KOENIG

step forward

Creating Custom APIs Using AI APIs with Node.js

Day 1: RESTful API Foundations with Node.js

RESTful API Foundations with Node.js

1.1 Introduction to APIs

- What is an API?
- REST vs GraphQL
- API request/response flow
- Status codes, headers, and content types

1.2 Setting Up Node.js Project

- Initializing package.json
- Installing Express, nodemon, cors
- Project folder structure: routes/, controllers/, services/

RESTful API Foundations with Node.js

1.3 Creating First REST API

- Basic routes: GET, POST, PUT, DELETE
- Parsing JSON payloads
- Using Postman to test endpoints

1.4 Error Handling and Middleware

- Centralized error handler
- Custom error messages
- Global middleware

Introduction to APIs

Introduction to APIs

- What is an API?
- REST vs GraphQL
- API request/response flow
- Status codes, headers, and content types

What is an API?

- An **API (Application Programming Interface)** is a set of rules and protocols that allows one software application to interact with another.
- An API is like a messenger that takes a request from one system, tells another system what to do, and then brings the response back.
- It acts like a bridge between different systems, enabling them to communicate without knowing how they're implemented internally.

What is an API?

- Real-Life Analogy
  - Think of an API as a restaurant menu:
    - **You (the client)** look at the **menu (API),**
    - Choose what you want (make a request),
    - The **kitchen (server)** prepares it,
    - And the **waiter (API)** brings the food back (response).
  - You don’t need to know how the kitchen works — just how to ask for what you want using the menu (API).
- In Software Terms:
  - If your **mobile app** wants to get weather data:
    - It sends a **request** to a weather **API**.
    - The API fetches the data from a server.
    - Then, it **responds** with the weather info.

What is an API?

- Key Components of an API
  - Endpoint
    - A specific URL where a resource can be accessed
  - Request
    - What the client sends to the server
  - Response
    - What the server sends back to the client
  - Methods
    - HTTP verbs like GET, POST, PUT, DELETE
  - Status Codes
    - Numbers that indicate the result (e.g., 200 OK, 404 Not Found)

REST vs GraphQL

- REST API (Representational State Transfer)
  - REST is an architectural style for APIs
  - Based on standard HTTP methods
  - Simple and widely used
  - Multiple endpoints for different resources (e.g., /users, /posts)
- GraphQL API
  - GraphQL is a query language for APIs developed by Facebook
  - Query exactly what you need
  - More flexible than REST
  - Single endpoint (/graphql) for all queries

REST vs GraphQL

Feature	REST	GraphQL
Response Format	Returns fixed data structure, even if more than needed	Returns only requested data, nothing more
Overfetching / Underfetching	Common problem (you get too much or too little)	Solved (you get exactly what you ask for)
Versioning	New versions created (e.g., /v1/users) to handle changes	No versioning needed — query structure defines response
Performance	Can lead to multiple round trips for nested data	Fetches nested and related data in one query
Flexibility	Less flexible; rigid structure	Highly flexible and customizable
Learning Curve	Easy to start, simple structure	Steeper learning curve, needs schema knowledge
Tooling	Postman, Swagger	GraphQL Playground, Apollo Studio
HTTP Methods	Uses GET, POST, PUT, DELETE etc.	Uses only POST (usually)

API request/response flow

- An API request/response flow describes how data travels between a client (like a browser or app) and a server through an API.
- Step-by-Step Flow
  1. Client Sends a Request
  2. API Endpoint Receives Request
  3. Server Processes the Request
  4. Server Sends a Response
  5. Client Receives and Uses the Response

Status codes, headers, and content types

- HTTP Status Codes
  - Status codes indicate the result of an API request.

Code	Meaning	Use Case Example
200	OK	Request succeeded (e.g., GET data)
201	Created	Resource successfully created (e.g., POST)
204	No Content	Success but no data returned (e.g., DELETE)
400	Bad Request	Invalid input from client
401	Unauthorized	User not authenticated
403	Forbidden	Authenticated but no permission
404	Not Found	Resource does not exist
500	Internal Server Error	Server crashed or error in logic

Status codes, headers, and content types

- HTTP Headers
  - Headers are key-value pairs sent in both request and response to carry meta-information.
- Common Request Headers

Header	Purpose	Example
Content-Type	Type of content being sent	application/json
Authorization	Authentication token or API key	Bearer eyJhbGci...
Accept	Expected response format	application/json, text/html

Status codes, headers, and content types

- Common Response Headers

Header	Purpose	Example
Content-Type	Type of content returned	application/json
Cache-Control	Caching rules	no-cache, max-age=3600
Access-Control-Allow-Origin	CORS policy	* or specific domain

Status codes, headers, and content types

- Content Types (MIME Types)
  - Content types define the format of data being sent or received.

