Day 3

1) Express.js Fundamentals

Example 1 — Employee Portal API (Routing + Middleware + Error Handling)

Scenario

A company needs a small Express.js API to manage employees.
They want:

- A logger middleware
- Routes inside `/routes` folder
- Custom error handler
- 3 endpoints: list, add, get employee
- Validation middleware

Project Structure

```
employee-api/
│── app.js
│── routes/
│     └── employees.js
│── middlewares/
│     ├── logger.js
│     └── validateEmployee.js
│── data/
│     └── employees.json
│── package.json
```

Code Implementation

app.js

```
const express = require("express");
const app = express();

const logger = require("./middlewares/logger");
const employeeRoutes = require("./routes/employees");

// Built-in middleware
app.use(express.json());

// Custom middleware
app.use(logger);

// Routes
app.use("/employees", employeeRoutes);

// Global Error Handler
app.use((err, req, res, next) => {
  console.error("ERROR:", err.message);
  res.status(500).json({ status: "error", message:
err.message });
});

app.listen(3000, () => console.log("Server running on port
3000"));
```

middlewares/logger.js

```
module.exports = function (req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next();
};
```

middlewares/validateEmployee.js

```
module.exports = function (req, res, next) {
  const { name, department } = req.body;
  if (!name || !department) {
    return res.status(400).json({ error: "Name & Department
required" });
  }
  next();
};
```

routes/employees.js

```
const express = require("express");
const router = express.Router();
const validateEmployee = require("../middlewares/
validateEmployee");

let employees = [
  { id: 1, name: "John Doe", department: "HR" },
  { id: 2, name: "Mary", department: "Finance" }
];

// GET all
router.get("/", (req, res) => {
  res.json(employees);
});

// GET by ID
router.get("/:id", (req, res) => {
  const emp = employees.find(e => e.id == req.params.id);
  if (!emp) return res.status(404).json({ error: "Employee
not found" });
  res.json(emp);
});

// POST (with validation middleware)
router.post("/", validateEmployee, (req, res, next) => {
  try {
    const newEmp = { id: employees.length + 1, ...req.body };
    employees.push(newEmp);
    res.status(201).json(newEmp);
  } catch (error) {
    next(error);
  }
});

module.exports = router;
```

Sample Output

Console

```
GET /employees
POST /employees
```

API Response — GET /employees

```
[
  { "id": 1, "name": "John Doe", "department": "HR" },
  { "id": 2, "name": "Mary", "department": "Finance" }
]
```

POST /employees (Invalid Body)

```
{ "error": "Name & Department required" }
```

Example 2 — Online Course Management (Routing + Multiple Middlewares)

Scenario

You are building a Course API with:

- Global authentication middleware
- Route-level middleware for admin role
- Error handler for invalid course ID

Project Structure

```
course-api/
│── app.js
│── routes/course.js
│── middlewares/auth.js
│── middlewares/admin.js
│── package.json
```

Code Implementation

middlewares/auth.js

```
module.exports = (req, res, next) => {
  const token = req.headers["authorization"];
  if (!token) return res.status(401).json({ error: 
"Unauthorized" });

  req.user = { id: 101, role: "admin" }; // mocked user

  next();
};
```

middlewares/admin.js

```
module.exports = (req, res, next) => {
  if (req.user.role !== "admin") {
    return res.status(403).json({ error: "Admin access 
required" });
  }
  next();
};
```

routes/course.js

```
const express = require("express");
const router = express.Router();

let courses = [
  { id: 1, name: "Node.js Basics" },
  { id: 2, name: "Advanced JavaScript" }
];

// GET
router.get("/", (req, res) => {
  res.json(courses);
});

// DELETE (Admin Only)
router.delete("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const course = courses.find(c => c.id === id);

  if (!course) return res.status(404).json({ error: "Course 
not found" });

  courses = courses.filter(c => c.id !== id);
  res.json({ message: "Deleted successfully" });
});
```

```
module.exports = router;
```

app.js

```
const express = require("express");
const app = express();
const auth = require("./middlewares/auth");
const admin = require("./middlewares/admin");
const courseRoutes = require("./routes/course");

app.use(express.json());

// Global Auth Middleware
app.use(auth);

// Admin middleware only for DELETE route
app.use("/courses", (req, res, next) => {
  if (req.method === "DELETE") return admin(req, res, next);
  next();
});

app.use("/courses", courseRoutes);

// Error handler
app.use((err, req, res, next) => {
  res.status(500).json({ error: err.message });
});

app.listen(4000, () => console.log("Server running"));
```

Sample Output

DELETE /courses/3

```
{ "error": "Course not found" }
```

DELETE /courses/1 (Admin User)

```
{ "message": "Deleted successfully" }
```

Example 3 — E-commerce Cart API (Error Handling Flow)

Scenario

An e-commerce app needs proper error propagation:

- Failing stock check triggers an error
- Error is passed to global handler
- Request logs middleware

Project Structure

```
cart-api/
│── app.js
│── routes/cart.js
│── middlewares/logger.js
│── package.json
```

Code Implementation

routes/cart.js

```
const express = require("express");
const router = express.Router();

router.post("/add", (req, res, next) => {
  const { qty } = req.body;

  if (qty > 5) {
    const err = new Error("Stock limit exceeded");
    err.status = 400;
    return next(err);
  }

  res.json({ message: "Item added to cart" });
});

module.exports = router;
```

app.js

```
const express = require("express");
const logger = require("./middlewares/logger");
const cartRoutes = require("./routes/cart");

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

app.use("/cart", cartRoutes);

// Global Error Handler
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ error: err.message });
});

app.listen(5000, () => console.log("Running on port 5000"));
```

middlewares/logger.js

```
module.exports = (req, res, next) => {
  console.log("Request:", req.method, req.url);
  next();
};
```

Output

POST /cart/add with { "qty": 10 }

```
{ "error": "Stock limit exceeded" }
```

2) Advanced Routing

(route parameters, query strings, nested routes)

Example 1 — Library System (Route Parameters + Query Filters)

Scenario

A library allows users to:

- Fetch book by **ID**
- Filter books using query strings: `/books?author=John&year=2021`
- Nested route: `/books/:id/reviews`

Project Structure

```
library-api/
│── app.js
│── routes/
│     └── books.js
│── package.json
```

Code Implementation

routes/books.js

```
const express = require("express");
const router = express.Router();

let books = [
  { id: 1, title: "Node Mastery", author: "John", year:
2021 },
  { id: 2, title: "Express Deep Dive", author: "Mary", year:
2020 }
];

// List books with filters
router.get("/", (req, res) => {
  const { author, year } = req.query;

  let result = books;

  if (author) result = result.filter(b => b.author ===
author);
  if (year) result = result.filter(b => b.year == year);
  res.json(result);
});

// Route parameter
router.get("/:id", (req, res) => {
  const book = books.find(b => b.id == req.params.id);
  if (!book) return res.status(404).json({ error: "Book not
found" });
  res.json(book);
});

// Nested route: /books/:id/reviews
router.get("/:id/reviews", (req, res) => {
  res.json({
    bookId: req.params.id,
    reviews: ["Good", "Excellent", "Must read"]
  });
});

module.exports = router;
```

app.js

```
const express = require("express");
const app = express();
const bookRoutes = require("./routes/books");

app.use("/books", bookRoutes);

app.listen(6000, () => console.log("Library API
running..."));
```

Sample Output

GET /books?author=John

```
[
  { "id": 1, "title": "Node Mastery", "author": "John",
"year": 2021 }
]
```

GET /books/1/reviews

```
{
  "bookId": "1",
  "reviews": ["Good", "Excellent", "Must read"]
}
```

Example 2 — Hotel Booking API (Query Search + Parameters)

Scenario

A hotel booking system needs:

- Query-based search: `/hotels?city=Chennai&stars=5`
- Route param for booking ID
- Nested routes for `/hotels/:id/rooms`

Code Implementation

routes/hotels.js

```
const express = require("express");
const router = express.Router();

const hotels = [
  { id: 1, name: "Grand Chennai", city: "Chennai", stars:
5 },
  { id: 2, name: "Silver Stay", city: "Bangalore", stars: 4 }
];

// Filter hotels
router.get("/", (req, res) => {
  const { city, stars } = req.query;
  let result = hotels;

  if (city) result = result.filter(h => h.city === city);
  if (stars) result = result.filter(h => h.stars == stars);

  res.json(result);
});
```

```
// Route param
router.get("/:id", (req, res) => {
  const hotel = hotels.find(h => h.id == req.params.id);
  if (!hotel) return res.status(404).json({ error: "Hotel not found" });
  res.json(hotel);
});

// Nested rooms
router.get("/:id/rooms", (req, res) => {
  res.json({
    hotelId: req.params.id,
    rooms: ["Deluxe", "Suite", "Premium"]
  });
});

module.exports = router;
```

