Table of Contents

# Bab 13: Debugging di TypeScript

## Penjelasan Materi

Debugging adalah proses menemukan dan memperbaiki bug dalam kode. TypeScript menyediakan beberapa fitur yang membantu dalam proses debugging, seperti type checking statis, source maps, dan integrasi dengan berbagai debugging tools. Dengan TypeScript, kita bisa menemukan bug lebih awal dalam proses development dan memiliki tools yang lebih baik untuk debugging runtime issues.

## Analogi yang Mudah Dipahami

Bayangkan debugging seperti proses mendiagnosis masalah pada mobil: - Type Checking seperti sistem diagnostik yang mendeteksi masalah sebelum mobil jalan - Breakpoints seperti pos pemeriksaan di jalan - Watch Expressions seperti sensor yang memantau kondisi mesin - Call Stack seperti rute perjalanan yang sudah dilalui - Source Maps seperti peta detail yang menunjukkan lokasi setiap komponen

## Point Penting

1. **Development Tools**
   * IDE/Editor setup
   * Source maps
   * Breakpoints
   * Watch expressions
2. **Debugging Techniques**
   * Console debugging
   * Step debugging
   * Conditional breakpoints
   * Exception handling
3. **Common Issues**
   * Type errors
   * Runtime errors
   * Async debugging
   * Memory leaks
4. **Performance Debugging**
   * Profiling
   * Memory analysis
   * Network debugging
   * Timeline analysis
5. **Best Practices**
   * Error handling
   * Logging
   * Testing
   * Documentation

