Introduction to APIs

Application Programming Interfaces (APIs) are critical componentns of the modern digital world. In the context of web services, APIs allow applications to request and receive data over the internet using standardized protocols. This section demonstrates how web APIs function through practical examples, examining two common approaches to API interaction and exploring the fundamental concepts that make these systems work.

Public Web API Demonstration

The best way to understand APIs is to interact with them directly. This section examines two different API patterns through hands-on examples that illustrate how HTTP requests work in practice.

GET Requests: Weather Data Retrieval

The first example demonstrates a GET request to retrieve current weather information for Honolulu. GET requests are the most common type of API call, used when you want to retrieve data without making any changes to the server.

To try this example, open Postman and import the following request:

```
curl "https://api.open-meteo.com/v1/forecast?latitude=21.3069&longitude=-
157.8583&current=temperature_2m,wind_speed_10m,is_day&timezone=auto"
```

When this request is sent to the Open-Meteo service, the server returns a JSON document containing the requested weather data:

```
{
  "latitude": 21.375,
  "longitude": -158.0,
  "generationtime_ms": 0.026106834411621094,
  "utc_offset_seconds": -36000,
  "timezone": "Pacific/Honolulu",
  "timezone_abbreviation": "GMT-10",
  "elevation": 4.0,
  "current_units": {
    "time": "iso8601",
    "interval": "seconds",
    "temperature_2m": "°C",
    "wind_speed_10m": "km/h",
    "is_day": ""
  },
  "current": {
    "time": "2025-08-29T14:45",
    "interval": 900,
    "temperature_2m": 31.3,
    "wind_speed_10m": 16.2,
    "is_day": 1
  }
}
```

This example demonstrates several key concepts. The URL contains a query string (the portion after the question mark) composed of key-value pairs separated by ampersands. Each parameter serves a specific purpose: `latitude` and `longitude` specify the geographic location, `current` defines which weather variables to return, and `timezone=auto` requests timestamps in the local time zone.

The API acts as a function boundary across the network. Just as you might call a function in a programming language with specific parameters, this API call provides parameters that declare the requester's intent. The server processes these parameters, performs the necessary computations, and returns machine-readable results.

APIs typically support controlled variation through different parameter combinations. For instance, the temperature unit can be changed from Celsius to Fahrenheit by adding `&temperature_unit=fahrenheit` to the request:

```
curl "https://api.open-meteo.com/v1/forecast?latitude=21.3069&longitude=-
157.8583&current=temperature_2m,wind_speed_10m,is_day&timezone=auto&temperature_unit=fahrenheit"
```

POST Requests: Country Information Lookup

The second example demonstrates a POST request, which is commonly used when sending more complex data to a server or when the request parameters are too large or complex for a URL query string.

This example queries a GraphQL endpoint for country information:

```
curl 'https://countries.trevorblades.com/' \
  -X POST \
  -H 'Content-Type: application/json' \
  --data '{"query":"{ country(code:\"US\") { name capital currency languages { code name } } }"}'
```

The server responds with structured data that mirrors the request format:

```
{
   "data":{
      "country":{
         "capital":"Washington D.C.",
         "currency":"USD,USN,USS",
         "languages":[
            {
               "code":"en",
               "name":"English"
            }
         ],
         "name":"United States"
      }
   }
}
```
Unlike the weather API, which uses URL parameters to specify what data to retrieve, this API uses a JSON body containing a query field. The server evaluates this query and returns a data object whose structure matches the requested fields. Changing the country code (for example, from "US" to "JP" for Japan or "BR" for Brazil) produces different results without modifying the endpoint URL, demonstrating how the POST body serves as the primary mechanism for customizing server behavior.

These two examples illustrate complementary approaches to API interaction: one parameterized through URL query strings and the other through JSON request bodies. Both return structured JSON data suitable for further processing, and both exemplify the stateless interaction model that underlies modern web services.

How Web APIs Work

The Request-Response Model

Web APIs operate on a fundamental request-response pattern over HTTP (Hypertext Transfer Protocol). When an application needs data from a remote service, it sends an HTTP request to a specific endpoint (URL). The server processes this request and returns an HTTP response containing the requested data or an indication of what went wrong.

Each HTTP request contains several key components:

Endpoint: The specific URL where the API can be accessed. This includes the base URL (like `https://api.open-meteo.com`) and the specific path to the resource (`/v1/forecast`).

Method: The type of operation being requested. Common methods include:

- GET: Retrieve data without making changes
- POST: Send data to create new resources or perform complex operations
- PUT: Update existing resources -- typically used to update some database values
- DELETE: Remove resources

Headers: Additional metadata about the request, such as authentication credentials, content type specifications, or API version preferences, when the API supports multiple versions.

Body: The actual data being sent with the request, typically in JSON format for modern APIs. GET requests usually don't include a body, while POST and PUT requests often do.

The server responds with an **status code** indicating whether the request succeeded (codes in the 200s), encountered a client error (codes in the 400s), or experienced a server error (codes in the 500s), along with the requested data or error information.

RESTful API Architecture

Many modern web APIs follow REST (Representational State Transfer) principles, which provide a standardized approach to designing web services. REST treats data as resources identified by URLs and uses standard HTTP methods to perform operations on these resources.

In RESTful design, each resource is represented as a noun in the URL structure. For example, a blog API might organize resources like this:

- `/articles` - represents the collection of all articles
- `/articles/123` - represents a specific article with ID 123
- `/authors/456` - represents a specific author with ID 456

The HTTP method indicates the type of operation:

- `GET /articles` retrieves a list of articles
- `POST /articles` creates a new article(s) (data sent as JSON determines how many articles are submitted)
- `GET /articles/123` retrieves a specific article
- `PUT /articles/123` updates the entire article (data sent as JSON determines what gets updates)
- `DELETE /articles/123` removes the article

RESTful APIs maintain statelessness, meaning each request contains all the information needed to process it. The server doesn't retain information about previous requests, which makes the system more predictable and easier to scale information about previous requests, which makes the system more predictable and easier to scale.