Output

```
GET /hotels?city=Chennai

[{ "id": 1, "name": "Grand Chennai", "city": "Chennai", "stars": 5 }]
```

Example 3 — Online Education Platform (Nested Routes + Multiple Params)

Scenario

Platform with:

- Nested course → lessons route
- Route params: `/courses/:courseId/lessons/:lessonId`
- Query filters: `/courses/:courseId/lessons?difficulty=medium`

Project Structure

```
edu-platform/
├── app.js
├── routes/courses.js
```

Code Implementation

routes/courses.js

```
const express = require("express");
const router = express.Router();

const lessons = {
  1: [
    { id: 1, title: "Intro", difficulty: "easy" },
    { id: 2, title: "Middleware", difficulty: "medium" }
  ]
};

// List lessons with query filter
router.get("/:courseId/lessons", (req, res) => {
  const { courseId } = req.params;
  const { difficulty } = req.query;

  let result = lessons[courseId];
  if (!result) return res.status(404).json({ error: "Course
not found" });

  if (difficulty)
    result = result.filter(l => l.difficulty === difficulty);

  res.json(result);
});

// Specific lesson
router.get("/:courseId/lessons/:lessonId", (req, res) => {
  const { courseId, lessonId } = req.params;
  const lesson = lessons[courseId]?.find(l => l.id ==
lessonId);

  if (!lesson) return res.status(404).json({ error: "Lesson
not found" });

  res.json(lesson);
});
```

```
module.exports = router;
```

app.js

```
const express = require("express");
const app = express();
const courseRoutes = require("./routes/courses");

app.use("/courses", courseRoutes);

app.listen(7000, () => console.log("Education API
running..."));
```

Output

GET /courses/1/lessons?difficulty=medium

```
[
  { "id": 2, "title": "Middleware", "difficulty": "medium" }
]
```

GET /courses/1/lessons/2

```
{ "id": 2, "title": "Middleware", "difficulty": "medium" }
```

1) Middleware Patterns

(authentication, logging, validation, custom error handlers)

Example 1 — Logging Middleware for API Requests

Scenario

We want to log every incoming request with its method, URL, and timestamp.

Project Structure

```
logging-api/
│── app.js
│── middlewares/
│     └── logger.js
│
```

```
├── package.json
```

Code Implementation

middlewares/logger.js

```
module.exports = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} $
{req.url}`);
  next();
};
```

app.js

```
const express = require("express");
const app = express();
const logger = require("./middlewares/logger");

app.use(logger);

app.get("/", (req, res) => res.send("Hello World!"));
app.get("/about", (req, res) => res.send("About Page"));

app.listen(3000, () => console.log("Server running on port
3000"));
```

Output

Console:

```
[2025-11-22T17:45:00.123Z] GET /
[2025-11-22T17:45:05.456Z] GET /about
```

Example 2 — Authentication Middleware

Scenario

Secure an endpoint using a token sent in headers.

middlewares/auth.js

```
module.exports = (req, res, next) => {
  const token = req.headers['authorization'];
  if (token !== "12345") {
    return res.status(401).json({ error: "Unauthorized" });
  }
  next();
};
```

app.js

```
const express = require("express");
const app = express();
const auth = require("./middlewares/auth");

app.use(express.json());

app.get("/secure", auth, (req, res) => {
  res.json({ message: "Secure data accessed" });
});

app.listen(3001, () => console.log("Server running on port 
3001"));
```

Output

- Without token:

```
{ "error": "Unauthorized" }
```

- With token `12345` in header:

```
{ "message": "Secure data accessed" }
```

Example 3 — Validation & Custom Error Handling

Scenario

POST `/users` requires `name` and `email`. Invalid input triggers a custom error handler.

middlewares/validateUser.js

```
module.exports = (req, res, next) => {
  const { name, email } = req.body;
  if (!name || !email) {
    const err = new Error("Name and email are required");
    err.status = 400;
    return next(err);
  }
  next();
};
```

app.js

```
const express = require("express");
const app = express();
const validateUser = require("./middlewares/validateUser");

app.use(express.json());

app.post("/users", validateUser, (req, res) => {
  res.status(201).json({ message: "User created", user:
req.body });
});

// Custom Error Handler
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ error: err.message });
});

app.listen(3002, () => console.log("Server running on port
3002"));
```

Output

- POST `/users` with `{}`

```
{ "error": "Name and email are required" }
```

- POST `/users` with `{ "name": "John", "email": "john@example.com" }`

```
{
  "message": "User created",
  "user": { "name": "John", "email": "john@example.com" }
}
```

2) Request/Response Lifecycle

(parsing bodies, headers, cookies, CORS)

Example 1 — Parsing JSON & URL-encoded Bodies

Scenario

Accept form submissions in JSON and URL-encoded formats.

Project Structure

```
body-api/
├── app.js
├── package.json
```

app.js

```
const express = require("express");
const app = express();

// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

app.post("/submit", (req, res) => {
  res.json({ message: "Data received", data: req.body });
});

app.listen(3003, () => console.log("Server running on port 
3003"));
```

Output

POST `/submit` with JSON:

```
{ "name": "Alice", "age": 25 }
```

Response:

```
{
  "message": "Data received",
  "data": { "name": "Alice", "age": 25 }
}
```

Example 2 — Reading Headers and Cookies

Scenario: Log a custom header and read cookies.

Install cookie-parser

```
npm install cookie-parser
```

app.js

```
const express = require("express");
const cookieParser = require("cookie-parser");
const app = express();

app.use(cookieParser());

app.get("/info", (req, res) => {
  const userAgent = req.headers["user-agent"];
  const sessionId = req.cookies.sessionId || "No cookie";
  res.json({ userAgent, sessionId });
});

app.listen(3004, () => console.log("Server running on port 
3004"));
```

Output

Headers: `User-Agent: Mozilla/5.0`
Cookies: `sessionId=abc123`

Response:

```
{ "userAgent": "Mozilla/5.0", "sessionId": "abc123" }
```

Example 3 — Enabling CORS for Cross-Origin Requests

Install cors

```
npm install cors
```

app.js

```
const express = require("express");
const cors = require("cors");
const app = express();

// Enable CORS for all origins
app.use(cors());

app.get("/data", (req, res) => {
  res.json({ message: "This is accessible from any 
domain" });
});

app.listen(3005, () => console.log("Server running on port 3005"));
```

Output

- Accessible via browser, Postman, or frontend app from any origin:

```
{ "message": "This is accessible from any domain" }
```

Example 1 — Employee API with Logging + Authentication + Validation

Scenario

A company wants an **Employee API** that allows adding and fetching employee data. Requirements:

- Log every request with timestamp and route
- Authenticate requests with a simple token in headers
- Validate employee data on creation (name and department required)
- Handle errors globally

Project Structure

```
employee-api/
│── app.js
│── routes/
│      └── employees.js
│── middlewares/
│      ├── logger.js
│      ├── auth.js
│      └── validateEmployee.js
│── package.json
```

Code Implementation

middlewares/logger.js

```
module.exports = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} $
{req.url}`);
  next();
};
```

middlewares/auth.js

```
module.exports = (req, res, next) => {
  const token = req.headers['authorization'];
  if (!token || token !== 'secret123') {
    return res.status(401).json({ error: 'Unauthorized' });
  }
  next();
};
```

middlewares/validateEmployee.js

```
module.exports = (req, res, next) => {
  const { name, department } = req.body;
  if (!name || !department) {
    const err = new Error('Name and Department are
required');
    err.status = 400;
    return next(err);
  }
  next();
};
```

routes/employees.js

```
const express = require('express');
const router = express.Router();
const validateEmployee = require('../middlewares/
validateEmployee');

let employees = [
  { id: 1, name: 'John Doe', department: 'HR' }
];

// GET all employees
router.get('/', (req, res) => {
  res.json(employees);
});
```

```
// POST create employee
router.post('/', validateEmployee, (req, res) => {
  const newEmp = { id: employees.length + 1, ...req.body };
  employees.push(newEmp);
  res.status(201).json(newEmp);
});

module.exports = router;
```

app.js

```
const express = require('express');
const app = express();
const logger = require('./middlewares/logger');
const auth = require('./middlewares/auth');
const employeeRoutes = require('./routes/employees');

app.use(express.json());
app.use(logger);
app.use(auth);
app.use('/employees', employeeRoutes);

// Global error handler
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ error: err.message });
});

app.listen(3000, () => console.log('Server running on port
3000'));
```

Explanation

1. **Logger middleware** logs timestamp + route for every request.
2. **Auth middleware** blocks requests without correct token.
3. **Validation middleware** ensures required fields are present before creating employee.
4. **Global error handler** catches validation errors or other exceptions.

