos usados en múltiples vistas
   - `/api/payments` - Dashboard de pagos
   - `/api/contracts` - Gestión de contratos

2. ⭐⭐ **Importantes (Optimizar pronto)**
   - `/api/analytics/*` - Reportes y estadísticas
   - `/api/tenants` - Lista de inquilinos
   - `/api/maintenance` - Solicitudes de mantenimiento
   - `/api/expenses` - Gastos y finanzas

3. ⭐ **Moderadas (Optimizar si hay tiempo)**
   - Otras APIs con queries complejas
   - APIs que se llaman frecuentemente

---

## 🛠️ Patrón de Optimización

### Paso 1: Identificar API a Optimizar

```
# Ver estructura de la API
cat app/api/buildings/route.ts
```

### Paso 2: Añadir Imports Necesarios

```
// Al inicio del archivo
import { cachedBuildings } from '@/lib/cache-helpers';
import { PerformanceTimer } from '@/lib/performance';
import { logger } from '@/lib/logger';
```

## Paso 3: Envolver Query con Cache

**ANTES:**

```
export async function GET(request: NextRequest) {
  const user = await requireAuth();
  const companyId = user.companyId;

  const buildings = await prisma.building.findMany({
    where: { companyId },
    include: { units: true }, // ❌ Carga TODO
  });

  return NextResponse.json(buildings);
}
```

**DESPUÉS:**

```
export async function GET(request: NextRequest) {
  const timer = new PerformanceTimer();

  try {
    const user = await requireAuth();
    const companyId = user.companyId;
    timer.mark('auth_complete');

    // Usar helper de caché apropiado
    const buildings = await cachedBuildings(companyId, async () => {
      logger.debug(`Fetching fresh buildings for ${companyId}`);

      // Optimizar query: solo campos necesarios
      return prisma.building.findMany({
        where: { companyId },
        select: {
          id: true,
          nombre: true,
          direccion: true,
          _count: { select: { units: true } }, // Solo contar
          // Solo incluir si es absolutamente necesario
          units: {
            select: {
              id: true,
              identificador: true,
              estado: true,
            },
            take: 50, // Limitar cantidad
          },
        },
      });
    });

    timer.mark('data_fetched');
    timer.logSummary('GET /api/buildings');

    return NextResponse.json(buildings);
  } catch (error: any) {
    logger.error('Error in GET /api/buildings:', error);
    // ... manejo de errores ...
  }
}
```

### Paso 4: Añadir Invalidación de Caché en Mutaciones

```
import { invalidateResourceCache } from '@/lib/cache-helpers';

export async function POST(request: NextRequest) {
  // ... crear recurso ...
  const newBuilding = await prisma.building.create({ ... });

  // ✅ Invalidar caché relacionado
  await invalidateResourceCache(companyId, 'buildings');
  await invalidateResourceCache(companyId, 'dashboard');

  return NextResponse.json(newBuilding);
}

export async function PUT(request: NextRequest) {
  // ... actualizar recurso ...
  const updated = await prisma.building.update({ ... });

  // ✅ Invalidar caché
  await invalidateResourceCache(companyId, 'buildings');

  return NextResponse.json(updated);
}

export async function DELETE(request: NextRequest) {
  // ... eliminar recurso ...
  await prisma.building.delete({ ... });

  // ✅ Invalidar caché
  await invalidateResourceCache(companyId, 'buildings');
  await invalidateResourceCache(companyId, 'dashboard');

  return NextResponse.json({ success: true });
}
```

---

## 📚 Helpers de Caché Disponibles

### En `lib/cache-helpers.ts`:

```
// Dashboard
await cachedDashboardStats(companyId, fetchFn);

// Buildings
await cachedBuildings(companyId, fetchFn);

// Units
await cachedUnits(companyId, buildingId, fetchFn);

// Tenants
await cachedTenants(companyId, fetchFn);

// Contracts
await cachedContracts(companyId, fetchFn);

// Payments
await cachedPayments(companyId, fetchFn);

// Analytics
await cachedAnalytics(companyId, period, fetchFn);
```

### Si necesitas crear un helper personalizado:

```
// En lib/cache-helpers.ts
export async function cachedExpenses<T>(
  companyId: string,
  fetchFn: () => Promise<T>
): Promise<T> {
  const key = companyKey(companyId, 'expenses');
  return getCached(key, fetchFn, CACHE_TTL.MEDIUM);
}
```

