Table of Contents

# Bab 15: Performance Optimization di TypeScript

## Penjelasan Materi

Performance Optimization di TypeScript melibatkan berbagai teknik untuk meningkatkan kinerja aplikasi, baik dalam hal waktu eksekusi, penggunaan memori, maupun ukuran bundle. Optimasi ini mencakup aspek kompilasi TypeScript, runtime performance, dan delivery optimization.

## Analogi yang Mudah Dipahami

Bayangkan performance optimization seperti mengoptimalkan kinerja mobil: - Compiler Optimization seperti tune-up mesin - Memory Management seperti mengatur konsumsi bahan bakar - Bundle Size Optimization seperti mengurangi berat kendaraan - Code Splitting seperti membagi beban ke beberapa kendaraan - Lazy Loading seperti mengangkut barang sesuai kebutuhan

## Point Penting

1. **Compiler Optimization**
   * tsconfig settings
   * Type checking
   * Declaration files
   * Build performance
2. **Runtime Performance**
   * Memory management
   * CPU optimization
   * Async operations
   * Data structures
3. **Bundle Optimization**
   * Tree shaking
   * Code splitting
   * Lazy loading
   * Minification
4. **Memory Management**
   * Memory leaks
   * Garbage collection
   * Resource pooling
   * Cache strategies
5. **Monitoring & Profiling**
   * Performance metrics
   * Memory profiling
   * CPU profiling
   * Network analysis

