Pertemuan 12: Error Handling dan Exception

Tujuan Pembelajaran

Setelah mengikuti pertemuan ini, mahasiswa diharapkan dapat:

1. Memahami konsep error handling dan exception dalam PHP OOP
2. Membuat dan menggunakan custom exception classes
3. Mengimplementasikan try-catch-finally blocks
4. Mengelola different types of errors dan exceptions
5. Membuat robust error handling system untuk aplikasi production
6. Menggunakan logging untuk debugging dan monitoring
7. Menerapkan defensive programming techniques

Konsep Error Handling

Definisi Exception

Exception adalah object yang mewakili kondisi error atau situasi unexpected yang terjadi selama eksekusi program. Exception memungkinkan program untuk "melompat" dari titik error ke error handler yang sesuai.

Perbedaan Error vs Exception

- **Error**: Masalah yang mencegah script berjalan (syntax error, fatal error)
- **Exception**: Kondisi exceptional yang dapat di-handle oleh program
- **Warning/Notice**: Non-fatal issues yang tidak menghentikan execution

Keuntungan Exception Handling

1. **Separation of Concerns**: Memisahkan business logic dari error handling
2. **Clean Code**: Code lebih readable tanpa banyak if-else untuk error checking
3. **Centralized Error Handling**: Satu tempat untuk handle berbagai jenis error
4. **Graceful Degradation**: Program dapat continue atau terminate dengan elegant

Basic Exception Handling

Try-Catch Block

```php
<?php
try {
    // Code yang mungkin throw exception
    $result = riskyOperation();
    echo $result;
} catch (Exception $e) {
    // Handle exception
    echo "Error occurred: " . $e->getMessage();
}
```

Multiple Catch Blocks

```php
<?php
try {
    $data = processData($input);
} catch (InvalidArgumentException $e) {
    echo "Invalid input: " . $e->getMessage();
} catch (RuntimeException $e) {
    echo "Runtime error: " . $e->getMessage();
} catch (Exception $e) {
    echo "General error: " . $e->getMessage();
}
```

Finally Block (PHP 5.5+)

```php
<?php
try {
    $file = fopen('data.txt', 'r');
    $data = fread($file, 1024);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
} finally {
    // Always executed
    if (isset($file) && $file) {
        fclose($file);
    }
    echo "Cleanup completed";
}
```

Built-in Exception Classes

Exception Hierarchy

```
Throwable (PHP 7+)
├── Error
│   ├── ArithmeticError
│   ├── AssertionError
│   ├── ParseError
│   ├── TypeError
│   └── ...
└── Exception
    ├── BadFunctionCallException
    ├── BadMethodCallException
    ├── DomainException
    ├── InvalidArgumentException
    ├── LengthException
    ├── LogicException
```

```
    ├── OutOfBoundsException
    ├── OutOfRangeException
    ├── OverflowException
    ├── RangeException
    ├── RuntimeException
    ├── UnderflowException
    ├── UnexpectedValueException
    └── ...
```

Commonly Used Exceptions

```php
<?php
// InvalidArgumentException
function divide($a, $b) {
    if (!is_numeric($a) || !is_numeric($b)) {
        throw new InvalidArgumentException("Arguments must be numeric");
    }

    if ($b == 0) {
        throw new DivisionByZeroError("Cannot divide by zero");
    }

    return $a / $b;
}

// RuntimeException
function readFile($filename) {
    if (!file_exists($filename)) {
        throw new RuntimeException("File not found: {$filename}");
    }

    $content = file_get_contents($filename);
    if ($content === false) {
        throw new RuntimeException("Failed to read file: {$filename}");
    }

    return $content;
}

// OutOfBoundsException
class ArrayList {
    private $items = [];

    public function get($index) {
        if ($index < 0 || $index >= count($this->items)) {
            throw new OutOfBoundsException("Index {$index} is out of
bounds");
        }

        return $this->items[$index];
```

```
    }
}
```

Custom Exception Classes

Basic Custom Exception

```php
<?php
class CustomException extends Exception {
    public function __construct($message = "", $code = 0, Throwable
$previous = null) {
        parent::__construct($message, $code, $previous);
    }

    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
    }
}
```