---

# 🔍 Optimización de Queries Prisma

## ❌ Evitar Estas Malas Prácticas

```
// 1. NO cargar todas las relaciones
const buildings = await prisma.building.findMany({
  include: {
    units: true, // Puede ser cientos
    units: {
      include: {
        contracts: true, // Aún más datos anidados
        contracts: {
          include: { payments: true }, // 💣 Explosión de datos
        },
      },
    },
  },
});

// 2. NO usar findMany sin where para tablas grandes
const allPayments = await prisma.payment.findMany(); // ❌ Miles de registros

// 3. NO hacer queries en bucles (N+1 problem)
for (const building of buildings) {
  const units = await prisma.unit.findMany({ // ❌ Query por cada building
    where: { buildingId: building.id },
  });
}
```

## ✅ Mejores Prácticas

```
// 1. Usar select para campos específicos
const buildings = await prisma.building.findMany({
  where: { companyId },
  select: {
    id: true,
    nombre: true,
    direccion: true,
    // NO incluir campos grandes o que no se usan
  },
});

// 2. Usar _count en lugar de cargar relaciones completas
const buildings = await prisma.building.findMany({
  where: { companyId },
  select: {
    id: true,
    nombre: true,
    _count: {
      select: {
        units: true,
        units: { where: { estado: 'ocupada' } }, // Contar con filtro
      },
    },
  },
});

// 3. Usar take para limitar resultados
const recentPayments = await prisma.payment.findMany({
  where: { companyId },
  take: 10,
  orderBy: { fechaVencimiento: 'desc' },
});

// 4. Usar aggregate para cálculos
const stats = await prisma.payment.aggregate({
  where: { companyId, estado: 'pagado' },
  _sum: { monto: true },
  _count: true,
  _avg: { monto: true },
});

// 5. Paralelizar queries con Promise.all
const [buildings, units, contracts] = await Promise.all([
  prisma.building.count({ where: { companyId } }),
  prisma.unit.count({ where: { building: { companyId } } }),
  prisma.contract.count({ where: { estado: 'activo' } }),
]);
```

---

## 🚦 Tabla de Invalidación de Caché

| Operación | Recursos a Invalidar |
|---|---|
| Crear/Editar Building | `buildings`, `dashboard` |
| Eliminar Building | `buildings`, `units`, `dashboard` |
| Crear/Editar Unit | `units`, `buildings`, `dashboard` |
| Crear/Editar Contract | `contracts`, `units`, `dashboard` |
| Crear/Editar Payment | `payments`, `contracts`, `dashboard` |
| Crear/Editar Tenant | `tenants`, `dashboard` |
| Crear/Editar Expense | `expenses`, `dashboard` |
| Cualquier dato financiero | `analytics:*`, `dashboard` |

### Código de Invalidación:

```
import { invalidateResourceCache, invalidateCompanyCache } from '@/lib/cache-helpers';

// Invalidar recurso específico
await invalidateResourceCache(companyId, 'buildings');

// Invalidar TODO el caché de una empresa (usar con precaución)
await invalidateCompanyCache(companyId);

// Invalidar múltiples recursos
await Promise.all([
  invalidateResourceCache(companyId, 'buildings'),
  invalidateResourceCache(companyId, 'units'),
  invalidateResourceCache(companyId, 'dashboard'),
]);
```

---

## 🧪 Testing y Validación

### 1. Probar Conexión Redis

```
cd /home/ubuntu/homming_vidaro/nextjs_space
yarn tsx scripts/init-redis.ts
```

### 2. Monitorear Performance

Después de optimizar, verifica los logs:

```
# Iniciar app en modo desarrollo
yarn dev

# En otra terminal, hacer requests y ver logs
curl http://localhost:3000/api/buildings

# Buscar en logs:
# - "Cache HIT" = Datos servidos desde Redis ✅
# - "Cache MISS" = Datos cargados desde DB 🟡
# - Tiempos de respuesta en ms
```

### 3. Verificar Headers de Performance

```
# Ver tiempo de respuesta en headers
curl -I http://localhost:3000/api/buildings
# Buscar: X-Response-Time: 245ms
```

### 4. Verificar Caché en Redis

```
# Conectar a Redis CLI
redis-cli

# Ver todas las keys
KEYS company:*

# Ver valor de una key específica
GET company:COMPANY_ID:buildings

# Ver TTL de una key
TTL company:COMPANY_ID:dashboard

# Limpiar todo el caché (solo para testing)
FLUSHALL
```

---

## 📊 Medir Mejoras

### Antes de Optimizar:

1. Ejecutar Lighthouse en Dashboard:

```
lighthouse http://localhost:3000/dashboard --output=html --output-path=./before.html
```

1. Medir tiempos de API manualmente:

```
# Repetir 10 veces y promediar
time curl http://localhost:3000/api/dashboard
```

### Después de Optimizar:

1. Ejecutar Lighthouse de nuevo:

```
lighthouse http://localhost:3000/dashboard --output=html --output-path=./after.html
```