Output

GET `/employees` (with token `secret123`)

```
[
  { "id": 1, "name": "John Doe", "department": "HR" }
]
```

POST `/employees` with body `{ "name": "Mary" }`

```
{ "error": "Name and Department are required" }
```

POST `/employees` with body `{ "name": "Mary", "department": "Finance" }`

```
{
  "id": 2,
  "name": "Mary",
  "department": "Finance"
}
```

Example 2 — Product API with Logging + Role-Based Auth + Error Handler

Scenario

An e-commerce API:

- Logs all requests
- Only admin users can delete products
- Returns proper errors for unauthorized or invalid operations

Project Structure

```
product-api/
│── app.js
│── routes/
│     └── products.js
│── middlewares/
│     ├── logger.js
│     └── adminAuth.js
│── package.json
```

Code Implementation

middlewares/logger.js

```
module.exports = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} $
{req.url}`);
  next();
};
```

middlewares/adminAuth.js

```
module.exports = (req, res, next) => {
  const role = req.headers['role'];
  if (role !== 'admin') {
    return res.status(403).json({ error: 'Admin access
required' });
  }
  next();
};
```

routes/products.js

```
const express = require('express');
const router = express.Router();

let products = [
  { id: 1, name: 'Laptop', price: 50000 },
  { id: 2, name: 'Mouse', price: 500 }
];

// GET all products
router.get('/', (req, res) => res.json(products));

// DELETE product (admin only)
router.delete('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = products.findIndex(p => p.id === id);
  if (index === -1) return res.status(404).json({ error:
'Product not found' });

  products.splice(index, 1);
  res.json({ message: 'Product deleted' });
});
```

```
module.exports = router;
```

app.js

```
const express = require('express');
const app = express();
const logger = require('./middlewares/logger');
const adminAuth = require('./middlewares/adminAuth');
const productRoutes = require('./routes/products');

app.use(express.json());
app.use(logger);

// Admin middleware only for DELETE route
app.use('/products/:id', (req, res, next) => {
  if (req.method === 'DELETE') return adminAuth(req, res,
next);
  next();
});

app.use('/products', productRoutes);

// Global error handler
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ error: err.message });
});

app.listen(3001, () => console.log('Product API running on
port 3001'));
```

Explanation

1. **Logger middleware** logs all requests.
2. **AdminAuth middleware** protects DELETE route.
3. **Global error handler** ensures proper JSON responses for errors.
4. Admin role is checked via custom `role` header.

Output

DELETE `/products/2` with header `role=user`

```
{ "error": "Admin access required" }
```

DELETE `/products/2` with header `role=admin`

```
{ "message": "Product deleted" }
```

1) Request/Response Lifecycle

(parsing bodies, headers, cookies, CORS)

Example 1 — Contact Form API with JSON & URL-encoded Bodies

Scenario

A website contact form submits data as **JSON** or **URL-encoded** form. Backend should parse both, read headers, and respond with a confirmation message.

Project Structure

```
contact-api/
│── app.js
│── package.json
```

Code Implementation

app.js

```
const express = require("express");
const app = express();

// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

// POST /contact
app.post("/contact", (req, res) => {
  const { name, email, message } = req.body;
  const userAgent = req.headers["user-agent"]; // read header
  res.json({
    message: `Thanks ${name}! Your message was received.`,
    submittedData: { name, email, message },
    userAgent
  });
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

Explanation

1. `express.json()` parses JSON bodies.
2. `express.urlencoded()` parses `application/x-www-form-urlencoded` forms.
3. `req.headers` allows reading HTTP headers like `User-Agent`.
4. Response includes parsed data + header info.

Sample Output

POST /contact (JSON Body)

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "message": "Hello!"
}
```

Response

```
{
  "message": "Thanks Alice! Your message was received.",
  "submittedData": {
    "name": "Alice",
    "email": "alice@example.com",
    "message": "Hello!"
  },
  "userAgent": "PostmanRuntime/7.29.0"
}
```

Example 2 — Cookie & CORS Handling

Scenario

A frontend website needs to send requests to the backend on a different domain. Backend should:

- Enable CORS
- Read cookies for session info

Project Structure

```
cookie-cors-api/
├── app.js
├── package.json
```

Code Implementation

```
const express = require("express");
const cors = require("cors");
const cookieParser = require("cookie-parser");
const app = express();

// Enable CORS for frontend domain
app.use(cors({ origin: "http://localhost:8080", credentials:
true }));

// Parse cookies
app.use(cookieParser());

// Example route
app.get("/profile", (req, res) => {
  const sessionId = req.cookies.sessionId || "No session";
  res.json({ message: "Profile data", sessionId });
});

app.listen(3001, () => console.log("Server running on port
3001"));
```

Explanation

1. `cors({ origin, credentials })` allows cross-origin requests with cookies.
2. `cookie-parser` parses cookies from incoming requests.
3. `req.cookies` retrieves session info.

Sample Output

- Request with cookie: `sessionId=abc123`

Response:

```
{
  "message": "Profile data",
  "sessionId": "abc123"
}
```

- Without cookie:

```
{
  "message": "Profile data",
  "sessionId": "No session"
}
```

2) API Versioning and Documentation (Swagger/OpenAPI)

Example 1 — Versioned User API with Swagger Docs

Scenario

A company wants versioned User APIs:

- `v1/users` → basic info
- `v2/users` → extended info with role
- Swagger documentation available at `/api-docs`

Project Structure

```
user-api/
│── app.js
│── routes/
│    └── users.js
│── package.json
```

Install Packages

```
npm install express swagger-ui-express swagger-jsdoc
```

Code Implementation

routes/users.js

```
const express = require("express");
const router = express.Router();

/**
 * @swagger
 * /users/v1:
 *   get:
 *     summary: Get all users (v1)
 *     tags:
 *       - Users
 *     responses:
 *       200:
 *         description: List of users
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 type: object
 *                 properties:
 *                   id:
 *                     type: integer
 *                   name:
 *                     type: string
 */
router.get("/v1", (req, res) => {
  res.json([
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" }
  ]);
});

/**
 * @swagger
 * /users/v2:
 *   get:
 *     summary: Get all users (v2 with role)
 *     tags:
 *       - Users
 *     responses:
 *       200:
 *         description: List of users with roles
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 type: object
 *                 properties:
```

```
 *                   id:
 *                     type: integer
 *                   name:
 *                     type: string
 *                   role:
 *                     type: string
 */
router.get("/v2", (req, res) => {
  res.json([
    { id: 1, name: "Alice", role: "admin" },
    { id: 2, name: "Bob", role: "user" }
  ]);
});

module.exports = router;
```

app.js

```
const express = require("express");
const app = express();
const userRoutes = require("./routes/users");

const swaggerUi = require("swagger-ui-express");
const swaggerJsdoc = require("swagger-jsdoc");

// Swagger setup
const options = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "User API",
      version: "1.0.0"
    }
  },
  apis: ["./routes/*.js"]
};

const specs = swaggerJsdoc(options);

// Use routes
app.use("/users", userRoutes);

// Swagger docs
app.use("/api-docs", swaggerUi.serve,
swaggerUi.setup(specs));
```

```
app.listen(3002, () => console.log("Server running on port
3002"));
```

Output

- `GET /users/v1` → `[ { id:1, name:"Alice" }, { id:2, name:"Bob" } ]`
- `GET /users/v2` → `[ { id:1, name:"Alice", role:"admin" }, { id:2, name:"Bob", role:"user" } ]`
- Swagger UI: `http://localhost:3002/api-docs` → interactive documentation.

Example 2 — Versioned Product API

Scenario

API has multiple versions:

- `v1/products` → basic info
- `v2/products` → includes category and price
- Swagger docs for both versions

Project Structure

```
product-api/
|── app.js
|── routes/
|     └── products.js
|── package.json
```

routes/products.js

```
const express = require("express");
const router = express.Router();

router.get("/v1", (req, res) => {
  res.json([
    { id: 1, name: "Laptop" },
    { id: 2, name: "Mouse" }
  ]);
});
```

```
router.get("/v2", (req, res) => {
  res.json([
    { id: 1, name: "Laptop", category: "Electronics", price: 
50000 },
    { id: 2, name: "Mouse", category: "Electronics", price: 
500 }
  ]);
});

module.exports = router;
```

app.js

```
const express = require("express");
const app = express();
const productRoutes = require("./routes/products");

const swaggerUi = require("swagger-ui-express");
const swaggerJsdoc = require("swagger-jsdoc");

const options = {
  definition: { openapi: "3.0.0", info: { title: "Product 
API", version: "1.0.0" } },
  apis: ["./routes/*.js"]
};

const specs = swaggerJsdoc(options);

app.use("/products", productRoutes);
app.use("/api-docs", swaggerUi.serve, 
swaggerUi.setup(specs));

app.listen(3003, () => console.log("Server running on port 
3003"));
```

Output

- `GET /products/v1` → basic products
- `GET /products/v2` → products with category + price
- `GET /api-docs` → interactive API documentation

- SQL:
  - Integrate with MySQL/PostgreSQL using sequelize or knex.

