

AI Integration for Web Development

I. Introduction to API Calling (Day 1)

Large Language Models (LLMs) are built upon massive computational systems due to their requirements to process large data each second. Because of this, companies usually host their own private servers where it runs all data processing and matrix calculations. An API acts as a key for developers to access just the model to work with, without risking any tamperings of the main servers.

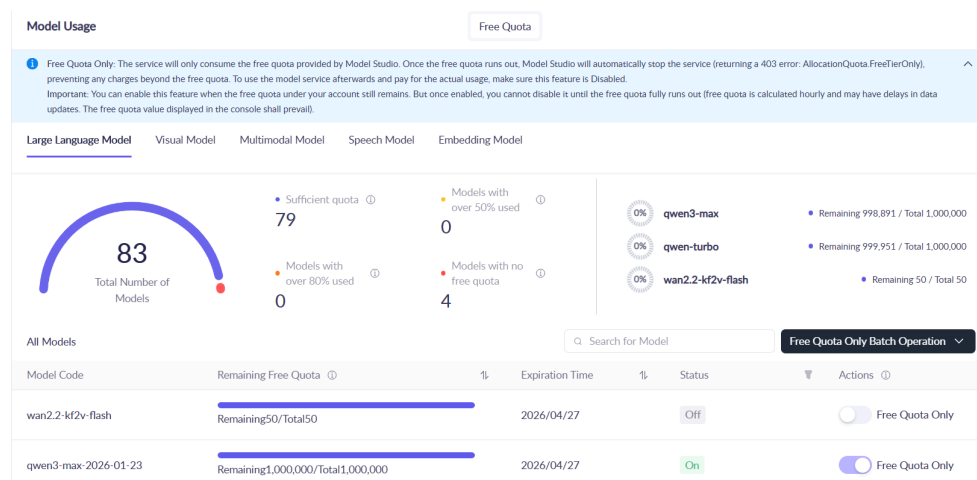
A. API Functions Towards Interface Standards and Security

An API sets the standard for all users on how they interact with the LLM. Developers using the API key to access the model obtain a standardized collection of functions and tools tailored to the use of that specific model. We can think of this as adding an extra library to include in an IDE, which lets developers write predictable code.

Having access only to the model also isolates your data and code from other developers and vice versa. Additionally, any improvements made on the AI model behind the API would not have any affect on a developer's code.

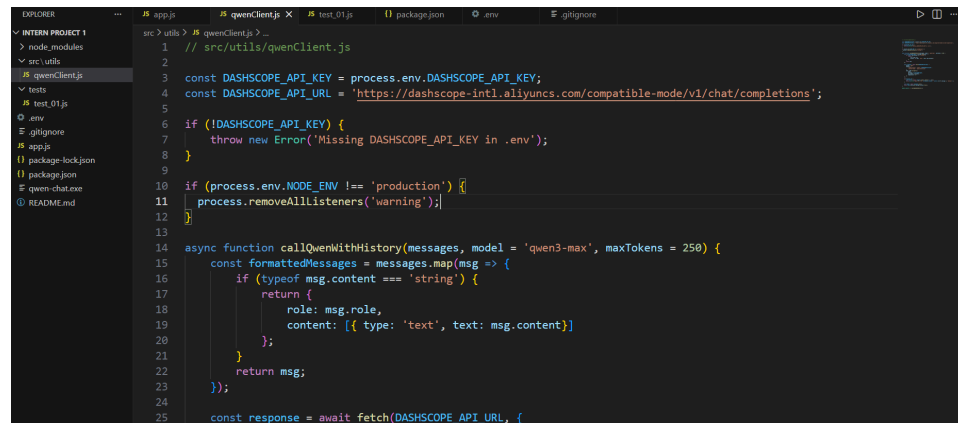
B. Basics of API Key Application

In order to carry out a test of working with an existing LLM, I chose the company Alibaba which has an abundance of working AI models comparable to OpenAI's chatGPT. After creating a free account and obtaining an API key, we gain access to more than 70 AI models with a **free** quota of 1 million output tokens.



Before working with any of those models, an essential step is to store the private API key in the environment variables or a .env file. This step ensures security on the data from other developers and avoids getting banned from the server for any abuse.

After creating a project folder, we use java script to create a default qwenClient.js file to call Qwen. This enables all other applications to access the model without code repetition. It should handle authentication, message formatting specific to Qwen, error handling, etc. Standardizing the input format is important since most LLMs use a multimodal input format.



```
1 // src/utlis/qwenClient.js
2
3 const DASHSCOPE_API_KEY = process.env.DASHSCOPE_API_KEY;
4 const DASHSCOPE_API_URL = 'https://dashscope-intl.aliyuncs.com/compatible-mode/v1/chat/completions';
5
6 if (!DASHSCOPE_API_KEY) {
7   throw new Error('Missing DASHSCOPE_API_KEY in .env');
8 }
9
10 if (process.env.NODE_ENV !== 'production') {
11   process.removeAllListeners('warning');
12 }
13
14 async function callQwenWithHistory(messages, model = 'qwen3-max', maxTokens = 250) {
15   const formattedMessages = messages.map(msg => {
16     if (typeof msg.content === 'string') {
17       return {
18         role: msg.role,
19         content: [{ type: 'text', text: msg.content }]
20       };
21     }
22     return msg;
23   });
24
25   const response = await fetch(DASHSCOPE_API_URL, {
```

