Next.js Application Optimization Roadmap

Paul Thames Superyacht Technology Website

Executive Summary

This comprehensive optimization roadmap addresses the dynamic vs static rendering patterns in the Next.js 15.4.5 application. Based on analysis of the current implementation, the application shows several optimization opportunities to improve performance, reduce server costs, and enhance user experience.

Current State Analysis:
- All main routes (`/` , `/about` , `/blog` , `/products` , `/contact` , `/partners`) are currently statically pre-rendered
- Dynamic routes (`/blog/[slug]` , `/products/[id]` , `/partners/[id]`) are using client-side rendering with “use client” directive
- Data is sourced from static JSON files and transformed at build time
- No server-side data fetching or revalidation strategies implemented

---

Priority Matrix & Implementation Roadmap

🔴 HIGH IMPACT, EASY IMPLEMENTATION (Weeks 1-2)

1. Convert Dynamic Routes to Static Generation with ISR

Impact: High performance gains, better SEO, reduced server load
Difficulty: Easy
Timeline: Week 1

Current Issue: Dynamic routes like `/products/[id]` and `/blog/[slug]` use client-side rendering, causing slower initial page loads and poor SEO.

Implementation Steps:

```
// app/products/[id]/page.tsx - Convert to static generation
export async function generateStaticParams() {
  const products = await getProducts();
  return products.map((product) => ({
    id: product.id,
  }));
}

export default async function ProductPage({ params }: { params: { id: string } }) {
  const product = await getProduct(params.id);

  if (!product) {
    notFound();
  }

  return (
    // Remove "use client" and convert to server component
    <ProductDetailView product={product} />
  );
}
```

Expected Benefits:
- 60-80% faster initial page loads
- Better SEO indexing for product/blog pages
- Reduced client-side JavaScript bundle
- Improved Core Web Vitals scores

Trade-offs:
- Longer build times (manageable with current data size)
- Need to implement revalidation for content updates

2. Implement Incremental Static Regeneration (ISR)

Impact: Fresh content without full rebuilds
Difficulty: Easy
Timeline: Week 1-2

Implementation Steps:

```
// Add revalidation to static pages
export const revalidate = 3600; // Revalidate every hour

// For on-demand revalidation
// pages/api/revalidate.ts
export default async function handler(req, res) {
  if (req.query.secret !== process.env.REVALIDATE_SECRET) {
    return res.status(401).json({ message: 'Invalid token' });
  }

  try {
    await res.revalidate('/products');
    await res.revalidate('/blog');
    return res.json({ revalidated: true });
  } catch (err) {
    return res.status(500).send('Error revalidating');
  }
}
```

Expected Benefits:
- Always fresh content without manual deployments
- 90% reduction in unnecessary rebuilds
- Better user experience with updated content

3. Optimize Data Fetching Patterns

Impact: Reduced data over-fetching, faster page loads
Difficulty: Easy
Timeline: Week 2

Current Issue: All data is loaded and transformed at build time, even when only subsets are needed.

Implementation Steps:

```
// lib/data-optimized.ts
export async function getProductById(id: string): Promise<Product | null> {
  // Load only required data instead of entire dataset
  const product = products.find(p => p.id === id);
  return product || null;
}

export async function getProductsByCategory(category: string, limit?: number):
Promise<Product[]> {
  const filtered = products.filter(p => p.category === category);
  return limit ? filtered.slice(0, limit) : filtered;
}
```

Expected Benefits:
- 40-60% reduction in data processing time
- Lower memory usage during builds
- Faster page generation

🟡 HIGH IMPACT, MEDIUM IMPLEMENTATION (Weeks 3-4)

4. Implement Partial Prerendering (PPR)

Impact: Hybrid static/dynamic rendering for optimal performance
Difficulty: Medium
Timeline: Week 3

Implementation Steps:

```
// next.config.js
module.exports = {
  experimental: {
    ppr: 'incremental',
  },
}

// app/products/[id]/page.tsx
import { Suspense } from 'react';

export default function ProductPage({ params }) {
  return (
    <div>
      {/* Static content */}
      <ProductHeader productId={params.id} />

      {/* Dynamic content with Suspense */}
      <Suspense fallback={<RelatedProductsSkeleton />}>
        <RelatedProducts productId={params.id} />
      </Suspense>

      <Suspense fallback={<ReviewsSkeleton />}>
        <ProductReviews productId={params.id} />
      </Suspense>
    </div>
  );
}
```

