Table of Contents

# Bab 19: Debugging dan Troubleshooting di TypeScript

## Penjelasan Materi

Debugging dan troubleshooting adalah keterampilan penting dalam pengembangan TypeScript. Proses ini melibatkan identifikasi, analisis, dan perbaikan bug serta masalah dalam kode TypeScript. TypeScript menyediakan berbagai tools dan teknik yang membantu developer dalam proses debugging.

## Analogi yang Mudah Dipahami

Bayangkan debugging seperti pekerjaan detektif: - Bug seperti kasus yang perlu dipecahkan - Debugger tools seperti peralatan investigasi - Breakpoints seperti titik pengamatan - Stack trace seperti jejak petunjuk - Console logs seperti catatan penyelidikan

## Point Penting

1. **IDE Debugging**
   * Breakpoints
   * Watch expressions
   * Call stack
   * Variable inspection
   * Step through code
2. **Browser Debugging**
   * Chrome DevTools
   * Source maps
   * Network tab
   * Performance profiling
   * Memory analysis
3. **Node.js Debugging**
   * Node inspector
   * Debug flags
   * Process monitoring
   * Memory leaks
   * CPU profiling
4. **Common Issues**
   * Type errors
   * Runtime errors
   * Build issues
   * Performance problems
   * Memory leaks
5. **Debugging Tools**
   * VS Code debugger
   * Chrome DevTools
   * Node.js debugger
   * TypeScript compiler
   * Logging tools

