Table of Contents

# Bab 8: Error Handling di TypeScript

## Penjelasan Materi

Error Handling di TypeScript adalah aspek penting dalam pengembangan aplikasi yang robust dan dapat diandalkan. TypeScript menyediakan berbagai cara untuk menangani error, dari penggunaan try-catch tradisional hingga tipe union yang sophisticated untuk penanganan error yang lebih elegan. Penanganan error yang baik membantu mencegah crash aplikasi dan memberikan feedback yang berguna kepada pengguna dan developer.

## Analogi yang Mudah Dipahami

Bayangkan Error Handling seperti jaring pengaman di sirkus: - Try-catch seperti jaring pengaman untuk menangkap pemain trapeze - Error types seperti jenis-jenis jaring untuk berbagai akrobat - Error propagation seperti sistem jaring bertingkat - Custom errors seperti jaring khusus untuk gerakan tertentu

Atau seperti prosedur keselamatan di pabrik: - Error handling seperti prosedur keselamatan - Try-catch seperti protokol penanganan kecelakaan - Error types seperti klasifikasi insiden - Error propagation seperti rantai pelaporan - Custom errors seperti prosedur khusus untuk kasus tertentu

## Point Penting

1. **Basic Error Handling**
   * Try-catch blocks
   * Error object
   * Finally clause
   * Throw statement
2. **Custom Error Types**
   * Extending Error class
   * Custom error properties
   * Error hierarchies
   * Type guards
3. **Async Error Handling**
   * Promise rejection
   * Async/await errors
   * Error boundaries
   * Error propagation
4. **Type-Safe Error Handling**
   * Union types
   * Result types
   * Option types
   * Error discrimination
5. **Best Practices**
   * Error logging
   * Error recovery
   * User feedback
   * Debugging

