Essential Node.js Modules

Content

- Logging & Monitoring
- Node Mailer
- Web Sockets
- Integrating APIs using Axios
- Model Pagination
- Handling Form Data Using Multer
- Listing Your Application Endpoints
- Scheduling Tasks Using Cron Jobs

Logging & Monitoring

Logging & Monitoring

- What is Logging?
  - Logging means recording details about what happens inside your application.
    In backend APIs, this can include:
    - User requests and responses
    - Server errors
    - Authentication attempts
    - Database operations
    - System events
  - These logs can be written to:
    - Console (for development)
    - Log files (for production)
    - Remote log management services (e.g., ELK, Datadog)

Logging & Monitoring

- What is Monitoring?
  - Monitoring is the continuous observation of your application's performance and behavior.
    It focuses on metrics like:
    - Response times
    - Error rates
    - Server CPU and memory
    - API usage patterns
    - Uptime/downtime
  - Monitoring tools analyze logs, track metrics, and send alerts when something goes wrong.

Logging & Monitoring

- Why Logging & Monitoring Matter:
  - **Debugging**: Helps you identify and fix issues faster
  - **Performance Tuning**: Detect slow endpoints or memory leaks
  - **Security**: Record suspicious or unauthorized access
  - **Auditing**: Track user actions for compliance
  - **Insights**: Understand how users interact with your API

Logging & Monitoring

ex

What is winston?

- Winston is the most widely used logging library in Node.js.
  It provides customizable, structured, and multi-channel logging.
- Support for multiple transports (file, console, HTTP, database)
- Flexible log levels (error, warn, info, debug, etc.)
- Custom formats and metadata
- Works with log analysis tools (ELK, Datadog)
- Supports timestamps, colors, and JSON output

Logging & Monitoring

ex

How to use winston?

- First, start by installing its module: **$ npm i winston**
- Then start implementing it in your app:

```
import winston from 'winston';
import fs from 'fs';
const logDir = './logs';                                    // set up file directory
if (!fs.existsSync(logDir)) fs.mkdirSync(logDir);

export const logger = winston.createLogger({               // create your logger
level: 'info',
format: winston.format.combine(
winston.format.timestamp(),                                 // use timestamps with json format
winston.format.json()),

transports: [
new winston.transports.Console(),
new winston.transports.File({ filename: 'logs/errors.log', level: 'error' }),
new winston.transports.File({ filename: 'logs/combined.log' })]});
```

Logging & Monitoring

ex

- Next start implementing it as a midlleware and start using it across your app:

```
import { logger } from '../utils/logger.js';

export const requestLogger = (req, res, next) => {                    // logs all requests
logger.info(` ${req.method} ${req.originalUrl} from ${req.ip}` );
next();
};

export const errorLogger = (err, req, res, next) => {                  // logs all errors
logger.error(` ${err.message} - ${req.method} ${req.originalUrl}` );
res.status(500).json({ error: 'Something went wrong!' });
};

// server.js
app.use('/', routes);
app.use(requestLogger);                                                 // global logging
app.use(errorLogger);
```

Node Mailer

Node Mailer

ex

- What is Nodemailer? :
  - Nodemailer is the most popular Node.js module for sending emails. It’s widely used in production apps for:
    - Welcome emails
    - Password reset links
    - Notifications and alerts
    - Contact forms
    - Transactional emails (e.g., invoices)

Node Mailer

ex

- What are the benefits of using Nodemailer? :
  - Easy to use: Simple API to send emails with attachments, HTML, etc.
  - Supports auth: Works with Gmail, Outlook, SMTP, and custom servers
  - Customizable: Supports HTML, plain text, embedded images
  - Test-friendly: Integrates with Mailtrap or Ethereal for safe dev testing
  - Attachments: Supports sending PDFs, images, etc. as attachments

Node Mailer

- How to start implementing it in your express app:
  - First start by creating the mailing service after installing **$ npm i nodemailer**:

```
const transporter = nodemailer.createTransport({
service: 'gmail',
auth: {
user: process.env.EMAIL_USER,
pass: process.env.EMAIL_PASS,}});

export const sendVerificationEmail = async (email, token) => {
const verifyUrl = `${process.env.CLIENT_URL}/api/auth/verify/${token}`;
const mailOptions = {
from: process.env.EMAIL_USER,
to: email,
subject: 'Email Verification',
html: `<p>Click <a href="\`${verifyUrl}">here</a> to verify your email.</p>`,};
await transporter.sendMail(mailOptions);};
```