The JSON You'll See Everywhere

Working with APIs inevitably means working with JSON (JavaScript Object Notation), the universal language of modern web services. While APIs can technically exchange data in various formats, JSON has become so dominant that understanding it is essential for anyone who wants to work with APIs effectively.

JSON Structure Fundamentals

JSON organizes data using a simple set of building blocks that combine to represent complex information structures. These building blocks mirror concepts found in most programming languages.

Objects are collections of key-value pairs enclosed in curly braces `{}`. They function similarly to dictionaries in Python. Each key must be a string, and values can be any valid JSON data type. The weather API example from earlier demonstrates this structure:

```
"current": {
  "temperature_2m": 31.3,
  "wind_speed_10m": 16.2,
  "is_day": 1
}
```

Arrays represent ordered lists of items enclosed in square brackets `[]`.

```
"languages": [
  { "code": "en", "name": "English" },
  { "code": "fr", "name": "French" },

]
```

This array contains a two objects. Arrays can hold multiple items of the same or different types.

Primitive values represent basic data types that don't contain other values. JSON supports several primitive types:

- Strings: text enclosed in double quotes, like `"Pacific/Honolulu"`
- Numbers: integers or decimals like `31.3` or `900`
- Booleans: `true` or `false` (always lowercase)
- Null: `null` to represent empty or missing values

These building blocks combine to create complex data structures.

Critical Syntax Rules

JSON's syntax rules are stricter than many programming languages, and small violations can cause entire API requests to fail. Understanding these requirements prevents common errors that can be frustrating to debug.

Double quotes are mandatory for all strings, including object keys. Single quotes, which are valid in many programming languages, will cause JSON parsing to fail. The key `"temperature_2m"` is valid, but `'temperature_2m'` or the unquoted `temperature_2m` are not.

Trailing commas are forbidden in JSON. While a trailing comma after the last element is acceptable (and often helpful) in many programming languages, JSON strictly prohibits this. The object `{"a": 1, "b": 2}` is valid, but `{"a": 1, "b": 2,}` will cause an error.

Boolean and null values have exact spelling requirements. JSON uses lowercase `true`, `false`, and `null`. Python's `True`, `False`, and `None` are not valid JSON.

These syntax rules exist because JSON is designed to be language-independent. The strict formatting ensures that any system can parse JSON consistently, regardless of the programming language being used.

JSON vs Python Dictionary

The similarity between JSON and Python dictionaries often causes confusion for new programmers. While they serve similar purposes and have comparable structures, they use different syntax rules and exist in different contexts.

A Python dictionary might look like this:

```
{'name': 'John', 'active': True, 'scores': [85, 92, 78]}
```

The equivalent JSON representation would be:

```
{"name": "John", "active": true, "scores": [85, 92, 78]}
```

The key differences are the quote types (double vs single), boolean capitalization (`true` vs `True`), and context. The Python dictionary exists as a data structure within a Python program, while JSON exists as a text format for storing data r transmitting it between systems.

Python provides built-in functions to convert between these formats. The `json.loads()` function converts a JSON string into Python data structures, while `json.dumps()` converts Python data structures into JSON strings. These functions handle the syntax translations automatically, allowing Python programs to work seamlessly with JSON APIs. The function `json.dumps()` convert a python object in valid json format

```
In [3]:python_dict = {"first": 'john', 'last': 'Doe', 'is_stuedent': False, 'salary': None}
    print(python_dict)
{'first': 'john', 'last': 'Doe', 'is_stuedent': False, 'salary': None}
In [12]:import json
     json_string = json.dumps(python_dict)
     print(json_string)
     type(json_string)
{"first": "john", "last": "Doe", "is_stuedent": false, "salary": null}
Out[12]:str
In [13]:print(json.loads(json_string))
Out[13]:{'first': 'john', 'last': 'Doe', 'is_stuedent': False, 'salary': None}
```

