

# Optimización de Rendimiento Full-Stack

El rendimiento en aplicaciones full-stack es crucial para la experiencia de usuario, SEO y costos de infraestructura. Este documento cubre estrategias avanzadas de optimización tanto en el backend (Node.js/Express) como en el frontend (React), abordando desde la base de datos hasta la renderización en el cliente.

## 1. Optimización en Backend (Node.js + Express)

### 1.1. Paginación Eficiente en MongoDB

**Problema:** Obtener grandes volúmenes de datos sin paginación sobrecarga el servidor y la red.

**Solución:** Implementar paginación con `limit()` y `skip()`:

```
javascript

// ✅ Paginación eficiente
app.get('/api/products', async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  const products = await Product.find()
    .select('name price category') // Solo campos necesarios
    .skip(skip)
    .limit(limit)
    .lean(); // Devuelve objetos planos (más rápido)
```

```
const total = await Product.countDocuments();

res.json({
  data: products,
  pagination: {
    current: page,
    totalPages: Math.ceil(total / limit),
    totalItems: total
  }
});
});
```

**Mejora:** Usar cursor-based pagination para datasets muy grandes (mejor rendimiento que skip()).

## 1.2. Caching con Redis

**Problema:** Consultas repetidas a la base de datos generan carga innecesaria.

**Solución:** Cachear respuestas frecuentes:

```
javascript

import redis from 'redis';
const client = redis.createClient();

// Middleware de caching
const cache = (key, expireTime = 3600) => {
  return (req, res, next) => {
    const cacheKey = key || req.originalUrl;
```

```

client.get(cacheKey, (err, data) => {
  if (err) throw err;
  if (data) {
    return res.json(JSON.parse(data));
  } else {
    res.sendResponse = res.json;
    res.json = (body) => {
      client.setex(cacheKey, expireTime, JSON.stringify(body));
      res.sendResponse(body);
    };
    next();
  }
});
});
};

// Uso en endpoint
app.get('/api/products/featured', cache('featured_products'), async (req, res) => {
  const products = await Product.find({ featured: true });
  res.json(products);
});

```

### 1.3. Compresión y Middlewares de Optimización

**Problema:** Respuestas grandes consumen más ancho de banda.

**Solución:** Comprimir respuestas con Gzip/Brotli:

```
javascript

import compression from 'compression';

// Habilitar compresión
app.use(compression({
  level: 6, // Nivel de compresión (1-9)
  threshold: 0, // Comprimir todas las respuestas
  filter: (req, res) => {
    if (req.headers['x-no-compression']) return false;
    return compression.filter(req, res);
  }
}));
```

## 1.4. Cluster Mode para Node.js

**Problema:** Node.js es single-threaded.

**Solución:** Usar el módulo cluster para multi-core:

```
javascript

import cluster from 'cluster';
import os from 'os';

if (cluster.isPrimary) {
  const numCPUs = os.cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  // Iniciar servidor Express aquí
  app.listen(3000);
}
```

## 2. Optimización en Frontend (React)

### 2.1. Lazy Loading de Componentes y Rutas

**Problema:** Carga inicial lenta por bundles grandes.

**Solución:** Code splitting con React.lazy:

```
javascript

import { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Cargando...</div>}>
      <LazyComponent />
    </Suspense>
  );
}

// Para rutas con React Router
const Home = lazy(() => import('./pages/Home'));
const Products = lazy(() => import('./pages/Products'));
```

### 2.2. Virtualización de Listas

**Problema:** Renderizar listas grandes afecta el performance.

**Solución:** Usar react-window para listas virtualizadas:

```
javascript

import { FixedSizeList as List } from 'react-window';

const Row = ({ index, style }) => (
  <div style={style}>Item {index}</div>
);

const VirtualizedList = () => (
  <List
    height={400}
    itemCount={1000}
    itemSize={35}
    width={300}
  >
    {Row}
  </List>
);

```

## 2.3. Memoización de Componentes

**Problema:** Re-renders innecesarios.

**Solución:** Usar React.memo, useMemo, useCallback:

```
javascript

const ExpensiveComponent = React.memo(({ data }) => {
  // Solo se re-renderiza si cambian las props
  return <div>{data}</div>;
});

function ParentComponent() {
  const [count, setCount] = useState(0);

  const expensiveCalculation = useMemo(() => {
    return calculateExpensiveValue(count);
  }, [count]);
}
```

```
const handleClick = useCallback(() => {
  setCount(prev => prev + 1);
}, []);

return (
  <>
    <ExpensiveComponent data={expensiveCalculation} />
    <button onClick={handleClick}>Increment</button>
  </>
);
}
```

## 2.4. Optimización de Bundles con Webpack

**Problema:** Bundle size demasiado grande.

**Solución:** Analizar y optimizar el bundle:

```

# Analizar bundle
npx webpack-bundle-analyzer

# Configurar split chunks en webpack.config.js
optimization: {
  splitChunks: {
    chunks: 'all',
    cacheGroups: {
      vendor: {
        test: /[\\/]node_modules[\\/]/,
        name: 'vendors',
        chunks: 'all',
      },
    },
  },
}

```

### 3. Optimización de Base de Datos (MongoDB)

#### 3.1. Índices para Consultas Frecuentes

**Problema:** Consultas lentas sin índices.

**Solución:** Crear índices estratégicos:

```

javascript

// Crear índices en modelos
productSchema.index({ category: 1, price: 1 });
productSchema.index({ name: 'text', description: 'text' });

// Verificar queries lentas
db.setProfilingLevel(1, { slowms: 100 });

```

#### 3.2. Aggregation Pipeline Optimizado

**Problema:** Aggregations complejos son lentos.

**Solución:** Optimizar pipelines:

```
javascript

// ✅ Pipeline optimizado
const pipeline = [
  { $match: { category: 'electronics' } }, // Filtro temprano
  { $project: { name: 1, price: 1 } }, // Solo campos necesarios
  { $sort: { price: -1 } },
  { $limit: 10 }
];
```

## 4. Estrategias de Caching

### 4.1. CDN para Assets Estáticos

**Problema:** Carga lenta de assets estáticos.

**Solución:** Usar CDN para imágenes, CSS, JS:

```
<!-- En React public/index.html -->
<script src="https://cdn.example.com/react.production.min.js"></script>
<link rel="stylesheet" href="https://cdn.example.com/styles.css">
```

### 4.2. Service Workers para Caching Offline

**Problema:** Usuarios sin conexión no pueden usar la app.

**Solución:** Implementar PWA con service workers:

```
javascript

// sw.js
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

## Monitorización y Métricas

- **Herramientas de Monitorización**
  - Backend: New Relic, Datadog, PM2 metrics
  - Frontend: Google Lighthouse, Web Vitals
  - Base de datos: MongoDB Atlas Performance Advisor
- **Métricas Clave Para Monitorizar**
  - Backend: Response time, CPU usage, Memory usage
  - Frontend: LCP (Largest Contentful Paint), FID (First Input Delay), CLS (Cumulative Layout Shift)
  - Base de datos: Query performance, Connection pool usage

## 5. Configuración para Producción

### 5.1. Variables de Entorno Críticas

```
env

# Node.js
NODE_ENV=production
UV_THREADPOOL_SIZE=128

# MongoDB
DB_URI=mongodb+srv://...&maxPoolSize=20&w=majority

# Redis
REDIS_URL=redis://...&max_clients=30
```

## 7.2. Configuración de Reverse Proxy (Nginx)

```
# /etc/nginx/conf.d/app.conf
server {
    listen 80;
    server_name tu-dominio.com;

    gzip on;
    gzip_types text/plain text/css application/json application/javascript;
```

```
location / {
    proxy_pass http://localhost:3000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
}

location /static/ {
    expires 1y;
    add_header Cache-Control "public, immutable";
}
```

La optimización full-stack requiere un enfoque holístico que aborde backend, frontend, base de datos y infraestructura. Implementar estas estrategias mejora significativamente el rendimiento, la escalabilidad y la experiencia de usuario.