- Migrations, transactions, connection pooling, query optimization.

Example 1 — Employee Management with Transactions

Scenario

A company wants to **add a new employee and a related bonus record atomically**. If any operation fails, the transaction should rollback.

- **Tables:** `Employees` and `Bonuses`
- Sequelize **transaction** is used to ensure both inserts succeed together.

Project Structure

```
employee-transaction-api/
│── app.js
│── models/
│     ├── index.js
│     ├── employee.js
│     └── bonus.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express sequelize mysql2
```

Step 2 — Sequelize Models

models/index.js

```
const { Sequelize } = require("sequelize");

const sequelize = new Sequelize("companydb", "root",
"password", {
  host: "localhost",
  dialect: "mysql",
  pool: {
    max: 5,
    min: 0,
    acquire: 30000,
    idle: 10000
  }
});

const db = {};
db.Sequelize = Sequelize;
db.sequelize = sequelize;
db.Employee = require("./employee")(sequelize, Sequelize);
db.Bonus = require("./bonus")(sequelize, Sequelize);

db.Employee.hasOne(db.Bonus, { foreignKey: "employeeId" });
db.Bonus.belongsTo(db.Employee, { foreignKey:
"employeeId" });

module.exports = db;
```

models/employee.js

```
module.exports = (sequelize, DataTypes) => {
  return sequelize.define("Employee", {
    name: { type: DataTypes.STRING, allowNull: false },
    department: { type: DataTypes.STRING, allowNull: false },
    salary: { type: DataTypes.FLOAT, allowNull: false }
  });
};
```

models/bonus.js

```
module.exports = (sequelize, DataTypes) => {
  return sequelize.define("Bonus", {
    amount: { type: DataTypes.FLOAT, allowNull: false }
  });
};
```

Step 3 — Express API with Transaction

app.js

```
const express = require("express");
const app = express();
app.use(express.json());
```

```
const db = require("./models");

db.sequelize.sync({ alter: true });

app.post("/employee", async (req, res) => {
  const t = await db.sequelize.transaction();
  try {
    const { name, department, salary, bonus } = req.body;

    const emp = await db.Employee.create({ name, department,
salary }, { transaction: t });

    await db.Bonus.create({ employeeId: emp.id, amount: bonus
}, { transaction: t });

    await t.commit();
    res.json({ message: "Employee and bonus created",
employee: emp });
  } catch (error) {
    await t.rollback();
    res.status(500).json({ error: error.message });
  }
});

app.listen(3000, () => console.log("Server running on port
3000"));
```

Code Explanation

1. `sequelize.transaction()` ensures **atomic operations**.
2. `db.sequelize.sync({ alter: true })` automatically creates tables.
3. Connection pooling is configured in `models/index.js` under `pool`.
4. If any insert fails, `t.rollback()` undoes all changes.

Sample Output

POST /employee

Request Body:

```
{
  "name": "John",
  "department": "HR",
  "salary": 50000,
  "bonus": 5000
}
```

Response:

```
{
  "message": "Employee and bonus created",
  "employee": {
    "id": 1,
    "name": "John",
    "department": "HR",
    "salary": 50000
  }
}
```

Project Structure

```
employee-api/
│── app.js
│── models/
│     ├── employee.js
│     └── index.js  (sequelize.js)
│── package.json
```

1) models/index.js (sequelize.js)

```
const { Sequelize } = require("sequelize");

// MySQL connection
const sequelize = new Sequelize("companydb", "root",
"password", {
  host: "localhost",
  dialect: "mysql",
  logging: false, // disable SQL logs, set true for debugging
  pool: {
    max: 5,
    min: 0,
    acquire: 30000,
    idle: 10000
  }
});

// Test connection
sequelize.authenticate()
  .then(() => console.log("MySQL connected..."))
  .catch(err => console.error("Unable to connect:", err));

const db = {};
db.sequelize = sequelize;
db.Sequelize = Sequelize;

// Models
db.Employee = require("./employee")(sequelize, Sequelize);

module.exports = db;
```

2) models/employee.js

```
module.exports = (sequelize, DataTypes) => {
  const Employee = sequelize.define("Employee", {
    name: {
      type: DataTypes.STRING,
      allowNull: false
    },
    salary: {
      type: DataTypes.FLOAT,
      allowNull: false
    },
    department: {
      type: DataTypes.STRING,
      allowNull: false
    }
  }, {
    tableName: "Employees",
    timestamps: true
  });

  return Employee;
};
```

3) Notes

1. **Database Name:** `companydb` (create it in MySQL first):

```
CREATE DATABASE companydb;
```

2. **Bonuses Table:** Since you are using raw query in `/employee/with-bonus`, create a simple table:

```
CREATE TABLE Bonuses (
  id INT AUTO_INCREMENT PRIMARY KEY,
  employeeId INT NOT NULL,
  amount FLOAT NOT NULL,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  FOREIGN KEY (employeeId) REFERENCES Employees(id)
);
```

3. Sequelize **connection pooling** is configured in `index.js`.

4. Model uses **timestamps** (`createdAt`, `updatedAt`).

4) How to Run

1. Install dependencies:

```
npm install express sequelize mysql2
```

2. Start server:

```
node app.js
```

3. Test endpoints in **Postman**:

- `POST /employee` → `{ "name": "John", "salary": 50000, "department": "HR" }`
- `POST /employee/with-bonus` → same body, automatically adds bonus 5000
- `GET /top-salaries` → top 5 salaries
- `GET /employees?dept=HR&page=1&limit=10` → paginated filtered results

Example 2 — Optimized Product Queries with Migrations

Scenario

An e-commerce app requires:

1. A `Products` table with migration setup
2. Fetch products filtered by category **efficiently** (indexed column)
3. Demonstrate **connection pooling** for multiple requests

Project Structure

```
product-api/
|── app.js
|── models/
|     ├── index.js
|     └── product.js
|── migrations/
|     └── 20251122-create-product.js
|── package.json
```

Step 1 — Sequelize Migration

migrations/20251122-create-product.js

```
'use strict';
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Products', {
      id: { type: Sequelize.INTEGER, autoIncrement: true,
primaryKey: true },
      name: { type: Sequelize.STRING, allowNull: false },
      category: { type: Sequelize.STRING, allowNull: false },
      price: { type: Sequelize.FLOAT, allowNull: false },
      createdAt: { type: Sequelize.DATE, allowNull: false,
defaultValue: Sequelize.literal('CURRENT_TIMESTAMP') },
      updatedAt: { type: Sequelize.DATE, allowNull: false,
defaultValue: Sequelize.literal('CURRENT_TIMESTAMP') }
    });
    await queryInterface.addIndex('Products',
['category']); // index for query optimization
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Products');
  }
};
```

Step 2 — Product Model

models/product.js

```
module.exports = (sequelize, DataTypes) => {
  return sequelize.define("Product", {
    name: { type: DataTypes.STRING, allowNull: false },
    category: { type: DataTypes.STRING, allowNull: false },
    price: { type: DataTypes.FLOAT, allowNull: false }
  });
};
```

models/index.js (same as Example 1, adjust for Product)

Step 3 — API with Optimized Queries

app.js

```
const express = require("express");
const app = express();
app.use(express.json());

const db = require("./models");

db.sequelize.sync({ alter: true });

app.get("/products", async (req, res) => {
  const { category, maxPrice } = req.query;

  const filter = {};
  if (category) filter.category = category;
  if (maxPrice) filter.price = { [db.Sequelize.Op.lte]:
parseFloat(maxPrice) };

  // Optimized query using where filter and indexed column
  const products = await db.Product.findAll({ where: filter,
order: [["price", "ASC"]] });

  res.json(products);
});
```

```
app.listen(3001, () => console.log("Server running on port
3001"));
```

Code Explanation

1. Migration creates table + **index on `category`** for query optimization.
2. Connection pooling handles multiple concurrent requests efficiently.
3. `findAll({ where: filter })` uses Sequelize operators for optimized queries.
4. Ordering results by price demonstrates query flexibility.