[qwenClient.js](#) exports a function ‘callQwenWithHistory’ which works to analyze a string of inputs using the default ‘qwen3-max’ model. Calling the function from any external .js file like the example below can be used to create a command line interface (CLI) where Qwen is integrated in. In this case, we can hold a conversation with the AI model with an expected output under 500 tokens.



```
16
17 // Creating a readline interface
18 const rl = readline.createInterface({
19   prompt: 'You: ',
20   input: process.stdin,
21   output: process.stdout
22 });
23
24 console.log('Qwen3-Max Chat Console (press Ctrl + C to exit)!\n');
25
26 rl.prompt();
27
28 let messages = [];
29
30 rl.on('line', async (input) => {
31   if (!input.trim()) {
32     rl.prompt();
33     return;
34   }
35
36   messages.push({ role: 'user', content: input });
37
38   try {
39     console.log('Qwen is thinking...\n');
40     const response = await callQwenWithHistory(messages);
41     messages.push({ role: 'assistant', content: response });
42     console.log('Qwen3-Max: ${response}\n');
43   } catch (err) {
44     console.error('Error: ${err.message}\n');
45   }
46
47   rl.prompt();
48 });
49
50 rl.on('close', () => {
51   console.log('\nShut down.\n');
52   process.exit(0);
53 });
```

```
Qwen3-Max Chat Console (press Ctrl + C to exit)!  
You: what is 5+5  
Qwen is thinking...  
Qwen3-Max: 5 + 5 equals 10.  
You: |
```

II. AI Integration In Inputting Product Attributes for Customers (Day 2)

A. Problem Scenario

Consider a sudden trend rush phenomenon that occurred in the span of a few days where more than 10 products are in high demand. A small online shop owner would like to stock up on these products so that they could be resold at a higher price. Commonly, making a list of the products and their attributes; name, market price, commodity type, mass, SKU, etc; requires manual work which lends itself to be boring and error-prone.

Goal: Create a solution to simplify the arduous process of listing out products with the use of AI integration.

Proposed solution: An interface where users can provide unstructured data (photo of the product) then the AI extracts structured data to be processed and formatted into a csv file. The csv file should remain updateable until the AI program is closed.

B. Method

In order to test the implementation, we try to use a hybrid from 2 AI providers. Alibaba's Qwen-VL-Max model has image input capability and a stunning ability to understand them, making it perfect to extract the product attributes from an image onto a single line. Google's gemini 2.0 flash also has great infrastructure as an LLM, which can be tailored to this specific task; to separate each attribute into its own group so that it becomes csv ready.

We start by creating a new [geminiClient.js](#) with the same purpose as we did with [qwenClient.js](#). This time we make a function to extract a product's description from a single line input into its own group.

```

14 ✓ async function structureProductData(rawText) {
15   const prompt = `
16   You are a product data validator for an Indonesian e-commerce system.
17   Extract and standardize the following fields from the input:
18
19   Input: "${rawText}"
20
21   Output strict JSON with these fields:
22   - "name": product name (string)
23   - "price (IDR)": price in IDR (number, no commas or "Rp")
24   - "sku": SKU code (string, uppercase if possible)
25   - "category": one of ["Beauty", "Food & Beverage", "Stationery", "Electronics", "Home & Living", "Other"]
26   - "unit": unit/volume/weight (e.g., "180ml", "500g", "1pc")
27
28   If a field is missing, use null.
29   Do not add extra fields.
30   `;
31
32   try {
33     const response = await client.models.generateContent({
34       model: "gemini-2.0-flash",
35       contents: [{ role: 'user', parts: [{ text: prompt }] }],
36       config: {
37         responseType: "application/json",
38         temperature: 0.1
39       }
40     });
41
42     const text = response.text;
43
44     if (!text) {
45       throw new Error("Empty response from Gemini");
46     }
47
48     return JSON.parse(text);
49   } catch (error) {

```

We can test the function by hardcoding a single line input from another .js test file.

```

1 // tests/geminiTest_01.js
2
3 const { structureProductData } = require('../src/Utils/geminiClient');
4
5 (async () => {
6   const raw = "Clear Men Anti-Dandruff Shampoo 180ml Rp 25.000 SKU: CLR-SHP-001";
7   const structured = await structureProductData(raw);
8   console.log(JSON.stringify(structured, null, 2));
9 })();

```

```

E:\Computer Science\Programming\VS CODE\Intern Project 1>node tests/geminiTest_01.js
[dotenv@17.2.3] injecting env (1) from .env -- tip: add secrets lifecycle management: https://dotenvx.com/ops
{
  "name": "Clear Men Anti-Dandruff Shampoo",
  "price (IDR)": 25000,
  "sku": "CLR-SHP-001",
  "category": "Beauty",
  "unit": "180ml"
}

```