Employee Management Platform — Case Study Overview

This case study demonstrates a **full-stack Node.js platform** for managing employees. It focuses on modern **Node.js backend concepts**, **RESTful APIs**, **authentication & authorization**, **security**, **file handling**, **database integration**, and **deployment to Azure**.

The goal is to create a **scalable, secure, and testable backend**.

1. Features Covered

Node.js Core Concepts

- **Event Loop**: Understand synchronous vs asynchronous execution.
- **Non-blocking I/O**: Using asynchronous file, network, and database operations.
- **Streams**: For file uploads and downloads using `fs` streams.
- **HTTP**: Raw server handling, request and response objects.
- **Utilities**: Using `path`, `os`, and `crypto`.
- **Modules**: CommonJS (`require`) and custom modules for separation of concerns.

Express.js API

- **Routing**: RESTful endpoints for employees and auth.
- **Middleware**:
  - Global middlewares (CORS, logging, rate limiting).
  - Custom error handling.
  - Authentication and authorization checks.
- **Advanced Routing**:
  - Route parameters (`/employees/:id`)
  - Query strings (`/employees?department=HR`)
  - Nested routes (`/admin/employees`)
- **Versioning & Documentation**: API versioning (`/api/v1/`) and Swagger/OpenAPI docs.

MongoDB + Mongoose

- **Schema Design**: Employee and User schemas with validation.

- **Population**: Linking employees with managers, departments, or users.
- **Aggregation**: Advanced queries like total salaries, department-wise summaries.
- **Validation & Sanitization**: Ensure only valid data enters the DB.

Authentication & Authorization

- **JWT Authentication**:
  - Generate access and refresh tokens.
  - Validate tokens for protected routes.
- **RBAC (Role-Based Access Control)**:
  - Admin vs User roles.
  - Only admins can create, update, delete employees.
- **Secure Password Storage**:
  - Using bcrypt for hashing.
- **Security Features**:
  - Rate limiting for brute-force protection.
  - Basic CSRF/XSS header protections.

Error Handling

- Centralized error handler.
- Custom error classes with error codes.
- Standardized error response format.

Logging & Monitoring

- **Logging**:
  - HTTP request logging with `morgan`.
  - Application logs with `winston`.
- **Monitoring**:
  - Configure Azure Application Insights for telemetry and metrics.

Caching

- Simple in-memory cache.
- Redis-ready hook for future scaling.

Streams

- File upload/download using `multer` and `fs` streams.
- Large files handled efficiently without blocking the event loop.

Profiling & Benchmarking

- Use `console.time()` for simple profiling.
- Use Node.js debugger: `node --inspect` and Chrome DevTools.

Testing

- **Unit Tests**: Using `Jest` for utilities and model validation.
- **Integration Tests**: Using `Supertest` to test APIs end-to-end.
- Test environment isolated from production DB.

Deployment

- **Azure App Service** (No Docker):
  - Git deployment or ZIP deployment.
  - Configure environment variables in App Service.
  - Enable App Insights for monitoring.
  - Scaling options for high availability.

2. Recommended Project Structure

```
employee-platform/
├─ app.js                      # Main Express app
├─ package.json
├─ .env                        # Environment variables
├─ uploads/                       # Uploaded files
├─ temp/                          # Temp storage for uploads
├─ src/
│  ├─ config/
│  │   └─ db.js                  # MongoDB connection
│  ├─ controllers/
│  │   ├─ auth.controller.js
│  │   └─ employees.controller.js
│  ├─ middleware/
│  │   ├─ auth.js
│  │   ├─ errorHandler.js
│  │   ├─ rateLimiter.js
│  │   └─ logger.js
│  ├─ models/
│  │   ├─ user.model.js
│  │   └─ employee.model.js
│  ├─ routes/
│  │   ├─ auth.routes.js
│  │   └─ employees.routes.js
│  └─ utils/
│      └─ streamUtils.js
├─ tests/
│  ├─ unit/
│  │   └─ utils.test.js
│  └─ integration/
│      └─ employee.test.js
│
```

3. Execution & Testing Flow

Step 1: Setup

1. Clone the repository.
2. Install dependencies:

   ```
   npm install
   ```

Setup `.env` with required variables:

```
PORT=3000

MONGO_URI=mongodb://localhost:27017/employee_db
MONGO_URI_TEST=mongodb://localhost:27017/employee_test
JWT_ACCESS_SECRET=access123
JWT_REFRESH_SECRET=refresh123
```

Step 2: Start Server

```
npm run start
```

- Server listens on port 3000 (or `.env` PORT).
- DB connection logs "MongoDB connected".