Sample Output

GET /products?category=Electronics&maxPrice=1000

```
[
  { "id": 2, "name": "Mouse", "category": "Electronics",
"price": 500 },
  { "id": 3, "name": "Keyboard", "category": "Electronics",
"price": 800 }
]
```

- NoSQL:
  - MongoDB with Mongoose: schema design, population, aggregation

Example 1 — Employee & Department Relationship (Population & Aggregation)

Scenario

A company wants to manage **employees and their departments** in MongoDB.

- Each employee belongs to a department.
- Fetch employees with department details using **population**.
- Calculate **average salary per department** using **aggregation**.

Project Structure

```
employee-mongo-api/
│── app.js
│── models/
│     ├── department.js
│     └── employee.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express mongoose
```

Step 2 — Mongoose Models

models/department.js

```
const mongoose = require("mongoose");

const departmentSchema = new mongoose.Schema({
  name: { type: String, required: true, unique: true },
  location: { type: String, required: true }
});

module.exports = mongoose.model("Department",
departmentSchema);
```

models/employee.js

```
const mongoose = require("mongoose");

const employeeSchema = new mongoose.Schema({
  name: { type: String, required: true },
  salary: { type: Number, required: true, min: 1000 },
  department: { type: mongoose.Schema.Types.ObjectId, ref:
"Department", required: true }
});

module.exports = mongoose.model("Employee", employeeSchema);
```

Step 3 — Express API with Population & Aggregation

app.js

```
const express = require("express");
const mongoose = require("mongoose");
const Employee = require("./models/employee");
const Department = require("./models/department");

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

// MongoDB connection
mongoose.connect("mongodb://127.0.0.1:27017/companydb", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB connected"))
  .catch(err => console.error("MongoDB connection error:",
err));

// Add department
app.post("/departments", async (req, res) => {
  const dept = await Department.create(req.body);
  res.json(dept);
});

// Add employee
app.post("/employees", async (req, res) => {
  const emp = await Employee.create(req.body);
  res.json(emp);
});

// Get employees with department info (population)
app.get("/employees", async (req, res) => {
  const employees = await
Employee.find().populate("department", "name location");
  res.json(employees);
});

// Average salary per department (aggregation)
app.get("/departments/average-salary", async (req, res) => {
  const result = await Employee.aggregate([
    { $group: { _id: "$department", avgSalary: { $avg:
"$salary" } } },
    {
      $lookup: {
        from: "departments",
        localField: "_id",
        foreignField: "_id",
        as: "department"
      }
    },
    { $unwind: "$department" },
    { $project: { _id: 0, department: "$department.name",
avgSalary: 1 } }
  ]);
  res.json(result);
});

app.listen(3000, () => console.log("Server running on port
3000"));
```

Code Explanation

1. **Population:** `Employee.find().populate("department", "name location")` fetches the department details in employee documents.
2. **Aggregation:** `$group` calculates average salary per department; `$lookup` joins with `departments` collection.
3. **Validation:** Mongoose schema ensures required fields (`name`, `salary`, `department`) and `salary` minimum.

Sample Output

POST /departments

```
{ "_id": "651f3b1e", "name": "HR", "location": "Chennai" }
```

POST /employees

```
{ "_id": "651f3c1f", "name": "Alice", "salary": 50000,
"department": "651f3b1e" }
```

GET /employees

```
[
  {
    "_id": "651f3c1f",
    "name": "Alice",
    "salary": 50000,
    "department": { "_id": "651f3b1e", "name": "HR",
"location": "Chennai" }
  }
]
```

GET /departments/average-salary

```
[
  { "department": "HR", "avgSalary": 50000 }
]
```

Example 2 — User Registration with Validation & Sanitization

Scenario

Create a **user registration API** with:

- Required fields: `username`, `email`, `password`
- Validation: username length, email format, password strength
- Sanitization: trim spaces, lowercase email

Project Structure

```
user-mongo-api/
│── app.js
│── models/
│     └── user.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express mongoose validator
```

Step 2 — Mongoose Model with Validation & Sanitization

models/user.js

```
const mongoose = require("mongoose");
const validator = require("validator");

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: [true, "Username required"],
    minlength: [3, "Username must be at least 3 characters"],
    trim: true
  },
  email: {
    type: String,
    required: [true, "Email required"],
    lowercase: true,
    trim: true,
    validate: [validator.isEmail, "Invalid email format"]
  },
  password: {
    type: String,
    required: [true, "Password required"],
    minlength: [6, "Password must be at least 6 characters"]
  }
});

module.exports = mongoose.model("User", userSchema);
```

Step 3 — Express API

app.js

```
const express = require("express");
const mongoose = require("mongoose");
const User = require("./models/user");

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

// MongoDB connection
mongoose.connect("mongodb://127.0.0.1:27017/userdb", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB connected"))
```

```
  .catch(err => console.error("MongoDB connection error:", 
err));

// Register user
app.post("/register", async (req, res) => {
  try {
    const user = await User.create(req.body);
    res.json({ message: "User registered successfully", 
user });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

app.listen(3000, () => console.log("Server running on port 
3000"));
```

Code Explanation

1. **Validation:**
   - Username: minimum 3 characters
   - Email: valid format using `validator.isEmail`
   - Password: minimum 6 characters
2. **Sanitization:**
   - `trim: true` removes extra spaces
   - `lowercase: true` ensures email stored in lowercase
3. **Error Handling:** returns Mongoose validation errors as JSON

Sample Output

POST /register
Request:

```
{
  "username": "  Alice ",
  "email": "ALICE@EXAMPLE.COM ",
  "password": "secret123"
}
```

Response:

```
{
  "message": "User registered successfully",
  "user": {
    "_id": "6520a1f2",
    "username": "Alice",
    "email": "alice@example.com",
    "password": "secret123"
  }
}
```

Validation Error Example:
Request with invalid email:

```
{
  "username": "Bob",
  "email": "bob@@example",
  "password": "123456"
}
```

Response:

```
{
  "error": "User validation failed: email: Invalid email format"
}
```

Summary of Concepts

Feature	Example 1	Example 2
Schema Design	Employee + Department with ObjectId ref	User schema with validation
Population	`.populate("department")`	—
Aggregation	Average salary per department	—
Validation	Mongoose required, min, unique	Username/email/password checks
Sanitization	—	Trim & lowercase email
Relationships	Employee → Department	—

- MongoDB with Mongoose: schema design,Data validation and sanitization.

Example 1 — Employee & Department (Schema Design with Population)

Scenario

A company wants to manage employees and departments:

- Each employee belongs to a department.
- Fetch employees with department details using **population**.
- Demonstrates **schema design and relationships**.

Project Structure

```
employee-mongo-api/
│── app.js
│── models/
│    ├── department.js
│    └── employee.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express mongoose
```

Step 2 — Mongoose Models

models/department.js

```
const mongoose = require("mongoose");

const departmentSchema = new mongoose.Schema({
  name: { type: String, required: true, unique: true },
  location: { type: String, required: true }
});

module.exports = mongoose.model("Department",
departmentSchema);
```

models/employee.js

```
const mongoose = require("mongoose");
const employeeSchema = new mongoose.Schema({
  name: { type: String, required: true },
  salary: { type: Number, required: true, min: 1000 },
  department: { type: mongoose.Schema.Types.ObjectId, ref:
"Department", required: true }
});

module.exports = mongoose.model("Employee", employeeSchema);
```

Step 3 — Express API

app.js

```
const express = require("express");
const mongoose = require("mongoose");
const Employee = require("./models/employee");
const Department = require("./models/department");

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

// MongoDB connection
mongoose.connect("mongodb://127.0.0.1:27017/companydb", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB connected"))
  .catch(err => console.error(err));

// Add department
app.post("/departments", async (req, res) => {
  const dept = await Department.create(req.body);
  res.json(dept);
});

// Add employee
app.post("/employees", async (req, res) => {
  const emp = await Employee.create(req.body);
  res.json(emp);
});

// Get employees with department info
app.get("/employees", async (req, res) => {
  const employees = await
Employee.find().populate("department", "name location");
```

```
  res.json(employees);
});

app.listen(3000, () => console.log("Server running on port 
3000"));
```

Explanation

1. Employee → Department is a **relationship** using `ObjectId` reference.
2. `.populate("department")` fetches full department info.
3. Mongoose schema enforces required fields and minimum salary.

Sample Output

POST /departments

```
{ "_id": "651f3b1e", "name": "HR", "location": "Chennai" }
```

POST /employees

```
{ "_id": "651f3c1f", "name": "Alice", "salary": 50000, 
"department": "651f3b1e" }
```

GET /employees

```
[
  {
    "_id": "651f3c1f",
    "name": "Alice",
    "salary": 50000,
    "department": { "_id": "651f3b1e", "name": "HR", 
"location": "Chennai" }
  }
]
```

Example 2 — User Registration (Validation & Sanitization)

Scenario

Create a **user registration API** with:

- Required fields: `username`, `email`, `password`
- Validation: username length, email format, password strength
- Sanitization: trim spaces, lowercase email

Project Structure

```
user-mongo-api/
│── app.js
│── models/
│    └── user.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express mongoose validator
```

Step 2 — User Model

models/user.js

```
const mongoose = require("mongoose");
const validator = require("validator");

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: [true, "Username required"],
    minlength: [3, "Username must be at least 3 characters"],
    trim: true
  },
  email: {
    type: String,
    required: [true, "Email required"],
    lowercase: true,
    trim: true,
    validate: [validator.isEmail, "Invalid email format"]
  },
  password: {
    type: String,
    required: [true, "Password required"],
    minlength: [6, "Password must be at least 6 characters"]
  }
});

module.exports = mongoose.model("User", userSchema);
```

Step 3 — Express API

app.js

```
const express = require("express");
const mongoose = require("mongoose");
const User = require("./models/user");

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

// MongoDB connection
mongoose.connect("mongodb://127.0.0.1:27017/userdb", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB connected"))
  .catch(err => console.error(err));

// Register user
app.post("/register", async (req, res) => {
  try {
    const user = await User.create(req.body);
    res.json({ message: "User registered successfully",
user });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

app.listen(3000, () => console.log("Server running on port
3000"));
```

Explanation

1. **Validation:** username length, email format, password length.

2. **Sanitization:** `trim` removes extra spaces, `lowercase` standardizes emails.
3. Mongoose handles schema enforcement automatically.

Sample Output

POST /register

```
{
  "username": "  Alice ",
  "email": "ALICE@EXAMPLE.COM ",
  "password": "secret123"
}
```

Response:

```
{
  "message": "User registered successfully",
  "user": {
    "_id": "6520a1f2",
    "username": "Alice",
    "email": "alice@example.com",
    "password": "secret123"
  }
}
```

Validation Error Example

```
{
  "username": "Bo",
  "email": "bob@@example",
  "password": "123"
}
```

Response:

```
{
  "error": "User validation failed: username: Username must be at least 3 characters, email: Invalid email format, password: Password must be at least 6 characters"
}
```

These **two examples** cover:

Feature	Example 1	Example 2

Schema Design	Employee + Department with ObjectId	User registration
Relationships	Population	—
Validation	Mongoose required & min	Username/email/password validation
Sanitization	—	Trim & lowercase email

Example 1 — Employee API with Centralized Error Handling

Scenario

A company maintains employee records. We want:

- To handle **errors centrally** instead of repeating `try/catch` in each route.
- Return **consistent JSON error responses**.

Project Structure

```
employee-error-api/
├── app.js
├── routes/
│    └── employee.js
├── middleware/
│    └── errorHandler.js
├── models/
│    └── employee.js
├── package.json
```

Step 1 — Install Dependencies

```
npm install express
```

Step 2 — Employee Model (In-memory for simplicity)

models/employee.js

```
let employees = [];
let idCounter = 1;

module.exports = {
  getAll: () => employees,
  getById: (id) => employees.find(e => e.id === id),
  create: (data) => {
    const emp = { id: idCounter++, ...data };
    employees.push(emp);
    return emp;
  }
};
```

Step 3 — Employee Routes

routes/employee.js

```
const express = require("express");
const router = express.Router();
const Employee = require("../models/employee");

// Get employee by ID
router.get("/:id", (req, res, next) => {
  const emp = Employee.getById(parseInt(req.params.id));
  if (!emp) return next({ status: 404, message: "Employee not
found" });
  res.json(emp);
});

// Create employee
router.post("/", (req, res, next) => {
  const { name, salary } = req.body;
  if (!name || !salary) return next({ status: 400, message:
"Name & salary required" });
  const emp = Employee.create({ name, salary });
  res.status(201).json(emp);
});

module.exports = router;
```

Step 4 — Centralized Error Middleware

middleware/errorHandler.js

```
module.exports = (err, req, res, next) => {
  const status = err.status || 500;
  const message = err.message || "Internal Server Error";
  res.status(status).json({ error: message, status });
};
```

Step 5 — App Setup

app.js

```
const express = require("express");
const app = express();
const employeeRoutes = require("./routes/employee");
const errorHandler = require("./middleware/errorHandler");

app.use(express.json());
app.use("/employees", employeeRoutes);

// Centralized error handler
app.use(errorHandler);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Sample Output

GET /employees/1 (employee not created yet)

```
{
  "error": "Employee not found",
  "status": 404
}
```

POST /employees (missing fields)

```
{
  "error": "Name & salary required",
  "status": 400
}
```

POST /employees (valid)

```
{
  "id": 1,
  "name": "Alice",
  "salary": 50000
}
```

Example 2 — Custom Error Classes with Error Codes

Scenario

We want a **custom error class** to handle:

- Validation errors
- Not found errors
- Include **custom codes** for client-side handling

Project Structure

```
custom-error-api/
│── app.js
│── routes/
│    └── product.js
│── errors/
│    └── CustomError.js
│── middleware/
│    └── errorHandler.js
│── models/
│    └── product.js
│── package.json
```

Step 1 — Custom Error Class

errors/CustomError.js

```
class CustomError extends Error {
  constructor(message, status = 500, code = "GENERIC_ERROR")
{
    super(message);
    this.status = status;
    this.code = code;
  }
}

module.exports = CustomError;
```

Step 2 — Product Model (In-memory)

models/product.js

```
let products = [];
let idCounter = 1;

module.exports = {
  getAll: () => products,
  getById: (id) => products.find(p => p.id === id),
  create: (data) => {
    const product = { id: idCounter++, ...data };
    products.push(product);
    return product;
  }
};
```

Step 3 — Product Routes

routes/product.js

```
const express = require("express");
const router = express.Router();
const Product = require("../models/product");
const CustomError = require("../errors/CustomError");

// Get product by ID
router.get("/:id", (req, res, next) => {
  const product = Product.getById(parseInt(req.params.id));
  if (!product) return next(new CustomError("Product not
found", 404, "PRODUCT_NOT_FOUND"));
  res.json(product);
});

// Create product
router.post("/", (req, res, next) => {
  const { name, price } = req.body;
  if (!name || !price) return next(new CustomError("Name &
price required", 400, "VALIDATION_ERROR"));
  const product = Product.create({ name, price });
  res.status(201).json(product);
});

module.exports = router;
```

Step 4 — Centralized Error Middleware

middleware/errorHandler.js

```
module.exports = (err, req, res, next) => {
  const status = err.status || 500;
  const code = err.code || "GENERIC_ERROR";
  const message = err.message || "Internal Server Error";
  res.status(status).json({ code, message, status });
};
```

Step 5 — App Setup

app.js

```
const express = require("express");
const app = express();
const productRoutes = require("./routes/product");
const errorHandler = require("./middleware/errorHandler");

app.use(express.json());
app.use("/products", productRoutes);

// Centralized error handling
app.use(errorHandler);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Sample Output

GET /products/1 (not exists)

```
{
  "code": "PRODUCT_NOT_FOUND",
  "message": "Product not found",
  "status": 404
}
```

POST /products (missing price)

```
{
  "code": "VALIDATION_ERROR",
  "message": "Name & price required",
  "status": 400
}
```

POST /products (valid)

```
{
  "id": 1,
  "name": "Laptop",
  "price": 1200
}
```

Key Concepts

Feature	Example 1	Example 2
Centralized error handling	`middleware/errorHandler.js`	Same, with custom codes
Custom error	Simple `{status,`	`CustomError` class with `status` &
Route-level error	`next({status,`	`next(new CustomError(...))`
JSON error response	`{error: "...", status: ...}`	`{code: "...", message: "...", status: ...}`

Project: Employee API with Logging & Monitoring

Project Structure

```
el10-logging-monitoring/
├── app.js
├── routes/
│    └── employee.js
├── middleware/
```

```
│    ├── logger.js
│    └── errorHandler.js
├── package.json
```

Step 1 — Install Dependencies

```
npm install express winston morgan @sentry/node
```