## Contoh Kode dan Penjelasan

```typescript // 1. Basic Debugging Setup // tsconfig.json { “compilerOptions”: { “sourceMap”: true, “outDir”: “./dist”, “rootDir”: “./src”, “strict”: true } }

// 2. Console Debugging function calculateTotal(items: { price: number; quantity: number }[]): number { console.log(‘Starting calculation…’); // Basic logging

let total = 0;  
for (const item of items) {  
 console.log(\`Processing item: \${JSON.stringify(item)}\`);  
 total += item.price \* item.quantity;  
 console.log(\`Running total: \${total}\`);  
}  
  
console.log(\`Final total: \${total}\`);  
return total;

}

// 3. Debug Class dengan Type Checking class ShoppingCart { private items: Map<string, number> = new Map();

addItem(id: string, quantity: number): void {  
 console.assert(quantity > 0, 'Quantity must be positive');  
   
 const currentQuantity = this.items.get(id) || 0;  
 this.items.set(id, currentQuantity + quantity);  
   
 console.debug(\`Added \${quantity} of item \${id}\`);  
}  
  
removeItem(id: string, quantity: number): void {  
 console.assert(this.items.has(id), \`Item \${id} not found\`);  
   
 const currentQuantity = this.items.get(id) || 0;  
 const newQuantity = Math.max(0, currentQuantity - quantity);  
   
 if (newQuantity === 0) {  
 this.items.delete(id);  
 console.debug(\`Removed item \${id} completely\`);  
 } else {  
 this.items.set(id, newQuantity);  
 console.debug(\`Updated quantity of item \${id} to \${newQuantity}\`);  
 }  
}  
  
getTotal(prices: Map<string, number>): number {  
 let total = 0;  
   
 this.items.forEach((quantity, id) => {  
 const price = prices.get(id);  
 console.assert(price !== undefined, \`Price not found for item \${id}\`);  
   
 if (price !== undefined) {  
 total += price \* quantity;  
 console.debug(\`Added \${price \* quantity} for item \${id}\`);  
 }  
 });  
   
 return total;  
}

}

// 4. Async Debugging async function fetchUserData(userId: string): Promise { try { console.time(‘fetchUserData’); console.log(`Fetching data for user ${userId}`);

const response = await fetch(\`/api/users/\${userId}\`);  
 if (!response.ok) {  
 throw new Error(\`HTTP error! status: \${response.status}\`);  
 }  
  
 const data = await response.json();  
 console.log('Received user data:', data);  
  
 return data;  
} catch (error) {  
 console.error('Error fetching user data:', error);  
 throw error;  
} finally {  
 console.timeEnd('fetchUserData');  
}

}

// 5. Performance Debugging class PerformanceMonitor { private measurements: Map<string, number[]> = new Map();

start(label: string): void {  
 performance.mark(\`\${label}-start\`);  
}  
  
end(label: string): void {  
 performance.mark(\`\${label}-end\`);  
 performance.measure(label, \`\${label}-start\`, \`\${label}-end\`);  
  
 const measure = performance.getEntriesByName(label).pop();  
 if (measure) {  
 const measurements = this.measurements.get(label) || [];  
 measurements.push(measure.duration);  
 this.measurements.set(label, measurements);  
 }  
}  
  
getStats(label: string): { avg: number; min: number; max: number } {  
 const measurements = this.measurements.get(label) || [];  
 if (measurements.length === 0) {  
 return { avg: 0, min: 0, max: 0 };  
 }  
  
 const sum = measurements.reduce((a, b) => a + b, 0);  
 return {  
 avg: sum / measurements.length,  
 min: Math.min(...measurements),  
 max: Math.max(...measurements)  
 };  
}

}

// 6. Error Boundary Pattern class ErrorBoundary { private static instance: ErrorBoundary; private errorHandlers: Set<(error: Error) => void> = new Set();

private constructor() {  
 window.addEventListener('error', (event) => {  
 this.handleError(event.error);  
 });  
  
 window.addEventListener('unhandledrejection', (event) => {  
 this.handleError(event.reason);  
 });  
}  
  
static getInstance(): ErrorBoundary {  
 if (!ErrorBoundary.instance) {  
 ErrorBoundary.instance = new ErrorBoundary();  
 }  
 return ErrorBoundary.instance;  
}  
  
addHandler(handler: (error: Error) => void): void {  
 this.errorHandlers.add(handler);  
}  
  
removeHandler(handler: (error: Error) => void): void {  
 this.errorHandlers.delete(handler);  
}  
  
private handleError(error: Error): void {  
 console.error('Global error caught:', error);  
 this.errorHandlers.forEach(handler => {  
 try {  
 handler(error);  
 } catch (e) {  
 console.error('Error in error handler:', e);  
 }  
 });  
}

}

// 7. Debug Utilities class DebugUtils { static logObject(obj: unknown, depth = 2): void { console.log( JSON.stringify(obj, null, depth) ); }

static async measureAsync<T>(  
 label: string,  
 fn: () => Promise<T>  
): Promise<T> {  
 console.time(label);  
 try {  
 return await fn();  
 } finally {  
 console.timeEnd(label);  
 }  
}  
  
static createLogger(namespace: string) {  
 return {  
 log: (message: string, ...args: unknown[]) =>  
 console.log(\`[\${namespace}] \${message}\`, ...args),  
 error: (message: string, ...args: unknown[]) =>  
 console.error(\`[\${namespace}] \${message}\`, ...args),  
 warn: (message: string, ...args: unknown[]) =>  
 console.warn(\`[\${namespace}] \${message}\`, ...args),  
 debug: (message: string, ...args: unknown[]) =>  
 console.debug(\`[\${namespace}] \${message}\`, ...args)  
 };  
}

}

// 8. Memory Leak Detection class MemoryLeakDetector { private static instance: MemoryLeakDetector; private intervalId?: number; private lastHeapSize: number = 0; private growthCount: number = 0;

private constructor() {}  
  
static getInstance(): MemoryLeakDetector {  
 if (!MemoryLeakDetector.instance) {  
 MemoryLeakDetector.instance = new MemoryLeakDetector();  
 }  
 return MemoryLeakDetector.instance;  
}  
  
startMonitoring(threshold: number = 3): void {  
 this.intervalId = window.setInterval(() => {  
 const heapSize = performance.memory?.usedJSHeapSize;  
 if (heapSize && heapSize > this.lastHeapSize) {  
 this.growthCount++;  
 if (this.growthCount >= threshold) {  
 console.warn('Possible memory leak detected!');  
 console.log('Heap size:', heapSize);  
 console.log('Growth count:', this.growthCount);  
 }  
 } else {  
 this.growthCount = 0;  
 }  
 this.lastHeapSize = heapSize || 0;  
 }, 1000);  
}  
  
stopMonitoring(): void {  
 if (this.intervalId) {  
 clearInterval(this.intervalId);  
 }  
}

} ```

## Cara Kerja Debugging

1. **Source Maps**:
   * Code mapping
   * Stack trace mapping
   * Breakpoint mapping
   * Variable inspection
2. **Debugging Flow**:
   * Breakpoint hit
   * Call stack analysis
   * Variable inspection
   * Step execution
3. **Error Handling**:
   * Error capture
   * Stack trace analysis
   * Error reporting
   * Error recovery

## Tips dan Trik

1. **Effective Console Usage**

* // ✅ Gunakan console methods yang tepat  
  const logger = {  
   info: (message: string) => console.log(\`ℹ️ \${message}\`),  
   warn: (message: string) => console.warn(\`⚠️ \${message}\`),  
   error: (message: string) => console.error(\`❌ \${message}\`),  
   debug: (message: string) => console.debug(\`🔍 \${message}\`)  
  };

1. **Conditional Breakpoints**

* // ✅ Gunakan debugger statement dengan kondisi  
  function processItems(items: any[]) {  
   items.forEach((item, index) => {  
   if (item.id === 'problematic') {  
   debugger; // Conditional break  
   }  
   // Process item  
   });  
  }

1. **Performance Monitoring**

* // ✅ Gunakan Performance API  
  class PerformanceMonitor {  
   static async measure<T>(  
   label: string,  
   fn: () => Promise<T>  
   ): Promise<T> {  
   const start = performance.now();  
   try {  
   return await fn();  
   } finally {  
   const duration = performance.now() - start;  
   console.log(\`\${label} took \${duration}ms\`);  
   }  
   }  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Console.log Debugging**

* // ❌ Buruk: Console.log berlebihan  
  function processData(data: any) {  
   console.log('Starting');  
   console.log('Data:', data);  
   // Process  
   console.log('Done');  
  }  
    
  // ✅ Baik: Gunakan debug levels  
  function processData(data: any) {  
   logger.debug('Processing data:', data);  
   // Process  
   logger.info('Processing complete');  
  }

1. **Tidak Menggunakan Source Maps**

* // ❌ Buruk: Tanpa source maps  
  {  
   "compilerOptions": {  
   "sourceMap": false  
   }  
  }  
    
  // ✅ Baik: Dengan source maps  
  {  
   "compilerOptions": {  
   "sourceMap": true,  
   "inlineSourceMap": false,  
   "inlineSources": true  
   }  
  }

1. **Mengabaikan Error Handling**

* // ❌ Buruk: Error diabaikan  
  async function fetchData() {  
   const response = await fetch('/api/data');  
   return response.json();  
  }  
    
  // ✅ Baik: Proper error handling  
  async function fetchData() {  
   try {  
   const response = await fetch('/api/data');  
   if (!response.ok) {  
   throw new Error(\`HTTP error! status: \${response.status}\`);  
   }  
   return await response.json();  
   } catch (error) {  
   logger.error('Error fetching data:', error);  
   throw error;  
   }  
  }

### Solusi:

1. Gunakan source maps
2. Implementasi proper logging
3. Manfaatkan debugging tools
4. Terapkan error handling
5. Monitor performance