Domain-Specific Exceptions

```php
<?php
// Database Exceptions
class DatabaseException extends Exception {
    protected $query;

    public function __construct($message, $query = null, $code = 0,
Throwable $previous = null) {
        parent::__construct($message, $code, $previous);
        $this->query = $query;
    }

    public function getQuery() {
        return $this->query;
    }
}

class DatabaseConnectionException extends DatabaseException {}
class QueryExecutionException extends DatabaseException {}

// Validation Exceptions
class ValidationException extends Exception {
    protected $errors = [];

    public function __construct($message, array $errors = [], $code = 0,
Throwable $previous = null) {
        parent::__construct($message, $code, $previous);
        $this->errors = $errors;
    }
```

```
    public function getErrors() {
        return $this->errors;
    }

    public function hasErrors() {
        return !empty($this->errors);
    }
}

// Business Logic Exceptions
class InsufficientFundsException extends Exception {
    private $balance;
    private $requestedAmount;

    public function __construct($balance, $requestedAmount) {
        $this->balance = $balance;
        $this->requestedAmount = $requestedAmount;

        $message = "Insufficient funds. Balance: {$balance}, Requested:
{$requestedAmount}";
        parent::__construct($message);
    }

    public function getBalance() { return $this->balance; }
    public function getRequestedAmount() { return $this->requestedAmount;
}
}

class UserNotFoundException extends Exception {}
class EmailAlreadyExistsException extends Exception {}
```

Exception with Context

```
<?php
class ContextualException extends Exception {
    protected $context = [];

    public function __construct($message, array $context = [], $code = 0,
Throwable $previous = null) {
        parent::__construct($message, $code, $previous);
        $this->context = $context;
    }

    public function getContext() {
        return $this->context;
    }

    public function addContext($key, $value) {
        $this->context[$key] = $value;
    }
```

```
    public function getFullMessage() {
        $message = $this->getMessage();

        if (!empty($this->context)) {
            $message .= "\nContext: " . json_encode($this->context,
JSON_PRETTY_PRINT);
        }

        return $message;
    }
}
```

Exception Handling Patterns

Repository Pattern dengan Exception

```
<?php
interface UserRepositoryInterface {
    public function findById($id);
    public function findByEmail($email);
    public function create(array $data);
    public function update($id, array $data);
    public function delete($id);
}

class DatabaseUserRepository implements UserRepositoryInterface {
    private $connection;

    public function __construct($connection) {
        $this->connection = $connection;
    }

    public function findById($id) {
        try {
            if (!is_numeric($id) || $id <= 0) {
                throw new InvalidArgumentException("User ID must be a
positive number");
            }

            $stmt = $this->connection->prepare("SELECT * FROM users WHERE
id = ?");
            $result = $stmt->execute([$id]);

            if (!$result) {
                throw new QueryExecutionException("Failed to execute user
query",
                    "SELECT * FROM users WHERE id = {$id}");
            }

            $user = $stmt->fetch(PDO::FETCH_ASSOC);
```

```
            if (!$user) {
                throw new UserNotFoundException("User with ID {$id} not
found");
            }

            return $user;

        } catch (PDOException $e) {
            throw new DatabaseException("Database error: " . $e-
>getMessage(), null, 0, $e);
        }
    }

    public function findByEmail($email) {
        try {
            if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
                throw new InvalidArgumentException("Invalid email format:
{$email}");
            }

            $stmt = $this->connection->prepare("SELECT * FROM users WHERE
email = ?");
            $stmt->execute([$email]);

            $user = $stmt->fetch(PDO::FETCH_ASSOC);

            if (!$user) {
                throw new UserNotFoundException("User with email {$email}
not found");
            }

            return $user;

        } catch (PDOException $e) {
            throw new DatabaseException("Database error: " . $e-
>getMessage(), null, 0, $e);
        }
    }

    public function create(array $data) {
        try {
            $this->validateUserData($data);

            // Check if email already exists
            try {
                $this->findByEmail($data['email']);
                throw new EmailAlreadyExistsException("Email
{$data['email']} already exists");
            } catch (UserNotFoundException $e) {
                // Email doesn't exist, which is what we want
            }

            $stmt = $this->connection->prepare(
```

```
                "INSERT INTO users (name, email, password, created_at)
VALUES (?, ?, ?, NOW())"
            );

            $result = $stmt->execute([
                $data['name'],
                $data['email'],
                password_hash($data['password'], PASSWORD_DEFAULT)
            ]);

            if (!$result) {
                throw new QueryExecutionException("Failed to create
user");
            }

            return $this->connection->lastInsertId();

        } catch (PDOException $e) {
            throw new DatabaseException("Database error: " . $e-
>getMessage(), null, 0, $e);
        }
    }

    private function validateUserData(array $data) {
        $errors = [];

        if (empty($data['name']) || strlen(trim($data['name'])) < 2) {
            $errors['name'] = 'Name must be at least 2 characters long';
        }

        if (empty($data['email']) || !filter_var($data['email'],
FILTER_VALIDATE_EMAIL)) {
            $errors['email'] = 'Valid email is required';
        }

        if (empty($data['password']) || strlen($data['password']) < 6) {
            $errors['password'] = 'Password must be at least 6 characters
long';
        }

        if (!empty($errors)) {
            throw new ValidationException("User validation failed",
$errors);
        }
    }
}
```