## Contoh Kode dan Penjelasan

```typescript // 1. Basic Debugging Setup // launch.json { “version”: “0.2.0”, “configurations”: [ { “type”: “node”, “request”: “launch”, “name”: “Debug TypeScript”, “program”: “${workspaceFolder}/src/index.ts”, “preLaunchTask”: “tsc: build”, “outFiles”: [“${workspaceFolder}/dist/\*\*/\*.js”], “sourceMaps”: true } ] }

// 2. Debug Logging Utility class DebugLogger { private static instance: DebugLogger; private logLevel: ‘debug’ | ‘info’ | ‘warn’ | ‘error’ = ‘info’;

private constructor() {}  
  
static getInstance(): DebugLogger {  
 if (!DebugLogger.instance) {  
 DebugLogger.instance = new DebugLogger();  
 }  
 return DebugLogger.instance;  
}  
  
setLogLevel(level: 'debug' | 'info' | 'warn' | 'error'): void {  
 this.logLevel = level;  
}  
  
debug(message: string, ...args: any[]): void {  
 if (this.logLevel === 'debug') {  
 console.debug(\`[DEBUG] \${message}\`, ...args);  
 }  
}  
  
info(message: string, ...args: any[]): void {  
 if (['debug', 'info'].includes(this.logLevel)) {  
 console.info(\`[INFO] \${message}\`, ...args);  
 }  
}  
  
warn(message: string, ...args: any[]): void {  
 if (['debug', 'info', 'warn'].includes(this.logLevel)) {  
 console.warn(\`[WARN] \${message}\`, ...args);  
 }  
}  
  
error(message: string, ...args: any[]): void {  
 console.error(\`[ERROR] \${message}\`, ...args);  
}

}

// 3. Error Tracking class ErrorTracker { private errors: Error[] = [];

trackError(error: Error): void {  
 this.errors.push(error);  
 console.error('Error tracked:', error);  
   
 // Send to error reporting service  
 this.reportError(error);  
}  
  
private reportError(error: Error): void {  
 // Implementation for error reporting  
 const errorReport = {  
 message: error.message,  
 stack: error.stack,  
 timestamp: new Date().toISOString()  
 };  
   
 // Send to service  
 console.log('Error report:', errorReport);  
}  
  
getErrorStats(): { count: number; lastError: Error | null } {  
 return {  
 count: this.errors.length,  
 lastError: this.errors[this.errors.length - 1] || null  
 };  
}

}

// 4. Performance Monitoring class PerformanceMonitor { private measurements: Map<string, number[]> = new Map();

startMeasurement(label: string): void {  
 performance.mark(\`\${label}-start\`);  
}  
  
endMeasurement(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) || [];  
 return {  
 avg: measurements.reduce((a, b) => a + b, 0) / measurements.length,  
 min: Math.min(...measurements),  
 max: Math.max(...measurements)  
 };  
}

}

// 5. Memory Leak Detection class MemoryLeakDetector { private snapshots: any[] = []; private interval: NodeJS.Timeout | null = null;

startMonitoring(intervalMs: number = 60000): void {  
 this.interval = setInterval(() => {  
 const snapshot = process.memoryUsage();  
 this.snapshots.push({  
 ...snapshot,  
 timestamp: Date.now()  
 });  
  
 this.analyzeMemoryUsage();  
 }, intervalMs);  
}  
  
stopMonitoring(): void {  
 if (this.interval) {  
 clearInterval(this.interval);  
 this.interval = null;  
 }  
}  
  
private analyzeMemoryUsage(): void {  
 if (this.snapshots.length < 2) return;  
  
 const latest = this.snapshots[this.snapshots.length - 1];  
 const previous = this.snapshots[this.snapshots.length - 2];  
  
 const heapGrowth = latest.heapUsed - previous.heapUsed;  
 if (heapGrowth > 100 \* 1024 \* 1024) { // 100MB threshold  
 console.warn('Potential memory leak detected');  
 console.warn(\`Heap grew by \${heapGrowth / 1024 / 1024}MB\`);  
 }  
}

}

// 6. Type Checking Utility class TypeChecker { static isNumber(value: unknown): value is number { return typeof value === ‘number’ && !isNaN(value); }

static isString(value: unknown): value is string {  
 return typeof value === 'string';  
}  
  
static isArray<T>(value: unknown): value is T[] {  
 return Array.isArray(value);  
}  
  
static isObject(value: unknown): value is object {  
 return typeof value === 'object' && value !== null && !Array.isArray(value);  
}  
  
static validateShape<T>(value: unknown, shape: Record<keyof T, string>): value is T {  
 if (!this.isObject(value)) return false;  
  
 return Object.entries(shape).every(([key, type]) => {  
 const val = (value as any)[key];  
 switch (type) {  
 case 'string': return this.isString(val);  
 case 'number': return this.isNumber(val);  
 case 'array': return this.isArray(val);  
 case 'object': return this.isObject(val);  
 default: return false;  
 }  
 });  
}

}

// 7. Stack Trace Analysis class StackTraceAnalyzer { static getStackTrace(): string { const error = new Error(); return error.stack || ’’; }

static parseStackTrace(stack: string): Array<{  
 function: string;  
 file: string;  
 line: number;  
 column: number;  
}> {  
 return stack  
 .split('\\n')  
 .slice(1)  
 .map(line => {  
 const match = line.match(/at (.+) \((.+):(\d+):(\d+)\)/);  
 if (!match) return null;  
  
 return {  
 function: match[1],  