1. Comparar resultados:
   - Performance Score
   - Time to Interactive (TTI)
   - First Contentful Paint (FCP)
   - API Response Times

### Resultados Esperados:

| Métrica | Antes | Después | Mejora |
|---|---|---|---|
| /api/dashboard | ~1500ms | ~200ms | -87% |
| /api/buildings | ~800ms | ~150ms | -81% |
| /api/units | ~600ms | ~120ms | -80% |
| /api/payments | ~900ms | ~180ms | -80% |

---

## ✅ Checklist por API

Para cada API que optimices:

- [ ] Añadir imports de caché helpers
- [ ] Envolver query principal con cached function
- [ ] Optimizar query Prisma (select, take, etc.)
- [ ] Añadir PerformanceTimer
- [ ] Invalidar caché en POST/PUT/DELETE
- [ ] Probar que funciona correctamente
- [ ] Verificar logs (HIT/MISS)
- [ ] Medir tiempo de respuesta

---

## 💡 Tips y Trucos

### 1. TTL (Time To Live) Apropiado

```
import { CACHE_TTL } from '@/lib/redis';

// Datos que cambian frecuentemente (dashboard, pagos pendientes)
await getCached(key, fetchFn, CACHE_TTL.SHORT); // 1 minuto

// Datos moderadamente dinámicos (buildings, units, tenants)
await getCached(key, fetchFn, CACHE_TTL.MEDIUM); // 5 minutos

// Datos relativamente estáticos (analytics, reportes)
await getCached(key, fetchFn, CACHE_TTL.LONG); // 30 minutos
```

### 2. Caché Condicional

```
// Solo cachear en producción
if (process.env.NODE_ENV === 'production') {
  return cachedBuildings(companyId, fetchFn);
} else {
  return fetchFn(); // Siempre fresh en desarrollo
}
```

### 3. Monitoring de Hit Rate

Añade esto a tu dashboard o logs:

```
// En lib/redis.ts, actualizar getCached para trackear métricas
let cacheHits = 0;
let cacheMisses = 0;

export function getCacheStats() {
  const total = cacheHits + cacheMisses;
  const hitRate = total > 0 ? (cacheHits / total) * 100 : 0;
  return { hits: cacheHits, misses: cacheMisses, hitRate: hitRate.toFixed(2) };
}
```

---

## 🆘 Troubleshooting

### Problema: “Datos no se actualizan después de crear/editar”

**Solución:** Asegúrate de invalidar el caché:

```
export async function POST(request: NextRequest) {
  // ... crear recurso ...

  // ✅ NO TE OLVIDES DE ESTO
  await invalidateResourceCache(companyId, 'resource-name');

  return NextResponse.json(newResource);
}
```

### Problema: “API sigue lenta incluso con caché”

**Diagnóstico:**

1. Verificar que Redis está corriendo
2. Verificar logs para ver si hay “Cache HIT”
3. Revisar la query Prisma - puede estar cargando demasiados datos

**Solución:**

```
// Optimizar query para cargar menos datos
select: {
  // Solo campos necesarios
  id: true,
  nombre: true,
  // Evitar campos grandes como 'descripcion', 'notas', etc.
},
take: 100, // Limitar resultados
```

### Problema: “Redis no conecta”

**Verificar:**

```
# ¿Redis está corriendo?
redis-cli ping
# Debe responder: PONG

# ¿REDIS_URL está configurado?
grep REDIS_URL .env

# Logs de Redis
tail -f /usr/local/var/log/redis.log
```

---

## 🚀 Deployment a Producción

### 1. Configurar Redis en Producción

```
# Opción 1: Redis Cloud (Recomendado)
# - Crear cuenta en https://redis.com/try-free/
# - Copiar URL de conexión
# - Añadir a variables de entorno de producción

# Opción 2: Upstash (Serverless)
# - Crear cuenta en https://upstash.com/
# - Crear base de datos Redis
# - Configurar REDIS_URL
```

### 2. Variables de Entorno

```
# Producción
REDIS_URL=redis://username:password@host:port
NODE_ENV=production
```

### 3. Monitoreo Post-Deployment

- Verificar logs para errores de Redis
- Monitorear hit/miss rates
- Verificar que las APIs responden < 500ms
- Usar herramientas como New Relic o DataDog si están disponibles

---

## 📚 Referencias

- Ver archivo: `app/api/buildings-optimized-example/route.ts`
- Ver documentación: `OPTIMIZACION_RENDIMIENTO.md`
- Redis docs: https://redis.io/docs/
- Prisma performance: https://www.prisma.io/docs/guides/performance-and-optimization

---

Última actualización: Diciembre 2024