Error Handling Standards

Estratégias por Linguagem

TypeScript/JavaScript

Result Pattern (Recomendado)

```
type Result<T, E = Error> =
  | { success: true; data: T }
  | { success: false; error: E };

// Usage
async function getUserById(id: number): Promise<Result<User, UserError>> {
  try {
    const user = await userRepository.findById(id);
    if (!user) {
      return { success: false, error: new UserNotFoundError(`User ${id} not found`) };
    }
    return { success: true, data: user };
  } catch (error) {
    return { success: false, error: error as UserError };
  }
}

// Consumer
const result = await getUserById(123);
if (result.success) {
  console.log(result.data.name); // Type-safe access
} else {
  console.error(result.error.message);
}
```

Error Types Hierarchy

```
abstract class AppError extends Error {
  abstract readonly code: string;
  abstract readonly statusCode: number;

  constructor(message: string, public readonly context?: Record<string, any>) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends AppError {
  readonly code = 'VALIDATION_ERROR';
  readonly statusCode = 400;
}

class BusinessError extends AppError {
  readonly code = 'BUSINESS_ERROR';
  readonly statusCode = 422;
}

class InfrastructureError extends AppError {
  readonly code = 'INFRASTRUCTURE_ERROR';
  readonly statusCode = 500;
}

class AuthenticationError extends AppError {
  readonly code = 'AUTHENTICATION_ERROR';
  readonly statusCode = 401;
}

class AuthorizationError extends AppError {
  readonly code = 'AUTHORIZATION_ERROR';
  readonly statusCode = 403;
}
```

React Error Boundaries

```
interface ErrorBoundaryState {
  hasError: boolean;
  error?: Error;
}

class ErrorBoundary extends React.Component<
  React.PropsWithChildren<{}>,
  ErrorBoundaryState
> {
  constructor(props: React.PropsWithChildren<{}>) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error: Error): ErrorBoundaryState {
    return { hasError: true, error };
  }

  componentDidCatch(error: Error, errorInfo: React.ErrorInfo) {
    console.error('Error caught by boundary:', error, errorInfo);
    // Send to error reporting service
    errorReportingService.captureException(error, {
      extra: errorInfo,
      tags: { component: 'ErrorBoundary' }
    });
  }

  render() {
    if (this.state.hasError) {
      return (
        <div className="error-fallback">
          <h2>Something went wrong</h2>
          <p>{this.state.error?.message}</p>
          <button onClick={() => this.setState({ hasError: false })}>
            Try again
          </button>
        </div>
      );
    }

    return this.props.children;
  }
}
```

Python

Exception Hierarchy

```python
class AppError(Exception):
    """Base application error."""

    def __init__(self, message: str, context: Optional[Dict[str, Any]] = None):
        super().__init__(message)
        self.message = message
        self.context = context or {}
        self.timestamp = datetime.utcnow()

class ValidationError(AppError):
    """Raised when input validation fails."""
    pass

class BusinessRuleError(AppError):
    """Raised when business rule is violated."""
    pass

class InfrastructureError(AppError):
    """Raised when infrastructure operation fails."""
    pass

class UserNotFoundError(BusinessRuleError):
    """Raised when user is not found."""
    pass

class PaymentFailedError(BusinessRuleError):
    """Raised when payment processing fails."""
    pass
```

Exception Chaining

```python
async def process_payment(payment_data: PaymentData) -> Payment:
    try:
        # Validate payment data
        validated_data = await validate_payment_data(payment_data)

        # Process with external service
        result = await payment_gateway.process(validated_data)

        # Save to database
        payment = await payment_repository.save(result)

        return payment

    except ValidationError as e:
        # Re-raise validation errors as-is
        raise
    except PaymentGatewayError as e:
        # Chain the original exception
        raise PaymentFailedError(
            f"Payment processing failed: {e.message}",
            context={"payment_id": payment_data.id, "gateway_error": str(e)}
        ) from e
    except DatabaseError as e:
        # Chain database errors
        raise InfrastructureError(
            "Failed to save payment",
            context={"payment_id": payment_data.id}
        ) from e
    except Exception as e:
        # Catch-all for unexpected errors
        logger.exception("Unexpected error in payment processing")
        raise InfrastructureError(
            "Unexpected error occurred",
            context={"payment_id": payment_data.id}
        ) from e
```