 file: match[2],  
 line: parseInt(match[3], 10),  
 column: parseInt(match[4], 10)  
 };  
 })  
 .filter((frame): frame is NonNullable<typeof frame> => frame !== null);  
}

}

// 8. Debug Configuration Manager class DebugConfig { private static config: Record<string, any> = { enableDebugLogs: false, logLevel: ‘info’, enablePerformanceMonitoring: false, enableMemoryMonitoring: false };

static setConfig(key: string, value: any): void {  
 this.config[key] = value;  
 console.log(\`Debug config updated: \${key} = \${value}\`);  
}  
  
static getConfig(key: string): any {  
 return this.config[key];  
}  
  
static enableDebugMode(): void {  
 this.setConfig('enableDebugLogs', true);  
 this.setConfig('logLevel', 'debug');  
 this.setConfig('enablePerformanceMonitoring', true);  
 this.setConfig('enableMemoryMonitoring', true);  
}

}

// 9. Async Error Handler class AsyncErrorHandler { static async wrap( promise: Promise, errorHandler?: (error: Error) => void ): Promise<[T | null, Error | null]> { try { const result = await promise; return [result, null]; } catch (error) { if (errorHandler && error instanceof Error) { errorHandler(error); } return [null, error instanceof Error ? error : new Error(String(error))]; } }

static createSafeAsync<T>(  
 fn: (...args: any[]) => Promise<T>  
): (...args: any[]) => Promise<[T | null, Error | null]> {  
 return async (...args: any[]) => {  
 return this.wrap(fn(...args));  
 };  
}

}

// 10. Debug State Manager class DebugStateManager { private static instance: DebugStateManager; private state: Map<string, any> = new Map();

private constructor() {}  
  
static getInstance(): DebugStateManager {  
 if (!DebugStateManager.instance) {  
 DebugStateManager.instance = new DebugStateManager();  
 }  
 return DebugStateManager.instance;  
}  
  
setState(key: string, value: any): void {  
 this.state.set(key, value);  
 console.debug(\`Debug state updated: \${key}\`, value);  
}  
  
getState(key: string): any {  
 return this.state.get(key);  
}  
  
clearState(): void {  
 this.state.clear();  
 console.debug('Debug state cleared');  
}  
  
dumpState(): Record<string, any> {  
 return Object.fromEntries(this.state);  
}

} ```

## Cara Kerja Debugging

1. **Proses Debugging**:
   * Identifikasi masalah
   * Set breakpoints
   * Inspect variables
   * Step through code
   * Analyze stack trace
2. **Tools Usage**:
   * IDE debugger setup
   * Browser tools configuration
   * Node.js debugging
   * Source maps handling
   * Logging setup
3. **Problem Resolution**:
   * Error analysis
   * Root cause identification
   * Solution implementation
   * Verification
   * Documentation

## Tips dan Trik

1. **Effective Logging**

* // ✅ Structured logging  
  class Logger {  
   log(level: string, message: string, context: object = {}) {  
   console.log(JSON.stringify({  
   timestamp: new Date().toISOString(),  
   level,  
   message,  
   ...context  
   }));  
   }  
  }

1. **Breakpoint Usage**

* // ✅ Conditional breakpoints  
  function processData(data: any[]) {  
   data.forEach((item, index) => {  
   // Add conditional breakpoint here when item meets certain criteria  
   if (item.status === 'error') {  
   debugger; // This will break only for error items  
   }  
   // Process item  
   });  
  }

1. **Error Handling**

* // ✅ Proper error handling  
  async function fetchData(): Promise<Data> {  
   try {  
   const response = await fetch('/api/data');  
   if (!response.ok) {  
   throw new Error(\`HTTP error! status: \${response.status}\`);  
   }  
   return await response.json();  
   } catch (error) {  
   console.error('Fetch error:', error);  
   throw error;  
   }  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Console.log Debugging**

* // ❌ Buruk: Unstructured logging  
  function processUser(user: User) {  
   console.log(user);  
   // Process user  
   console.log('done');  
  }  
    
  // ✅ Baik: Structured debugging  
  function processUser(user: User) {  
   logger.debug('Processing user', { userId: user.id });  
   // Process user  
   logger.info('User processing completed', { userId: user.id });  
  }

1. **Tidak Menggunakan Source Maps**

* // ❌ Buruk: No source maps  
  {  
   "compilerOptions": {  
   "sourceMap": false  
   }  
  }  
    
  // ✅ Baik: Enable source maps  
  {  
   "compilerOptions": {  
   "sourceMap": true  
   }  
  }

1. **Mengabaikan Error Handling**

* // ❌ Buruk: Ignoring errors  
  async function getData() {  
   const data = await fetch('/api/data');  
   return data.json();  
  }  
    
  // ✅ Baik: Proper error handling  
  async function getData() {  
   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('Failed to fetch data', { error });  
   throw error;  
   }  
  }

### Solusi:

1. Gunakan proper debugging tools
2. Implementasi structured logging
3. Enable source maps
4. Implement proper error handling
5. Use TypeScript strict mode