## Contoh Kode dan Penjelasan

```typescript // 1. Basic Error Handling function divide(a: number, b: number): number { if (b === 0) { throw new Error(“Tidak bisa membagi dengan nol”); } return a / b; }

try { const result = divide(10, 0); console.log(result); } catch (error) { if (error instanceof Error) { console.error(“Terjadi kesalahan:”, error.message); } } finally { console.log(“Operasi selesai”); }

// 2. Custom Error Types class ValidationError extends Error { constructor( message: string, public field?: string ) { super(message); this.name = “ValidationError”; } }

class DatabaseError extends Error { constructor( message: string, public code: number ) { super(message); this.name = “DatabaseError”; } }

// 3. Type Guard untuk Error function isValidationError(error: unknown): error is ValidationError { return error instanceof ValidationError; }

function isDatabaseError(error: unknown): error is DatabaseError { return error instanceof DatabaseError; }

// 4. Error Handling dengan Type Guards function handleError(error: unknown) { if (isValidationError(error)) { console.error(`Validasi gagal pada field ${error.field}: ${error.message}`); } else if (isDatabaseError(error)) { console.error(`Database error [${error.code}]: ${error.message}`); } else if (error instanceof Error) { console.error(“Error umum:”, error.message); } else { console.error(“Terjadi kesalahan yang tidak diketahui”); } }

// 5. Async Error Handling async function fetchData(url: string): Promise { try { const response = await fetch(url); if (!response.ok) { throw new Error(`HTTP error! status: ${response.status}`); } return await response.json(); } catch (error) { if (error instanceof Error) { throw new DatabaseError(error.message, 500); } throw error; } }

// 6. Result Type Pattern interface Success { success: true; data: T; }

interface Failure { success: false; error: Error; }

type Result = Success | Failure;

function tryOperation(operation: () => T): Result { try { return { success: true, data: operation() }; } catch (error) { return { success: false, error: error instanceof Error ? error : new Error(String(error)) }; } }

// 7. Option Type Pattern type Option = Some | None;

interface Some { type: “some”; value: T; }

interface None { type: “none”; }

function findUser(id: number): Option { const user = users.find(u => u.id === id); return user ? { type: “some”, value: user } : { type: “none” }; }

// 8. Error Boundary Pattern class ErrorBoundary { private static instance: ErrorBoundary; private errorHandlers: Map<string, (error: Error) => void>;

private constructor() {  
 this.errorHandlers = new Map();  
}  
  
static getInstance(): ErrorBoundary {  
 if (!ErrorBoundary.instance) {  
 ErrorBoundary.instance = new ErrorBoundary();  
 }  
 return ErrorBoundary.instance;  
}  
  
register(type: string, handler: (error: Error) => void): void {  
 this.errorHandlers.set(type, handler);  
}  
  
handle(error: Error): void {  
 const handler = this.errorHandlers.get(error.name);  
 if (handler) {  
 handler(error);  
 } else {  
 console.error("Unhandled error:", error);  
 }  
}

}

// 9. Praktik Real-world interface User { id: number; name: string; email: string; }

class UserService { async createUser(data: Omit<User, “id”>): Promise<Result> { try { // Validasi if (!data.email.includes(“@”)) { throw new ValidationError(“Email tidak valid”, “email”); }

// Simulasi database operation  
 const user: User = {  
 id: Math.random(),  
 ...data  
 };  
  
 return {  
 success: true,  
 data: user  
 };  
 } catch (error) {  
 if (isValidationError(error)) {  
 return {  
 success: false,  
 error  
 };  
 }  
   
 return {  
 success: false,  
 error: new Error("Gagal membuat user")  
 };  
 }  
}  
  
async getUser(id: number): Promise<Option<User>> {  
 try {  
 // Simulasi database query  
 const user = await this.findUserById(id);  
 return user  
 ? { type: "some", value: user }  
 : { type: "none" };  
 } catch (error) {  
 throw new DatabaseError(  
 \`Gagal mengambil user dengan id \${id}\`,  
 500  
 );  
 }  
}  
  
private async findUserById(id: number): Promise<User | null> {  
 // Simulasi database query  
 return null;  
}

}

// 10. Penggunaan async function main() { const userService = new UserService(); const errorBoundary = ErrorBoundary.getInstance();

// Register error handlers  
errorBoundary.register("ValidationError", (error: Error) => {  
 if (isValidationError(error)) {  
 console.error(\`Validasi: \${error.field} - \${error.message}\`);  
 }  
});  
  
errorBoundary.register("DatabaseError", (error: Error) => {  
 if (isDatabaseError(error)) {  
 console.error(\`Database [\${error.code}]: \${error.message}\`);  
 }  
});  
  
try {  
 // Create user  
 const createResult = await userService.createUser({  
 name: "John",  
 email: "invalid-email"  
 });  
  
 if (!createResult.success) {  
 errorBoundary.handle(createResult.error);  
 return;  
 }  
  
 console.log("User created:", createResult.data);  
  
 // Get user  
 const getResult = await userService.getUser(1);  
   
 if (getResult.type === "none") {  
 console.log("User tidak ditemukan");  
 return;  
 }  
  
 console.log("User found:", getResult.value);  
} catch (error) {  
 if (error instanceof Error) {  
 errorBoundary.handle(error);  
 }  
}

} ```

## Cara Kerja Error Handling

1. **Error Propagation**:
   * Error mengalir ke atas call stack
   * Dapat ditangkap di level mana saja
   * Bisa dilempar ulang setelah ditangani
2. **Type Safety**:
   * TypeScript memvalidasi tipe error
   * Memastikan penanganan yang tepat
   * Mencegah runtime errors
3. **Async Handling**:
   * Promise rejection chain
   * Async/await error flow
   * Error boundary pattern

## Tips dan Trik

1. **Specific Error Types**

* // ✅ Gunakan error type yang spesifik  
  class NotFoundError extends Error {  
   constructor(resource: string, id: string) {  
   super(\`\${resource} dengan id \${id} tidak ditemukan\`);  
   this.name = "NotFoundError";  
   }  
  }

1. **Error Recovery**

* // ✅ Implementasi recovery strategy  
  async function retryOperation<T>(  
   operation: () => Promise<T>,  
   retries: number = 3  
  ): Promise<T> {  
   try {  
   return await operation();  
   } catch (error) {  
   if (retries > 0) {  
   return retryOperation(operation, retries - 1);  
   }  
   throw error;  
   }  
  }

1. **Structured Error Handling**

* // ✅ Gunakan pattern yang terstruktur  
  type ApiResponse<T> = {  
   data?: T;  
   error?: {  
   code: number;  
   message: string;  
   };  
  };  
    
  async function fetchApi<T>(url: string): Promise<ApiResponse<T>> {  
   try {  
   const response = await fetch(url);  
   const data = await response.json();  
   return { data };  
   } catch (error) {  
   return {  
   error: {  
   code: 500,  
   message: error instanceof Error ? error.message : "Unknown error"  
   }  
   };  
   }  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Catching Everything**

* // ❌ Buruk: Menangkap semua error tanpa diskriminasi  
  try {  
   // some operation  
  } catch (error) {  
   console.log(error);  
  }  
    
  // ✅ Baik: Handle error secara spesifik  
  try {  
   // some operation  
  } catch (error) {  
   if (error instanceof ValidationError) {  
   // handle validation error  
   } else if (error instanceof DatabaseError) {  
   // handle database error  
   } else {  
   // handle unknown error  
   }  
  }

1. **Ignoring Error Types**

* // ❌ Buruk: Mengabaikan tipe error  
  function process(data: any): void {  
   try {  
   // process data  
   } catch (error: any) {  
   console.log(error.message);  
   }  
  }  
    
  // ✅ Baik: Menggunakan type guard  
  function process(data: unknown): void {  
   try {  
   // process data  
   } catch (error) {  
   if (error instanceof Error) {  
   console.log(error.message);  
   }  
   }  
  }

1. **Silent Failures**

* // ❌ Buruk: Menelan error  
  function doSomething() {  
   try {  
   riskyOperation();  
   } catch {  
   // nothing here  
   }  
  }  
    
  // ✅ Baik: Proper error handling  
  function doSomething() {  
   try {  
   riskyOperation();  
   } catch (error) {  
   logError(error);  
   notifyUser("Operasi gagal");  
   throw error;  
   }  
  }

### Solusi:

1. Gunakan custom error types
2. Implementasi proper error recovery
3. Lakukan logging yang memadai
4. Berikan feedback yang jelas ke user
5. Gunakan type-safe error handling patterns