Purwadhika
Digital Technology School

AI Fullstack Software Development

Performance and Reliability: Caching, Errors, and Debugging

Job Connector Program

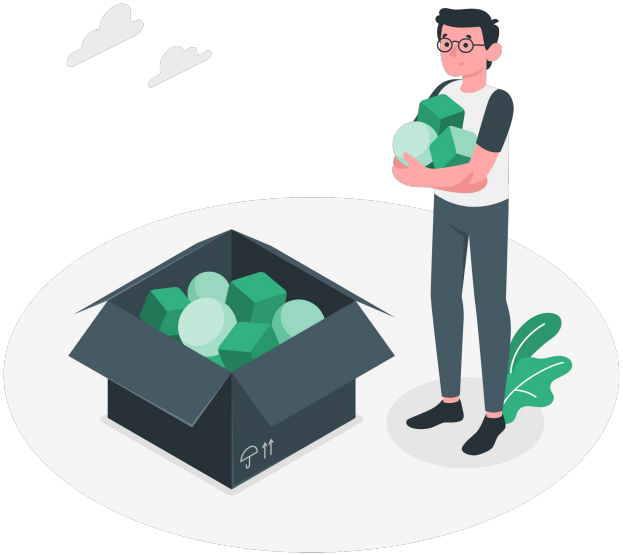
Outline

Purwadhika
Digital Technology School

- Caching with Redis
- Error Handler
- Debugging

What is caching ?

Caching is the process of storing copies of files in a cache or a temporary storage location so that they can be accessed more quickly.



Why do we cache ?

- To save cost. Such as paying for bandwidth or even volume of data sent over the network.
- To reduce app response time.

What is Redis?

- Redis (short for Remote Dictionary Server) is an open-source, in-memory data structure store that acts as a **database**, **message broker**, and **caching**.
- Unlike traditional databases that store data on disk, Redis keeps data in memory, which makes it lightning-fast for both read and write operations.
- It supports various data types such as strings, hashes, lists, sets, and sorted sets, allowing developers to handle a wide range of data use cases.

Why Use Redis for Caching?

Caching means temporarily storing frequently accessed data so that your application can retrieve it faster. Redis is one of the best caching tools because:

- It provides sub-millisecond data access.
- It reduces database load significantly.
- It supports auto-expiration using TTL (Time To Live).
- It helps handle high traffic with better scalability.

Example: *You can cache user profiles, session tokens, or API responses to make your app load instantly.*

Redis Installation

To install and start Redis on your system, follow the installation steps described in the documentation below carefully and in order:

Redis Installation Guidance

Starting Redis Server

- Once installed (via WSL, Linux, or macOS), you can start the Redis server manually:

  redis-server

- Or, for background service:

  sudo service redis-server start

- To check if it's active:

  redis-cli
  127.0.0.1:6379> ping
  # PONG

Using Redis in Express.js

- Install the required packages:

  npm install express ioredis

- Setup redis instance:

```
import Redis from "ioredis";
export const redis = new Redis(); // Default port: 6379
```