## Contoh Kode dan Penjelasan

```typescript // 1. Compiler Optimization // tsconfig.json { “compilerOptions”: { “target”: “ES2020”, “module”: “ESNext”, “strict”: true, “skipLibCheck”: true, “forceConsistentCasingInFileNames”: true, “incremental”: true, “tsBuildInfoFile”: “./buildcache/cache.json” } }

// 2. Memory Management class ResourcePool { private pool: T[] = []; private inUse = new Set();

constructor(  
 private factory: () => T,  
 private reset: (item: T) => void,  
 private initialSize: number  
) {  
 this.initialize();  
}  
  
private initialize(): void {  
 for (let i = 0; i < this.initialSize; i++) {  
 this.pool.push(this.factory());  
 }  
}  
  
acquire(): T {  
 let item: T;  
 if (this.pool.length > 0) {  
 item = this.pool.pop()!;  
 } else {  
 item = this.factory();  
 }  
 this.inUse.add(item);  
 return item;  
}  
  
release(item: T): void {  
 if (this.inUse.has(item)) {  
 this.inUse.delete(item);  
 this.reset(item);  
 this.pool.push(item);  
 }  
}

}

// 3. Efficient Data Structures class LRUCache<K, V> { private cache = new Map<K, V>(); private readonly maxSize: number;

constructor(maxSize: number) {  
 this.maxSize = maxSize;  
}  
  
get(key: K): V | undefined {  
 const value = this.cache.get(key);  
 if (value !== undefined) {  
 // Move to front (most recently used)  
 this.cache.delete(key);  
 this.cache.set(key, value);  
 }  
 return value;  
}  
  
set(key: K, value: V): void {  
 if (this.cache.size >= this.maxSize) {  
 // Remove least recently used item  
 const firstKey = this.cache.keys().next().value;  
 this.cache.delete(firstKey);  
 }  
 this.cache.set(key, value);  
}

}

// 4. Async Operation Optimization class AsyncOperationManager { private queue: Array<() => Promise> = []; private isProcessing = false; private concurrentLimit: number; private activeCount = 0;

constructor(concurrentLimit = 3) {  
 this.concurrentLimit = concurrentLimit;  
}  
  
async add(operation: () => Promise<void>): Promise<void> {  
 this.queue.push(operation);  
 this.processQueue();  
}  
  
private async processQueue(): Promise<void> {  
 if (this.isProcessing) return;  
 this.isProcessing = true;  
  
 while (this.queue.length > 0 && this.activeCount < this.concurrentLimit) {  
 const operation = this.queue.shift();  
 if (operation) {  
 this.activeCount++;  
 try {  
 await operation();  
 } finally {  
 this.activeCount--;  
 }  
 }  
 }  
  
 this.isProcessing = false;  
 if (this.queue.length > 0) {  
 this.processQueue();  
 }  
}

}

// 5. Performance Monitoring class PerformanceMonitor { private static instance: PerformanceMonitor; private metrics: Map<string, number[]> = new Map();

private constructor() {}  
  
static getInstance(): PerformanceMonitor {  
 if (!PerformanceMonitor.instance) {  
 PerformanceMonitor.instance = new PerformanceMonitor();  
 }  
 return PerformanceMonitor.instance;  
}  
  
async measure<T>(  
 label: string,  
 fn: () => Promise<T>  
): Promise<T> {  
 const start = performance.now();  
 try {  
 return await fn();  
 } finally {  
 const duration = performance.now() - start;  
 const measurements = this.metrics.get(label) || [];  
 measurements.push(duration);  
 this.metrics.set(label, measurements);  
 }  
}  
  
getStats(label: string): {  
 avg: number;  
 min: number;  
 max: number;  
 count: number;  
} {  
 const measurements = this.metrics.get(label) || [];  
 if (measurements.length === 0) {  
 return { avg: 0, min: 0, max: 0, count: 0 };  
 }  
  
 const sum = measurements.reduce((a, b) => a + b, 0);  
 return {  
 avg: sum / measurements.length,  
 min: Math.min(...measurements),  
 max: Math.max(...measurements),  
 count: measurements.length  
 };  
}

}

// 6. Code Splitting Example // app.ts const loadModule = async (moduleName: string) => { switch (moduleName) { case ‘users’: return import(‘./modules/users’); case ‘products’: return import(‘./modules/products’); case ‘orders’: return import(‘./modules/orders’); default: throw new Error(`Unknown module: ${moduleName}`); } };

// 7. Memory Leak Prevention class EventEmitterWithCleanup { private handlers: Map<string, Set> = new Map(); private weakHandlers: Map<string, WeakSet> = new Map();

on(event: string, handler: Function): void {  
 const handlers = this.handlers.get(event) || new Set();  
 handlers.add(handler);  
 this.handlers.set(event, handlers);  
}  
  
onWeak(event: string, target: object): void {  
 const handlers = this.weakHandlers.get(event) || new WeakSet();  
 handlers.add(target);  
 this.weakHandlers.set(event, handlers);  
}  
  
off(event: string, handler: Function): void {  
 const handlers = this.handlers.get(event);  
 if (handlers) {  
 handlers.delete(handler);  
 if (handlers.size === 0) {  
 this.handlers.delete(event);  
 }  
 }  
}  
  
emit(event: string, ...args: any[]): void {  
 const handlers = this.handlers.get(event);  
 if (handlers) {  
 handlers.forEach(handler => handler(...args));  
 }  
}

}

// 8. CPU-Intensive Task Optimization class WorkerPool { private workers: Worker[] = []; private taskQueue: Array<{ task: any; resolve: (value: any) => void; reject: (reason: any) => void; }> = []; private availableWorkers: Worker[] = [];

constructor(private workerScript: string, private poolSize: number) {  
 this.initialize();  
}  
  
private initialize(): void {  
 for (let i = 0; i < this.poolSize; i++) {  
 const worker = new Worker(this.workerScript);  
 worker.onmessage = (e) => this.handleWorkerMessage(worker, e);  
 worker.onerror = (e) => this.handleWorkerError(worker, e);  
 this.workers.push(worker);  
 this.availableWorkers.push(worker);  
 }  
}  
  
private handleWorkerMessage(worker: Worker, e: MessageEvent): void {  
 const task = this.taskQueue.shift();  
 if (task) {  
 task.resolve(e.data);  
 }  
 this.availableWorkers.push(worker);  
 this.processQueue();  
}  
  
private handleWorkerError(worker: Worker, e: ErrorEvent): void {  
 const task = this.taskQueue.shift();  
 if (task) {  
 task.reject(e.error);  
 }  
 this.availableWorkers.push(worker);  
 this.processQueue();  
}  
  
private processQueue(): void {  