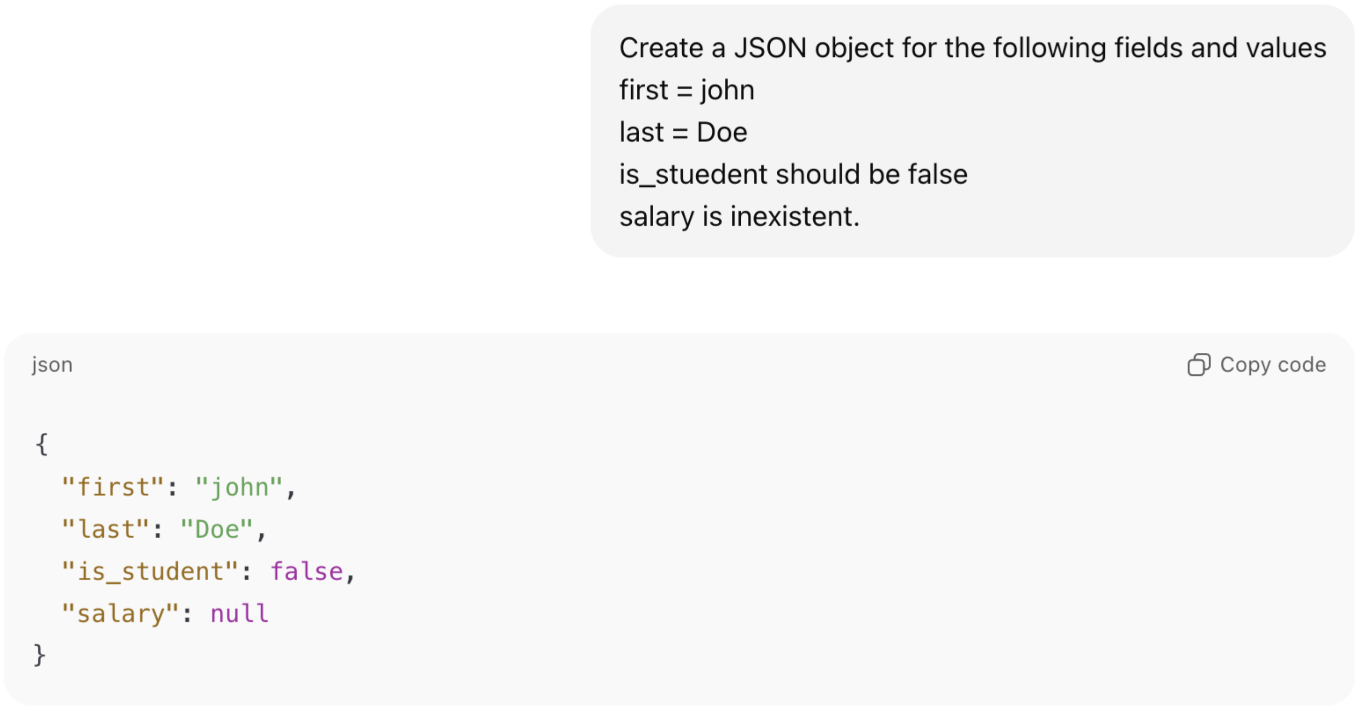
Why JSON Dominates APIs

JSON's widespread adoption in API design stems from several practical advantages that make it well-suited for modern web services.

Language independence allows JSON to work across different programming environments. Whether you're writing Python, or virtually any other modern language, libraries exist to parse and generate JSON. This universality makes JSON an ideal format for APIs that need to serve diverse client applications.

Human readability makes JSON easy to understand and debug. Unlike binary formats that require specialized tools to interpret, JSON can be read directly by developers. This readability becomes invaluable when troubleshooting API integrations or examining data structures returned by unfamiliar services.

LLM compatibility has become increasingly important as artificial intelligence tools integrate into more workflows. Language models excel at generating and parsing JSON, making it the preferred format for AI-powered applications. All modern commercial language models can produce valid JSON, and many can be instructed to format their responses as structured JSON objects.


Create a JSON object for the following fields and values
first = john
last = Doe
is_stuedent should be false
salary is inexistent.

```json
{
  "first": "john",
  "last": "Doe",
  "is_student": false,
  "salary": null
}
```

The combination of these factors has made JSON the de facto standard for API communication. While other formats still exist in legacy systems, JSON has become so dominant that most developers expect it as the default format for new API designs.

Making API Requests with Python

The best way to understand how APIs work is to interact with one directly. While API testing tools like Postman are useful for experimentation, real applications use API libraries make such requests. Python's `requests` library is the standard tool for this purpose—it handles all the HTTP complexity while providing a simple, intuitive interface.

Setting Up the Demo

The demonstration uses the Open-Meteo weather API because it requires no authentication or registration. This removes barriers that might complicate a first API experience and allows immediate focus on the core request-response mechanics.

First, ensure you have the `requests` library installed:

```
pip install requests
```

Making the First GET Request

Here's the same weather API request implemented in Python:

In [14]:
```python
import requests

# Make the GET request
url = "https://api.open-meteo.com/v1/forecast"

params = {
    "latitude": 21.3069,
    "longitude": -157.8583,
    "current": "temperature_2m"
}

response = requests.get(url, params=params)
response
```

Out[14]:
```
<Response [200]>
```

In [15]:
```python
if response.status_code == 200:
    data = response.json()
    print(data)
else:
    print(f"Error: {response.status_code}")
```

```
{'latitude': 21.375, 'longitude': -158.0, 'generationtime_ms': 0.013113021850585938, 'utc_offset_seconds': 0, 'timezone': 'GMT', 'timezone_abbreviation': 'GMT', 'elevation': 4.0, 'current_units': {'time': 'iso8601', 'interval': 'seconds', 'temperature_2m': '°C'}, 'current': {'time': '2025-09-02T00:00', 'interval': 900, 'temperature_2m': 32.1}}
```

This Python code does exactly what clicking "Send" in Postman does. The `requests.get()` function formulates an HTTP GET request, transmits it to Open-Meteo's servers, and returns a response object containing the server's reply.

The response contains the same structured JSON data:

```json
{
   "latitude":21.375,
   "longitude":-158.0,
   "generationtime_ms":0.013113021850585938,
   "utc_offset_seconds":0,
   "timezone":"GMT",
   "timezone_abbreviation":"GMT",
   "elevation":4.0,
   "current_units":{
      "time":"iso8601",
      "interval":"seconds",
      "temperature_2m":"°C"
   },
   "current":{
      "time":"2025-09-02T00:00",
      "interval":900,
      "temperature_2m":32.1
   }
}
```

The Power of the Requests Library

The `requests` library abstracts away the complexity of HTTP communication. Essentially, `requests` is the tool most Python developers use for API interactions. Behind the scenes, it handles URL encoding, header formatting, connection management, and response parsing. This single line of code:

```
response = requests.get(url, params=params)
```

The library automatically converts your `params` dictionary into the proper URL query string format and handles the HTTP protocol details transparently.

Understanding What Just Happened

This simple interaction demonstrates the fundamental API pattern that powers countless applications. Your computer sent a structured request to a remote server saying, in effect, "Give me current temperature data for these coordinates." The server processed this request by consulting its weather databases, performing any necessary calculations, and formatting the results according to its API specification.

The server's response includes both the requested data and important metadata. The HTTP status code (200 OK) indicates that the request succeeded without problems. Other status codes, as we will see at the end of this module, provide a standardized way for servers to communicate the outcome of requests, allowing client applications to respond appropriately to different situations.

The response data itself in this example follows the JSON format discussed in the previous section. Notice how the information is organized into nested objects with descriptive keys like `current_units` and `current`. This structure makes the data self-documenting. So, you can often understand what each piece of information represents without referring to separate documentation.