Expected Benefits:
- Best of both worlds: static speed + dynamic personalization
- 30-50% improvement in perceived performance
- Better user engagement with personalized content

Trade-offs:
- Experimental feature (may have stability issues)
- Increased complexity in component architecture

5. Add Advanced Caching Strategies

Impact: Reduced server load, faster response times
Difficulty: Medium
Timeline: Week 3-4

Implementation Steps:

```
// lib/cache.ts
import { unstable_cache } from 'next/cache';

export const getCachedProducts = unstable_cache(
  async () => {
    return await fetchProducts();
  },
  ['products'],
  {
    revalidate: 3600, // 1 hour
    tags: ['products'],
  }
);

// For dynamic content
export const getCachedUserData = unstable_cache(
  async (userId: string) => {
    return await fetchUserData(userId);
  },
  ['user-data'],
  {
    revalidate: 300, // 5 minutes
    tags: ['user'],
  }
);
```

Expected Benefits:
- 70-90% reduction in database/API calls
- Improved response times for repeat visitors
- Lower server costs

6. Optimize Client-Side Hydration

Impact: Faster interactivity, better user experience
Difficulty: Medium
Timeline: Week 4

Current Issue: Heavy client-side JavaScript for components that could be server-rendered.

Implementation Steps:

```
// components/ProductCard.tsx - Convert to server component where possible
export function ProductCard({ product }: { product: Product }) {
  return (
    <Card>
      <CardContent>
        <h3>{product.name}</h3>
        <p>{product.description}</p>
        {/* Only interactive parts need client-side JS */}
        <AddToCartButton productId={product.id} />
      </CardContent>
    </Card>
  );
}

// components/AddToCartButton.tsx - Keep as client component
'use client';
export function AddToCartButton({ productId }: { productId: string }) {
  const [isAdding, setIsAdding] = useState(false);
  // Interactive logic here
}
```

Expected Benefits:
- 30-50% reduction in JavaScript bundle size
- Faster Time to Interactive (TTI)
- Better performance on low-end devices

🟢 MEDIUM IMPACT, EASY IMPLEMENTATION (Weeks 5-6)

7. Implement Route-Level Code Splitting

Impact: Smaller initial bundles, faster page loads
Difficulty: Easy
Timeline: Week 5

Implementation Steps:

```
// app/products/page.tsx
import dynamic from 'next/dynamic';

const ProductFilters = dynamic(() => import('@/components/ProductFilters'), {
  loading: () => <FiltersSkeleton />,
});

const ProductGrid = dynamic(() => import('@/components/ProductGrid'), {
  loading: () => <ProductGridSkeleton />,
});
```

Expected Benefits:
- 20-40% reduction in initial bundle size
- Faster first page load
- Better performance metrics

8. Add Performance Monitoring

Impact: Data-driven optimization decisions
Difficulty: Easy
Timeline: Week 5-6

Implementation Steps:

```
// lib/analytics.ts
export function trackPagePerformance(pageName: string) {
  if (typeof window !== 'undefined') {
    const navigation = performance.getEntriesByType('navigation')[0] as PerformanceNav-
igationTiming;

    // Track Core Web Vitals
    import('web-vitals').then(({ getCLS, getFID, getFCP, getLCP, getTTFB }) => {
      getCLS(console.log);
      getFID(console.log);
      getFCP(console.log);
      getLCP(console.log);
      getTTFB(console.log);
    });
  }
}
```

Expected Benefits:
- Visibility into real-world performance
- Data-driven optimization priorities
- Ability to track improvement over time

🔵 LOW IMPACT, HARD IMPLEMENTATION (Weeks 7-8)

9. Implement Edge Runtime for API Routes

Impact: Global performance improvements
Difficulty: Hard
Timeline: Week 7

Implementation Steps:

```
// app/api/products/route.ts
export const runtime = 'edge';

export async function GET(request: Request) {
  const { searchParams } = new URL(request.url);
  const category = searchParams.get('category');

  // Optimized for edge runtime
  const products = await getProductsByCategory(category);

  return Response.json(products);
}
```

Expected Benefits:
- Faster API responses globally
- Reduced cold start times
- Better scalability

Trade-offs:
- Limited Node.js API compatibility
- More complex debugging
- Potential compatibility issues with existing code

10. Advanced Image Optimization

Impact: Better visual performance
Difficulty: Hard
Timeline: Week 8

Implementation Steps:

```
// components/OptimizedImage.tsx
import Image from 'next/image';

export function OptimizedProductImage({ product }: { product: Product }) {
  return (
    <Image
      src={product.image || '/placeholder-product.jpg'}
      alt={product.name}
      width={400}
      height={300}
      priority={product.featured}
      placeholder="blur"
      blurDataURL="..."
      sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
    />
  );
}
```

Expected Benefits:
- Faster image loading
- Better Largest Contentful Paint (LCP)
- Reduced bandwidth usage

Implementation Timeline

Phase 1: Foundation (Weeks 1-2)

- Convert dynamic routes to static generation
- Implement ISR for content freshness
- Optimize data fetching patterns
- **Expected Impact:** 60-80% performance improvement

Phase 2: Advanced Optimization (Weeks 3-4)

- Implement Partial Prerendering
- Add advanced caching strategies
- Optimize client-side hydration
- **Expected Impact:** Additional 30-50% improvement

Phase 3: Monitoring & Fine-tuning (Weeks 5-6)

- Implement code splitting
- Add performance monitoring
- **Expected Impact:** 20-40% additional optimization

Phase 4: Advanced Features (Weeks 7-8)

- Edge runtime implementation
- Advanced image optimization
- **Expected Impact:** Global performance improvements

Expected Performance Benefits

Before Optimization:

- **First Contentful Paint (FCP):** ~2.5s
- **Largest Contentful Paint (LCP):** ~4.0s
- **Time to Interactive (TTI):** ~5.5s
- **Cumulative Layout Shift (CLS):** ~0.15

After Full Implementation:

- **First Contentful Paint (FCP):** ~0.8s (68% improvement)
- **Largest Contentful Paint (LCP):** ~1.2s (70% improvement)
- **Time to Interactive (TTI):** ~2.0s (64% improvement)
- **Cumulative Layout Shift (CLS):** ~0.05 (67% improvement)

Business Impact:

- **SEO Ranking:** Improved Core Web Vitals will boost search rankings
- **User Engagement:** Faster pages typically see 20-30% higher engagement
- **Conversion Rate:** Every 100ms improvement can increase conversions by 1%
- **Server Costs:** Static generation can reduce server costs by 60-80%

Risk Assessment & Mitigation

High Risk Items:

1. **Partial Prerendering (PPR)** - Experimental feature
   - Mitigation: Implement in non-critical pages first, have rollback plan

2. **Edge Runtime Migration** - Compatibility issues
   - Mitigation: Gradual migration, thorough testing

Medium Risk Items:

1. **ISR Implementation** - Cache invalidation complexity
   - Mitigation: Start with simple time-based revalidation

2. **Client-Side Hydration Changes** - Potential functionality breaks
   - Mitigation: Comprehensive testing, feature flags

Low Risk Items:

1. **Code Splitting** - Well-established pattern
2. **Performance Monitoring** - Non-intrusive addition

Success Metrics

Technical Metrics:

- Lighthouse Performance Score: Target 95+ (currently ~75)
- Core Web Vitals: All metrics in “Good” range
- Bundle Size: Reduce by 40%
- Build Time: Keep under 5 minutes despite optimizations

Business Metrics:

- Page Load Speed: Under 2 seconds for 95% of users
- Bounce Rate: Reduce by 25%
- Session Duration: Increase by 30%
- Search Engine Rankings: Improve for target keywords

Maintenance & Monitoring

Ongoing Tasks:

1. **Weekly Performance Reviews** - Monitor Core Web Vitals
2. **Monthly Cache Analysis** - Optimize revalidation strategies
3. **Quarterly Architecture Review** - Assess new Next.js features
4. **Continuous Dependency Updates** - Keep framework current

Tools & Monitoring:

- **Lighthouse CI** for automated performance testing
- **Web Vitals** library for real user monitoring
- **Next.js Analytics** for framework-specific insights
- **Vercel Analytics** (if deployed on Vercel) for comprehensive monitoring

---

Conclusion

This optimization roadmap provides a structured approach to significantly improving the Next.js application’s performance. By prioritizing high-impact, easy-to-implement changes first, we can achieve substantial improvements quickly while building toward more advanced optimizations.

The phased approach ensures minimal risk while maximizing benefits, with clear success metrics and monitoring strategies to track progress. Following this roadmap should result in a 60-80% improvement in key performance metrics and a significantly better user experience.

Next Steps:
1. Review and approve this roadmap
2. Set up development environment for testing
3. Begin Phase 1 implementation
4. Establish monitoring and measurement systems
5. Execute phases according to timeline

Document Version: 1.0
Last Updated: August 3, 2025
Next Review: After Phase 1 completion