 while (this.availableWorkers.length > 0 && this.taskQueue.length > 0) {  
 const worker = this.availableWorkers.pop()!;  
 const task = this.taskQueue[0];  
 worker.postMessage(task.task);  
 }  
}  
  
executeTask<T>(task: any): Promise<T> {  
 return new Promise((resolve, reject) => {  
 this.taskQueue.push({ task, resolve, reject });  
 this.processQueue();  
 });  
}  
  
terminate(): void {  
 this.workers.forEach(worker => worker.terminate());  
 this.workers = [];  
 this.availableWorkers = [];  
}

}

// 9. Network Optimization class NetworkOptimizer { private cache = new Map<string, { data: any; timestamp: number; ttl: number; }>();

async fetch<T>(  
 url: string,  
 options: {  
 ttl?: number;  
 forceRefresh?: boolean;  
 retries?: number;  
 timeout?: number;  
 } = {}  
): Promise<T> {  
 const {  
 ttl = 5 \* 60 \* 1000, // 5 minutes  
 forceRefresh = false,  
 retries = 3,  
 timeout = 5000  
 } = options;  
  
 const cached = this.cache.get(url);  
 if (  
 !forceRefresh &&  
 cached &&  
 Date.now() - cached.timestamp < cached.ttl  
 ) {  
 return cached.data;  
 }  
  
 let lastError: Error | null = null;  
 for (let i = 0; i < retries; i++) {  
 try {  
 const controller = new AbortController();  
 const timeoutId = setTimeout(  
 () => controller.abort(),  
 timeout  
 );  
  
 const response = await fetch(url, {  
 signal: controller.signal  
 });  
  
 clearTimeout(timeoutId);  
  
 if (!response.ok) {  
 throw new Error(\`HTTP error! status: \${response.status}\`);  
 }  
  
 const data = await response.json();  
 this.cache.set(url, {  
 data,  
 timestamp: Date.now(),  
 ttl  
 });  
  
 return data;  
 } catch (error) {  
 lastError = error as Error;  
 await new Promise(resolve =>  
 setTimeout(resolve, Math.pow(2, i) \* 1000)  
 );  
 }  
 }  
  
 throw lastError;  
}

}

// 10. Bundle Size Optimization // webpack.config.js module.exports = { mode: ‘production’, entry: ‘./src/index.ts’, output: { filename: ‘[name].[contenthash].js’, chunkFilename: ‘[name].[contenthash].js’ }, optimization: { splitChunks: { chunks: ‘all’, maxInitialRequests: Infinity, minSize: 0, cacheGroups: { vendor: { test: /[\/]node\_modules[\/]/, name(module) { const packageName = module.context.match( /[\/]node\_modules[\/](.*?)([\/]|$)/ )[1]; return `vendor.${packageName.replace(‘@’, ’’)}`; } } } } } }; ```

## Cara Kerja Performance Optimization

1. **Compilation Process**:
   * Type checking
   * Code generation
   * Bundle creation
   * Tree shaking
2. **Memory Management**:
   * Allocation
   * Garbage collection
   * Reference tracking
   * Memory pooling
3. **Runtime Optimization**:
   * JIT compilation
   * Code execution
   * Async operations
   * Resource utilization

## Tips dan Trik

1. **Efficient Data Structures**

* // ✅ Gunakan struktur data yang tepat  
  class Cache<K, V> {  
   private data = new Map<K, {  
   value: V;  
   expiry: number;  
   }>();  
    
   set(key: K, value: V, ttl: number): void {  
   this.data.set(key, {  
   value,  
   expiry: Date.now() + ttl  
   });  
   }  
    
   get(key: K): V | undefined {  
   const item = this.data.get(key);  
   if (!item) return undefined;  
   if (Date.now() > item.expiry) {  
   this.data.delete(key);  
   return undefined;  
   }  
   return item.value;  
   }  
  }

1. **Memory Management**

* // ✅ Implement proper cleanup  
  class ResourceManager {  
   private resources = new Set<Resource>();  
    
   acquire(): Resource {  
   const resource = new Resource();  
   this.resources.add(resource);  
   return resource;  
   }  
    
   release(resource: Resource): void {  
   if (this.resources.has(resource)) {  
   resource.dispose();  
   this.resources.delete(resource);  
   }  
   }  
    
   dispose(): void {  
   this.resources.forEach(resource => {  
   resource.dispose();  
   });  
   this.resources.clear();  
   }  
  }

1. **Async Operation Optimization**

* // ✅ Optimize async operations  
  class BatchProcessor {  
   private batch: any[] = [];  
   private timeout: NodeJS.Timeout | null = null;  
    
   add(item: any): void {  
   this.batch.push(item);  
   this.scheduleProcess();  
   }  
    
   private scheduleProcess(): void {  
   if (!this.timeout) {  
   this.timeout = setTimeout(() => {  
   this.processBatch();  
   }, 100);  
   }  
   }  
    
   private async processBatch(): Promise<void> {  
   const items = [...this.batch];  
   this.batch = [];  
   this.timeout = null;  
   await this.process(items);  
   }  
    
   private async process(items: any[]): Promise<void> {  
   // Process batch  
   }  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Memory Leaks**

* // ❌ Buruk: Memory leak dalam event listeners  
  class Component {  
   constructor() {  
   window.addEventListener('resize', this.onResize);  
   }  
    
   onResize = () => {  
   // Handle resize  
   }  
  }  
    
  // ✅ Baik: Proper cleanup  
  class Component {  
   constructor() {  
   window.addEventListener('resize', this.onResize);  
   }  
    
   onResize = () => {  
   // Handle resize  
   }  
    
   destroy() {  
   window.removeEventListener('resize', this.onResize);  
   }  
  }

1. **Inefficient Data Processing**

* // ❌ Buruk: Inefficient array processing  
  function processItems(items: number[]): number[] {  
   return items  
   .filter(x => x > 0)  
   .map(x => x \* 2)  
   .filter(x => x < 100);  
  }  
    
  // ✅ Baik: Single pass processing  
  function processItems(items: number[]): number[] {  
   return items.reduce((acc, x) => {  
   if (x > 0) {  
   const result = x \* 2;  
   if (result < 100) {  
   acc.push(result);  
   }  
   }  
   return acc;  
   }, [] as number[]);  
  }

1. **Bundle Size Issues**

* // ❌ Buruk: Import entire library  
  import \* as lodash from 'lodash';  
    
  // ✅ Baik: Import specific functions  
  import { debounce, throttle } from 'lodash';

### Solusi:

1. Implementasi proper memory management
2. Optimize data structures dan algoritma
3. Gunakan code splitting dan lazy loading
4. Monitor performance metrics
5. Implement caching strategies