Step 3: Execute APIs in Postman

1. **Register User**

   POST `http://localhost:3000/api/v1/auth/register`

   Body (JSON):

   ```
   {

     "username": "admin",
     "email": "admin@example.com",
     "password": "pass123",
     "role": "admin"
   }
   ```

   ```
   Response:
   ```

   ```
   {
     "message": "User registered successfully"
   }
   ```

2. **Login User**

   POST `http://localhost:3000/api/v1/auth/login`

   Body (JSON):

   ```
   {

     "username": "admin",
     "password": "pass123"
   }
   ```

   ```
   Response:
   ```

```
{
  "accessToken": "jwt-access-token",
  "refreshToken": "jwt-refresh-token"
}
```

3. **Create Employee**

   POST `http://localhost:3000/api/v1/employees`

   Headers:

   ```
   Authorization: Bearer <accessToken>
   ```

   ```
   Body (JSON):
   ```

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

   ```
   Response:
   {
     "_id": "employee-id",
     "name": "Alice",
     "department": "HR",
     "salary": 50000
   }
   ```

4. **List Employees**

   GET `http://localhost:3000/api/v1/employees`

   Headers:

   ```
   Authorization: Bearer <accessToken>
   ```

   Response:

   ```
   [

     {
       "_id": "employee-id",
       "name": "Alice",
   ```

```
      "department": "HR",
      "salary": 50000
    }
  ]
```

5. **File Upload**

   POST `http://localhost:3000/api/v1/upload`

   Body: `form-data` → key: `file`, type: `File`

   Response:

   ```
   {

     "message": "uploaded",
     "path": "/uploads/filename.txt"
   }
   ```

Step 4: Run Tests

```
npm run test
```

- Unit tests for utilities.
- Integration tests for API endpoints.
- Test database is dropped automatically after tests.

Step 5: Deployment to Azure App Service

1. Create an **App Service** in Azure.
2. Set **environment variables** (`PORT`, `MONGO_URI`, `JWT_*`) in App Service settings.
3. Deploy using **Git deployment**:

   ```
   git push azure main
   ```

4.
5. Enable **App Insights** for logging and monitoring.
6. Scale using Azure App Service scaling options.

4. Sample Output

- **Server log**:

```
MongoDB connected
Server listening on 3000
```

- **Postman JSON responses** as shown in Step 3.

5. Notes

- Always use `.env` for secrets (JWT keys, DB URI).
- Ensure `uploads/` and `temp/` directories exist for file uploads.
- Use `Authorization` header for protected routes in Postman.
- For tests, use a separate test DB to avoid interfering with production/dev DB.

Employee Management Platform – Node.js Core Concepts Integration

Project structure (simplified for core concepts demo):

```
employee-platform/
│
├─ src/
│   ├─ utils/
│   │   └─ fileHandler.js        # Streams for file uploads/
downloads
│   ├─ models/
│   │   └─ employee.model.js      # Mongoose model
│   ├─ routes/
│   │   └─ employees.routes.js    # Express routes
│   └─ controllers/
│       └─ employee.controller.js # Controller logic
│
├─ app.js                         # HTTP server + routing
├─ math.js                         # Custom module (CommonJS
example)
├─ sample.txt                      # Sample file for streaming
demo
└─ package.json
```

1. Event Loop + Async File I/O + Streams

File: `src/utils/fileHandler.js`

```
const fs = require("fs");
const path = require("path");

/**
 * Save uploaded file using streams (non-blocking I/O)
 */
function saveFile(tempPath, filename) {
  return new Promise((resolve, reject) => {
    const target = path.join(__dirname, "../../uploads",
filename);
    const readStream = fs.createReadStream(tempPath);
    const writeStream = fs.createWriteStream(target);

    readStream.pipe(writeStream);

    writeStream.on("finish", () => resolve(target));
    writeStream.on("error", reject);
  });
}

module.exports = { saveFile };
```

Explanation:

- Asynchronous streams ensure the server **does not block** while saving large files.
- `pipe()` transfers data from temp upload to final storage.

