NodeJS – ERROR Handling

## Exceptions

Exceptions/apiError.ts

export class ApiError extends Error {

message: string;

errorCode: ErrorCode;

statusCode: number;

errors: any

constructor(message: string, errorCode: any, statusCode: number, errors?: any) {

super(message)

this.message = message;

this.errorCode = errorCode;

this.statusCode = statusCode;

this.errors = errors

Error.captureStackTrace(this, this.constructor);

}

}

Exceptions/errorCode.ts

export enum ErrorCode {

// 4xxx series: Client-Side Errors

// Group 4000-4099: Authentication & Authorization

UNAUTHORIZED = 4001,

FORBIDDEN = 4002,

INCORRECT\_PASSWORD = 4003,

// Group 4100-4199: Request & Validation Issues

BAD\_REQUEST = 4100, // Malformed syntax

//VALIDATION\_FAILED = 4101,

UNPROCESSABLE\_ENTITY = 4102,

// Group 4200-4299: Resource & State Issues

NOT\_FOUND = 4200,

USER\_NOT\_FOUND = 4201,

CONFLICT = 4202,

USER\_ALREADY\_EXISTS = 4203,

// 5xxx series: Server-Side Errors

SERVER\_ERROR = 5001,

}

Exceptions/errors.ts

import { ApiError } from "./apiError";

import { ErrorCode } from "./errorCode";

/\*\*

\* 400 Bad Request: General client-side error for malformed request syntax,

\* invalid request message framing, or deceptive request routing.

\*/

export class BadRequestError extends ApiError {

constructor(message: string, errorCode: ErrorCode = ErrorCode.BAD\_REQUEST) {

super(message, errorCode, 400);

}

}

/\*\*

\* 401 Unauthorized: Authentication is required and has either failed or not yet been provided.

\*/

export class UnauthorizedError extends ApiError {

constructor(message: string, errorCode: ErrorCode = ErrorCode.UNAUTHORIZED) {

super(message, errorCode, 401);

}

}

/\*\*

\* 403 Forbidden: The server understood the request but refuses to authorize it.

\* Authentication will not help, and the request should not be repeated.

\*/

export class ForbiddenError extends ApiError {

constructor(message: string, errorCode: ErrorCode = ErrorCode.FORBIDDEN) {

super(message, errorCode, 403);

}

}

/\*\*

\* 404 Not Found: The server cannot find the requested resource.

\*/

export class NotFoundError extends ApiError {

constructor(message: string, errorCode: ErrorCode = ErrorCode.NOT\_FOUND) {

super(message, errorCode, 404);

}

}

/\*\*

\* 409 Conflict: The request could not be completed due to a conflict with the

\* current state of the resource.

\*/

export class ConflictError extends ApiError {

constructor(message: string, errorCode: ErrorCode = ErrorCode.CONFLICT) {

super(message, errorCode, 409);

}

}

/\*\*

\* 422 Unprocessable Entity: The server understands the request but cannot

\* process the contained instructions due to invalid data (e.g., failed validation).

\*/

export class UnprocessableEntityError extends ApiError {

constructor(errors: any, message: string = "Validation Failed", errorCode: ErrorCode = ErrorCode.UNPROCESSABLE\_ENTITY) {

super(message, errorCode, 422, errors);

}

}

## Middleware

Middleware/errors.ts

import { NextFunction, Request, Response } from "express";

import { ApiError } from "../exceptions/apiError";

import { ErrorCode } from "../exceptions/errorCode";

import { UnprocessableEntityError } from "../exceptions/errors";

export const errorMiddleware = (err: unknown, req: Request, res: Response, next: NextFunction) => {

// Check if the error is an instance of our custom ApiError class

if (err instanceof ApiError) {

// Log the operational error for debugging purposes (optional)

console.error('API Error:', err);

const errorResponse: { [key: string]: any } = {

statusCode: err.statusCode,

message: err.message,

errorCode: err.errorCode,

};

if (err instanceof UnprocessableEntityError) {

// Add specific validation details if it's a ValidationError

errorResponse.errors = err.errors;

}

return res.status(err.statusCode).json(errorResponse);

}

// Handle other unexpected (non-operational) programming errors

const isDevelopment = process.env.NODE\_ENV !== 'production';

const isErrorInstance = err instanceof Error;

// Log the full unexpected error, including stack, in all environments

if (isErrorInstance) {

console.error('UNEXPECTED SERVER ERROR:', err.stack);

} else {

console.error('UNEXPECTED SERVER ERROR:', err);

}

// Send a generic 500 response

const statusCode = 500;

const message = isDevelopment && isErrorInstance ? err.message : 'An unexpected server error occurred';

const errorResponse: { [key: string]: any } = {

statusCode: statusCode,

message: message,

errorCode: ErrorCode.SERVER\_ERROR,

};

if (isDevelopment && isErrorInstance) {

errorResponse.stack = err.stack;

}

return res.status(statusCode).json(errorResponse);

};

### Zod Validation middleware

Middleware/validate.ts

import z, { ZodType } from "zod";

import { Request, Response, NextFunction } from "express";

import { UnprocessableEntityError } from "../exceptions/errors";

export const validateData = <T extends ZodType>(schema: T) => {

return (req: Request<{}, {}, z.infer<T>>, res: Response, next: NextFunction) => {

const result = schema.safeParse(req.body);

if (!result.success) {

const formattedErrors = result.error.issues.map(issue => ({

field: issue.path.join(".") || "body",

message: issue.message

}));

throw new UnprocessableEntityError(formattedErrors)

}

req.body = result.data;

next();

};

};