Comparing Different Requests

The power of APIs becomes more apparent when you see how changing the request parameters affects the response. Here's a second Python request that asks for both temperature and wind speed:

In [18]:
```
import requests

# Second request with additional parameter
url = "https://api.open-meteo.com/v1/forecast"
params = {
    "latitude": 21.3069,
    "longitude": -157.8583,
    "current": "temperature_2m,wind_speed_10m"  # Added wind_speed_10m
}

response = requests.get(url, params=params)
data = response.json()
print(data)
```

```
{'latitude': 21.375, 'longitude': -158.0, 'generationtime_ms': 0.03600120544433594, 'utc_offset_seconds': 0, 'timezone': 'GMT', 'timezone_a
bbreviation': 'GMT', 'elevation': 4.0, 'current_units': {'time': 'iso8601', 'interval': 'seconds', 'temperature_2m': '°C', 'wind_speed_10m': 'km/h'
}, 'current': {'time': '2025-09-02T00:15', 'interval': 900, 'temperature_2m': 32.1, 'wind_speed_10m': 9.4}}
```

The only difference is changing the `current` parameter from `"temperature_2m"` to `"temperature_2m,wind_speed_10m"`. This second request asks for both temperature and wind speed data from the same weather station. The response to this expanded request includes the additional information.

```
{
{
   "latitude":21.375,
   "longitude":-158.0,
   "generationtime_ms":0.03600120544433594,
   "utc_offset_seconds":0,
   "timezone":"GMT",
   "timezone_abbreviation":"GMT",
   "elevation":4.0,
   "current_units":{
      "time":"iso8601",
      "interval":"seconds",
      "temperature_2m":"°C",
      "wind_speed_10m":"km/h"
   },
   "current":{
      "time":"2025-09-02T00:15",
      "interval":900,
      "temperature_2m":32.1,
      "wind_speed_10m":9.4
   }
}
}
```

Notice how the `requests` library makes it simple to modify API parameters—just change the dictionary values and the library handles the URL formatting automatically.

Recognizing the Pattern

These two requests demonstrate a crucial principle of API design: you control what data you receive by modifying your request parameters. Both requests queried the same weather station at the same coordinates, but they asked different questions and received correspondingly different responses.

The first request essentially asked, "What's the current temperature at this location?" The server responded with temperature data and the metadata needed to interpret it correctly.

The second request asked, "What's the current temperature and wind speed at this location?" The server provided both pieces of information, along with units for each measurement.

This pattern appears consistently across well-designed APIs. Whether you're requesting user profiles from a social media platform (e.g., what the address and email associated with this username), product information from an e-commerce service (e.g., what's the rating associated with the product), or research data from an academic database (what's the abstract associated with this paper doi), you typically have control over which fields and how much information the API returns.

Understanding this principle helps explain why APIs are so powerful for building applications. Instead of retrieving massive datasets and filtering them locally, applications can request exactly the information they need. This approach reduces bandwidth usage, improves response times, and allows servers to optimize their responses for specific use cases.

Sending Data with POST Requests

The weather API demonstration showed how to retrieve information from servers using GET requests. However, GET requests have limitations when you need to send complex data or large amounts of information to a server. For instance, if you need to send a long address with spaces, special characters, or an image, this becomes very difficult, if not impossible, to encode in a URL. In such instances, it's common to limit GET requests to queries that require little (and simple) to no information to be sent, and use POST for requests that do need to send substantial data.

Understanding POST vs GET

The fundamental difference between GET and POST is about how data is transmitted and the practical limitations of each method.

GET requests put data in the URL. This works fine for simple parameters like coordinates or search terms, but becomes problematic when you need to send:

- Large amounts of data (URLs have length limits)
- Complex nested data structures
- Sensitive information (URLs are logged and cached)
- Data containing special characters that break URL formatting

POST requests put data in the request body. This allows you to send structured JSON data of any size without URL limitations. Even when you're just retrieving information, some APIs use POST because:

- The query is too complex for URL parameters
- The request contains sensitive data that shouldn't appear in URLs
- The API needs to accept rich, structured input

For example, a search API might use POST when you're sending complex filters, multiple search criteria, or when the search query itself is very long. That said, POST requests are also very common when creating data on the server, as one tends to send the data to store on the server, e.g., creating a new user in the database.

This is why many modern APIs, including most AI language model APIs, use POST requests even for operations that are fundamentally about retrieving information - the request body format is simply more flexible and appropriate for complex data exchange.

Setting Up a POST Request

To demonstrate POST functionality, this example uses JSONPlaceholder, a testing service that provides realistic API endpoints without requiring authentication or affecting real data. The service accepts new user data at the endpoint `https://jsonplaceholder.typicode.com/users` and simulates creating user records.

Creating a POST request requires configuring two essential elements that weren't necessary for GET requests:

HTTP Method: Change your API client from the default GET method to POST. This tells the server that you're sending data rather than requesting it.

Content-Type Header: Add a header specifying `Content-Type: application/json`. This informs the server that your request body contains JSON-formatted data, allowing it to parse and process the information correctly.

These configurations prepare the request to carry data payload, which is the primary distinction between GET and POST operations.

Structuring the Request Body

POST requests carry their data in the request body rather than in URL parameters. This approach allows for more complex data structures and larger payloads than would be practical in a URL.

For a complex query example, let's request detailed information about multiple countries using a GraphQL query:

```
{
  "query": "{ countries(filter: { continent: { eq: \"EU\" } }) { name capital currency languages {
name } population } }"
}
```

This JSON structure travels with the request as its payload, containing a query that requests all European countries with their capitals, currencies, languages, and population. Unlike GET request parameters that become part of the URL, this data remains in the request body, making it more secure and allowing for arbitrary complexity.

The request body can contain nested objects, arrays, and any valid JSON structure. This flexibility makes POST requests suitable for complex operations that require detailed input data.