2. HTTP Server & Express Routing

File: `app.js`

```
require("dotenv").config();
const express = require("express");
const cors = require("cors");
const multer = require("multer");
const { saveFile } = require("./src/utils/fileHandler");
const employeeRoutes = require("./src/routes/
employees.routes");

const app = express();
```

```
app.use(cors());
app.use(express.json());

// Upload endpoint
const upload = multer({ dest: "temp/" });
app.post("/api/v1/upload", upload.single("file"), async (req, 
res, next) => {
  try {
    const filePath = await saveFile(req.file.path, 
req.file.originalname);
    res.json({ message: "File uploaded", path: filePath });
  } catch (err) {
    next(err);
  }
});

// Employee routes
app.use("/api/v1/employees", employeeRoutes);

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port $
{PORT}`));
```
Explanation:

- Express handles **HTTP requests**.
- `/upload` uses streams from `fileHandler.js` to handle file uploads efficiently.

3. Using Utilities (`path`, `os`, `crypto`)

File: `src/controllers/employee.controller.js`

```
const path = require("path");
const os = require("os");
const crypto = require("crypto");

/**
 * Example: create employee ID and log system info
 */
function createEmployeeID(name) {
  const hash = crypto.createHash("sha256").update(name + 
Date.now()).digest("hex");
  console.log("Server OS:", os.platform(), "Free Memory:", 
os.freemem());
```

```
  return hash.slice(0, 8); // short unique ID
}

module.exports = { createEmployeeID };
```

Explanation:

- Uses `crypto` for unique employee ID.
- `os` to log server info for monitoring.
- `path` can be used for file paths in uploads.

4. Custom Module Example

File: `math.js`

```
// Utility module for salary calculations
function calculateBonus(salary, percentage) {
  return salary + salary * (percentage / 100);
}

module.exports = { calculateBonus };
```

File usage in controller: `src/controllers/employee.controller.js`

```
const { calculateBonus } = require("../../math");

function addBonus(employee) {
  employee.salary = calculateBonus(employee.salary, 10);
  return employee;
}
```

Explanation:

- `math.js` is a **CommonJS module**, reusable across platform.
- Helps separate **business logic** (bonus calculation) from controllers.

5. Employee Model Example (Mongoose)

File: `src/models/employee.model.js`

```
const mongoose = require("mongoose");
const employeeSchema = new mongoose.Schema({
  name: { type: String, required: true },
  department: { type: String },
  salary: { type: Number, required: true },
  employeeID: { type: String, unique: true }
});

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

6. Employee Routes Example

File: `src/routes/employees.routes.js`

```
const express = require("express");
const { createEmployeeID } = require("../controllers/
employee.controller");
const Employee = require("../models/employee.model");

const router = express.Router();

// Create Employee
router.post("/", async (req, res) => {
  try {
    const empID = createEmployeeID(req.body.name);
    const employee = await Employee.create({ ...req.body,
employeeID: empID });
    res.status(201).json(employee);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// List Employees
router.get("/", async (req, res) => {
  const employees = await Employee.find();
  res.json(employees);
});

module.exports = router;
```

How This Connects Core Concepts to the Platform

Core Concept	Platform Usage

Event Loop + Async I/	File uploads handled asynchronously without blocking requests.
Streams	Efficient reading/writing of uploaded files (`fs.createReadStream`/ `createWriteStream`).
HTTP Server	Express routes handle CRUD APIs for employees.
Utilities (`path`, `os`, `crypto`)	Path management for uploads, server info logging, unique employee ID generation.
Modules	`math.js` module for business logic (bonus calculation), reusable across
Non-blocking DB	Mongoose async methods (`create`, `find`) follow event loop model.

Testing the Output via Postman

1. **Start Server:**

```
node app.js
```

2. **Upload File:**

- Method: POST
- URL: `http://localhost:3000/api/v1/upload`
- Body: form-data, Key: `file`, Type: File, choose any `.txt` or `.csv` file.
- Expected Response:

```
{
  "message": "File uploaded",
  "path": "/absolute/path/to/uploads/sample.txt"
}
```

3. **Create Employee:**

- Method: POST
- URL: `http://localhost:3000/api/v1/employees`
- Body (JSON):

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

- Expected Response:

```
{
  "_id": "6432c7ae0f1b3d3c4a2a1234",
  "name": "Alice",
  "department": "HR",
  "salary": 50000,
  "employeeID": "a1b2c3d4"
}
```

4. **List Employees:**

- Method: GET
- URL: `http://localhost:3000/api/v1/employees`
- Expected Response:

```
[
  {
    "_id": "6432c7ae0f1b3d3c4a2a1234",
    "name": "Alice",
    "department": "HR",
    "salary": 50000,
    "employeeID": "a1b2c3d4"
  }
]
```

Project Structure (Expanded for Express.js concepts)

```
employee-platform/
│
├─ src/
│  ├─ controllers/
│  │  └─ employee.controller.js
│  ├─ middleware/
│  │  ├─ auth.js
│  │  ├─ errorHandler.js
│  │  ├─ logger.js
│  │  └─ rateLimiter.js
│  ├─ models/
│  │  └─ employee.model.js
│  ├─ routes/
│  │  ├─ auth.routes.js
│  │  └─ employees.routes.js
│  └─ utils/
│     └─ fileHandler.js
│
```

```
│
├─ app.js
├─ swagger.json
└─ package.json
```

1. Global Middlewares

File: `app.js`

```
require("dotenv").config();
const express = require("express");
const cors = require("cors");
const morganMiddleware = require("./src/middleware/logger");
const rateLimiter = require("./src/middleware/rateLimiter");
const { errorHandler } = require("./src/middleware/
errorHandler");

const authRoutes = require("./src/routes/auth.routes");
const employeeRoutes = require("./src/routes/
employees.routes");

const app = express();

// Global Middlewares
app.use(cors());
app.use(express.json());
app.use(morganMiddleware

// Routes (API versioning)
app.use("/api/v1/auth", authRoutes);
app.use("/api/v1/employees", employeeRoutes);

// Error handler (must be last)
app.use(errorHandler);

module.exports = app;
Explanation:
cors(): Enables cross-origin requests.
express.json(): Parses JSON request bodies.
morganMiddleware: Logs HTTP requests.
rateLimiter: Limits request frequency to prevent abuse.
errorHandler: Centralized error handler catches errors from
routes/middleware.
API versioning with /api/v1/ ensures future-proofing.
```

2. Employee Routes

File: src/routes/employees.routes.js

```
const express = require("express");
const router = express.Router();
const employeeController = require("../controllers/
employee.controller");
const { authenticate, authorize } = require("../middleware/
auth");

// Advanced Routing Examples:

// Get all employees (supports query: department)
router.get("/", authenticate,
employeeController.listEmployees);

// Get employee by ID (route parameter)
router.get("/:id", authenticate,
employeeController.getEmployeeById);

// Admin nested routes
router.post("/", authenticate, authorize("admin"),
employeeController.createEmployee);
router.put("/:id", authenticate, authorize("admin"),
employeeController.updateEmployee);
router.delete("/:id", authenticate, authorize("admin"),
employeeController.deleteEmployee);

module.exports = router;
```

Explanation:
authenticate: JWT middleware verifies token.
authorize("admin"): Role-based access for admin-only routes.
Route parameters: /employees/:id allows fetching, updating, or deleting specific employee.
Query strings: listEmployees can filter by ?department=HR.
Nested routes: Admin routes ensure restricted access.

3. Employee Controller

File: src/controllers/employee.controller.js

```
const Employee = require("../models/employee.model");

exports.listEmployees = async (req, res, next) => {
  try {
    const filter = {};
    if (req.query.department) filter.department =
req.query.department;
    const employees = await Employee.find(filter);
    res.json(employees);
  } catch (err) {
    next(err);
  }
};

exports.getEmployeeById = async (req, res, next) => {
  try {
    const employee = await Employee.findById(req.params.id);
    if (!employee) return res.status(404).json({ message:
"Employee not found" });
    res.json(employee);
  } catch (err) {
    next(err);
  }
};

exports.createEmployee = async (req, res, next) => {
  try {
    const employee = await Employee.create(req.body);
    res.status(201).json(employee);
  } catch (err) {
    next(err);
  }
};