Content-Type	Meaning / Usage
application/json	JSON format (most common for APIs)
application/xml	XML format
text/html	HTML format
text/plain	Plain text
multipart/form-data	Used for file uploads
application/x-www-form-urlencoded	Traditional form data

Setting Up Node.js Project

Setting Up Node.js Project

- Initializing package.json
- Installing Express, nodemon, cors
- Project folder structure: routes/, controllers/, services/

Initializing package.json

- Prerequisites
  - Make sure you have Node.js and npm installed, Check using:

    $ node -v

    $ npm -v
    - If not installed, download from https://nodejs.org
- Initialize Project
  - Create a new folder and initialize a Node.js project:

    $ mkdir my-node-app

    $ cd my-node-app

    $ npm init -y
    - This creates a package.json file with default settings.

Installing Express, nodemon, cors

- Install express
  - Express is a web framework for Node.js that simplifies building APIs and web apps.

    $ npm install express
- Install cors
  - cors is a middleware that allows your API to be accessed from different domains (Cross-Origin Resource Sharing).

    $ npm install cors
- Install nodemon (for development)
  - nodemon restarts your app automatically when you make file changes.

    $ npm install --save-dev nodemon

Project folder structure: routes/, controllers/, services/

- Clean and scalable Node.js project folder structure using the following standard folders:
  - routes/ → For API endpoints
  - controllers/ → For handling request logic
  - services/ → For business logic, DB access, etc.

Creating First REST API

Creating First REST API

- Basic routes: GET, POST, PUT, DELETE
- Parsing JSON payloads
- Using Postman to test endpoints

Basic routes: GET, POST, PUT, DELETE

- Creating first REST API in a Node.js app using Express with basic HTTP methods: GET, POST, PUT, and DELETE.
- **GET – Retrieve Data**
  - Fetch data from the server without modifying it.
- **POST – Create a New Resource**
  - Submit data to the server to create a new resource.
- **PUT – Update an Existing Resource**
  - Replace an existing resource entirely with new data.
- **DELETE – Remove a Resource**
  - Delete an existing resource from the server.

Parsing JSON payloads

- When a client (browser, Postman, frontend app) sends a request with a JSON body, your Express app needs to parse that data so you can access it in your routes.
- In Express, use express.json() to parse incoming requests with JSON payloads:

```
const express = require('express');
const app = express();
// Parse JSON payloads
app.use(express.json());
```

  - This line ensures that JSON data in the body of incoming requests is parsed and available in **req.body.**

Using Postman to test endpoints

- Postman is a GUI tool to send HTTP requests to your API and view responses. It’s ideal for testing REST APIs.
- Download from https://www.postman.com/downloads/, install and launch the app
- Start Your Node.js Server and Send Requests with Postman.
- Example GET Request — Fetch All Users
  - Method: GET
  - URL: http://localhost:3000/api/users
  - Click **Send**
  - You should see a JSON list of users in response.

Error Handling and Middleware

Error Handling and Middleware

- Centralized error handler
- Custom error messages
- Global middleware

Centralized error handler

- A centralized error handler improves code readability, consistency, and maintainability by handling all errors in one place rather than within each route/controller.
- Creating a Centralized Error Handler with Express
  - Create a Custom Error Handler Middleware
    - errorHandler.js - Captures and formats all errors
  - Use the Error Middleware in app.js
    - next(err) - Forwards the error from anywhere in the app

Custom error messages

- Custom error messages help users and developers understand what went wrong in a clean, descriptive way.
- They're especially powerful when paired with centralized error handling.
- Custom Error Messages in Express.js
  - Create a Custom Error Class
    - AppError class - Reusable structure for all errors
  - Use Custom Errors in Controllers
    - next(new AppError(...)) - Clean and consistent error throwing

Global middleware

- Global middleware functions in Express are functions that run before any specific route handler, giving you a place to apply logic like logging, authentication, data parsing, etc., across your entire application.
- What is Middleware?
  - Middleware in Express is a function that has access to: (req, res, next)
  - It can:
    - Modify the request/response
    - End the response
    - Call next() to move to the next middleware
- Use app.use() to register middleware before your route handlers.

Global middleware

- Common Built-in Global Middleware

Middleware	Purpose
express.json()	Parses JSON request bodies
express.urlencoded()	Parses URL-encoded data (e.g. forms)
cors()	Enables Cross-Origin Resource Sharing
Custom middleware	Logging, timing, etc.