Making the POST Request

When you send this POST request through your API client, the complete HTTP request includes the method, headers, and body working together. Open Postman and import the following query the click send.

```
curl -X POST https://countries.trevorblades.com/ \
 -H "Content-Type: application/json" \
 -d '{
   "query": "{ countries(filter: { continent: { eq: \"EU\" } }) { name capital currency languages {
name } } }"
 }'
```

The server receives this request, parses the JSON data from the body, and processes the GraphQL query to filter and return the requested country information.

Interpreting the Response

When you send this complex query request, the server responds with:

Status Code 200 (OK): This indicates successful processing of the query.

Response Body: Contains the actual country data matching your criteria:

```
{
  "data": {
    "countries": [
      {
        "name": "France",
        "capital": "Paris",
        "currency": "EUR",
        "languages": [
          { "name": "French" }
        ],
        "population": 67391582
      },
      {
        "name": "Germany",
        "capital": "Berlin",
        "currency": "EUR",
        "languages": [
          { "name": "German" }
        ],
        "population": 83240525
      }
    ]
  }
}
```

Notice how the response includes the requested nested data structure. This demonstrates why POST is often preferred for complex queries—the rich request structure enables sophisticated filtering and field selection that would be impossible with URL parameters.

In [23]:
```
import requests

url = "https://countries.trevorblades.com/"
data = {
    "query": "{ countries(filter: { continent: { eq: \"EU\" } }) { name capital currency languages {
}

response = requests.post(url, json=data)
print(response.json()['data']['countries'][:2], "...")
```

```
[{'capital': 'Andorra la Vella', 'currency': 'EUR', 'languages': [{'name': 'Catalan'}], 'name': 'Andorra'}, {'capital': 'Tirana', 'currency': 'ALL', 'lang
uages': [{'name': 'Albanian'}], 'name': 'Albania'}] ...
```

The Request-Response Pattern

This POST example demonstrates the bidirectional nature of API communication. While GET requests represent a simple question-and-answer interaction, POST requests represent a more complex exchange where the client provides substantial information for the server to process (e.g., filter it's data, create a new instance (entry) in the db, run some computation, etc.).

The JSON format serves as a structured envelope in both directions. Your client wraps the user information in JSON to send it to the server, and the server wraps its response in JSON to confirm the operation's success and provide relevant details.

This pattern appears consistently across different types of POST operations. POST requests power most interactive functionality on the modern web. Every time you fill out a contact form, create a social media post, upload a file, register for a service, or make a purchase online, POST requests carry your information to the appropriate servers.

API Authentication and Tokens

The public APIs demonstrated so far required no authentication—anyone could access weather data or test user creation without providing credentials. However, most real-world APIs require authentication to control access, track usage, and protect sensitive data. This section explores how API authentication works and why it's essential for working with production services.

Understanding API Tokens

API tokens function as digital identification cards that serve two critical purposes: proving who you are and determining what you're permitted to access. Like a keycard for a secure building, an API token grants entry to a service and controls which resources and operations are available to you.

These tokens are typically long, seemingly random strings of characters generated by the API provider. For example, a token might look like: `sk-abc123def456ghi789jkl012mno345pqr678stu901vwx234yz567`

The complexity and length of these strings make them extremely difficult to guess, providing security through cryptographic randomness rather than through secrecy of the authentication mechanism itself.

Most APIs require tokens to be included in the Authorization header of every request. This header follows a standard format: `Authorization: Bearer <your-token-here>` . The word "Bearer" indicates the type of authentication scheme being used, followed by your actual token.

Unlike request parameters that become part of the URL or data that travels in the request body, authentication headers provide identification information that accompanies every request without affecting the core functionality being requested.

How Token Authentication Works

The authentication process follows a straightforward verification pattern that happens transparently with each API request:

First, you include your token in the Authorization header of your HTTP request. This might look like:

```
GET https://api.example.com/v1/data
Authorization: Bearer sk-abc123def456ghi789jkl012mno345pqr678stu901vwx234yz567
Content-Type: application/json
```

When the server receives your request, it extracts the token from the Authorization header and validates it against its internal records. This validation confirms both that the token is legitimate and that it hasn't been revoked or expired (token have expiration dates, although many are set years ahead that that may not effectively expire).

If the token is valid, the server processes your request normally and returns the requested data or performs the requested operation. The response follows the same patterns demonstrated with public APIs, but the server has verified your authorization before processing the request.

If the token is invalid, missing, or expired, the server denies access and typically returns a 401 (Unauthorized) or 403 (Forbidden) status code along with an error message explaining the authentication failure.

This system allows API providers to control access to their services, track usage by individual users or applications, and enforce different permission levels. Some tokens might allow read-only access to certain resources, while others enable full create, update, and delete operations.

Token Security Best Practices

API tokens represent sensitive credentials that require careful handling throughout their lifecycle. Poor token security can lead to unauthorized access to your accounts, unexpected charges, or exposure of sensitive data.

Keep tokens confidential: Treat API tokens with the same care you would passwords or credit card numbers. Never share them in public forums, include them in screenshots, or copy them into documentation that others might see. If you need to show API requests in examples or tutorials, replace the actual token with placeholder text.

Avoid version control exposure: One of the most common security mistakes involves accidentally committing tokens to GitHub or other public repositories. Instead of hardcoding tokens in your source code, use environment variables or configuration files that are excluded from version control. Most programming environments provide mechanisms for loading sensitive configuration from external sources.

Use secure storage: Store tokens in environment variables, dedicated secret management services, or secure configuration files rather than embedding them directly in application code. Cloud platforms typically provide secure storage options specifically designed for API credentials and other sensitive configuration data.

Rotate compromised tokens: If a token is accidentally exposed—whether through a public repository, screenshot, or other means—regenerate it immediately through your API provider's dashboard. Most services allow you to create new tokens and revoke old ones without disrupting your applications if you update the credentials promptly.

Respect rate limits: API providers typically implement rate limiting to prevent abuse and ensure fair usage among all customers. These limits might restrict the number of requests per minute, hour, or month. Exceeding these limits can result in temporary blocking, token suspension, or additional charges. Monitor your usage and implement appropriate retry logic in your applications.