Service Layer dengan Error Handling

```
<?php
class UserService {
```

```
    private $userRepository;
    private $logger;

    public function __construct(UserRepositoryInterface $userRepository,
LoggerInterface $logger) {
        $this->userRepository = $userRepository;
        $this->logger = $logger;
    }

    public function registerUser(array $userData) {
        try {
            $this->logger->info("Attempting to register user", ['email' =>
$userData['email'] ?? 'unknown']);

            $userId = $this->userRepository->create($userData);

            $this->logger->info("User registered successfully", ['user_id'
=> $userId]);

            return [
                'success' => true,
                'user_id' => $userId,
                'message' => 'User registered successfully'
            ];

        } catch (ValidationException $e) {
            $this->logger->warning("User registration validation failed",
[
                'errors' => $e->getErrors(),
                'email' => $userData['email'] ?? 'unknown'
            ]);

            return [
                'success' => false,
                'errors' => $e->getErrors(),
                'message' => 'Validation failed'
            ];

        } catch (EmailAlreadyExistsException $e) {
            $this->logger->warning("Registration attempt with existing
email", [
                'email' => $userData['email'] ?? 'unknown'
            ]);

            return [
                'success' => false,
                'message' => 'Email already exists'
            ];

        } catch (DatabaseException $e) {
            $this->logger->error("Database error during user
registration", [
                'error' => $e->getMessage(),
                'email' => $userData['email'] ?? 'unknown'
```

```
            ]);

            return [
                'success' => false,
                'message' => 'Registration failed due to technical error'
            ];

        } catch (Exception $e) {
            $this->logger->critical("Unexpected error during user
registration", [
                'error' => $e->getMessage(),
                'trace' => $e->getTraceAsString(),
                'email' => $userData['email'] ?? 'unknown'
            ]);

            return [
                'success' => false,
                'message' => 'An unexpected error occurred'
            ];
        }
    }

    public function getUserProfile($userId) {
        try {
            $user = $this->userRepository->findById($userId);

            // Don't return sensitive data
            unset($user['password']);

            return [
                'success' => true,
                'data' => $user
            ];

        } catch (UserNotFoundException $e) {
            $this->logger->info("Profile request for non-existent user",
['user_id' => $userId]);

            return [
                'success' => false,
                'message' => 'User not found'
            ];

        } catch (Exception $e) {
            $this->logger->error("Error retrieving user profile", [
                'user_id' => $userId,
                'error' => $e->getMessage()
            ]);

            return [
                'success' => false,
                'message' => 'Failed to retrieve user profile'
            ];
        }
```

```
    }
}
```

Error Logging dan Monitoring

Simple Logger Class

```php
<?php
interface LoggerInterface {
    public function emergency($message, array $context = []);
    public function alert($message, array $context = []);
    public function critical($message, array $context = []);
    public function error($message, array $context = []);
    public function warning($message, array $context = []);
    public function notice($message, array $context = []);
    public function info($message, array $context = []);
    public function debug($message, array $context = []);
}

class FileLogger implements LoggerInterface {
    private $logPath;
    private $dateFormat;

    public function __construct($logPath = 'logs/app.log', $dateFormat =
'Y-m-d H:i:s') {
        $this->logPath = $logPath;
        $this->dateFormat = $dateFormat;

        // Create log directory if it doesn't exist
        $logDir = dirname($logPath);
        if (!is_dir($logDir)) {
            mkdir($logDir, 0755, true);
        }
    }

    public function emergency($message, array $context = []) {
        $this->log('EMERGENCY', $message, $context);
    }

    public function alert($message, array $context = []) {
        $this->log('ALERT', $message, $context);
    }

    public function critical($message, array $context = []) {
        $this->log('CRITICAL', $message, $context);
    }

    public function error($message, array $context = []) {
        $this->log('ERROR', $message, $context);
    }
```

```
    public function warning($message, array $context = []) {
        $this->log('WARNING', $message, $context);
    }

    public function notice($message, array $context = []) {
        $this->log('NOTICE', $message, $context);
    }

    public function info($message, array $context = []) {
        $this->log('INFO', $message, $context);
    }

    public function debug($message, array $context = []) {
        $this->log('DEBUG', $message, $context);
    }

    private function log($level, $message, array $context = []) {
        $timestamp = date($this->dateFormat);
        $contextStr = !empty($context) ? ' ' . json_encode($context) : '';

        $logEntry = "[{$timestamp}] {$level}: {$message}{$contextStr}" .
PHP_EOL;

        file_put_contents($this->logPath, $logEntry, FILE_APPEND |
LOCK_EX);
    }
}
```

Exception Handler dan Error Reporting

```
<?php
class GlobalExceptionHandler {
    private $logger;
    private $isProduction;

    public function __construct(LoggerInterface $logger, $isProduction =
false) {
        $this->logger = $logger;
        $this->isProduction = $isProduction;

        // Register handlers
        set_exception_handler([$this, 'handleException']);
        set_error_handler([$this, 'handleError']);
        register_shutdown_function([$this, 'handleShutdown']);
    }

    public function handleException(Throwable $exception) {
        $this->logException($exception);

        if ($this->isProduction) {
            $this->displayProductionError();
```

```
        } else {
            $this->displayDevelopmentError($exception);
        }
    }

    public function handleError($severity, $message, $file, $line) {
        if (!(error_reporting() & $severity)) {
            return false;
        }

        $errorTypes = [
            E_ERROR => 'ERROR',
            E_WARNING => 'WARNING',
            E_PARSE => 'PARSE',
            E_NOTICE => 'NOTICE',
            E_CORE_ERROR => 'CORE_ERROR',
            E_CORE_WARNING => 'CORE_WARNING',
            E_COMPILE_ERROR => 'COMPILE_ERROR',
            E_COMPILE_WARNING => 'COMPILE_WARNING',
            E_USER_ERROR => 'USER_ERROR',
            E_USER_WARNING => 'USER_WARNING',
            E_USER_NOTICE => 'USER_NOTICE',
            E_STRICT => 'STRICT',
            E_RECOVERABLE_ERROR => 'RECOVERABLE_ERROR',
            E_DEPRECATED => 'DEPRECATED',
            E_USER_DEPRECATED => 'USER_DEPRECATED',
        ];

        $errorType = $errorTypes[$severity] ?? 'UNKNOWN';

        $this->logger->error("PHP {$errorType}: {$message}", [
            'file' => $file,
            'line' => $line,
            'severity' => $severity
        ]);

        return true;
    }

    public function handleShutdown() {
        $error = error_get_last();

        if ($error && in_array($error['type'], [E_ERROR, E_PARSE,
E_CORE_ERROR, E_COMPILE_ERROR])) {
            $this->logger->critical("Fatal error occurred", $error);

            if ($this->isProduction) {
                $this->displayProductionError();
            }
        }
    }

    private function logException(Throwable $exception) {
        $context = [
```

```
            'class' => get_class($exception),
            'message' => $exception->getMessage(),
            'file' => $exception->getFile(),
            'line' => $exception->getLine(),
            'trace' => $exception->getTraceAsString()
        ];

        if ($exception instanceof ContextualException) {
            $context['exception_context'] = $exception->getContext();
        }

        $this->logger->error("Uncaught exception", $context);
    }

    private function displayProductionError() {
        http_response_code(500);
        echo json_encode([
            'error' => true,
            'message' => 'An internal server error occurred. Please try
again later.'
        ]);
    }

    private function displayDevelopmentError(Throwable $exception) {
        http_response_code(500);

        echo "<h1>Exception: " . get_class($exception) . "</h1>";
        echo "<p><strong>Message:</strong> " .
htmlspecialchars($exception->getMessage()) . "</p>";
        echo "<p><strong>File:</strong> " . $exception->getFile() . "
</p>";
        echo "<p><strong>Line:</strong> " . $exception->getLine() . "
</p>";

        if ($exception instanceof ContextualException) {
            echo "<h2>Context:</h2>";
            echo "<pre>" . htmlspecialchars(json_encode($exception-
>getContext(), JSON_PRETTY_PRINT)) . "</pre>";
        }

        echo "<h2>Stack Trace:</h2>";
        echo "<pre>" . htmlspecialchars($exception->getTraceAsString()) .
"</pre>";
    }
}
```