RESTful API Foundations with Node.js

Lab 1:

Build a simple task-manager API with:

- Create, Read, Delete tasks
- Use middleware for logging and error handling

Day 2: API Security, Modularization, and Environment Setup

API Security, Modularization, and Environment Setup

2.1 Environment Configuration

- Storing API keys securely with. env
- Using dotenv in Node.js

2.2 API Structuring Best Practices

- Creating modular routes and controllers
- Reusable services
- Organizing validation and constants

API Security, Modularization, and Environment Setup

2.3 CORS, Rate Limiting & Security Basics

- Enabling CORS for frontend apps
- Adding basic security headers
- Rate limiting with express-rate-limit

2.4 Using Git & GitHub for Version Control

- Creating a GitHub repo
- Pushing project code
- Managing .env in .gitignore

Environment Configuration

Environment Configuration

- Storing API keys securely with. env
- Using dotenv in Node.js

Storing API keys securely with. env

- In any Node.js project, sensitive configuration like API keys, database URLs, or secret tokens should not be hardcoded. Instead, we use environment variables.
- Create a file named .env (no file extension) in the root of your project.

  PORT=3000

  API_KEY=my-secret-api-key

Using dotenv in Node.js

- Install dotenv package to load .env Variables in Your Node.js App

  $ npm install dotenv

- At the very top of your index.js or app.js, add:

  require('dotenv').config();

- Then use environment variables via process.env

  const PORT = process.env.PORT || 5000;

  const apiKey = process.env.API_KEY;

API Structuring Best Practices

API Structuring Best Practices

- Creating modular routes and controllers
- Reusable services
- Organizing validation and constants

Creating modular routes and controllers

- Why modular routes and controllers?
  - Keeps code organized, scalable, and maintainable.
  - Separates routing logic from business logic.
  - Makes it easier to work in teams and debug.
- Route File (e.g., routes/userRoutes.js)
  - Handles HTTP method and URL path.
  - Delegates the request to a controller.
- Controller File (e.g., controllers/userController.js)
  - Handles the logic for each route.
  - May call a service or directly interact with models.

Reusable services

- Services handle business logic (e.g., business logics, calculations).
- They are independent of HTTP or route logic.
- Promote code reusability across controllers.
- Keeps controllers clean and focused on request/response.
- Allows unit testing of logic without HTTP dependencies.
- Makes code modular and easy to maintain.
- Avoid writing business logic directly in controllers

Organizing validation and constants

- Why Organize Validation and Constants?
  - Improves readability, reusability, and maintainability
  - Avoids duplicate logic and magic values scattered across code
- Validation Layer
  - express-validator is a popular validation middleware for Express.js.
  - It helps you validate incoming HTTP request data like:
    - Checking if fields are present
    - Validating formats (email, URL, etc.)
- Constants Layer
  - Store fixed values like status codes, or error messages in a constants/ folder
  - Centralize constants for easy updates

CORS, Rate Limiting & Security Basics

CORS, Rate Limiting & Security Basics

- Enabling CORS for frontend apps
- Adding basic security headers
- Rate limiting with express-rate-limit

Enabling CORS for frontend apps

- CORS (Cross-Origin Resource Sharing) is a security feature in browsers.
- It blocks requests from different origins (e.g., frontend at localhost:4200 calling backend at localhost:3000) unless explicitly allowed.
- Enable CORS to allow your frontend app (on a different domain/port) to access your backend API.

Enabling CORS for frontend apps

- Install CORS Middleware

  $ npm install cors

- Basic Usage

```
const cors = require('cors');
app.use(cors());
```

  - This allows all origins (not recommended for production).

- Restrict to Specific Frontend

```
const corsOptions = {
  origin: 'http://localhost:4200',  // frontend URL
  methods: ['GET', 'POST', 'PUT', 'DELETE']
};
app.use(cors(corsOptions));
```

Adding basic security headers

- Security headers help protect your web application from common vulnerabilities by instructing the browser how to behave with your site.
- In Express, these can be added easily using the **helmet** middleware.
- **helmet** is a Node.js middleware that automatically sets various HTTP headers to enhance your app's security.
- Protects your app from common web vulnerabilities like:
  - XSS (Cross-site scripting)
  - Clickjacking
  - MIME sniffing

Adding basic security headers

- Installation

  $ npm install helmet

- Basic Usage

```
const helmet = require('helmet');
app.use(helmet());
```

- Customize Helmet Headers

```
app.use(
  helmet({
    contentSecurityPolicy: false, // disable if causing issues in dev
  })
);
```

Rate limiting with express-rate-limit

- Rate limiting helps:
  - Prevent API abuse, brute-force, and DDoS attacks
  - Limit requests per IP address within a specific time window
  - Improve API stability and protect backend resources
- Install the Package
  - npm install express-rate-limit

Rate limiting with express-rate-limit

- Configure the Rate Limiter
  - Create a rate limiter in index.js (or a middleware file):