Alternative Authentication Methods

While bearer tokens are the most common authentication method for modern APIs, you'll encounter other approaches depending on the service and use case.

Basic authentication encodes username and password combinations in request headers. While simple to implement, this method is less secure than token-based systems and is primarily used for internal services or legacy systems.

OAuth 2.0 powers the familiar "Sign in with Google," "Login with Facebook," and similar authorization flows. This protocol allows users to grant your application permission to access their data without sharing their actual passwords. OAuth is particularly common for APIs that access user-generated content or personal information, as it provides a secure way for users to control what data they share and with which applications.

Important Distinction: API Tokens vs LLM Tokens

When working with language models and AI services, you'll encounter the word "token" in two completely different contexts, which can cause significant confusion for newcomers.

API tokens are the authentication credentials discussed in this section—long strings that identify you to the API provider and authorize your access to their services. These remain consistent throughout your interactions with a service and function as your digital identity.

LLM tokens refer to the units of text that language models use to process and generate content. In this context, a "token" might represent a word, part of a word, or even punctuation marks. Language models break text into these tokens for processing, and many AI services charge based on the number of tokens consumed by your requests.

These concepts are entirely unrelated despite sharing the same name. When someone mentions "tokens" in the context of AI APIs, the specific meaning depends on whether they're discussing authentication (API tokens) or text processing and billing (LLM tokens).

In [ ]:
```
### Hands-On: Using Your OpenAI API Token

Having covered the principles of API authentication, the best way to understand how tokens work in pra

Before proceeding, you'll need an OpenAI API token. If you haven't obtained one yet, please ask the co

### Setting Up the Authenticated Request

This example demonstrates a complete API request that includes authentication, proper headers, and a J

**URL:**
```
https://api.openai.com/v1/chat/completions

Method: POST (because you're sending data, including an API token that should not be part of a URI

```bash
curl -X POST https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer YOUR_API_KEY_HERE" \
  -d '{
    "model": "gpt-4o",
    "messages": [
      {
        "role": "system",
        "content": "You are a helpful assistant that only speaks in pidgin."
      },
      {
        "role": "user",
        "content": "Hello!"
```

~~~
          }
        ]
      }'
    ```
~~~

Understanding the Request Structure

This API call replicates the functionality available through ChatGPT's web interface, but through programmatic access. Each component serves a specific purpose in communicating with the AI model:

The model parameter specifies which AI model to use for generating the response. OpenAI offers several models with different capabilities, performance characteristics, and pricing structures. The `gpt-3.5-turbo` model provides a good balance of capability and cost for most applications.

The messages array contains the conversation context that the AI model uses to generate appropriate responses. This array can include multiple messages representing an ongoing conversation, allowing the model to maintain context across exchanges.

System message establishes the AI's behavior, personality, and any specific instructions about how it should respond. This message type doesn't appear in the conversation history visible to users but influences how the AI interprets and responds to user messages.

User message contains the actual question, prompt, or input that you want the AI to respond to. In a multi-turn conversation, you would include previous user and assistant messages in the array to maintain context.

Expected Response Structure

When you send this request, OpenAI's servers process your input through their language model and return a JSON response containing the AI's reply along with metadata about the request:

```
{
    "id": "chatcmpl-CB9f4cffaWRN1AM2inLB1CZox8rye",
    "object": "chat.completion",
    "created": 1756774782,
    "model": "gpt-4o-2024-08-06",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": "Wetin dey! How you dey?",
                "refusal": null,
                "annotations": []
            },
            "logprobs": null,
            "finish_reason": "stop"
        }
    ],
    "usage": {
        "prompt_tokens": 25,
        "completion_tokens": 8,
        "total_tokens": 33,
        "prompt_tokens_details": {
            "cached_tokens": 0,
            "audio_tokens": 0
        },
        "completion_tokens_details": {
            "reasoning_tokens": 0,
            "audio_tokens": 0,
            "accepted_prediction_tokens": 0,
            "rejected_prediction_tokens": 0
        }
    },
    "service_tier": "default",
    "system_fingerprint": "fp_f33640a400"
}
```

The AI's actual response appears in `choices[0].message.content`. The `usage` object shows token consumption for billing purposes—these are LLM tokens (text units) rather than API tokens (authentication credentials).

Authentication in Action

This example demonstrates token authentication working in practice. Your API token in the Authorization header proves to OpenAI's servers that you're authorized to access their services and provides them with your account information for billing and usage tracking.

Without a valid token, the request would fail with a 401 Unauthorized error, and you'd receive a JSON response explaining that authentication is required. With a valid but expired or revoked token, you might receive a 403 Forbidden error indicating that the credentials are recognized but no longer valid.

In [26]:
```
import requests

url = "https://api.openai.com/v1/chat/completions"
headers = {
   "Content-Type": "application/json",
   "Authorization": "Bearer YOUR_API_KEY_HERE"
}
data = {
   "model": "gpt-3.5-turbo",
   "messages": [
       {
           "role": "system",
           "content": "You are a helpful assistant."
       },
       {
           "role": "user",
           "content": "Hello!"
       }
   ]
}

response = requests.post(url, headers=headers, json=data)
print(response.json())
```

```
{'id': 'chatcmpl-CB9grJTR4niHqi4w93JcBnG45HEzx', 'object': 'chat.completion', 'created': 1756774893, 'model': 'gpt-3.5-turbo-0125', 'cho
ices': [{'index': 0, 'message': {'role': 'assistant', 'content': 'Hello! How can I assist you today?', 'refusal': None, 'annotations': []}, 'logprobs':
None, 'finish_reason': 'stop'}], 'usage': {'prompt_tokens': 19, 'completion_tokens': 9, 'total_tokens': 28, 'prompt_tokens_details': {'cached_to
kens': 0, 'audio_tokens': 0}, 'completion_tokens_details': {'reasoning_tokens': 0, 'audio_tokens': 0, 'accepted_prediction_tokens': 0, 'rejecte
d_prediction_tokens': 0}}, 'service_tier': 'default', 'system_fingerprint': None}
```

LLMs and APIs: Statelessness and Streaming

Language model APIs introduce two important concepts that distinguish them from simpler web services: statelessness and streaming responses.