Step 2 — Logger Setup (Winston)

middleware/logger.js

```
const winston = require("winston");

// Create Winston logger
const logger = winston.createLogger({
  level: "info",
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: "error.log",
level: "error" }),
    new winston.transports.Console()
  ]
});

module.exports = logger;
```

Step 3 — Centralized Error Handler

middleware/errorHandler.js

```
const Sentry = require("@sentry/node");
const logger = require("./logger");

module.exports = (err, req, res, next) => {
  // Log to Winston
  logger.error(err.message);

  // Send error to Sentry
  Sentry.captureException(err);

  res.status(err.status || 500).json({
    error: err.message,
    status: err.status || 500
  });
};
```

Step 4 — Employee Routes

routes/employee.js

```
const express = require("express");
const router = express.Router();
const logger = require("../middleware/logger");

let employees = [];
let idCounter = 1;

// Get all employees
router.get("/", (req, res) => {
  res.json(employees);
});

// Add employee
router.post("/", (req, res, next) => {
  const { name, salary } = req.body;
  if (!name || !salary) {
    const err = new Error("Name & salary required");
    err.status = 400;
    return next(err);
  }
  const emp = { id: idCounter++, name, salary };
  employees.push(emp);
  logger.info(`Employee created: ${name}`);
  res.status(201).json(emp);
});

// Simulate crash for monitoring
router.get("/crash", (req, res) => {
  throw new Error("Unexpected server crash!");
});

module.exports = router;
```

Step 5 — App Setup

app.js

```
const express = require("express");
const morgan = require("morgan");
const Sentry = require("@sentry/node");
const employeeRoutes = require("./routes/employee");
const errorHandler = require("./middleware/errorHandler");
const logger = require("./middleware/logger");

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

// Initialize Sentry
Sentry.init({
  dsn: "YOUR_SENTRY_DSN_HERE", // replace with your DSN
  tracesSampleRate: 1.0
});

// Request handler for Sentry
app.use(Sentry.Handlers.requestHandler());

// HTTP request logging (Morgan)
app.use(morgan("combined"));

// Routes
app.use("/employees", employeeRoutes);

// Sentry error handler
app.use(Sentry.Handlers.errorHandler());

// Centralized error handler
app.use(errorHandler);

app.listen(3000, () => logger.info("Server running on port
3000"));
```

Explanation

1. **Morgan** logs **all incoming HTTP requests** to console.

2. **Winston** logs **errors and important events** (like employee creation) to console and `error.log`.
3. **Sentry** monitors uncaught errors (`/employees/crash`) in real-time.
4. Centralized error middleware handles **all errors consistently** and reports them to Sentry.

Testing the API

1. Add Employee

POST /employees

```
{
  "name": "Alice",
  "salary": 50000
}
```

Response:

```
{
  "id": 1,
  "name": "Alice",
  "salary": 50000
}
```

Console / error.log:

```
{"level":"info","message":"Employee created: 
Alice","timestamp":"..."}
```

2. Add Employee with Missing Salary

POST /employees

```
{
  "name": "Bob"
}
```

Response:

```
{
  "error": "Name & salary required",
  "status": 400
}
```

Console / error.log:

```
{"level":"error","message":"Name & salary
required","timestamp":"..."}
```

Sentry Dashboard: Captures the error.

3. Simulate Crash

GET /employees/crash
Response:

```
{
  "error": "Unexpected server crash!",
  "status": 500
}
```

Sentry Dashboard: Error with full stack trace appears for monitoring.

✅ Key Features Implemented

Feature	Implementation
HTTP Request Logging	Morgan
Error Logging	Winston (console + file)
Error Monitoring	Sentry
Centralized Error Handling	middleware/errorHandler.js
Simulated Crash	`/employees/crash` route
Info Logging	Employee creation logged via Winston

Example 1 — Logging Errors with Winston & Request Logging with Morgan

Scenario

We want to build an API to manage **employees**:

- Log **all incoming requests** using **Morgan**.
- Log **errors to a file** and **console** using **Winston**.

- Centralized error handling returns JSON errors.

Project Structure

```
employee-logging-api/
├── app.js
├── routes/
│     └── employee.js
├── middleware/
│     ├── errorHandler.js
│     └── logger.js
├── package.json
```

Step 1 — Install Dependencies

```
npm install express winston morgan
```

Step 2 — Logger Setup

middleware/logger.js

```
const winston = require("winston");

const logger = winston.createLogger({
  level: "info",
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: "error.log",
level: "error" }),
    new winston.transports.Console()
  ]
});

module.exports = logger;
```

Step 3 — Employee Routes (Simulated)

routes/employee.js

```
const express = require("express");
const router = express.Router();
const logger = require("../middleware/logger");

let employees = [];
let idCounter = 1;

router.get("/", (req, res) => {
  res.json(employees);
});

router.post("/", (req, res, next) => {
  const { name, salary } = req.body;
  if (!name || !salary) {
    const err = new Error("Name & salary required");
    err.status = 400;
    logger.error(err.message); // log error
    return next(err);
  }
  const emp = { id: idCounter++, name, salary };
  employees.push(emp);
  res.status(201).json(emp);
});

module.exports = router;
```

Step 4 — Centralized Error Handler

middleware/errorHandler.js

```
module.exports = (err, req, res, next) => {
  const status = err.status || 500;
  res.status(status).json({ error: err.message, status });
};
```

Step 5 — App Setup

app.js

```
const express = require("express");
const morgan = require("morgan");
const employeeRoutes = require("./routes/employee");
const errorHandler = require("./middleware/errorHandler");
const logger = require("./middleware/logger");

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

// Morgan logs HTTP requests to console
app.use(morgan("combined"));

// Routes
app.use("/employees", employeeRoutes);

// Centralized error handler
app.use(errorHandler);

app.listen(3000, () => logger.info("Server running on port
3000"));
```

Explanation

1. **Morgan** logs all HTTP requests in a standard format.
2. **Winston** logs errors to both console and `error.log` file.
3. Centralized error handler returns consistent JSON errors.

Sample Output

Console (Morgan + Winston)

```
::1 - - [22/Nov/2025:23:45:12 +0530] "POST /employees HTTP/
1.1" 400 -
{"level":"error","message":"Name & salary
required","timestamp":"2025-11-22T18:45:12.345Z"}
```

POST /employees (missing salary)

```
{
  "error": "Name & salary required",
  "status": 400
}
```

POST /employees (valid)

```
{
  "id": 1,
  "name": "Alice",
  "salary": 50000
}
```

Example 2 — Monitoring Errors with Sentry

Scenario

We want an API for **products**:

- Any uncaught exceptions or errors should be reported to **Sentry**.
- Centralized error handler returns JSON errors to clients.

Project Structure

```
product-monitor-api/
│── app.js
│── routes/
│    └── product.js
│── middleware/
│    └── errorHandler.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express @sentry/node
```

Step 2 — Product Routes

routes/product.js

```
const express = require("express");
const router = express.Router();

let products = [];
let idCounter = 1;

router.get("/", (req, res) => {
  res.json(products);
});

router.post("/", (req, res, next) => {
  const { name, price } = req.body;
  if (!name || !price) return next(new Error("Name & price
required"));
  const product = { id: idCounter++, name, price };
  products.push(product);
  res.status(201).json(product);
});

// Simulate uncaught error
router.get("/crash", (req, res, next) => {
  throw new Error("Server crashed unexpectedly");
});

module.exports = router;
```

Step 3 — Centralized Error Handler

middleware/errorHandler.js

```
const Sentry = require("@sentry/node");

module.exports = (err, req, res, next) => {
  Sentry.captureException(err); // Send error to Sentry
  res.status(err.status || 500).json({ error: err.message });
};
```

Step 4 — App Setup with Sentry

app.js

```
const express = require("express");
const Sentry = require("@sentry/node");
const productRoutes = require("./routes/product");
const errorHandler = require("./middleware/errorHandler");

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

// Initialize Sentry
```

```
Sentry.init({
  dsn: "YOUR_SENTRY_DSN_HERE", // replace with your Sentry
DSN
  tracesSampleRate: 1.0
});

// Request Handler
app.use(Sentry.Handlers.requestHandler());

app.use("/products", productRoutes);

// Error Handler
app.use(Sentry.Handlers.errorHandler());
app.use(errorHandler);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Explanation

1. **Sentry.init()** initializes monitoring with DSN.
2. `Sentry.captureException(err)` sends any error to Sentry dashboard.
3. Centralized error handler still returns **JSON response** to clients.
4. `/products/crash` simulates an uncaught server error for monitoring.

Sample Output

POST /products (missing price)

```
{
  "error": "Name & price required"
}
```

GET /products/crash

- **JSON Response:**