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

// Limit to 100 requests per 15 minutes per IP
const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100,                 // limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP, please try again later.',
  standardHeaders: true,   // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false     // Disable the `X-RateLimit-*` headers
});
```

Rate limiting with express-rate-limit

- Apply to Routes
  - You can apply the limiter globally or to specific routes.
- Globally for all /api routes:

  app.use('/api', apiLimiter);
- Or apply to specific route:

  app.use('/api/auth', authLimiter);

Using Git & GitHub for Version Control

Using Git & GitHub for Version Control

- Creating a GitHub repo
- Pushing project code
- Managing .env in .gitignore

Creating a GitHub repo

- Initialize Git in Your Project

  $ git init

  - This creates a hidden .git/ folder and starts tracking your code.
- Stage and Commit Your Code

  $ git add .

  $ git commit -m "Initial commit"
- Create a GitHub Repository
  - Go to https://github.com
  - Click **"New repository"**
  - Give it a name (e.g., task-manager-api)
  - Click **Create** repository

Pushing project code

- Connect to GitHub

  $ git remote add origin https://github.com/your-username/task-manager-api.git

- Use main as the default branch (replacing master)

  $ git branch -M main

- Push local code to GitHub

  $ git push -u origin main

Managing .env in .gitignore

- To protect sensitive credentials like API keys, DB URLs, and tokens, you should never commit your .env file to GitHub.
- Create a .gitignore File
  - Add files/folders you don’t want to track, like:

```
.env
node_modules/
```

  - This tells Git to ignore the .env file and node_modules/ folder so, it’s never committed.

API Security, Modularization, and Environment Setup

Lab 2:

Refactor the task-manager API:

- Move logic to controllers/ and services/
- Add .env, rate limiting, and GitHub versioning

Day 3: OpenAI API Integration (ChatGPT, DALL·E)

OpenAI API Integration (ChatGPT, DALL·E)

3.1 Introduction to OpenAI APIs

- Overview: GPT, DALL·E, Whisper
- OpenAI API documentation walkthrough
- Prompt engineering fundamentals

3.2 ChatGPT Integration (gpt-3.5-turbo)

- Install axios for HTTP calls
- Secure API key with .env
- Create /chat endpoint that accepts prompt and returns a GPT response

OpenAI API Integration (ChatGPT, DALL·E)

3.3 Image Generation with DALL·E

- Understanding DALL·E parameters (prompt, size, format)
- Creating /generate-image endpoint

3.4 Prompt Engineering Practice

- Writing system vs user prompts
- Controlling tone, format, creativity

Introduction to OpenAI APIs

Introduction to OpenAI APIs

- Overview: GPT, DALL·E, Whisper
- OpenAI API documentation walkthrough
- Prompt engineering fundamentals

Overview: GPT, DALL·E, Whisper

- GPT (Generative Pre-trained Transformer)
  - GPT is a language model that understands and generates human-like text.
- Use Cases:
  - Chatbots and virtual assistants
  - Text summarization
  - Code generation and explanation
  - Writing help (emails, blogs, etc.)
  - Translation and Q&A

Overview: GPT, DALL·E, Whisper

- DALL·E (Text-to-Image Generation)
  - DALL·E generates realistic images and art from text prompts.
- Use Cases:
  - Marketing & creative design
  - UI/UX mockups
  - Game assets
  - Visual storytelling

Overview: GPT, DALL·E, Whisper

- Whisper (Speech-to-Text)
  - Whisper is a speech recognition system to transcribe and translate audio files.
- Use Cases:
  - Meeting transcription
  - Podcast indexing
  - Voice assistants
  - Video subtitling

OpenAI API documentation walkthrough

- OpenAI API Documentation:
- Visit: https://platform.openai.com/docs
- Sections in the Documentation
  - Getting Started
    - Overview of how to use the API
    - API key setup
    - Rate limits and usage quotas
  - API Reference (Most Important)
    - This section contains all endpoints and payload formats.
      - Chat (GPT)
      - Images (DALL·E)
      - Audio (Whisper)

OpenAI API documentation walkthrough

- Sections in the Documentation
  - Models
    - Learn about available models and their capabilities:
      - GPT-4 vs GPT-3.5
      - DALL·E models
      - Embedding models
  - Authentication
    - Explains how to use your API key with headers.
  - Libraries & SDKs
    - Official OpenAI SDKs are provided for Node.js, Python
  - Rate Limits and Usage
    - Based on the model (e.g. GPT-4 has more limited throughput)
  - Pricing
    - Know your costs before you build.

Prompt engineering fundamentals

- Prompt engineering is the practice of crafting inputs (prompts) to guide AI models like GPT to produce accurate, relevant, and useful outputs.
- Core Goals
  - Improve accuracy
  - Control tone and style
  - Achieve consistent results
  - Reduce hallucinations
  - Make outputs structured and parseable

Prompt engineering fundamentals

- Clear Instructions
  - Be direct and specific.
- Set the Role or Behavior
  - Use system messages to set context.
- Use Examples (Few-shot Learning)
  - Provide input-output pairs to guide behavior.
- Ask for Structured Output
  - Guide GPT to format results as JSON, bullet points, tables, etc.
- Use Temperature and Max Tokens
  - temperature: Controls randomness
    - 0 = factual/deterministic
    - 1 = creative/random
  - max_tokens: Controls output length

Prompt engineering fundamentals

- Prompt Styles

Style	Purpose	Example
Instructional	Ask for actions	“Write a polite apology email to a customer.”
Contextual	Add background info	“You are a resume expert. Improve this summary…”
Conversational	Multiturn dialog	“User: Tell me a joke. Assistant: …”
Formatting Prompt	Control structure	“List the points in bullet form.”

Prompt engineering fundamentals

- DALL·E Prompt Tips
  - **Be descriptive**: Include subject, style, lighting, angle

    “A futuristic robot reading a book, digital art, high detail”
  - **Specify style**: "oil painting", "isometric", "pixel art", etc.
  - **Use commas to separate details**
- Whisper Prompt Tips
  - Whisper does not need a prompt; it's a speech-to-text model.
  - However, you can:
    - Choose language for transcription
    - Use translation mode (translate: true)

ChatGPT Integration (gpt-3.5-turbo)

ChatGPT Integration (gpt-3.5-turbo)

- Install axios for HTTP calls
- Secure API key with .env
- Create /chat endpoint that accepts prompt and returns a GPT response

Install axios for HTTP calls

- Axios is a promise-based HTTP client for making API requests from Node.js or the browser.
- Installation

  $ npm install axios
- Basic Usage in Node.js