Node Mailer

ex

- Now in your registration controller, you can use this mailing service:

```
export const register = async (req, res) => {
const { name, email, password } = req.body;
try {
const hashedPassword = await bcrypt.hash(password, 10);
const verificationToken = crypto.randomBytes(32).toString('hex');

const user = await User.create({
name,
email,
password: hashedPassword,
verificationToken,});

sendVerificationEmail(user.email, verificationToken);
res.status(201).json({ message: 'Registration successful, check your email to verify
your account.' });
} catch (error) {
res.status(400).json({ error: error.message });}};
```

Web Sockets

Web Sockets

- What are WebSockets?
  - WebSockets allow full-duplex communication between client and server meaning both can send and receive data at any time, without needing to make new HTTP requests.
  - This makes them ideal for applications requiring real-time updates, such as chat applications, online games, and live dashboards.

Web Sockets

- What are the key features of WebSockets?:
  - Persistent Connection:
    - Unlike HTTP, which requires a new connection for each request, WebSockets maintain an open connection, allowing for continuous, low-latency communication.
  - Full-Duplex Communication:
    - Both the client and server can send data to each other simultaneously, unlike HTTP where the client initiates the communication.

Web Sockets

- What are the key features of WebSockets?:
  - Real-time Updates:
    - WebSockets enable real-time data exchange, allowing for instant updates on the client side whenever the server has new information.
  - Low Overhead:
    - Compared to HTTP, WebSockets have lower overhead, meaning less data is transmitted for each communication exchange, making them more efficient.
  - Stateful Protocol:
    - WebSockets maintain a persistent connection, unlike HTTP which is stateless, meaning the server remembers the state of the connection between requests.

Web Sockets

- How does WebSockets Work? :
  - Handshake:
    - The connection is established through an initial HTTP "upgrade" request, signaling the server to switch to the WebSocket protocol.
  - Persistent Connection:
    - Once established, the connection remains open, allowing for data exchange using WebSocket frames.
  - Data Exchange:
    - Data is transmitted as binary or text messages between the client and server.
  - Connection Closure:
    - The connection can be closed by either the client or the server.

Web Sockets

- **Use Cases:**
  - Chat Applications: Real-time messaging and group chats.
  - Online Games: Multiplayer games with real-time interactions and updates.
  - Live Dashboards: Displaying real-time data and analytics.
  - Collaborative Tools: Allowing multiple users to work on the same document simultaneously.
  - Financial Applications: Streaming real-time stock prices and market data.
  - Social Media Feeds: Displaying real-time updates and notifications.

Socket.io

- socket.io is the most popular library to implement WebSockets in Node.
- Benefits of socket.io:
  - Works on top of native WebSocket (adds fallback support)
  - Easy to integrate into Express apps
  - Handles events, rooms, acknowledgments
  - Handles reconnection & cross-browser compatibility

Socket.io

ex

- How to start implementing it in your express app:
  - First install socket.io: **$ npm i socket.io**
  - **Then start by implementing you socket authentication & configurations:**

```
io.use(async (socket, next) => {
const token = socket.handshake.auth.token;
try {
const decoded = jwt.verify(token, process.env.JWT_SECRET);
const user = await User.findById(decoded.id);

if (!user || !user.isVerified) return next(new Error("Unauthorized"));
socket.user = user;
next();
}
catch (err) {
next(new Error("Unauthorized"));}});
```

Socket.io

ex

- We use **io.on('connection')** to fire an event when a user connects to our socket:

```
io.on('connection', async (socket) => {
console.log(`${socket.user.name} connected`);
....
```

  - We can use **socket.on()** for client events like sending messages:

```
socket.on('chat:message', async (msg) => {const savedMsg = await
Message.create({
sender: socket.user._id,
content: msg,
timestamp: new Date().getTime()
});
```

Socket.io

- Then we can use io.emit() to puplish the message to all clients:

```
io.emit('chat:message', {
user: socket.user.name,
msg,
timestamp: savedMsg.timestamp
});
});
```

- Finally, you can handle users disconnections using the disconnect parameter:

```
socket.on('disconnect', () => {
console.log(`${socket.user.name} disconnected`);
});
```

Integrating APIs using Axios

Axios

ex

- What is Axios?
  - Axios is a JavaScript library that allows you to make HTTP requests (like GET, POST, PUT, DELETE) to communicate with APIs or servers.
  - Normally, backends handle requests from frontends. But sometimes the backend itself needs to fetch data from somewhere else another API or service for example:
    - Calling a weather API to get weather info
    - Using Stripe API to charge users
    - Fetching GitHub repos, news headlines, or exchange rates
    - Sending a request to another microservice (in microservice architectures)