```
{
  "error": "Server crashed unexpectedly"
}
```

- **Sentry Dashboard:** Error appears with stack trace and timestamp.

✅ **Key Concepts**

Feature	Example 1	Example 2
Logging requests	Morgan	—
Logging errors	Winston (file & console)	—
Monitoring	—	Sentry
Centralized error handling	middleware/ errorHandler.js	middleware/errorHandler.js
Simulated error	Validation error	Uncaught exception / `crash`

Example 1 — Express API with Graceful Shutdown on SIGINT/SIGTERM

Scenario

We have a simple **employee API**.

- When the server receives a **shutdown signal** (Ctrl+C or Docker stop), it should:
  - Stop accepting new requests
  - Complete pending requests
  - Close DB connections (simulated)
  - Exit gracefully

Project Structure

```
graceful-shutdown-api/
│── app.js
│── routes/
│     └── employee.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express
```

Step 2 — Employee Routes

routes/employee.js

```
const express = require("express");
const router = express.Router();

let employees = [];
let idCounter = 1;

// Get employees
router.get("/", (req, res) => {
  res.json(employees);
});

// Add employee
router.post("/", (req, res) => {
  const { name, salary } = req.body;
  const emp = { id: idCounter++, name, salary };
  employees.push(emp);
  res.status(201).json(emp);
});

module.exports = router;
```

Step 3 — App Setup with Graceful Shutdown

app.js

```
const express = require("express");
const employeeRoutes = require("./routes/employee");

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

app.use("/employees", employeeRoutes);

const server = app.listen(3000, () => console.log("Server
running on port 3000"));

// Graceful shutdown function
const shutdown = () => {
  console.log("\nGraceful shutdown initiated...");
  server.close(() => {
    console.log("All requests finished. Server closed.");
    // Simulate DB disconnect
    console.log("DB connections closed.");
    process.exit(0);
  });

  // Force shutdown if requests do not finish in 10s
  setTimeout(() => {
    console.error("Forcing shutdown...");
    process.exit(1);
  }, 10000);
};

// Handle termination signals
process.on("SIGINT", shutdown);  // Ctrl+C
process.on("SIGTERM", shutdown); // Docker stop / kill
```

Explanation

1. `server.close()` stops new requests but allows **existing requests** to finish.
2. `setTimeout` ensures the server **forces exit** if pending requests take too long.
3. `SIGINT` and `SIGTERM` are standard shutdown signals.
4. Simulated DB disconnect is printed in logs.

Sample Output

Ctrl+C during request

```
Server running on port 3000
Graceful shutdown initiated...
All requests finished. Server closed.
DB connections closed.
```

Example 2 — Express API with Recovery on Uncaught Exceptions and Promises

Scenario

- We want the server to **catch uncaught exceptions and unhandled promise rejections**.
- Log the error, attempt **cleanup**, and **exit gracefully**.
- Demonstrates **recovery strategy** for unexpected crashes.

Project Structure

```
recovery-api/
│── app.js
│── routes/
│    └── product.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express
```

Step 2 — Product Routes (Simulate Error)

routes/product.js

```
const express = require("express");
const router = express.Router();

let products = [];
let idCounter = 1;

// Add product
router.post("/", (req, res) => {
  const { name, price } = req.body;
  if (!name || !price) throw new Error("Name & price
required"); // Simulate crash
  const product = { id: idCounter++, name, price };
  products.push(product);
  res.status(201).json(product);
});

// Simulate async error
```

```
router.get("/async-error", async (req, res) => {
  await Promise.reject(new Error("Async failure!"));
});

module.exports = router;
```

Step 3 — App Setup with Recovery

app.js

```
const express = require("express");
const productRoutes = require("./routes/product");

const app = express();
app.use(express.json());
app.use("/products", productRoutes);

const server = app.listen(3000, () => console.log("Server
running on port 3000"));

// Graceful shutdown
const shutdown = (err) => {
  if (err) console.error("Error:", err);
  console.log("Cleaning up resources...");
  server.close(() => {
    console.log("Server closed.");
    process.exit(err ? 1 : 0);
  });
  setTimeout(() => process.exit(1), 5000);
};

// Catch uncaught exceptions
process.on("uncaughtException", (err) => {
  console.error("Uncaught Exception:", err);
  shutdown(err);
});

// Catch unhandled promise rejections
process.on("unhandledRejection", (reason, promise) => {
  console.error("Unhandled Rejection:", reason);
  shutdown(reason);
});
```

Explanation

1. `uncaughtException` handles **synchronous runtime errors**.
2. `unhandledRejection` handles **promise rejections not caught**.
3. Both trigger **graceful shutdown** and optional **cleanup**.
4. `server.close()` ensures existing requests finish before exit.

Sample Output

POST /products (missing price)

```
Uncaught Exception: Error: Name & price required
Cleaning up resources...
Server closed.
```

GET /products/async-error

```
Unhandled Rejection: Error: Async failure!
Cleaning up resources...
Server closed.
```

✅ **Key Concepts**

Feature	Example 1	Example 2
Graceful shutdown	SIGINT/SIGTERM	Uncaught exception / unhandled promise
Pending requests	Completed before shutdown	Completed before shutdown
Cleanup	Simulated DB disconnect	Simulated cleanup logs
Forced exit	Timeout (10s)	Timeout (5s)
Recovery	—	Handles unexpected crashes

H3 Framework Introduction and Comparison with Express.js

1. What is H3?

- **H3** is a **minimalistic Node.js HTTP framework** designed for building web applications and APIs.

- Developed primarily for use with **Nuxt 3** and **server-side rendering**, but it can also be used standalone.

- Focuses on **simplicity, performance, and native TypeScript support**.

- Inspired by other frameworks like Express.js but with a **lightweight core** and modern features.

Key Features of H3

1. **Lightweight and Fast**: Minimal middleware overhead, optimized for performance.

2. **TypeScript Friendly**: Native TypeScript types and definitions.

3. **Composable Handlers**: Uses simple **request handlers** instead of full middleware stacks.

4. **Promise-based**: Supports **async/await** natively.

5. **Integrated Utilities**: Includes helpers for headers, cookies, query parsing, and JSON responses.

6. **Error Handling**: Built-in support for structured error handling.

2. Basic Example of H3

Standalone H3 server example:

```
import { createServer, send, readBody } from "h3";

const server = createServer(async (req, res) => {
  if (req.url === "/") {
    send(res, 200, { message: "Hello from H3!" });
  } else if (req.url === "/echo" && req.method === "POST") {
    const body = await readBody(req);
    send(res, 200, { echo: body });
  } else {
    send(res, 404, { error: "Not Found" });
  }
});

server.listen(3000, () => {
  console.log("H3 server running on port 3000");
});
```

Explanation:

- `createServer` is similar to Node’s native HTTP server.

- `send` helps to respond with JSON or text easily.

- `readBody` parses the request body (supports JSON, URL-encoded).

3. Express.js Overview

- **Express.js** is a **widely-used Node.js web framework**.
- Provides a **robust set of features** for building APIs and web apps.
- Features:
    1. Routing system with **middleware support**.
    2. Support for JSON, URL-encoded bodies, cookies, sessions.
    3. Large ecosystem of plugins and middleware.
    4. Works with both REST APIs and server-side rendered apps.

Basic Express.js example:

```
const express = require("express");
const app = express();

app.use(express.json());

app.get("/", (req, res) => {
  res.json({ message: "Hello from Express!" });
});

app.post("/echo", (req, res) => {
  res.json({ echo: req.body });
});

app.listen(3000, () => console.log("Express server running on
port 3000"));
```

4. H3 vs Express.js — Comparison

Feature	H3	Express.js
Size & Performance	Lightweight, minimal overhead	Slightly heavier, middleware-based
Middleware	Composable handlers, minimal middleware	Full middleware stack
TypeScript	Native, strong typing	Requires external types (@types/express)
Ease of Use	Simple, functional style	Mature, feature-rich

Routing	Simple URL/path handling	Rich routing API, params, query, nested
Ecosystem	Small, mostly for Nuxt 3	Huge, many plugins and community
Error Handling	Built-in, structured	Customizable via middleware
Use Case	Lightweight APIs, Nuxt 3 backend	REST APIs, MVC apps, complex middleware stacks

5. When to Use H3 vs Express.js

Use H3 When:

- You want a **lightweight API** server.
- You are building **Nuxt 3 server endpoints**.
- Native **TypeScript support** is important.
- You prefer **minimalistic, functional approach**.

Use Express.js When:

- You need **full-featured web server** with middleware ecosystem.
- Building **complex REST APIs** with authentication, sessions, or file handling.
- Working on a **legacy Node.js project** or need **community support**.