```
const axios = require('axios');
axios.get('https://api.example.com/data').then(response => {
  console.log(response.data);
}).catch(error => {
  console.error(error.message);
});
```

Secure API key with .env

- Create .env file

  OPENAI_API_KEY=sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

- Never share or commit your API key.
- Secure .env in .gitignore
  - Create a .gitignore:

    node_modules/

    .env

Create /chat endpoint that accepts prompt and returns a GPT response

```
app.post("/chat", async (req, res) => {
  const { prompt } = req.body;
try {
    const response = await axios.post(
      OPENAI_API_URL,
      {
        model: "gpt-3.5-turbo",
        messages: [
          { role: "system", content: "You are a helpful assistant." },
          { role: "user", content: prompt },
        ],
      },     {
        headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer ${process.env.OPENAI_API_KEY}`,
        },     }    );
    const gptReply = response.data.choices[0].message.content;
    res.json({ reply: gptReply });
  } catch (error) {
    console.error("Error from OpenAI:", error.response?.data || error.message);
    res.status(500).json({ error: "Failed to fetch GPT response" });
  }});
```

Image Generation with DALL·E

Image Generation with DALL·E

- Understanding DALL·E parameters (prompt, size, format)
- Creating /generate-image endpoint

Understanding DALL·E parameters (prompt, size, format)

- prompt
  - Description of the image you want to generate.
  - The most important parameter.
  - Be clear and specific for better results.
- Example:
  - "prompt": "A futuristic cityscape at sunset with flying cars“

Understanding DALL·E parameters (prompt, size, format)

- size
  - Defines the resolution of the generated image.
  - Common options:
    - "256x256" – fast, low quality
    - "512x512" – balanced
    - "1024x1024" – high quality, slower
- Example:
  - "size": "512x512"

Understanding DALL·E parameters (prompt, size, format)

- format
  - Specifies the image output format (optional).
  - Default: "url" – returns a link to the image.
  - Other option: "b64_json" – returns image as a base64 string (useful for embedding or direct use in frontend apps).
- Example:
  - "response_format": "b64_json"

Creating /generate-image endpoint

```
app.post("/generate-image", async (req, res) => {
  const { prompt, size = "512x512" } = req.body;
try {
    const response = await axios.post(
      "https://api.openai.com/v1/images/generations",
      {
        prompt,
        n: 1,
        size,
        response_format: "url",
      },
      {
        headers: {
          "Content-Type": "application/json",
          Authorization: `Bearer
${process.env.OPENAI_API_KEY}`,
        },     }    );
    const imageUrl = response.data.data[0].url;
    res.json({ imageUrl });
  } catch (error) {
    console.error("DALL·E API error:",
error.response?.data || error.message);
    res.status(500).json({ error: "Failed to generate
image" });
  }});
```

Prompt Engineering Practice

Prompt Engineering Practice

- Writing system vs user prompts
- Controlling tone, format, creativity

Writing system vs user prompts

- Prompt Roles in ChatGPT API (in messages array)

Role	Purpose	Example
system	Sets overall behavior and tone of the assistant	"You are a professional resume writer."
user	Input or question from the human	"Write a cover letter for a product manager role."

Writing system vs user prompts

- Example Chat Prompt Structure:

```
messages: [
  { role: "system", content: "You are a kind and formal email assistant." },
  { role: "user", content: "Write an apology email to a client for a missed meeting." }
]
```

Controlling tone, format, creativity

- Controlling Tone
  - Formal Tone
    - { "role": "system", "content": "You are a polite and professional business assistant." }
  - Humorous Tone
    - { "role": "system", "content": "You are a witty comedian who answers with humor." }
  - Example Comparison:

Prompt	Tone Style Result
"Explain AI in simple terms"	(default tone)
+ system: "You are a stand-up comic"	"AI is like a toddler with internet access..."
+ system: "You are a PhD professor"	"Artificial Intelligence refers to computational..."

Controlling tone, format, creativity

- Controlling Format
- Ask for JSON
  - { "role": "user", "content": "Summarize the article and return it as JSON with keys 'title', 'summary', and 'keywords'." }
- Ask for Bullet Points
  - { "role": "user", "content": "List the pros and cons of remote work in bullet points." }

Controlling tone, format, creativity

- Controlling Creativity with temperature

Temperature	Behavior	Use Case
0.0	Factual, precise (deterministic)	Math, technical explanations
0.7	Balanced creativity	Conversational, general summaries
1.0+	Highly creative & unpredictable	Poems, jokes, storytelling

OpenAI API Integration (ChatGPT, DALL·E)

Lab 3:

Create a content-generator API with:

- /blog-summary – summarizes blog posts
- /image-cover – generates an image cover using DALL·E

Day 4: Google Cloud AI & Whisper API Integration

Google Cloud AI & Whisper API Integration

4.1 Using Google Cloud Vision API

- Enabling APIs in Google Cloud
- Creating service account and API key
- Analyzing:
  - o Text (OCR)
  - o Labels and objects

4.2 Using Google Natural Language API

- Analyzing sentiment, entities, syntax
- Creating /analyze-text endpoint

Google Cloud AI & Whisper API Integration

4.3 Uploading Files with Multer

- Handling audio/image uploads
- Saving to disk or using memory buffer

4.4 Whisper API for Speech-to-Text

- Upload .mp3 or .wav files
- Send to OpenAI Whisper API
- Create /transcribe-audio endpoint

Uploading Files with Multer

Uploading Files with Multer

- Handling audio/image uploads
- Saving to disk or using memory buffer

Uploading Files with Multer

- Multer is a middleware for handling multipart/form-data, primarily used for uploading files in Express apps.
- Install Multer

  $ npm install multer
- Basic Setup

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' }); // files stored in /uploads
```

Handling audio/image uploads

- Handle Image/Audio Uploads Example

```
// Single image/audio file
app.post('/upload', upload.single('file'), (req, res) => {
  console.log(req.file); // file metadata
  res.send('File uploaded');
});
```

  - Use upload.single('file') for file upload.
  - File Metadata in req.file
    - originalname – original file name
    - mimetype – e.g., image/png, audio/mpeg
    - filename – generated file name
    - path – full path on disk

Saving to disk or using memory buffer

- Two Storage Options in Multer
  - Disk Storage
    - Saves files directly to a folder (e.g., uploads/).
    - Good for temporary/local development or when processing files later.
    - Access files using req.file.path or req.file.filename
  - Memory Storage (Buffer)
    - Stores files in RAM as Buffer (req.file.buffer).
    - Best when you want to process immediately (e.g., send to cloud, analyze in-memory).
    - Access files using req.file.buffer
  - When to Use Each

Use Case	Use Storage Type
Save files on local server	Disk
Stream/upload to S3 or API directly	Memory
Temporary processing (no save)	Memory
Upload then process asynchronously	Disk

Whisper API for Speech-to-Text

Whisper API for Speech-to-Text

- Upload .mp3 or .wav files
- Send to OpenAI Whisper API
- Create /transcribe-audio endpoint

Whisper API for Speech-to-Text

- Whisper API is a OpenAI's speech-to-text model.
- Converts spoken audio (.mp3, .wav, etc.) to text transcription.
- Ideal for voice commands, audio summaries, subtitles, etc.
- Accepts audio formats like:
  - .mp3
  - .mp4
  - .mpeg
  - .mpga
  - .wav
  - .webm

Upload .mp3 or .wav files

- Multer is a middleware for handling multipart/form-data, primarily used for uploading files in Express apps.
- Install Multer

  $ npm install multer

- Setup Storage (Disk or Memory)

```
const multer = require('multer');
// For disk storage
const upload = multer({ dest: 'uploads/' });
// OR for memory buffer
const memoryUpload = multer({ storage: multer.memoryStorage() });
```

- Accept Only .mp3 or .wav Files

```
fileFilter: (req, file, cb) => {
   const allowed = ["audio/mpeg", "audio/wav"];
   cb(null, allowed.includes(file.mimetype));
 }
```

Send to OpenAI Whisper API

- Uploading Audio (Node.js + Axios + FormData)

```
const form = new FormData();
form.append('file', fs.createReadStream('./audio/sample.mp3'));
form.append('model', 'whisper-1'); // required
axios.post('https://api.openai.com/v1/audio/transcriptions', form, {
  headers: {
    'Authorization': `Bearer YOUR_OPENAI_API_KEY`,
    ...form.getHeaders()
  }
}).then(res => console.log(res.data.text))
.catch(err => console.error(err.response.data));
```