// Update & Delete similar
```

Explanation:
Handles CRUD operations for employees.
Supports query filtering (?department=HR) and route params (:id).
Errors are passed to centralized errorHandler.

4. Auth Middleware (Example)

File: src/middleware/auth.js

```
const jwt = require("jsonwebtoken");

exports.authenticate = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  if (!authHeader) return res.status(401).json({ message:
"Unauthorized" });

  const token = authHeader.split(" ")[1];
  try {
    req.user = jwt.verify(token,
process.env.JWT_ACCESS_SECRET);
```

```
    next();
  } catch (err) {
    return res.status(403).json({ message: "Forbidden" });
  }
};

exports.authorize = (role) => (req, res, next) => {
  if (req.user.role !== role) return
res.status(403).json({ message: "Forbidden" });
  next();
};
```

Explanation:
authenticate: Verifies JWT token from Authorization header.
authorize: Checks user role (admin/user).
5. Swagger Documentation (Optional)
File: swagger.json (partial)

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Employee Management API",
    "version": "1.0.0"
  },
  "paths": {
    "/api/v1/employees": {
      "get": {
        "summary": "List employees",
        "parameters": [
          { "name": "department", "in": "query", "schema":
{ "type": "string" } }
        ],
        "responses": { "200": { "description": "Success" } }
      }
    }
  }
}
```

Explanation:
Allows auto-generation of API docs.
Supports route query params, parameters, and response definitions.
6. Testing Routes in Postman
Base URL: http://localhost:3000/api/v1/
Register + Login: POST /auth/register and POST /auth/login
Save JWT access token.
Get All Employees: GET /employees?department=HR
Include Authorization: Bearer <token> header.

```
Get Employee by ID: GET /employees/<id>
Admin Actions: POST /employees, PUT /employees/:id, DELETE /
employees/:id
Only accessible with admin token.
```

. Project Structure (Relevant to MongoDB + Auth + Security)

```
employee-platform/
│
├─ src/
│   ├─ config/
│   │   └─ db.js                    # MongoDB connection
│   ├─ models/
│   │   ├─ user.model.js            # User schema
│   │   ├─ employee.model.js        # Employee schema
│   ├─ controllers/
│   │   ├─ auth.controller.js       # JWT login/register
│   │   └─ employee.controller.js
│   ├─ routes/
│   │   ├─ auth.routes.js
│   │   └─ employees.routes.js
│   ├─ middleware/
│   │   ├─ auth.js                  # JWT authentication & RBAC
│   │   ├─ errorHandler.js          # centralized error handling
│   │   └─ rateLimiter.js           # brute-force protection
│   └─ utils/
│       └─ passwordUtils.js         # bcrypt hashing helper
│
├─ app.js                           # Express server
├─ .env
└─ package.json
```

2. MongoDB Connection

File: `src/config/db.js`

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

const MONGO_URI = process.env.MONGO_URI || "mongodb://
localhost:27017/employee_db";
```

```
async function connectDb() {
  try {
    await mongoose.connect(MONGO_URI);
    console.log("MongoDB connected");
  } catch (err) {
    console.error("MongoDB connection error:", err);
    process.exit(1);
  }
}

module.exports = { connectDb };
```
Explanation:

- Connects to MongoDB.
- Handles connection errors.

3. User Schema (with Validation & Secure Password Storage)

File: `src/models/user.model.js`

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

const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true,
trim: true },
  email: { type: String, required: true, unique: true, trim:
true },
  password: { type: String, required: true },
  role: { type: String, enum: ["admin", "user"], default:
"user" },
}, { timestamps: true });

// Hash password before saving
userSchema.pre("save", async function(next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});
```

```
// Compare password
userSchema.methods.comparePassword = async
function(candidatePassword) {
  return bcrypt.compare(candidatePassword, this.password);
};

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

- Creating user `admin` stores hashed password.
- Password is never stored in plain text.

4. Employee Schema (Validation + Population)

File: `src/models/employee.model.js`

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

const employeeSchema = new mongoose.Schema({
  name: { type: String, required: true },
  department: { type: String, required: true },
  salary: { type: Number, required: true, min: 0 },
  manager: { type: mongoose.Schema.Types.ObjectId, ref:
"User" },
}, { timestamps: true });

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

- `manager` field links to a `User` model (population).
- Validates `salary` is positive and required fields.

5. Auth Controller (JWT Tokens)

File: `src/controllers/auth.controller.js`