Understanding Stateless APIs

Most APIs, including language model services, are stateless, meaning each request stands completely alone without any memory of previous interactions. The server doesn't retain information about what you asked before, what responses it provided, or any context from earlier in your session.

In the context of APIs, "state" refers to information about the current situation or context—what happened previously in a conversation, what data was processed, or what preferences were established. A stateful system would remember these details between requests, while a stateless system starts fresh with each interaction.

This design choice offers significant advantages for API providers. Stateless servers are easier to scale because any server can handle any request without needing to synchronize context information. They're also more reliable because there's no session state that can become corrupted or lost.

However, statelessness can seem counterintuitive when working with conversational AI. When you chat with ChatGPT through the web interface, it appears to remember your entire conversation. This apparent memory is actually an illusion—each API request includes the full conversation history that provides context to the language model.

Maintaining Context in Stateless Systems

To work effectively with stateless language model APIs, applications must include all necessary context in each request. For conversational applications, this means sending the complete message history that the AI needs to understand the current exchange.

Consider a multi-turn conversation where you first ask "What's the capital of France?" and then follow up with "What's the population?" The second question only makes sense with the context of the first. Since the API is stateless, your second request must include both the original question about France's capital and the AI's response, so the model understands that "What's the population?" refers to Paris.

This pattern appears in the OpenAI API's messages array structure:

```
{
  "model": "gpt-3.5-turbo",
  "messages": [
    {
      "role": "user",
      "content": "What's the capital of France?"
    },
    {
      "role": "assistant",
      "content": "The capital of France is Paris."
    },
    {
      "role": "user",
      "content": "What's the population?"
    }
  ]
}
```
Each API request includes the complete conversation context that the model needs to generate an appropriate response. Applications must manage this context accumulation, deciding what history to include and when to truncate older messages to stay within token limits.

Streaming with Server-Sent Events

Traditional API requests follow a simple pattern: send a request, wait for the complete response, then process the results. However, language models generate text incrementally, producing one token (roughly word-like piece) at a time. For longer responses, waiting for the complete generation can create noticeable delays that hurt user experience.

Many LLM APIs address this challenge by supporting streaming responses through Server-Sent Events (SSE), a web standard that allows servers to push data to clients as it becomes available. Instead of waiting for the complete response, the client receives each token as the model generates it, enabling real-time display of the AI's output.

The streaming process follows a modified request-response pattern. The client establishes a connection and sends a request that includes a `stream: true` parameter. Instead of returning a single JSON response, the server maintains the connection and sends a series of events, each containing the next generated token.

A streaming response might look like this sequence of events:

```
data: {"choices":[{"delta":{"content":"The"}}]}

data: {"choices":[{"delta":{"content":" capital"}}]}

data: {"choices":[{"delta":{"content":" of"}}]}

data: {"choices":[{"delta":{"content":" France"}}]}

data: {"choices":[{"delta":{"content":" is"}}]}

data: [DONE]
```

Each event contains a small piece of the response, allowing the client application to display text as it's generated rather than waiting for completion. This creates the familiar typing effect seen in ChatGPT and similar interfaces.

Connection Patterns and Implementation

Server-Sent Events operate over standard HTTP connections but modify the typical request-response cycle. The client initiates the connection with a normal HTTP request, but instead of receiving a complete response and closing the connection, the server keeps the connection open and pushes multiple response events.

This pattern is unidirectional—once the streaming begins, data flows only from server to client. The client cannot send additional information over the same connection. If the conversation continues, the client must establish a new connection for the next request.

Implementing streaming requires handling several technical considerations. The client must parse each event as it arrives, accumulate the tokens to build the complete response, and handle connection errors or interruptions gracefully. Many applications also implement timeout handling to avoid indefinitely waiting for responses that may never complete.

LLM Libraries and SDKs

While it's possible to interact with language model APIs using raw HTTP requests, most developers use specialized libraries or SDKs (Software Development Kits) that handle the complexity of API communication, authentication, error handling, and streaming.

These libraries package common HTTP operations into convenient functions, abstracting away details like header formatting, request signing, retry logic, and response parsing. They often provide streaming support that handles the Server-Sent Events protocol transparently, delivering tokens to your application through callbacks or iterators.

Popular libraries also include features like automatic rate limiting, request queuing, and error recovery that would be tedious to implement from scratch. Some provide higher-level abstractions that manage conversation context, token counting, and response formatting.

Orchestration tools represent another layer of abstraction, providing frameworks for building complex AI applications that might involve multiple API calls, data processing pipelines, and integration with various services. These tools handle workflow management, state coordination, and error handling across multi-step processes.

However, libraries and SDKs ultimately communicate with language model services using the same HTTP and JSON protocols demonstrated throughout this chapter. Understanding the underlying API mechanics helps you troubleshoot issues, implement custom functionality, or work with services that don't have dedicated libraries.

The principles covered in this chapter—HTTP methods, JSON formatting, authentication, and request-response patterns—remain relevant regardless of what tools you use to interact with APIs. Whether you're making raw HTTP calls or using sophisticated orchestration frameworks, the fundamental communication patterns stay the same.

Reading Errors Like a Pro

Working with APIs inevitably means encountering errors. Rather than viewing these as roadblocks, experienced developers treat error messages as valuable diagnostic information that guides troubleshooting and problem resolution. Understanding common error patterns and their meanings can transform frustrating debugging sessions into quick fixes.

Common HTTP Status Codes

HTTP status codes provide standardized communication about request outcomes. These three-digit codes appear in every API response and follow predictable patterns that indicate what happened and what action you might need to take.