ZOD Schema

import { z } from "zod";

export const CreateUserSchema = z.object({

name: z.string().min(1, "Name is required"),

email: z.email("Invalid email").min(1, "Email is required"),

password: z.string("Password is required").min(6, "Password must be at least 6 characters")

});

export type CreateUserInput = z.infer<typeof CreateUserSchema>;

// Using PICK/OMIT

export const LoginSchema = CreateUserSchema.pick({

email: true,

password: true

});

/\* // Using OMIT

export const LoginSchema = CreateUserSchema.omit({

name: true

});

\*/

export type LoginInput = z.infer<typeof LoginSchema>;

Auth Routes

import { Router } from "express";

import { loginHandler, signUpHandler } from "../controllers/auth";

import { validateData } from "../middlewares/validate";

import { CreateUserSchema, LoginSchema } from "../schema/auth.schema";

const authRoutes = Router()

authRoutes.post('/signup', validateData(CreateUserSchema), signUpHandler)

authRoutes.post("/login", validateData(LoginSchema), loginHandler);

export default authRoutes

### Auth Middleware (Optional)

Middleware/auth.ts

import { Request, Response, NextFunction } from 'express';

import jwt from 'jsonwebtoken';

import { UnauthorizedError, ForbiddenError } from '../exceptions/errors'; // Import your custom errors

import { JwtPayload } from '../types/express';

// Read the JWT secret from your environment variables

const jwtSecret = process.env.JWT\_SECRET;

if (!jwtSecret) {

throw new Error('JWT\_SECRET must be defined in your environment variables.');

}

export const authMiddleware = (req: Request, res: Response, next: NextFunction) => {

// 1. Get the 'Authorization' header from the request

const authHeader = req.headers.authorization;

// 2. Check if the header and token exist

if (!authHeader || !authHeader.startsWith('Bearer ')) {

throw new UnauthorizedError('No token provided.');

}

// 3. Extract the token from the header

const token = authHeader.split(' ')[1];

// 4. Verify the token

try {

const decoded = jwt.verify(token, jwtSecret) as JwtPayload;

// 5. Attach the decoded user information to the request object

req.user = decoded;

next();

} catch (err) {

// Handle specific JWT errors

if (err instanceof jwt.JsonWebTokenError) {

throw new ForbiddenError('Invalid token.');

}

if (err instanceof jwt.TokenExpiredError) {

throw new UnauthorizedError('Token expired.');

}

// Handle any other verification errors

throw new UnauthorizedError('Failed to authenticate token.');

}

};

Testing Endpoints

// A public route that does not require authentication

app.get('/api/public', (req, res) => {

res.json({ message: 'This is a public route.' });

});

// A protected route that requires a valid JWT

app.get('/api/protected', authMiddleware, (req, res) => {

// req.user is available here and is correctly typed

res.json({ message: `Hello, user ${req.user?.userId}! This is protected data.` });

});

## Main APP

**In the Main express app starting code**

Src/index.ts

const app = express();

process.on('uncaughtException', (err) => {

console.log(`ERROR: ${err}`);

console.log('Shutting down server due to uncaught exception');

process.exit(1)

})

app.use(express.json())

**Invalid URL, Error Middleware and Unhandled Rejection**

// --- Place the 404 handler middleware AFTER all other routes ---

app.use((req, res, next) => {

throw new NotFoundError('Invalid URL', ErrorCode.NOT\_FOUND)

});

// Error handling middleware (optional)

app.use(errorMiddleware);

const server = app.listen(appConfig.PORT, () => {

console.log(`Server is running on port ${appConfig.PORT}`);

})

// Handle Unhandled Promise rejections

process.on('unhandledRejection', (err) => {

console.log(`ERROR: ${err}`);

console.log('Shutting down server due to unhandled promise rejection');

server.close(() => {

process.exit(1)

})

})

## Controllers

import { Request, RequestHandler, Response } from "express";

import { compareSync, hashSync } from "bcrypt";

import prisma from "../prisma";

import { CreateUserInput, LoginInput } from "../schema/auth.schema";

import { signToken } from "../utils/jwt";

import { ConflictError, UnauthorizedError } from "../exceptions/errors";

import { ErrorCode } from "../exceptions/errorCode";

export const loginHandler = async (

req: Request<{}, {}, LoginInput>,

res: Response

) => {

const { email, password } = req.body;

// Find user by email

const user = await prisma.user.findUnique({

where: { email }

});

if (!user) {

throw new UnauthorizedError("Invalid email or password")

//return res.status(401).json({ message: "Invalid email or password" });

}

// 2️ Compare passwords

const passwordMatches = compareSync(password, user.password);

if (!passwordMatches) {

throw new UnauthorizedError("Invalid email or password")

//return res.status(401).json({ message: "Invalid email or password" });

}

// 3️⃣ Generate JWT token

const token = signToken({ userId: user.id, email: user.email });

// 4️⃣ Return token + user info (omit password)

const { password: \_pwd, createdAt, updatedAt, ...userWithoutSensitive } = user;

return res.status(200).json({

token,

user: userWithoutSensitive,

});

};

export const signUpHandler = async (

req: Request<{}, {}, CreateUserInput>,

res: Response

) => {

const { email, name, password } = req.body

// Check if email already exists

const existing = await prisma.user.findUnique({ where: { email } });

if (existing) {

//return res.status(409).json({ message: "Email already in use" });

throw new ConflictError("Email already in use", ErrorCode.USER\_ALREADY\_EXISTS)

}

// Create new user

const user = await prisma.user.create({

data: { email, name, password: hashSync(password, 10) }

});

res.status(201).json(user);

};