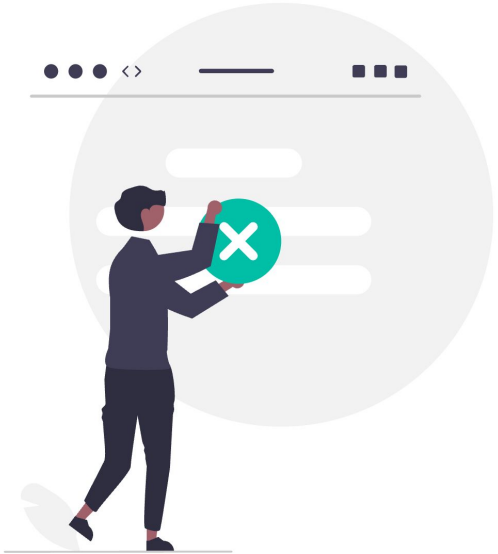
Example : Cache API Response

This example caches the response for 10 seconds, so repeated requests are instantly served from Redis.

```ts
import express, { Request, Response } from "express";
import { redis } from "./cache/redisClient";

const app = express();
const PORT = 3000;

async function getDataFromDB() {
  console.log("Fetching from database...");
  return { message: "Hello from DB", time: new Date().toISOString() };
}

app.get("/data", async (req: Request, res: Response) => {
  const cacheKey = "myData";

  const cached = await redis.get(cacheKey);
  if (cached) {
    return res.json({ source: "cache", data: JSON.parse(cached) });
  }

  const data = await getDataFromDB();
  await redis.set(cacheKey, JSON.stringify(data), "EX", 10); // 10 seconds cache

  return res.json({ source: "database", data });
});

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Error Handling

In modern web applications, robust error handling is essential not only for ensuring a smooth user experience but also for helping developers trace and fix issues quickly. This session explores best practices for handling errors in Express.js using TypeScript. We'll cover structured error classes, async error wrappers, global error middleware, and Prisma-specific error handling. The strategies presented are applicable to both function-based and class-based architectures, and are designed to make debugging easier and responses more meaningful and consistent.



Principles of Error Handling

A well-structured error handling system is a critical part of any scalable application. These principles serve as a foundation for designing a consistent and maintainable error management approach that works across different technologies and architectures.

- **Structured and Traceable Error Handling**
  Design a centralized system that clearly defines error types and stack traces to make debugging faster and easier.
- **Centralized Management of Errors from Dependencies**
  Handle errors from third-party tools (e.g., databases, external APIs, libraries) in one place, instead of scattering try-catch blocks throughout the codebase.
- **Separation of Error Logic from Business Logic**
  Keep error-related logic isolated from controllers and services to maintain clean and testable code.
- **Support for Both Functional and Object-Oriented Design**
  Apply error handling consistently, whether you're using function-based or class-based coding styles, to ensure flexibility and code reuse.

Create Error Class

Custom ***AppError*** helps us standardize error handling by allowing each error to carry an HTTP status code and a flag (isOperational) to indicate whether it's expected or not. This makes it easier to log, trace, and respond to errors consistently across the app.

```ts
// src/errors/AppError.ts
export class AppError extends Error {
  public readonly statusCode: number;
  public readonly isOperational: boolean;

  constructor(message: string, statusCode = 500, isOperational = true) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = isOperational;
    Error.captureStackTrace(this, this.constructor);
  }
}
```

Prepare Error Middleware

errorHandler is a centralized Express middleware for handling various types of errors. It logs errors to the console and responds with the appropriate HTTP status:

- Handles PrismaClientKnownRequestError with status 400 and detailed message.
- Handles custom AppError by using the defined statusCode and message.
- Falls back to status 500 for unknown errors with a generic internal error message.

```ts
// src/middleware/errorHandler.ts
import { Request, Response, NextFunction } from 'express';
import { AppError } from '../errors/AppError';
import { Prisma } from '@prisma/client';

export const errorHandler = (
  err: any,
  req: Request,
  res: Response,
  next: NextFunction
) => {
  console.error(`[Error] ${err.name}: ${err.message}`);
  if (err.stack) console.error(err.stack);

  // Prisma Error Handling
  if (err instanceof Prisma.PrismaClientKnownRequestError) {
    return res.status(400).json({ message: err.message, code: err.code });
  }

  // AppError
  if (err instanceof AppError) {
    return res.status(err.statusCode).json({ message: err.message });
  }

  // Unknown Error
  return res.status(500).json({
    message: 'Internal Server Error',
    detail: err.message || 'Something went wrong',
  });
};
```

Implement Error Handling at Controller

```ts
// src/controllers/userController.ts
import { Request, Response, NextFunction } from 'express';
import { userService } from '../services/userService';
import AppError from '../errors/AppError';

export const getUserById = async (req: Request, res: Response, next: NextFunction) => {
    try {
        const { id } = req.params;
        const user = await userService.findById(Number(req.params.id));

        if (!user) {
            throw new AppError('User not found', 404);
        }

        return res.status(200).send({
            message: 'User found',
            user,
        });
    } catch (error) {
        next(error);
    }
};
```

Implement Error Handling at Service