Create /transcribe-audio endpoint

- Steps to integrate OpenAI’s Whisper API for speech-to-text in Node.js app
  - Setup Node project and Dependencies
  - Accepts .mp3 or .wav audio file uploads
  - Sends audio to Whisper API
  - Returns transcribed text

Using Google Cloud Vision API

Using Google Cloud Vision API

- Enabling APIs in Google Cloud
- Creating service account and API key
- Analyzing Text (OCR), Labels and objects

Using Google Cloud Vision API

- The Google Cloud Vision API is a powerful tool that allows you to understand the content of your images. With this API, you can detect faces, recognize text, identify objects, and more.
- Prerequisites
  - Before you can get started, you'll need to have the following:
    - A Google Cloud Platform account
    - A new project created in the Google Cloud Platform console
    - The Cloud Vision API enabled for your project
    - A service account created for your project
    - The service account key downloaded to your computer

Enabling APIs in Google Cloud

- Navigate to the API Library:
  - Open the Google Cloud Console.
  - From the project dropdown menu at the top of the page, select the project for which you want to enable an API.
  - Click on the navigation menu (the hamburger icon) in the top-left corner.
  - Go to **APIs & Services > Library**.
- Find and Select the API:
  - The API Library displays all available APIs. You can use the search bar to find a specific API (e.g., "Cloud Vision API").
  - Click on the desired API from the search results to open its overview page.
- Enable the API:
  - On the API's overview page, click the **Enable** button.
    - If the API has dependencies on other APIs, they will be enabled automatically.

Creating service account and API key

- A service account is a special type of Google account intended to represent a non-human user that needs to authenticate and be authorized to access data in Google APIs.
- It is the recommended way for applications to authenticate with Google Cloud services.

Creating service account and API key

- Steps to Create a Service Account:
  1. Navigate to the Service Accounts Page:
     - In the Google Cloud Console, go to the navigation menu.
     - Select **IAM & Admin > Service Accounts**.
  2. Initiate Service Account Creation:
     - Click the **+ CREATE SERVICE ACCOUNT** button at the top of the page.
  3. Provide Service Account Details:
     - **Service account name:** Enter a descriptive name for your service account (e.g., "my-vision-api-user").
     - **Service account ID:** This ID is automatically generated based on the name you provide. You can edit it if needed.
     - Click CREATE AND CONTINUE.

Creating service account and API key

- Creating and Downloading a Service Account Key:
  - After creating the service account, you need to generate a key that your application will use to authenticate.
  - Select the Service Account:
    - From the list of service accounts, click on the email address of the service account you just created.
  - Navigate to the Keys Tab:
    - Click on the **KEYS** tab.
  - Add a New Key:
    - Click on ADD KEY and then select Create new key.
  - Choose Key Type and Download:
    - Select **JSON** as the key type. This is the recommended format for most applications.
    - Click **CREATE**.
      - A JSON file containing your service account key will be automatically downloaded to your computer. **Store this file securely**, as it provides access to your Google Cloud resources. This is the only time you will be able to download this key.

Analyzing Text (OCR), Labels and objects

- You can easily analyze images for text, labels, and objects using the Google Cloud Vision API with the @google-cloud/vision Node.js client library.
- Text Analysis (OCR)
  - To extract text from an image, you'll use the **textDetection** method.
  - This is great for Optical Character Recognition (OCR) on images with sparse text, like street signs or posters.
  - For dense text in a document, the **documentTextDetection** method is optimized and provides a more structured response, including pages, blocks, paragraphs, and words.

Analyzing Text (OCR), Labels and objects

- Label Detection
  - Label detection provides a list of content categories for an image.
  - It can identify broad categories like "sky" or "animal" as well as more specific ones.
  - You use the **labelDetection** method for this.
- Object Detection (Object Localization)
  - If you need to identify specific objects within an image and get their locations, you'll use the **objectLocalization** method.
  - This returns the name of each detected object along with the coordinates of a bounding box that outlines it in the image.

Using Google Natural Language API

Using Google Natural Language API

- Analyzing sentiment, entities, syntax
- Creating /analyze-text endpoint

Using Google Natural Language API

- You can analyze text for emotional leaning, known entities, and language structure using the Google Natural Language API with the @google-cloud/language Node.js client.

Analyzing sentiment, entities, syntax

- Sentiment Analysis
  - Sentiment analysis inspects text to identify the fundamental emotional opinion, determining if it's positive, negative, or neutral.
  - The API provides a score (the emotional leaning from -1.0 for negative to 1.0 for positive) and a magnitude (the overall strength of emotion, from 0 to +infinity).
- Entity Analysis
  - Entity analysis scans text to find and classify known entities—such as people, organizations, locations, events, products, and media—into predefined categories.
  - It also provides a salience score (from 0 to 1.0) that indicates the importance of the entity to the overall text.