Context Managers for Resource Cleanup

```python
from contextlib import asynccontextmanager
from typing import AsyncGenerator

@asynccontextmanager
async def database_transaction(
    session: AsyncSession
) -> AsyncGenerator[AsyncSession, None]:
    """Context manager for database transactions with automatic rollback."""
    try:
        await session.begin()
        yield session
        await session.commit()
    except Exception as e:
        await session.rollback()
        logger.error(f"Transaction rolled back due to error: {e}")
        raise
    finally:
        await session.close()

# Usage
async def create_user_with_profile(user_data: UserCreateData) -> User:
    async with database_transaction(get_session()) as session:
        # Create user
        user = User(**user_data.dict())
        session.add(user)
        await session.flush()  # Get user.id

        # Create profile
        profile = UserProfile(user_id=user.id, **user_data.profile_data)
        session.add(profile)

        return user  # Commit happens automatically
```

Logging e Observabilidade

Structured Logging

```typescript
interface LogContext {
  requestId?: string;
  userId?: string;
  operation?: string;
  duration?: number;
  [key: string]: any;
}

class Logger {
  private static instance: Logger;

  static getInstance(): Logger {
    if (!Logger.instance) {
      Logger.instance = new Logger();
    }
    return Logger.instance;
  }

  error(message: string, error: Error, context: LogContext = {}) {
    const logEntry = {
      level: 'error',
      message,
      error: {
        name: error.name,
        message: error.message,
        stack: error.stack,
      },
      context,
      timestamp: new Date().toISOString(),
    };

    console.error(JSON.stringify(logEntry));

    // Send to external service
    this.sendToExternalService(logEntry);
  }

  warn(message: string, context: LogContext = {}) {
    const logEntry = {
      level: 'warn',
      message,
      context,
      timestamp: new Date().toISOString(),
    };

    console.warn(JSON.stringify(logEntry));
  }

  info(message: string, context: LogContext = {}) {
    const logEntry = {
      level: 'info',
      message,
      context,
      timestamp: new Date().toISOString(),
    };

    console.info(JSON.stringify(logEntry));
  }
}
```

Python Structured Logging

```python
import logging
import json
from typing import Dict, Any, Optional
from datetime import datetime

class StructuredLogger:
    def __init__(self, name: str):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(logging.INFO)

        # JSON formatter
        handler = logging.StreamHandler()
        handler.setFormatter(self._get_json_formatter())
        self.logger.addHandler(handler)

    def _get_json_formatter(self):
        class JSONFormatter(logging.Formatter):
            def format(self, record):
                log_entry = {
                    'timestamp': datetime.utcnow().isoformat(),
                    'level': record.levelname,
                    'message': record.getMessage(),
                    'module': record.module,
                    'function': record.funcName,
                    'line': record.lineno,
                }

                # Add extra context if available
                if hasattr(record, 'context'):
                    log_entry['context'] = record.context

                # Add exception info if available
                if record.exc_info:
                    log_entry['exception'] = {
                        'type': record.exc_info[0].__name__,
                        'message': str(record.exc_info[1]),
                        'traceback': self.formatException(record.exc_info)
                    }

                return json.dumps(log_entry)

        return JSONFormatter()

    def error(self, message: str, context: Optional[Dict[str, Any]] = None, exc_info: b
ool = True):
        self.logger.error(message, extra={'context': context}, exc_info=exc_info)

    def warning(self, message: str, context: Optional[Dict[str, Any]] = None):
        self.logger.warning(message, extra={'context': context})

    def info(self, message: str, context: Optional[Dict[str, Any]] = None):
        self.logger.info(message, extra={'context': context})

# Usage
logger = StructuredLogger(__name__)

async def process_user_registration(user_data: UserRegistrationData):
    context = {
        'operation': 'user_registration',
        'email': user_data.email,
        'request_id': get_request_id()
    }
    try:
        logger.info("Starting user registration", context)

        user = await user_service.create_user(user_data)

        logger.info("User registration completed", {
            **context,
            'user_id': user.id,
            'duration_ms': get_duration()
        })

        return user

    except ValidationError as e:
        logger.warning("User registration validation failed", {
            **context,
            'validation_errors': e.errors
        })
        raise
    except Exception as e:
        logger.error("User registration failed", context)
        raise
```