```ts
// src/services/userService.ts
import { prisma } from '../lib/prisma';
import { AppError } from '../errors/AppError';

export const userService = {
    findById: async (id: number) => {
        try {
            return await prisma.user.findUnique({
                where: { id }
            });
        } catch (error) {
            throw new AppError('Failed to retrieve user', 500);
        }
    },

    create: async (data: any) => {
        try {
            return await prisma.user.create({
                data
            });
        } catch (error: any) {
            if (error.code === 'P2002') {
                throw new AppError('Duplicate entry', 409);
            }
            throw error;
        }
    }
};
```

Implement Error Handler

```ts
// src/app.ts
import express from 'express';
import { errorHandler } from './middleware/errorHandler';
import userRoutes from './routes/userRoutes';

const app = express();
app.use(express.json());

app.use('/api/users', userRoutes);

app.use(errorHandler);

export default app;
```

Store Error to Logger

Storing error information in a logger is essential for debugging, monitoring, and auditing purposes.
It helps developers trace the root cause of issues — even after users receive a clean error response — and is critical for identifying patterns, unexpected behavior, or security concerns in production environments.

```
import { createdLogger, format, transports } from "winston";
import path from "path";

const logFormat = format.printf({( timestamp, level, message )}) => {
    return '$(timestamp) [$(level.tolpperCase()]]: $(message)';
});

const logger = createdLogger({
    level: 'info',
    format: format.combine(
        format.timestamp({ format: "YYYY-MM-DD HH:mm:ss" ),
        format.errors({ stack: true ),
        format.splat(),
        format.json(),
        logFormat
    ),
    transports: [
        new transports.File({
            filename: path.join(__dirname, "./../logs/error.log"),
            level: "error",
        });
        new transports.File({
            filename: path.join(__dirname, "./../logs/combined.log"),
        });
    ],
});

if (process.env.NODE_ENV !== "production") {
    logger.add(
        new transports.Console({
            format: format.combine(fermat.colorize(), logFormat),
        });
    );
}

export default logger;
```

Update Error Handler

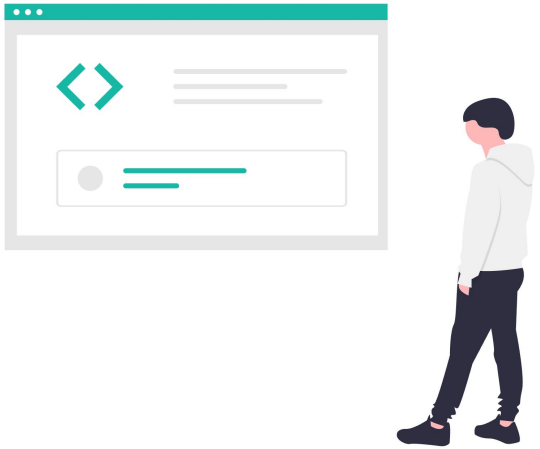
This middleware captures all thrown errors and determines the appropriate HTTP response:

- Logs detailed error info using a custom logger
- Detects and handles PrismaClientKnownRequestError specifically
- Recognizes custom AppError to send client-friendly messages
- Falls back to a generic 500 Internal Server Error for unknown issues
- Centralizing error handling improves code readability, consistency, and debugging.