Analyzing sentiment, entities, syntax

- Syntactic Analysis
  - Syntactic analysis extracts language information by breaking down text into a series of tokens (words) and providing a grammatical analysis for each one.
  - This includes its part of speech (e.g., noun, verb, adjective) and its role in the sentence's dependency tree (how words relate to each other).

Creating /analyze-text endpoint

- Steps to create /analyze-text endpoint using Google NLP in Node.js
  - Google Cloud Project Setup
    - Create a Google Cloud Project
    - Enable the Natural Language API
    - Set up Authentication (Service Account Key)
  - Node.js Project Setup
    - Initialize Project
    - Install Dependencies
    - Analyzing Sentiment, Entities, and Syntax
    - Creating the /analyze-text Endpoint

Google Cloud AI & Whisper API Integration

Lab 4:

Build a "Media Intelligence API":

- /analyze-image: returns labels/text from image
- /transcribe-audio: returns text from uploaded audio
- /summarize-transcript: sends transcribed text to ChatGPT for summary

Day 5: Final Project, Testing, and Deployment

Final Project, Testing, and Deployment

5.1 Combining AI APIs in Workflow

- AI workflow: audio → text → analysis → summary
- Chaining multiple AI APIs in one request
- Create /smart-report:
  - o Accepts audio
  - o Transcribes → summarizes → returns JSON summary

5.2 Input Validation & Robust Error Handling

- Using express-validator or custom checks
- Handling timeouts, malformed inputs, API quota errors

Final Project, Testing, and Deployment

5.3 API Documentation with Postman

- Creating Postman collections
- Documenting request/response schemas
- Exporting and sharing collections

5.4 Deployment

- Deploy to Render or Vercel
- Securing deployed API

Combining AI APIs in Workflow

Combining AI APIs in Workflow

- AI workflow: audio → text → analysis → summary
- Chaining multiple AI APIs in one request
- Create /smart-report:
  - Accepts audio
  - Transcribes → summarizes → returns JSON summary

Input Validation & Robust Error Handling

Input Validation & Robust Error Handling

- Using express-validator or custom checks
- Handling timeouts, malformed inputs, API quota errors

Using express-validator or custom checks

- Input Validation prevents invalid or malicious data from reaching your logic/database
- It Helps ensure data integrity and security
- Use express-validator for structured, scalable validation
- Centralize validation logic in a validators/ folder
- Always return helpful error messages to the client

Handling timeouts, malformed inputs, API quota errors

- Handling Timeouts
  - Set a timeout limit on external API calls (e.g., using Axios).

    ```
    axios.get('https://api.example.com', { timeout: 5000 }) // 5 seconds
    ```
- Handling Malformed Inputs
  - Use express-validator or manual checks to validate user inputs before processing.

    ```
    if (!req.body.email || !req.body.email.includes('@')) {
      return res.status(400).json({ error: 'Invalid email format' });
    }
    ```
  - Always check types, required fields, and ranges.

Handling timeouts, malformed inputs, API quota errors

- Handling API Quota Errors (429)
  - Many third-party APIs return 429 Too Many Requests when rate limits are hit.
  - Catch and respond gracefully:

```
axios.get('https://api.example.com')
  .catch(err => {
    if (err.response && err.response.status === 429) {
      return res.status(429).json({ error: 'API rate limit exceeded. Try again later.' });
    }
  });
```

  - Optionally retry with exponential backoff or show a helpful message to the user.

API Documentation with Postman

API Documentation with Postman

- Creating Postman collections
- Documenting request/response schemas
- Exporting and sharing collections

Deployment

Deployment

- Deploy to Render or Vercel
- Securing deployed API

Deploy to Render

- Create a new repository on GitHub
- Push your project
- Go to https://dashboard.render.com
- Click “New Web Service”
- Connect your GitHub repository
- Fill in details:
  - Environment: Node
  - Build Command: npm install
  - Start Command: node server.js
  - Environment Variables: OPENAI_API_KEY and Any others from .env
- Click **Deploy**

Securing deployed API

Security Layer	Implementation Suggestion
API key in .env	Don’t hardcode in code. Use environment vars (Render > Environment tab)
Restrict CORS	Only allow known frontend origins
Rate Limiting	Use express-rate-limit to prevent abuse
Quota Monitoring	Monitor OpenAI usage to avoid overcharges
Remove uploaded files	fs.unlinkSync() after processing audio/image
Input validation	Use express-validator as shown earlier
Hide .env	Ensure .env is listed in .gitignore

Final Project, Testing, and Deployment

Final Lab 5:

Build a “Smart AI Assistant API”:

- Features:
  - o /chat: ChatGPT response
  - o /transcribe: Whisper audio transcription
  - o /analyze: Vision + NLP
  - o /smart-assist: Uploads audio/image, performs AI operations, and summarizes results

Happy Learning :)