MakeEstimate Performance Guide

Overview

This document covers the performance optimizations and monitoring capabilities implemented to scale MakeEstimate to 10k-50k companies.

Performance Milestones

Milestone 1: Instrumentation & Observability ✅

Files:
- `lib/performance.ts` - Core performance monitoring
- `lib/api-instrumentation.ts` - API route wrapper
- `app/api/admin/metrics/route.ts` - Admin metrics endpoint

Features:
- Request timing for all instrumented endpoints
- Slow query detection (>500ms threshold)
- Error tracking with context
- In-memory metrics buffer (last 1000 requests)

Usage:

```
// Wrap API routes
export const GET = instrumentRoute(
  '/api/endpoint',
  'GET',
  async (request, { params }, { companyId }) => {
    // Your handler
  }
);

// Time specific operations
const result = await timedOperation('db.query', () =>
  prisma.boq.findFirst({ where: { id } })
);
```

Admin Metrics (GET /api/admin/metrics):

```
{
  "summary": { "totalRequests": 1234, "slowQueries": 5 },
  "endpointStats": [...],
  "rateLimitStats": {...},
  "recentSlowQueries": [...],
  "recentRequests": [...]
}
```

Milestone 2: Database Hardening ✅

Schema Indexes (prisma/schema.prisma):

Model	Index	Purpose
CompanyMembership	`[companyId]`	Fast user lookup by company
CompanyMembership	`[userId]`	Fast company lookup for user
Customer	`[companyId, name]`	Customer search within company
Boq	`[companyId, updatedAt(desc)]`	Recent BOQs list (most common)
Boq	`[companyId, projectName]`	BOQ search by name
Boq	`[customerId]`	BOQs for customer
PdfExportJob	`[companyId, createdAt(desc)]`	Recent jobs list
PdfExportJob	`[boqId]`	Jobs for BOQ
PdfExportJob	`[status]`	Query by status

Connection Pooling (lib/db.ts):

```
DATABASE_URL="postgresql://...?connection_limit=20&pool_timeout=30"
```

Milestone 3: Async PDF Export ✅

Problem: PDF generation blocks the request thread for 2-10+ seconds.

Solution: Job queue pattern with background processing.

Flow:
1. `POST /api/boqs/{id}/pdf/async` → Creates job, returns `{ jobId }`
2. Background: `POST /api/boqs/{id}/pdf?process=true` → Generates PDF
3. Client polls `GET /api/pdf-jobs/{jobId}` → Status + URL when complete

Job States:
- `pending` → Initial state
- `processing` → Being generated
- `completed` → PDF ready (includes `pdfUrl`)
- `failed` → Error occurred (includes `errorMessage`)

Frontend Integration:

```
// Export button triggers async flow
const response = await fetch(`/api/boqs/${id}/pdf/async`, { method: 'POST' });
const { jobId } = await response.json();

// Poll for completion
const poll = setInterval(async () => {
  const status = await fetch(`/api/pdf-jobs/${jobId}`).then(r => r.json());
  if (status.status === 'completed') {
    clearInterval(poll);
    downloadPdf(status.pdfUrl);
  }
}, 2000);
```

Milestone 4: Rate Limiting ✅

File: `lib/rate-limiter.ts`

Rate Limit Profiles:

Endpoint	Limit	Scope	Block Duration
PDF Export	10/min	Company	30 seconds
Item Update	120/min	User	—
BOQ Create	20/min	Company	—
API General	200/min	User	—

Implementation:

```
// Using instrumentRoute with rate limiting
export const PUT = instrumentRoute(
  '/api/items/[id]',
  'PUT',
  handler,
  { requireAuth: true, rateLimit: { type: 'ITEM_UPDATE', keyBy: 'user' } }
);

// Or manually
const rateKey = rateLimitKey('PDF_EXPORT', companyId);
const result = checkRateLimit(rateKey, RATE_LIMITS.PDF_EXPORT);
if (!result.allowed) {
  return rateLimitResponse(result);
}
```

Response (429 Too Many Requests):

```
{
  "error": "Too many requests",
  "message": "Rate limit exceeded. Please try again later.",
  "retryAfter": 30
}
```

Headers:
- `Retry-After: 30`
- `X-RateLimit-Remaining: 0`
- `X-RateLimit-Reset: 30`

Milestone 5: Load Testing ✅

Scripts:
- `scripts/load-test.ts` - Load test runner
- `scripts/generate-test-data.ts` - Test data generator

Generate Test Data:

```
# Create 50 companies with 100 BOQs each
npx tsx scripts/generate-test-data.ts --companies=50 --boqs=100

# Clean and regenerate
npx tsx scripts/generate-test-data.ts --clean --companies=10
```

Run Load Tests:

```
# Test autosave with 20 concurrent users for 2 minutes
npx tsx scripts/load-test.ts autosave --users=20 --duration=120

# Mixed scenario (realistic usage)
npx tsx scripts/load-test.ts mixed --users=50 --duration=300

# PDF export stress test
npx tsx scripts/load-test.ts pdf-export --users=10 --duration=60
```

Scenarios:
- `dashboard` - Fetch BOQs, customers, billing status
- `autosave` - Rapid item updates (simulates real-time editing)
- `pdf-export` - Async PDF generation requests
- `mixed` - 50% autosave, 30% dashboard, 20% PDF export

Sample Output:

```
============================================================
LOAD TEST RESULTS
============================================================
Scenario:               mixed
Total Requests:         4523
Successful:             4412
Failed:                 111
Rate Limited (429):     45
------------------------------------------------------------
Avg Response Time:      89ms
P50 Response Time:      45ms
P95 Response Time:      234ms
P99 Response Time:      512ms
Max Response Time:      1823ms
------------------------------------------------------------
Requests/Second:        37.69
Error Rate:             2.45%
============================================================
```

Performance Best Practices

Database Queries

1. **Always filter by companyId first** - Uses indexes efficiently
2. **Select only needed fields** - Reduces data transfer
3. **Use `findFirst` over `findUnique` when checking existence** - Can use partial indexes

```
// Good
const boq = await prisma.boq.findFirst({
  where: { id, companyId },
  select: { id: true },
});

// Avoid - fetches all fields
const boq = await prisma.boq.findFirst({
  where: { id, companyId },
});
```

API Routes

1. **Wrap with instrumentRoute** - Automatic timing and logging
2. **Use timedOperation for DB calls** - Identifies slow queries
3. **Apply rate limits** - Protects system from abuse

Frontend

1. **Debounce autosave** - 500ms minimum between saves
2. **Use optimistic updates** - Better perceived performance
3. **Prefetch likely navigation targets** - Instant page loads

---

Monitoring Checklist

- [ ] Check `/api/admin/metrics` for slow endpoints
- [ ] Monitor rate limit stats for abuse patterns
- [ ] Review slow query logs for optimization opportunities
- [ ] Run load tests before major releases
- [ ] Set up alerts for error rate > 5%
- [ ] Monitor database connection pool utilization

Scaling Recommendations

For 50k+ companies:

1. **Redis for rate limiting** - Replace in-memory store
2. **External job queue** - Bull/BullMQ for PDF processing
3. **Read replicas** - For dashboard/list queries
4. **CDN for PDFs** - Cache generated PDFs
5. **Connection pooler** - PgBouncer in front of database