```ts
// src/middleware/errorHandler.ts
import { Request, Response, NextFunction } from 'express';
import { AppError } from '../errors/AppError';
import { Prisma } from '@prisma/client';
import logger from '../utils/logger';

export const errorHandler = (
    err: any,
    req: Request,
    res: Response,
    next: NextFunction
) => {
    logger.error('[${err.name}] ${err.message}', { stack: err.stack, ... });
    if (err.stack) console.error(err.stack);

    if (err instanceof Prisma.PrismaClientKnowRequestError) {
        return res.status(400.json({ message: err.message, code: err.code });
    }

    if (err instanceof AppError) {
        return res.status(err.statusCode.json({ message: err.message });
    }

    return res.status(500.json({
        message: 'Internal Server Error',
        detail: err.message || 'Something went wrong',
    });
});
```

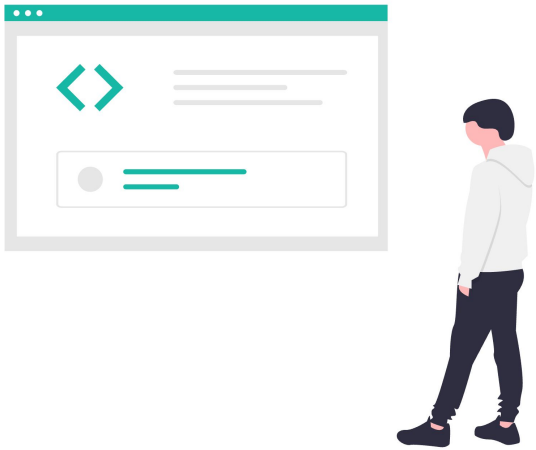
Debugging

After implementing error handling and logging, the next essential step is debugging — the process of identifying, analyzing, and fixing unexpected behavior in your application.
By combining meaningful logs with debugging tools, developers can quickly trace the root cause of issues, reduce downtime, and ensure a smoother development experience.



Why Debugging Matters

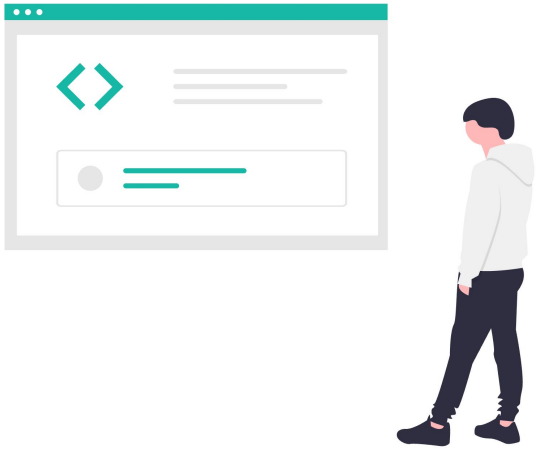
- Even with error handling, bugs will happen.
- Debugging turns cryptic logs into actionable insights.
- Efficient debugging = Faster development + More stable apps.



Log-Driven Debugging Workflow

After implementing global error handling + logging (e.g., with pino), we now use logs to:

- Locate the source of the error
- Understand context around the error
- Reproduce and fix the bug efficiently



How to Read a Logger Message

```
2025-05-12 08:54:45 [ERROR]: [Error] GET /user/getBy/120 : User not found {"statusCode": 404, "isOperational": true}
```

Part	Meaning
2025-05-12 08:54:45	Timestamp when the error occurred
[ERROR]	Log level – indicates this is an error
[Error]	Error type (could be default Error, or custom like AppError)
GET /user/getBy/120	The HTTP method and endpoint that caused the error
User not found	Human-readable error message
{"statusCode":404,"isOperational":true}	Additional metadata about the error

Debugging Based on Error Type

Error Type	Debug Strategy
PrismaClientKnownRequestError	Check query + DB schema mismatch
AppError	Verify validation logic, business rules
500 Internal Server Error	Trace the stack – look for null/undefined bugs

Exercise

In this task, you will improve the **Blog App** you previously built by adding **caching** to **speed up article retrieval**, **showing the performance** difference through **logs**, and implementing **logging** to **monitor application activity** inside a Docker container.

Showing Performance Difference (Logs)

- Log data source:
  - `[DB] Fetching articles from database`
  - `[Cache] Returning articles from cache`

- Log request execution time:
  - `[Performance] GET /articles - 120ms (DB)`
  - `[Performance] GET /articles - 5ms (CACHE)`

- Call the same endpoint at least twice:
  - First call ➜ DB log
  - Second call ➜ Cache log

Thank you