```
const User = require("../models/user.model");
const jwt = require("jsonwebtoken");

exports.register = async (req, res, next) => {
  try {
    const user = await User.create(req.body);
    res.status(201).json({ message: "User registered",
userId: user._id });
  } catch (err) { next(err); }
};

exports.login = async (req, res, next) => {
  try {
    const { username, password } = req.body;
    const user = await User.findOne({ username });
    if (!user) return res.status(401).json({ message:
"Invalid credentials" });

    const valid = await user.comparePassword(password);
    if (!valid) return res.status(401).json({ message:
"Invalid credentials" });

    const accessToken = jwt.sign({ id: user._id, role:
user.role }, process.env.JWT_ACCESS_SECRET, { expiresIn:
process.env.ACCESS_TOKEN_EXPIRES });
    const refreshToken = jwt.sign({ id: user._id, role:
user.role }, process.env.JWT_REFRESH_SECRET, { expiresIn:
process.env.REFRESH_TOKEN_EXPIRES });

    res.json({ accessToken, refreshToken });
  } catch (err) { next(err); }
};
```

Sample Output (Postman Login):

```
{
  "accessToken": "eyJhbGciOiJIUzI1...",
  "refreshToken": "eyJhbGciOiJIUzI1..."
}
```

6. Auth Middleware (JWT + RBAC)

File: `src/middleware/auth.js`

```
const jwt = require("jsonwebtoken");

exports.authenticate = (req, res, next) => {
  const token = req.headers["authorization"]?.split(" ")[1];
  if (!token) return res.status(401).json({ message: 
"Unauthorized" });

  try {
    req.user = jwt.verify(token, 
process.env.JWT_ACCESS_SECRET);
    next();
  } catch (err) {
    return res.status(403).json({ message: "Forbidden" });
  }
};

exports.authorize = (role) => (req, res, next) => {
  if (req.user.role !== role) return 
res.status(403).json({ message: "Forbidden" });
  next();
};
```

Explanation:

- `authenticate`: Checks token validity.
- `authorize`: Restricts admin-only routes.

7. Rate Limiter (Brute-force Protection)

File: `src/middleware/rateLimiter.js`

```
const rateLimit = require("express-rate-limit");

const limiter = rateLimit({
  windowMs: process.env.RATE_LIMIT_WINDOW_MS || 10 * 60 * 
1000, // 10 minutes
  max: process.env.RATE_LIMIT_MAX || 100,
  message: "Too many requests from this IP, try again later."
});

module.exports = limiter;
```

8. Error Handler

File: `src/middleware/errorHandler.js`

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

9. Sample Postman Requests

1. **Register User**
   - `POST /api/v1/auth/register`
   - Body: `{ "username": "admin", "email": "a@a.com", "password": "pass123", "role": "admin" }`
2. **Login**
   - `POST /api/v1/auth/login`
   - Response contains `accessToken` and `refreshToken`.
3. **Create Employee (Admin only)**
   - `POST /api/v1/employees`
   - Headers: `Authorization: Bearer <accessToken>`
   - Body: `{ "name": "Alice", "department": "HR", "salary": 50000 }`
4. **List Employees**
   - `GET /api/v1/employees?department=HR`
   - Headers: `Authorization: Bearer <accessToken>`
5. **Population Example**
   - `GET /api/v1/employees/:id`
   - `manager` field can populate user info using `Employee.findById(id).populate("manager")`.

. Project Structure (Extended)

```
employee-platform/
│
├─ src/
│  ├─ config/
│  │  └─ db.js
│  ├─ controllers/
│  │  ├─ auth.controller.js
│  │  └─ employee.controller.js
│  ├─ middleware/
│  │  ├─ auth.js
│  │  ├─ errorHandler.js       # centralized error handling
│  │  ├─ logger.js             # winston logger
│  │  └─ rateLimiter.js
│  ├─ models/
│  │  ├─ user.model.js
│  │  └─ employee.model.js
│  ├─ routes/
│  │  ├─ auth.routes.js
│  │  └─ employees.routes.js
│  ├─ utils/
│  │  ├─ cache.js              # in-memory + Redis-ready
│  │  └─ streamUtils.js        # file streams
│  └─ tests/
│     ├─ unit/
│     │  └─ utils.test.js
│     └─ integration/
│        └─ employee.test.js
│
├─ app.js
├─ .env
└─ package.json
```

2. Centralized Error Handling

File: `src/middleware/errorHandler.js`

```
class AppError extends Error {
  constructor(message, statusCode = 500, code =
"ERR_INTERNAL") {
    super(message);
    this.statusCode = statusCode;
    this.code = code;
  }
}
```

```
const errorHandler = (err, req, res, next) => {
  console.error(err); // for server logs
  res.status(err.statusCode || 500).json({
    code: err.code || "ERR_INTERNAL",
    message: err.message || "Internal Server Error"
  });
};

module.exports = { AppError, errorHandler };
```
Explanation:

- `AppError` class to standardize errors.
- Middleware returns **JSON with code + message**.

Sample Output (Postman):

```
{
  "code": "ERR_INTERNAL",
  "message": "Invalid employee ID"
}
```

3. Logging & Monitoring

a) Winston Logger

File: `src/middleware/logger.js`

```
const { createLogger, transports, format } =
require("winston");

const logger = createLogger({
  level: "info",
  format: format.combine(
    format.timestamp(),
    format.printf(({ timestamp, level, message }) => `$
{timestamp} [${level}]: ${message}`)
  ),
  transports: [
    new transports.Console(),
    new transports.File({ filename: "logs/app.log" })
  ]
});
```

```
module.exports = logger;
```

Usage Example in App:

```
const logger = require("./middleware/logger");
logger.info("Server started");
```

b) HTTP Request Logging with Morgan

File: `src/middleware/morgan.js`

```
const morgan = require("morgan");
const logger = require("./logger");

const morganMiddleware = morgan("combined", {
  stream: {
    write: (message) => logger.info(message.trim())
  }
});

module.exports = morganMiddleware;
```

c) Monitoring (Azure App Insights)

```
// Optional: In production
// const appInsights = require("applicationinsights");
//
appInsights.setup(process.env.APPINSIGHTS_INSTRUMENTATIONKEY)
.start();
```

4. Caching

File: `src/utils/cache.js`

```
const memoryCache = new Map();

// Simple in-memory cache
function setCache(key, value, ttlMs = 60000) {
  memoryCache.set(key, { value, expire: Date.now() +
ttlMs });
}
```

```
function getCache(key) {
  const cached = memoryCache.get(key);
  if (!cached) return null;
  if (cached.expire < Date.now()) {
    memoryCache.delete(key);
    return null;
  }
  return cached.value;
}

// Redis-ready placeholder
// const redis = require("redis");
// const client = redis.createClient();
module.exports = { setCache, getCache };
```

Sample Usage:

```
const { setCache, getCache } = require("../utils/cache");
setCache("employee_1", { name: "Alice" }, 10000);
console.log(getCache("employee_1")); // { name: "Alice" }
```

5. Streams (File Upload / Download)

File: `src/utils/streamUtils.js`

```
const fs = require("fs");
const path = require("path");

function streamToFile(tempPath, filename) {
  const target = path.join(__dirname, "../../uploads",
filename);
  return new Promise((resolve, reject) => {
    const read = fs.createReadStream(tempPath);
    const write = fs.createWriteStream(target);
    read.pipe(write);
    write.on("finish", () => resolve(target));
    write.on("error", reject);
  });
}

module.exports = { streamToFile };
```

Execution in Postman:

- `POST /api/v1/upload`
- Body: `form-data` with key `file` (type: File)
- Response:

```
{
  "message": "uploaded",
  "path": "/.../uploads/filename.txt"
}
```

6. Profiling & Benchmarking

File: `src/utils/profiling.js`

```
function profileExample() {
  console.time("loopTime");
  let sum = 0;
  for (let i = 0; i < 1e6; i++) sum += i;
  console.timeEnd("loopTime");
}

module.exports = { profileExample };
```

Execution:

```
node
> const { profileExample } = require("./src/utils/
profiling");
> profileExample();
loopTime: 4.123ms
```

7. Testing

a) Unit Test (Jest)

File: `src/tests/unit/utils.test.js`

```
const { setCache, getCache } = require("../../utils/cache");

test("cache set and get", () => {
  setCache("key1", "value1", 1000);
  expect(getCache("key1")).toBe("value1");
});
```

b) Integration Test (Supertest)

File: `src/tests/integration/employee.test.js`

```
const request = require("supertest");
const app = require("../../app");
const mongoose = require("mongoose");

beforeAll(async () => {
  await mongoose.connect(process.env.MONGO_URI_TEST ||
"mongodb://localhost:27017/employee_test");
});
afterAll(async () => {
  await mongoose.connection.db.dropDatabase();
  await mongoose.disconnect();
});

describe("Employee API", () => {
  let token;
  test("register + login", async () => {
    await request(app).post("/api/v1/auth/
register").send({ username: "admin", email: "a@a.com",
password: "pass123", role: "admin" });
    const res = await request(app).post("/api/v1/auth/
login").send({ username: "admin", password: "pass123" });
    token = res.body.accessToken;
    expect(token).toBeTruthy();
  });

  test("create employee", async () => {
    const res = await request(app).post("/api/v1/
employees").set("Authorization", `Bearer ${token}
`).send({ name: "Alice", department: "HR", salary: 50000 });
    expect(res.statusCode).toBe(201);
  });

  test("list employees", async () => {
    const res = await request(app).get("/api/v1/
employees").set("Authorization", `Bearer ${token}`);
    expect(res.statusCode).toBe(200);
    expect(Array.isArray(res.body)).toBe(true);
  });
});
```
Execution:

```
npm test
```
Sample Output:

```
PASS  utils.test.js
PASS  employee.test.js
Tests: 4 passed, 0 failed
```

✅ Summary

- **Error Handling:** `AppError` + `errorHandler`.
- **Logging:** Morgan + Winston.
- **Monitoring:** Azure App Insights optional.
- **Caching:** In-memory + Redis-ready.
- **Streams:** Efficient file upload/download.
- **Profiling:** `console.time()` and `--inspect`.
- **Testing:** Jest (unit) + Supertest (integration) with isolated test DB.

Azure

```
# Docs for the Azure Web Apps Deploy action: https://github.com/Azure/webapps-deploy
# More GitHub Actions for Azure: https://github.com/Azure/actions

name: Build and deploy Node.js app to Azure Web App - employee-platform-app

on:
  push:
    branches:
      - main
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read #This is required for actions/checkout

    steps:
      - uses: actions/checkout@v4

      - name: Set up Node.js version
        uses: actions/setup-node@v3
        with:
          node-version: '20.x'

      - name: npm install, build, and test
        run: |
          npm install
          npm run build --if-present
          npm run test --if-present

      - name: Upload artifact for deployment job
        uses: actions/upload-artifact@v4
        with:
          name: node-app
          path: .

  deploy:
    runs-on: ubuntu-latest
    needs: build
    permissions:
      id-token: write #This is required for requesting the 
JWT
      contents: read #This is required for actions/checkout

    steps:
      - name: Download artifact from build job
        uses: actions/download-artifact@v4
        with:
          name: node-app

      - name: Login to Azure
        uses: azure/login@v2
        with:
          client-id: ${{ secrets.__clientidsecretname__ }}
          tenant-id: ${{ secrets.__tenantidsecretname__ }}
          subscription-id: $
{{ secrets.__subscriptionidsecretname__ }}

      - name: 'Deploy to Azure Web App'
        id: deploy-to-webapp
        uses: azure/webapps-deploy@v3
        with:
          app-name: 'employee-platform-app'
          slot-name: 'Production'
          package: .
```