Axios

ex

- To start using axios, simply install it first: **$ npm install axios**
- **Then use its methods to fetch, post, delete or update data from another services:**

```
app.get('/api/countries', async (req, res) => {
try {
const response =
await axios.get('https://restcountries.com/v3.1/all?fields=name,capital,region,population');

const countries = response.data.map(country => ({
name: country.name.common,
capital: country.capital?.[0] || 'N/A',
region: country.region,
population: country.population
}));

res.json(countries);
```

Pagination

Pagination

- What is Pagination? :
  - Pagination means breaking up large sets of data into smaller “pages”.
  - Pagination helps us with:
    - Reducing data load
    - Improving performance
    - Making APIs more usable
    - You mainly use two query parameters for pagination:
      - Page (Which page to fetch)
      - Limit (How many items per page)
      - Ex: GET /api/countries?page=2&limit=10

Pagination

ex

- How to use pagination:
  - To use pagination, You can install the following module: **$ npm i mongoose-paginate-v2**
  - Then simply add it in your model before exporting:

```
import mongoosePaginate from 'mongoose-paginate-v2';
countrySchema.plugin(mongoosePaginate);
```

  - Finally before sending your results, paginate it:

```
import mongoosePaginate from 'mongoose-paginate-v2';
countrySchema.plugin(mongoosePaginate);
```

Handling Form Data Using Multer

Multer

- What is Multer?
  - Multer **is a Node.js middleware for handling multipart/form-data, which is** primarily used for **uploading files.**
  - **Think of it as the tool that lets** you **handle:**
    - Image uploads (profile pictures, products)
    - File uploads (PDFs, documents, resumes)
    - Any form that includes files, not just JSON/text

Multer

ex

- How to accept form data using Multer:
  - To use pagination, You can install the following module: **$ npm i multer**
  - Then add start configuring it as a middleware:

```
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/');                                  // Save to /uploads folder
  },
  filename: function (req, file, cb) {
  const uniqueName = `${Date.now()}-${file.originalname}`;
  cb(null, uniqueName);
  }
  });
```

Listing Your Application Endpoints

Listing Your Endpoints

ex

- As your backend app grows, you'll have many routes: /login, /register, /api/countries, /upload, etc.
- But how do you know what routes exist and what methods they use?:
  - Instead of manually searching through all your files, use the swagger tool to list all routes in one place.
  - Installation:

```
$ npm i swagger-jsdoc swagger-ui-express
```

Swagger

- Then configure it:

```
export const swaggerOptions = {
definition: {
openapi: '3.0.0',
info: {
title: 'My API',
version: '1.0.0',
description: 'Auto-generated Swagger docs for your Express app',
},
servers: [
{ url: 'http://localhost:3000' }
],
},
apis: ['./routes/*.js', './docs/*.js'],
};
```

-

Swagger

- And Finally, add a doc file for the UI:

```
/**
* @swagger
* /api/examples/hello:
* get:
* summary: Greet the world
* description: Returns a hello message
* responses:
* 200:
* description: Successful response
* 500:
* description: Server error
*/
```

  - **This will result in a clean web UI that you can use to view all of your routes.**

Scheduling Tasks Using Cron Jobs

Cron Jobs

- What Are Cron Jobs?
  - Cron jobs are automated, scheduled tasks that run at fixed intervals (every minute, hour, day, etc.).
  - In Node.js, they're used for:
    - Sending reminder or newsletter emails
    - Cleaning up expired tokens or data
    - Backing up databases
    - Generating reports
    - Notifying users (e.g., "you haven't logged in for 7 days")

Cron Jobs

ex

The cron expression is made of five fields. Each field can have the following values.

*	*	*	*	*
minute (0-59)	hour (0 - 23)	day of the month (1 - 31)	month (1 - 12)	day of the week (0 - 6)

Examples

- **0 0 * * *** -----> **Every day at midnight**
- **30 14 * * 5** -----> **Every Friday at 2:30 PM**
- ***/5 * * * *** -----> **Every 5 minutes**
- **0 9 * * 1-5** -----> **Weekdays (Mon–Fri) at 9 AM**
- **0 0 1 * *** -----> **First day of each month at midnight**

Cron Jobs

ex

- How to setup your Cron Jobs? :
  - The node-cron module is lightweight, simple, and doesn’t require a DB or Redis
  - Installation: **$ npm install node-cron**
  - Then Simply in a cron.js file, add whatever functionality you went to repeat and its schedule:

```
import cron from 'node-cron';

cron.schedule('* * * * *', () => {
  console.log('This runs every minute');
});
```

Any Questions ?