Success codes (200-299) indicate that your request was processed successfully:

- **200 OK**: Your request succeeded and the server returned the requested data
- **201 Created**: Your POST request successfully created a new resource, commonly seen when creating user accounts or submitting forms

Client error codes (400-499) indicate problems with your request:

- **400 Bad Request**: Your request contains invalid data, malformed JSON, or missing required parameters. Check your request body structure and ensure all required fields are present
- **401 Unauthorized**: Your authentication credentials are missing, invalid, or expired. Verify your API token and ensure it's properly formatted in the Authorization header
- **403 Forbidden**: Your credentials are valid, but you don't have permission to access this specific resource or perform this operation
- **404 Not Found**: The requested resource doesn't exist. Check your URL path and ensure you're using the correct endpoint or resource ID
- **429 Too Many Requests**: You've exceeded the API's rate limits. Implement exponential backoff or reduce your request frequency

Server error codes (500-599) indicate problems on the API provider's side:

- **500 Internal Server Error**: The server encountered an unexpected condition. Try your request again after a brief delay
- **503 Service Unavailable**: The server is temporarily unable to handle requests, often due to maintenance or high load. Implement retry logic with increasing delays

Systematic Debugging Approach

When an API request fails, follow a methodical approach to identify and resolve the issue quickly.

Verify the basics first: Confirm that your URL is correct, including the protocol (https://), domain, path, and any required parameters. Ensure you're using the correct HTTP method—POST for sending data, GET for retrieving information.

Check authentication carefully: Verify that your Authorization header is present and properly formatted. The bearer token should be preceded by "Bearer " and match exactly what your API provider generated. Authentication issues cause a large percentage of API failures.

Validate your JSON: Use a JSON validator to ensure your request body is properly formatted. Common issues include trailing commas, single quotes instead of double quotes, or unescaped special characters within strings.

Review required headers: Ensure you're including all necessary headers, particularly `Content-Type: application/json` for requests that send JSON data. Missing or incorrect content type headers can cause servers to reject otherwise valid requests.

Consider rate limits: If you're receiving 429 errors, implement exponential backoff—wait briefly before retrying, then increase the delay with each subsequent failure. Many APIs include headers that indicate when you can make the next request.

Examine error response bodies: Don't just look at status codes—read the actual error messages returned by the API. These often contain specific information about what went wrong and how to fix it.

Error Response Patterns

Most modern APIs return structured error information that helps identify problems quickly. A typical error response might look like:

```
{
  "error": {
    "message": "Invalid JSON format in request body",
    "type": "invalid_request_error",
    "code": "malformed_json"
  }
}
```

This structure provides multiple levels of detail: a human-readable message explaining what went wrong, an error type that categorizes the problem, and sometimes a specific error code that your application can handle programmatically.

Some APIs include additional debugging information like the specific field that caused the error or suggestions for fixing the problem. Always read these details carefully—they often contain the exact information needed to resolve the issue.

Building Resilience

Professional applications handle errors gracefully rather than simply failing when problems occur. Implement retry logic for temporary failures like network timeouts or server errors, but don't retry client errors like 400 or 401 responses—these indicate problems with your request that won't be fixed by repetition.

Log error details for debugging purposes, but be careful not to log sensitive information like API tokens. Include enough context in your logs to reproduce the problem, such as the request URL, method, and sanitized parameters.

Consider implementing circuit breaker patterns for production applications that make many API requests. If an API becomes consistently unavailable, temporarily stop making requests rather than continuing to fail repeatedly.

Understanding error patterns and implementing robust error handling distinguishes professional API integrations from fragile prototypes. The debugging skills developed through working with APIs transfer to other areas of software development and make you more effective at diagnosing and resolving technical problems across different systems and technologies.

Putting It All Together: A Conversation Example

To demonstrate how all these concepts work together in practice, let's walk through a three-turn conversation with a language model API. This example shows how stateless APIs maintain context by including conversation history in each request.

First Request: Starting the Conversation

```
{
  "model": "gpt-3.5-turbo",
  "messages": [
    {
      "role": "user",
      "content": "What's the largest planet in our solar system?"
    }
  ]
}
```

The API responds:

```
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "Jupiter is the largest planet in our solar system."
      }
    }
  ]
}
```

Second Request: Building Context

For the follow-up question, we must include the entire conversation history:

```
{
  "model": "gpt-3.5-turbo",
  "messages": [
    {
      "role": "user",
      "content": "What's the largest planet in our solar system?"
    },
    {
      "role": "assistant",
      "content": "Jupiter is the largest planet in our solar system."
    },
    {
      "role": "user",
      "content": "How many moons does it have?"
    }
  ]
}
```

The API responds:

```
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "Jupiter has 95 confirmed moons, making it the planet with the most moons in our
solar system."
```

```
      }
    }
  ]
}
```

Third Request: Maintaining Full Context

The third request includes the complete conversation:

```
{
  "model": "gpt-3.5-turbo",
  "messages": [
    {
      "role": "user",
      "content": "What's the largest planet in our solar system?"
    },
    {
      "role": "assistant",
      "content": "Jupiter is the largest planet in our solar system."
    },
    {
      "role": "user",
      "content": "How many moons does it have?"
    },
    {
      "role": "assistant",
      "content": "Jupiter has 95 confirmed moons, making it the planet with the most moons in our
solar system."
    },
    {
      "role": "user",
      "content": "Which is the largest one?"
    }
  ]
}
```

Notice how each request contains the complete conversation history. Without this context, the API wouldn't understand that "How many moons does it have?" refers to Jupiter, or that "Which is the largest one?" asks about Jupiter's moons. The stateless API relies on your application to provide all necessary context in each request.

This pattern demonstrates why understanding APIs is essential for working with modern AI tools. Whether you're building chatbots, analysis tools, or creative applications, you'll use these same fundamental concepts: HTTP requests, JSON formatting, authentication tokens, and context management. The skills you've learned in this chapter provide the foundation for working with any web API, from simple data services to sophisticated AI platforms.