Defensive Programming

Input Validation dan Sanitization

```php
<?php
class InputValidator {
    public static function validateEmail($email) {
        if (empty($email)) {
            throw new InvalidArgumentException("Email is required");
        }

        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException("Invalid email format:
{$email}");
        }

        return filter_var($email, FILTER_SANITIZE_EMAIL);
    }

    public static function validatePassword($password) {
        if (empty($password)) {
            throw new InvalidArgumentException("Password is required");
        }

        if (strlen($password) < 8) {
            throw new InvalidArgumentException("Password must be at least
8 characters long");
        }

        if (!preg_match('/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/', $password)) {
            throw new InvalidArgumentException("Password must contain at
least one lowercase letter, one uppercase letter, and one digit");
        }

        return $password;
    }

    public static function validateInteger($value, $min = null, $max =
null) {
        if (!is_numeric($value)) {
            throw new InvalidArgumentException("Value must be numeric:
{$value}");
        }

        $intValue = (int) $value;

        if ($min !== null && $intValue < $min) {
            throw new OutOfRangeException("Value {$intValue} is below
minimum {$min}");
        }

        if ($max !== null && $intValue > $max) {
            throw new OutOfRangeException("Value {$intValue} is above
maximum {$max}");
        }

        return $intValue;
```

```
    }

    public static function validateArray($data, array $requiredKeys = [])
{
        if (!is_array($data)) {
            throw new InvalidArgumentException("Data must be an array");
        }

        foreach ($requiredKeys as $key) {
            if (!array_key_exists($key, $data)) {
                throw new InvalidArgumentException("Required key '{$key}'
is missing");
            }
        }

        return $data;
    }
}
```

Circuit Breaker Pattern

```
<?php
class CircuitBreaker {
    private $failureThreshold;
    private $recoveryTimeout;
    private $failureCount = 0;
    private $lastFailureTime = null;
    private $state = 'CLOSED'; // CLOSED, OPEN, HALF_OPEN

    public function __construct($failureThreshold = 5, $recoveryTimeout =
60) {
        $this->failureThreshold = $failureThreshold;
        $this->recoveryTimeout = $recoveryTimeout;
    }

    public function call(callable $operation) {
        if ($this->state === 'OPEN') {
            if (time() - $this->lastFailureTime >= $this->recoveryTimeout)
{
                $this->state = 'HALF_OPEN';
            } else {
                throw new RuntimeException("Circuit breaker is OPEN");
            }
        }

        try {
            $result = $operation();

            if ($this->state === 'HALF_OPEN') {
                $this->reset();
            }
```

```
            return $result;

        } catch (Exception $e) {
            $this->recordFailure();
            throw $e;
        }
    }

    private function recordFailure() {
        $this->failureCount++;
        $this->lastFailureTime = time();

        if ($this->failureCount >= $this->failureThreshold) {
            $this->state = 'OPEN';
        }
    }

    private function reset() {
        $this->failureCount = 0;
        $this->lastFailureTime = null;
        $this->state = 'CLOSED';
    }

    public function getState() {
        return $this->state;
    }
}
```

Testing Exception Handling

PHPUnit Testing

```
<?php
use PHPUnit\Framework\TestCase;

class UserServiceTest extends TestCase {
    private $userRepository;
    private $logger;
    private $userService;

    protected function setUp(): void {
        $this->userRepository = $this-
>createMock(UserRepositoryInterface::class);
        $this->logger = $this->createMock(LoggerInterface::class);
        $this->userService = new UserService($this->userRepository, $this-
>logger);
    }

    public function testRegisterUserWithValidData() {
        $userData = [
```

```
            'name' => 'John Doe',
            'email' => 'john@example.com',
            'password' => 'SecurePass123'
        ];

        $this->userRepository
            ->expects($this->once())
            ->method('create')
            ->with($userData)
            ->willReturn(123);

        $result = $this->userService->registerUser($userData);

        $this->assertTrue($result['success']);
        $this->assertEquals(123, $result['user_id']);
    }

    public function testRegisterUserWithInvalidData() {
        $userData = [
            'name' => 'J',
            'email' => 'invalid-email',
            'password' => '123'
        ];

        $errors = [
            'name' => 'Name must be at least 2 characters long',
            'email' => 'Valid email is required',
            'password' => 'Password must be at least 6 characters long'
        ];

        $this->userRepository
            ->expects($this->once())
            ->method('create')
            ->with($userData)
            ->willThrowException(new ValidationException("Validation
failed", $errors));

        $result = $this->userService->registerUser($userData);

        $this->assertFalse($result['success']);
        $this->assertEquals($errors, $result['errors']);
    }

    public function testRegisterUserWithDuplicateEmail() {
        $userData = [
            'name' => 'John Doe',
            'email' => 'existing@example.com',
            'password' => 'SecurePass123'
        ];

        $this->userRepository
            ->expects($this->once())
            ->method('create')
            ->with($userData)
```

```
            ->willThrowException(new EmailAlreadyExistsException("Email
already exists"));

        $result = $this->userService->registerUser($userData);

        $this->assertFalse($result['success']);
        $this->assertEquals('Email already exists', $result['message']);
    }

    public function testGetUserProfileNotFound() {
        $this->userRepository
            ->expects($this->once())
            ->method('findById')
            ->with(999)
            ->willThrowException(new UserNotFoundException("User not
found"));

        $result = $this->userService->getUserProfile(999);

        $this->assertFalse($result['success']);
        $this->assertEquals('User not found', $result['message']);
    }
}
```

Best Practices

1. Exception Naming dan Organization

```
<?php
// Domain-specific exception namespace
namespace App\Exceptions\User;

class UserException extends \Exception {}
class UserNotFoundException extends UserException {}
class InvalidUserDataException extends UserException {}
class UserPermissionException extends UserException {}

namespace App\Exceptions\Database;

class DatabaseException extends \Exception {}
class ConnectionException extends DatabaseException {}
class QueryException extends DatabaseException {}
class TransactionException extends DatabaseException {}
```

2. Exception Message Guidelines

```
<?php
// ❌ Bad - Vague message
throw new Exception("Error occurred");
```

```
// ✅ Good - Specific, actionable message
throw new InvalidArgumentException("User ID must be a positive integer,
got: {$userId}");

// ✅ Good - Include context
throw new RuntimeException("Failed to connect to database server {$host}:
{$port} - Connection timeout after {$timeout}s");
```

3. Error Response Standardization

```
<?php
class ApiResponse {
    public static function success($data = null, $message = 'Success') {
        return [
            'success' => true,
            'message' => $message,
            'data' => $data,
            'timestamp' => date('c')
        ];
    }

    public static function error($message, $errors = null, $code = 500) {
        $response = [
            'success' => false,
            'message' => $message,
            'timestamp' => date('c')
        ];

        if ($errors) {
            $response['errors'] = $errors;
        }

        http_response_code($code);
        return $response;
    }

    public static function validation($errors, $message = 'Validation
failed') {
        return self::error($message, $errors, 422);
    }

    public static function notFound($message = 'Resource not found') {
        return self::error($message, null, 404);
    }

    public static function unauthorized($message = 'Unauthorized') {
        return self::error($message, null, 401);
    }

    public static function forbidden($message = 'Forbidden') {
```

```
            return self::error($message, null, 403);
        }
    }
```

Contoh Implementasi

Lihat file `example.php` untuk berbagai contoh implementasi error handling dan exception dalam PHP OOP.

Latihan

1. Buat custom exception hierarchy untuk aplikasi e-commerce
2. Implementasikan try-catch-finally untuk file operations
3. Buat error handler global dengan logging
4. Implementasikan circuit breaker untuk external API calls

Tugas Rumah

Buat system error handling yang komprehensif untuk:

- User management dengan validation exceptions
- Database operations dengan connection handling
- File upload dengan size dan type validation
- API integration dengan retry mechanism dan circuit breaker
- Comprehensive logging dan monitoring system