Recovery Strategies

Retry with Exponential Backoff

```typescript
interface RetryOptions {
  maxAttempts: number;
  baseDelay: number;
  maxDelay: number;
  backoffFactor: number;
}

async function withRetry<T>(
  operation: () => Promise<T>,
  options: RetryOptions = {
    maxAttempts: 3,
    baseDelay: 1000,
    maxDelay: 10000,
    backoffFactor: 2
  }
): Promise<T> {
  let lastError: Error;

  for (let attempt = 1; attempt <= options.maxAttempts; attempt++) {
    try {
      return await operation();
    } catch (error) {
      lastError = error as Error;

      if (attempt === options.maxAttempts) {
        throw lastError;
      }

      // Calculate delay with exponential backoff
      const delay = Math.min(
        options.baseDelay * Math.pow(options.backoffFactor, attempt - 1),
        options.maxDelay
      );

      logger.warn(`Operation failed, retrying in ${delay}ms`, {
        attempt,
        maxAttempts: options.maxAttempts,
        error: lastError.message
      });

      await new Promise(resolve => setTimeout(resolve, delay));
    }
  }

  throw lastError!;
}

// Usage
const result = await withRetry(
  () => externalApiClient.fetchUserData(userId),
  { maxAttempts: 3, baseDelay: 1000, maxDelay: 5000, backoffFactor: 2 }
);
```

Circuit Breaker Pattern

```
enum CircuitState {
  CLOSED = 'CLOSED',
  OPEN = 'OPEN',
  HALF_OPEN = 'HALF_OPEN'
}

class CircuitBreaker {
  private state = CircuitState.CLOSED;
  private failureCount = 0;
  private lastFailureTime?: Date;
  private successCount = 0;

  constructor(
    private readonly failureThreshold: number = 5,
    private readonly recoveryTimeout: number = 60000, // 1 minute
    private readonly successThreshold: number = 3
  ) {}

  async execute<T>(operation: () => Promise<T>): Promise<T> {
    if (this.state === CircuitState.OPEN) {
      if (this.shouldAttemptReset()) {
        this.state = CircuitState.HALF_OPEN;
        this.successCount = 0;
      } else {
        throw new Error('Circuit breaker is OPEN');
      }
    }

    try {
      const result = await operation();
      this.onSuccess();
      return result;
    } catch (error) {
      this.onFailure();
      throw error;
    }
  }

  private onSuccess(): void {
    this.failureCount = 0;

    if (this.state === CircuitState.HALF_OPEN) {
      this.successCount++;
      if (this.successCount >= this.successThreshold) {
        this.state = CircuitState.CLOSED;
      }
    }
  }

  private onFailure(): void {
    this.failureCount++;
    this.lastFailureTime = new Date();

    if (this.failureCount >= this.failureThreshold) {
      this.state = CircuitState.OPEN;
    }
  }

  private shouldAttemptReset(): boolean {
    return this.lastFailureTime &&
           (Date.now() - this.lastFailureTime.getTime()) >= this.recoveryTimeout;
  }
}
```

Anti-Padrões Proibidos

TypeScript/JavaScript

- ❌ `any` type usage sem justificativa
- ❌ Silent error swallowing (`catch` vazio ou apenas `console.log`)
- ❌ Hardcoded error messages
- ❌ Throwing strings ao invés de Error objects
- ❌ Nested try/catch sem necessidade

Python

- ❌ Bare except clauses (`except:` sem tipo)
- ❌ Catching `Exception` sem re-raise
- ❌ Using `pass` em exception handlers
- ❌ Not using exception chaining (`raise ... from e`)
- ❌ Logging e re-raising na mesma função

Geral

- ❌ Exposing internal error details para usuários
- ❌ Not logging sufficient context
- ❌ Inconsistent error response formats
- ❌ Missing correlation IDs
- ❌ Not implementing proper cleanup

Referências

- @ref:global-standards#quality-principles
- @ref:naming-conventions#error-types
- @docs:https://docs.python.org/3/tutorial/errors.html
- @docs:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling